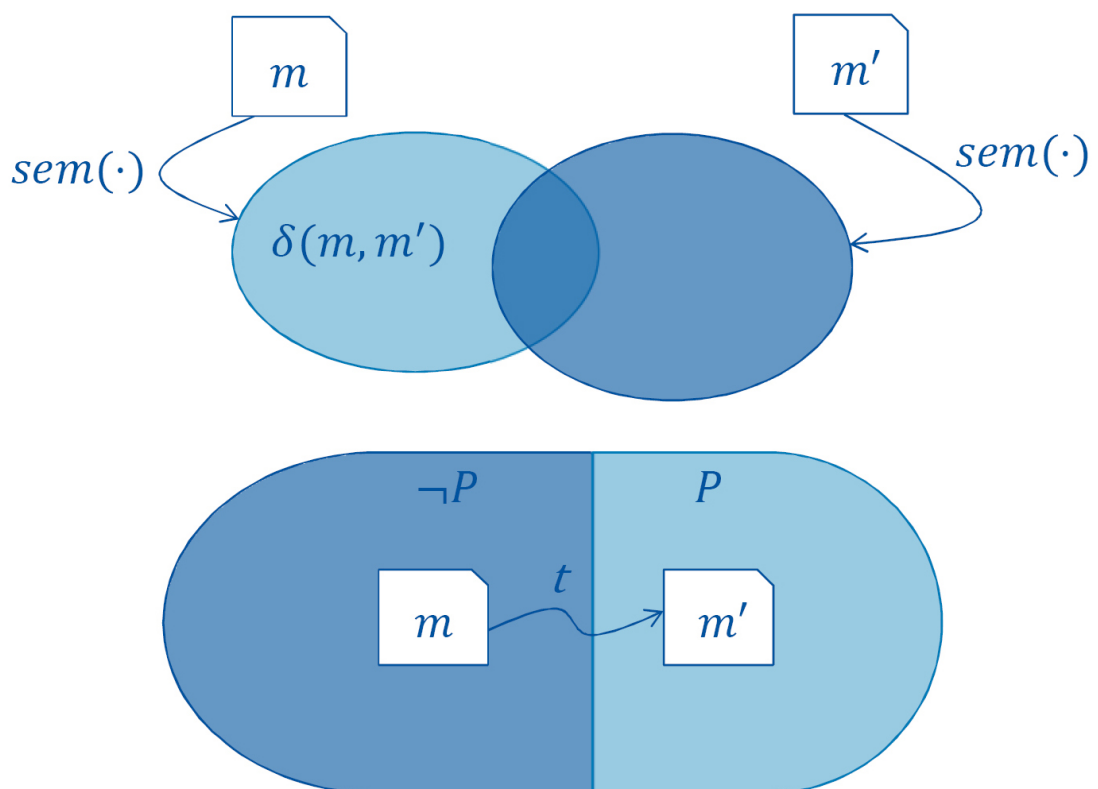


Oliver Kautz

Model Analyses Based on Semantic Differencing and Automatic Model Repair



Model Analyses Based on Semantic Differencing and Automatic Model Repair

Von der Fakultät für Mathematik, Informatik und Naturwissenschaften der
RWTH Aachen University zur Erlangung des akademischen Grades
eines Doktors der Naturwissenschaften genehmigte Dissertation

vorgelegt von

**M.Sc. RWTH
Oliver Kautz**

aus

Menden (Sauerland), Deutschland

Berichter: Universitätsprofessor Dr. rer. nat. Bernhard Rumpe
 University Professor Shahar Maoz, PhD

Tag der mündlichen Prüfung: 10. Februar 2021



[Kau21] O. Kautz:
Model Analyses Based on Semantic Differencing and Automatic Model Repair.
In: Aachener Informatik-Berichte, Software Engineering, Band 46. ISBN 978-3-8440-7926-5. Shaker Verlag, April 2021.
www.se-rwth.de/publications/

D 82 (Diss. RWTH Aachen University, 2021)

Eidesstattliche Erklärung

I, Oliver Kautz

erklärt hiermit, dass diese Dissertation und die darin dargelegten Inhalte die eigenen sind und selbstständig, als Ergebnis der eigenen originären Forschung, generiert wurden.

Hiermit erkläre ich an Eides statt

1. Diese Arbeit wurde vollständig oder größtenteils in der Phase als Doktorand dieser Fakultät und Universität angefertigt;
2. Sofern irgendein Bestandteil dieser Dissertation zuvor für einen akademischen Abschluss oder eine andere Qualifikation an dieser oder einer anderen Institution verwendet wurde, wurde dies klar angezeigt;
3. Wenn immer andere eigene- oder Veröffentlichungen Dritter herangezogen wurden, wurden diese klar benannt;
4. Wenn aus anderen eigenen- oder Veröffentlichungen Dritter zitiert wurde, wurde stets die Quelle hierfür angegeben. Diese Dissertation ist vollständig meine eigene Arbeit, mit der Ausnahme solcher Zitate;
5. Alle wesentlichen Quellen von Unterstützung wurden benannt;
6. Wenn immer ein Teil dieser Dissertation auf der Zusammenarbeit mit anderen basiert, wurde von mir klar gekennzeichnet, was von anderen und was von mir selbst erarbeitet wurde;
7. Teile dieser Arbeit wurden zuvor veröffentlicht und zwar in: [BKRW17, BKRW19, DKMR19, KR18a, KR18b, DKMR20].

Abstract

Models are the primary development artifacts used in model-driven software development. Therefore, models continuously evolve during the design, development, and maintenance of software systems. Thus, model differencing is an important task to understand the syntactic and semantic differences between model versions.

Previous work produced general (and thus language-independent) approaches for syntactic model differencing, but only a few language-dependent approaches for semantic model differencing. Approaches combining syntactic with semantic model differencing by relating the syntactic changes of models to their semantic differences rarely exist. Previous work neglected the development of language-independent approaches abstracting from a concrete model property for detecting the syntactic elements of a model, which cause that the model does not satisfy the property. If the property encodes a requirement and the non-satisfaction represents the existence of a bug, then detecting the syntactic model elements causing the non-satisfaction of the property facilitates developers in detecting the syntactic model elements causing the bug.

This thesis presents a framework for precisely defining modeling languages, including syntax, semantics, and model evolution possibilities. To demonstrate its feasibility, the framework is instantiated with four concrete modeling languages: Time-synchronous port automata, feature diagrams, sequence diagrams, and activity diagrams. For each of these modeling languages, this thesis presents syntactic and semantic differencing operators. The operators facilitate developers in understanding the syntactic and semantic differences between models of the languages. Based on the framework for precisely defining modeling languages, this thesis presents a modeling language and property-independent framework for automatic model repairs. The framework facilitates developers in detecting the syntactic elements of a model causing that the model does not satisfy a property. Instantiating the framework with a concrete modeling language and a concrete model property enables the automatic calculation of syntactic changes that transform a model not satisfying the property to a model that satisfies the property. The syntactic model elements affected by the syntactic changes can be interpreted to cause the non-satisfaction of the property. Developers can review the affected elements as evidence for the identification of the required changes for fixing the bug that causes the non-satisfaction of the property. Alternatively, the automatically calculated syntactic changes can be directly applied to the model to obtain a model that satisfies the property. The framework relies on the assumption that it is possible to partition the syntactic changes applicable to each model into finitely many model-specific and property-specific equivalence classes.

This thesis presents formal proofs for the correctness of the language-independent and language-dependent results. The applicability and usefulness of the modeling language-independent frameworks are demonstrated by instantiating the frameworks with four modeling languages and the properties refinement, generalization, and refactoring.

The instantiations of the frameworks with the four modeling languages and the example properties can be directly employed in development processes. The process of instantiating the frameworks is a methodology for the development of syntactic and semantic differencing procedures as well as precise model evolution analyses for detecting syntactic model elements causing the non-satisfaction of properties.

Danksagung

Zunächst danke ich meinem Doktorvater Prof. Dr. Bernhard Rumpe für die Möglichkeit zur Promotion am Lehrstuhl für Software Engineering. Die zahlreichen Diskussionen über die Inhalte dieser Arbeit und andere Themen waren sehr lehrreich und motivierend. Außerdem bin ich sehr dankbar für die gewährten thematischen Freiheiten bezüglich der Inhalte dieser Arbeit. Ich möchte mich auch für die Möglichkeit der Bearbeitung anderer, akademischer und industrienaher, Projekte neben dieser Dissertation bedanken.

Des Weiteren möchte ich Prof. Shahar Maoz, PhD, für die Zweitbegutachtung dieser Arbeit danken. Ich danke außerdem Prof. Dr. Klaus Wehrle für die Leitung des Prüfungskomitees und Prof. Dr. Erika Ábrahám für die Abnahme der Prüfung im Bereich der theoretischen Informatik.

Bedanken möchte ich mich außerdem bei meinen weiteren, teilweise ehemaligen, Kolleginnen und Kollegen für die schöne und erfolgreiche Zeit am Lehrstuhl. Diesbezüglich gilt mein Dank Jun.-Prof. Dr. Andreas Wortmann, Dr. Katrin Hölldobler, Dr. Judith Michael, Vincent Bertram, Arvid Butting, Joel Charles, Manuela Dalibor, Imke Drave, Robert Eikermann, Arkadii Gerasimov, Steffen Hillemacher, Nico Jansen, Jörg Christian Kirchhof, Evgeny Kusmenko, Achim Lindt, Matthias Markthaler, Lukas Netz, Deni Raco, David Schmalzing, Sebastian Stüber, Simon Varga, Louis Wachtmeister, Vassily Aliseyko, Marita Breuer, Niklas Dienstknecht, Christoph Engels, Joshua Mingers, Brian Sinkovec, Nina Pichler, Annika Donath, Galina Volkova, Lennart Bucher, Jerome Pfeiffer, Sylvia Gunder, Sonja Müßigbrodt, Dr. Timo Greifenberg, Dr. Markus Look, Dr. Klaus Müller, Dr. Pedram Mir Seyed Nazari, Dr. Christoph Schulze und Dr. Michael von Wenckstern. Für das Korrekturlesen früherer Fassungen dieser Arbeit bedanke ich mich bei Andreas, Arvid, David, Evgeny, Imke, Judith, Katrin, Nico, Robert, Sebastian, Simon und Steffen.

Ich möchte mich auch bei meiner Familie und meinen Freunden bedanken. Mein besonderer Dank gilt meinen Eltern Karin und Joachim sowie meiner Schwester Sabrina für die kontinuierliche Unterstützung meiner Vorhaben, die zu Anfertigung dieser Arbeit geführt haben. Zusätzlich bedanke ich mich bei Theodor, Hildegard, Till, Pia und Steffen. Auf euer Interesse und eure motivierende Unterstützung war immer Verlass.

Zuletzt bedanke ich mich von ganzem Herzen bei meiner Ehefrau Romina. Ich konnte mich immer auf deine volle Unterstützung verlassen, du hast mich immer motiviert und einige Opfer in unserem Privatleben erbracht, sodass ich mich bestmöglich meinem Promotionsvorhaben widmen konnte. Dafür bin ich dir zutiefst dankbar.

Contents

I	Prologue	1
1	Introduction	3
1.1	Context of the Thesis	5
1.2	Main Goals and Contribution	6
1.3	Thesis Organization	6
1.4	Notational Conventions and Mathematical Foundations	7
1.4.1	Sets and Functions	7
1.4.2	Finite and Infinite Words	8
1.4.3	Countable Sets	9
1.4.4	Nondeterministic Finite Automata	9
1.4.5	Büchi Automata	10
1.4.6	Graphs and Trees	11
1.5	Own Related Publications	11
2	A Generic Framework for Defining Modeling Languages	13
2.1	Modeling Language	14
2.2	Change Operations and Syntactic Differencing	19
2.3	Universe of Names	22
2.4	A Template for Describing Change Operations	22
2.5	Related Work	24
2.5.1	Modeling Language Definition and Variability	24
2.5.2	Syntactic Model Differencing and Change Operations	25
2.5.3	Semantic Model Differencing	26
II	Concrete Instantiations of the Generic Framework	31
3	Finite Time-Synchronous Port Automata	33
3.1	Time-synchronous Port Automata Syntax	35
3.2	Time-synchronous Port Automata Semantics	36
3.3	Semantic Differencing of Time-synchronous Port Automata	38
3.4	Time-synchronous Port Automata Change Operations	43
3.4.1	State-Addition Operations	44

3.4.2	State-Deletion Operations	45
3.4.3	Transition-Addition Operations	46
3.4.4	Transition-Deletion Operations	47
3.4.5	Input-Channel-Addition Operations	49
3.4.6	Output-Channel-Addition Operations	50
3.4.7	Channel-Deletion Operations	51
3.4.8	Initial-State-Change Operations	52
3.5	Time-Synchronous Port Automaton Modeling Language	53
3.6	Related Work	54
4	Feature Diagrams	57
4.1	Feature Diagram Syntax	58
4.2	Feature Diagram Semantics	60
4.3	Semantic Differencing of Feature Diagrams	63
4.4	Feature Diagram Change Operations	71
4.4.1	Feature-Addition Operations	72
4.4.2	Feature-Deletion Operations	73
4.4.3	Implies-Constraint-Addition Operations	74
4.4.4	Implies-Constraint-Deletion Operations	75
4.4.5	Excludes-Constraint-Addition Operations	76
4.4.6	Excludes-Constraint-Deletion Operations	77
4.4.7	Or-Group-Creation Operations	78
4.4.8	Xor-to-Or-Conversion Operations	79
4.4.9	Or-to-Xor-Conversion Operations	80
4.4.10	Mandatory-to-Optional-Conversion Operations	81
4.4.11	Optional-to-Mandatory-Conversion Operations	82
4.4.12	Feature-Group-Insertion Operations	83
4.4.13	Feature-Group-Exclusion Operations	85
4.4.14	Root-Rename Operations	85
4.5	Feature Diagram Modeling Language	86
4.6	Related Work and Discussion	88
5	Sequence Diagrams	91
5.1	Sequence Diagram Syntax	93
5.2	Sequence Diagram Semantics	95
5.3	Semantic Differencing of Sequence Diagrams	97
5.4	Sequence Diagram Change Operations	107
5.4.1	Object-Addition Operations	109
5.4.2	Object-Deletion Operations	110
5.4.3	Tag-Object-as-Complete Operations	112
5.4.4	Untag-Object-as-Complete Operations	113

5.4.5	Tag-Object-as-Visible Operations	114
5.4.6	Untag-Object-as-Visible Operations	115
5.4.7	Tag-Object-as-Initial Operations	116
5.4.8	Untag-Object-as-Initial Operations	118
5.4.9	Action-Addition Operations	119
5.4.10	Action-Deletion Operations	120
5.4.11	Interaction-Addition Operations	121
5.4.12	Interaction-Deletion Operations	122
5.5	Sequence Diagram Modeling Language	123
5.6	Related Work	124
6	Activity Diagrams	129
6.1	Activity Graph Syntax	131
6.2	Activity Graph Trace Semantics	133
6.3	Semantic Differencing of Activity Graphs	137
6.4	Activity Graph Change Operations	139
6.4.1	Label-Addition Operations	142
6.4.2	Label-Deletion Operations	143
6.4.3	Action-Insertion Operations	144
6.4.4	Action-Deletion Operations	144
6.4.5	Xor-Fragment-Insertion Operations	145
6.4.6	Xor-Fragment-Deletion Operations	146
6.4.7	And-Fragment-Insertion Operations	147
6.4.8	And-Fragment-Deletion Operations	148
6.4.9	Cyclic-Fragment-Insertion Operations	149
6.4.10	Cyclic-Fragment-Deletion Operations	150
6.4.11	Fragment-Branch-Insertion Operations	152
6.4.12	Fragment-Branch-Deletion Operations	152
6.5	Activity Diagram Modeling Language	154
6.6	Related Work	156
III	Automatic Model Repairs	159
7	A Framework for Automatic Model Repairs	161
7.1	Motivating Examples in Context of Repairing Refinement	163
7.1.1	Shortest Repair of a Failed Activity Diagram Refinement Step . .	164
7.1.2	Shortest Repair of a Time-Synchronous Port Automaton to Achieve the Satisfaction of a Requirement	166
7.1.3	Understanding a Feature Diagram Evolution Step	168
7.1.4	Understanding the Semantic Differences between Sequence Diagrams	169

7.2	Model Repair Problems	171
7.3	Change Operation Properties	176
7.4	Computing Shortest Repairing Change Sequences	183
7.5	Algorithms for Computing Shortest Solutions	189
7.5.1	Algorithm Performance	191
7.5.2	Checking Change Operation Properties	192
7.5.3	Propagating Properties Implying the Complement Model Property	194
7.5.4	Detecting Previously Explored Models	196
7.6	Applicability and Development Methodology	200
7.7	Composing Model Repair Problems	202
7.7.1	Derivation of Operations that Delay Solutions	203
7.7.2	Derivation of Operations that Induce Equally Long Shortest Solutions	206
7.7.3	Repair-Representative Function Derivation	209
7.8	Related Work and Discussion	211
8	Concrete Instantiations of the Model Repair Framework	215
8.1	Refines, Generalizes, and Refactors Model Repair Problems	215
8.2	Instantiations with the Time-Synchronous Port Automaton Language	217
8.2.1	Time-Synchronous Port Automaton Refinement Repair	218
8.2.2	Time-Synchronous Port Automaton Generalization Repair	220
8.2.3	Time-Synchronous Port Automaton Refactoring Repair	221
8.2.4	Repair-Representative Function and Example Applications	222
8.2.5	Implementation and Experiments	223
8.3	Instantiations with the Feature Diagram Language	231
8.3.1	Feature Diagram Refinement Repair	232
8.3.2	Feature Diagram Generalization Repair	234
8.3.3	Feature Diagram Refactoring Repair	236
8.3.4	Example Repair-Representative Function and Application	236
8.3.5	Implementation and Experiments	237
8.4	Instantiations with the Sequence Diagram Language	245
8.4.1	Sequence Diagram Refinement Repair	246
8.4.2	Sequence Diagram Generalization Repair	249
8.4.3	Sequence Diagram Refactoring Repair	251
8.4.4	Repair-Representative Function and Example Applications	252
8.4.5	Implementation and Experiments	253
8.5	Instantiations with the Activity Diagram Language	257
8.5.1	Activity Diagram Refinement Repair	259
8.5.2	Activity Diagram Generalization Repair	262
8.5.3	Activity Diagram Refactoring Repair	264
8.5.4	Repair-Representative Function and Example Applications	265

8.5.5	Implementation and Experiments	266
IV	Epilogue	273
9	Conclusion and Future Work	275
9.1	Summary and Main Results	275
9.2	Possible Future Work Directions	279
	Bibliography	281
A	Time-Synchronous Port Automata for Experimental Evaluations	301
B	Feature Diagrams for Experimental Evaluations	303
C	Sequence Diagrams for Experimental Evaluations	307
D	Activity Diagrams for Experimental Evaluations	309
	List of Figures	313
	Acronyms	319
	Glossary of Notation for Foundations	321
	General Glossary of Notation	323
	Index	329

Part I

Prologue

Chapter 1

Introduction

Software engineering has emerged to cope with the development complexity of large software systems [NR69]. The complexity of software systems is often caused by the conceptual gap between the problem domain and the low abstraction level of general-purpose programming languages (GPLs) [FR07]. One promising possibility to cope with the complexity is to describe software systems on a more abstract level than GPL source code. Model-driven development (MDD) is a paradigm for increasing the abstraction of software system descriptions by leveraging models as primary development artifacts [Sel03, Sch06, FR07, BCW17, KRR18]. “A model is an abstraction of an aspect of reality (as-is or to-be) that is built for a given purpose” [CFJ⁺16].

A model is an abstraction of an original that fulfills a purpose concerning the original [Sta73, MFBC12, Rum16]. In MDD, a model is thus an abstract representation of (parts of) a software system that fulfills a purpose concerning the software system. Developers use models to describe systems on a level of abstraction that is closer to the problem domain as GPL source code [Sel03, KRR18, FR07]. Models are used constructively to automatically generate (parts of) an executable software system or are interpreted (executed) in the context of a running system [Sel03, Sch06, FR07, KRR18]. Due to iterative development methodologies, changing requirements, and bug fixes, models continuously evolve during the design, development, and maintenance of software systems [MD08, MRR11a]. Therefore, model differencing is an important task [BKL⁺12] in MDD for detecting bugs and exploring design alternatives [MRR11a].

The fundamental entities for model differencing are *differencing operators* that can be used for model comparisons [MRR11a]. Research on model differencing includes syntactic model differencing, semantic model differencing, and hybrid approaches that combine syntactic and semantic model differencing.

Syntactic differencing focuses on detecting the differences between two models in terms of change operations (*e.g.*, add, delete, move, update) [AP03, EPK06, ASW09, XS07, GKLE13, BKL⁺12, KKT13, TELW14]. A syntactic differencing operator takes two models as input and outputs change operations [BKL⁺12]. The application of the change operations to one model yields the other model [AP03, MR15, MR18, KR18a]. However, syntactic differencing abstracts from the semantics of models. Two models may be syntactically different but semantically equivalent. Vice versa, two syntactically similar

models may have very different semantics. Syntactic differencing operators cannot detect these circumstances.

Semantic differencing approaches differ from syntactic differencing by focusing on differences in terms of the elements in the semantics of models [MRR11a, MRR12, MR18, KR18b]. For example, the semantic difference from a class diagram cd_1 to a class diagram cd_2 contains all object models in the semantics of cd_1 that are not object models in the semantics of cd_2 [MRR11e]. Similarly, the semantic difference from an activity diagram ad_1 to an activity diagram ad_2 contains all execution traces in the semantics of ad_1 that are not execution traces in the semantics of ad_2 [MRR11b, KR18b]. Semantic differencing approaches usually assume that models have a set-based denotational semantics. The elements in the semantics of a model represent mathematical abstractions of the possible realizations of the model. The *semantic difference* from a model m to a model m' is defined as the set of elements in the semantics of the model m that are not elements in the semantics of the model m' [MRR11a, MRR11g, MRR12, MR15, MR18, KR18a]. Each element in the semantic difference is called a *diff witness* and represents a possible realization of the model m that is not a possible realization of the model m' . If the semantics of the model m is included in the semantics of the model m' , then the semantic difference is empty. In this case, the model m can be interpreted to be a *refinement* [MRR11a, KR18a] of the model m' because every possible realization of m is also a possible realization of m' . Research in semantic differencing focuses on the development of language-specific semantic differencing operators [MRR11e, MRR11b, BKRW17, AHC⁺12, KR18b, DKMR19, DEKR19]. A semantic differencing operator is an automatic procedure to determine whether there are semantic differences from one model to another model. Previous research has only produced a few such procedures for well-accepted modeling languages as, for example, for parts of the Unified Modeling Language (UML) [OMG15]. Semantic differencing operators take two models of the same language as input and output either that no semantic difference exists or a finite non-empty set of diff witnesses. Developers can survey the diff witnesses to increase their understanding of the semantic difference from one model to another model. The diff witnesses also facilitate developers in detecting the syntactic model elements that cause semantic differences. However, semantic differencing approaches almost completely abstract from the syntax of models. Thus, they usually have difficulties to facilitate the detection of the syntactic model elements that cause the existence of semantic differences.

Approaches that combine syntactic differencing with semantic differencing rarely exist [MR15, MR18, KR18a]. A framework for determining the syntactic model elements that cause the existence of semantic differences from a model to another model is introduced in [KR18a]. The framework considers two models in isolation (without considering a changelog) and identifies syntactic changes that transform one model to a refinement of the other model. A framework for determining the changes contained in a changelog causing that a successor model version does not permit a specific realization is presented in [MR15, MR18]. The framework considers two models and a changelog between the

models. The analyses of the framework can be used to identify the syntactic changes contained in the changelog, which cause that a successor model version does not permit a concrete realization of its predecessor model version. The existing approaches combining syntactic with semantic model differencing are tailored towards concrete problems (*i.e.*, refinement and containment of a concrete realization) and, therefore, do not abstract from the model property of interest.

Model differencing is almost the opposite of model merging [MRR11a]. Model differencing is concerned with detecting the syntactic or semantic differences from one model to another model. One of the main goals of model differencing is to facilitate developers in understanding the differences between models. In contrast, merging is concerned with merging multiple models [Ber86, Wes10, TELW14, LvO92, Men02, BKL⁺12]. To this effect, model merging aims at identifying and resolving the conflicts [LvO92, BKL⁺12, Men02, Wes10, TELW14] between the models for constructing a merged model that incorporates aspects of all versions.

The goal of this thesis is to provide contributions in the contexts of semantic differencing and hybrid approaches that combine semantic and syntactic differencing with

1. one semantic differencing operator for each modeling language under consideration. The languages have heterogeneous semantic domains and semantic mappings.
2. a concrete modeling language- and property-independent method to repair models with respect to model properties (*e.g.*, refinement) for the development of model evolution analyses relating the syntax of models to their semantics.

In the following, Section 1.1 classifies this thesis in the context of the research conducted at the chair of Software Engineering RWTH Aachen University. Section 1.2 highlights the main goals and contributions. Afterwards, Section 1.3 overviews the organization of this thesis. Section 1.4 introduces notational conventions and mathematical foundations. Section 1.5 overviews the publications that have been published before and in the context of this thesis.

1.1 Context of the Thesis

The foundations for the research conducted in the context of this thesis are grounded in previous research undertaken at the chair of Software Engineering RWTH Aachen University. This section overviews this research.

The fundamental insight that the definition of every modeling language should constitute a syntax definition, the definition of a semantic domain, and the definition of a semantic mapping (independent of the degree of formality) is explicated in [HR04]. Based on this, the vision for the development of semantic differencing operators is presented in [MRR11a]. The vision led to the development of concrete semantic differencing

operators for class diagrams [MRR11e] and activity diagrams [MRR11b]. A method for summarizing similar semantic model differences to condensate and increase the amount of information presented to an engineer is presented in [MRR11g]. An interim summary on semantic model differencing [MRR12] summarizes the above research and presents possible future work directions in semantic model differencing. One future work direction includes the development of more language-specific differencing operators. Another future work direction is concerned with the combination of syntactic and semantic differencing approaches. This thesis addresses both of these directions.

1.2 Main Goals and Contribution

The main contributions of this thesis are:

1. A formal and concrete modeling language-independent methodology for the development of precise syntactic and semantic model evolution analyses.
2. A semantic differencing operator for finite time-synchronous port automata, an automaton variant for modeling finite message-driven interactive systems. The semantics of a time-synchronous port automaton consists of infinite communication histories of messages communicated via channels.
3. A semantic differencing operator for a feature diagram variant. The semantics of a feature diagram consists of the configurations that do not violate the constraints induced by the feature diagram.
4. A semantic differencing operator for a sequence diagram variant. The semantics of a sequence diagram consists of systems runs (encoding finite execution traces) that do not violate the constraints induced by the sequence diagram.
5. A semantic differencing operator for an activity diagram variant. The semantics of an activity diagram consists of the finite execution traces that are explicitly modeled in the activity diagram.
6. A concrete modeling language- and property-independent framework based on meaningful assumptions for computing syntactic changes to repair models with respect to their properties.
7. Instantiations of the model repair framework with the four concrete modeling languages and the concrete properties refinement, generalization, and refactoring.

1.3 Thesis Organization

This thesis is organized as follows:

- Chapter 2 presents a framework for precisely defining modeling languages and model evolution possibilities.
- Chapter 3 presents the time-synchronous port automaton modeling language, a semantic differencing operator, change operations, and a syntactic differencing operator for the time-synchronous port automaton modeling language.
- Chapter 4 presents the feature diagram modeling language, a semantic differencing operator, change operations, and a syntactic differencing operator.
- Chapter 5 presents the sequence diagram modeling language, a semantic differencing operator, change operations, and a syntactic differencing operator.
- Chapter 6 presents the activity diagram modeling language, a semantic differencing operator, change operations, and a syntactic differencing operator.
- Chapter 7 introduces the model repair framework, including an assumption on modeling languages that is sufficient to guarantee the computability of repairing syntactic changes, and algorithms for the computation of repairing changes.
- Chapter 8 presents instantiations of the model repair framework with the concrete modeling languages and refinement, generalization, and refactoring properties.
- Chapter 9 summarizes the thesis and presents possible directions for future work.

1.4 Notational Conventions and Mathematical Foundations

This section introduces notational conventions for foundations of this thesis.

1.4.1 Sets and Functions

The set of natural numbers is denoted by \mathbb{N} . Let A and B be two sets. $A \cup B$ denotes the union of A and B . $A \cap B$ denotes the intersection of A and B . $A \setminus B$ denotes the relative complement of B in A . $A \times B$ denotes the cartesian product of A and B .

The powerset of a set A is denoted by $\wp(A)$. The set of all finite subsets of a set A is denoted by $\wp_{fin}(A)$. A partition of a set A is a set $P \subseteq \wp(A)$ such that $\forall B, C \in P : B \cap C = \emptyset$ and $\bigcup_{B \in P} B = A$, *i.e.*, the sets in P are pairwise disjoint and the union of the sets in P is equal to A .

A binary relation \sim on a set A is said to be an equivalence relation iff it is reflexive ($a \sim a$ for all $a \in A$), symmetric ($a \sim b$ iff $b \sim a$ for all $a, b \in A$), and transitive ($a \sim b$ and $b \sim c$ implies $a \sim c$ for all $a, b, c \in A$).

We write $f : A \rightarrow B$ to denote that f is a total function from A to B . Similarly, $f : A \rightharpoonup B$ denotes that f is a partial function from A to B . We denote by $dom(f) \subseteq A$

the domain of a (partial) function $f : A \rightarrow B$ and write $f(x) = \perp$ to denote that $x \notin \text{dom}(f)$. Thus, $f(x) = \perp$ denotes that f is undefined on $x \in A$.

A function f is said to be finite iff $\text{dom}(f)$ is finite. For finite functions, we sometimes use a set-based notation that describes the underlying relation. For instance, the set $\{(1, a), (2, b)\}$ denotes the function $f : \{1, 2\} \rightarrow \{a, b\}$ where $f(1) = a$ and $f(2) = b$. We use the notation $x : y$ for denoting a tuple (x, y) in the set of a function. For instance, $\{(1, a), (2, b)\}$ and $\{1 : a, 2 : b\}$ denote the same function. For every (total or partial) function $f : A \rightarrow B$ and $R \subseteq A$, the restriction of f to R is defined as the function $f|_R : R \rightarrow B$ that satisfies $f|_R(x) = f(x)$ for all $x \in R$.

A function $f : A \rightarrow B$ is said to be injective iff $\forall x, y \in A : f(x) = f(y) \Rightarrow x = y$. A function $f : A \rightarrow B$ is said to be surjective iff $\forall y \in B : \exists x \in A : f(x) = y$. A function f is said to be bijective iff it is injective and surjective.

1.4.2 Finite and Infinite Words

Let Σ be an arbitrary set. We denote by

- Σ^* the set of all finite sequences (finite words) over the set Σ .
- ε the empty sequence, which is an element of Σ^* .
- Σ^∞ the set of all infinite sequences over Σ .
- $|s|$ the length of a sequence $s \in \Sigma^* \cup \Sigma^\infty$ where $|\varepsilon| = 0$ and $|s| = \infty$ for all $s \in \Sigma^\infty$.
- $\underline{s} \stackrel{\text{def}}{=} \{n \in \mathbb{N} \mid n \leq |s|\}$ the set of all natural numbers that are smaller than or equal to the length of a finite sequence $s \in \Sigma^*$.
- $s.i$ the $(i + 1)$ -th element of a sequence s where $i < |s|$.
- $s\&t$ the concatenation of two sequences $s, t \in \Sigma^* \cup \Sigma^\infty$. If $s \in \Sigma^\infty$, then $s\&t \stackrel{\text{def}}{=} s$. Similarly, $s\&a$ denotes the concatenation of the sequence $s \in \Sigma^* \cup \Sigma^\infty$ with the unique sequence of length one containing the symbol $a \in \Sigma$. If the context is clear, we simply write st instead of $s\&t$.
- \sqsubseteq the prefix relation over finite and infinite sequences, which is defined as usual $\forall s, t \in \Sigma^* \cup \Sigma^\infty : s \sqsubseteq t \Leftrightarrow \exists u : s\&u = t$.
- \sqsubset the true prefix relation, which is defined by $\forall s, t \in \Sigma^* \cup \Sigma^\infty : s \sqsubset t \Leftrightarrow s \sqsubseteq t \wedge s \neq t$.
- $s\downarrow i$ the prefix of the sequence s with length $0 \leq i \leq |s|$ where $s\downarrow 0 = \varepsilon$.
- $rt(s)$ the result from removing the first element from a non-empty sequence s .
- $a : s$ the result from prepending the symbol $a \in \Sigma$ to the sequence $s \in \Sigma^* \cup \Sigma^\infty$.

We lift the concatenation operator to sets of finite words as usual: Let Σ be a set. If $A \subseteq \Sigma^*$ and $B \subseteq \Sigma^*$ are two sets of finite words, then $A \& B \stackrel{\text{def}}{=} \{vw \mid v \in A \wedge w \in B\}$ denotes the set of words obtained from the pairwise concatenation of the words contained in A with the words contained in B . Similarly, if $A \subseteq \Sigma^*$ is a set of words and $c \in \Sigma$, then $A \& c \stackrel{\text{def}}{=} A \& \{c\}$ and $c \& A \stackrel{\text{def}}{=} \{c\} \& A$.

1.4.3 Countable Sets

A set C is said to be countable iff there exists an injective function $f : C \rightarrow \mathbb{N}$. This thesis uses the following standard results (e.g., [Hal60, End72, Rud76, Kan00]):

- The set of all finite sequences over a countable set is countable.
- Every finite cartesian product of countable sets is countable.
- The union of countably many countable sets is countable.
- The set of all finite subsets of a countable set is countable.
- Every subset of a countable set is countable.

1.4.4 Nondeterministic Finite Automata

Nondeterministic finite automata [RS59, HMU06] are abstract machines that act as acceptors for finite words over alphabets. An alphabet is a non-empty finite set that does not contain the empty word ε . The syntax of nondeterministic finite automata with epsilon-moves is defined as follows:

Definition 1.1. *A nondeterministic finite automaton with epsilon moves (NFA) is a tuple $(S, \Sigma, \delta, i, F)$ where*

- S is a finite set of states,
- Σ is an alphabet,
- $\delta \subseteq S \times (\Sigma \cup \{\varepsilon\}) \times S$ is a transition relation,
- $i \in S$ is an initial state, and
- $F \subseteq S$ is a set of final states.

The epsilon closure of a state in an NFA is the set of all states reachable from the state in the NFA by only following epsilon-transitions.

Definition 1.2. *Let $A = (S, \Sigma, \delta, i, F)$ be an NFA. The epsilon closure $E_A(q)$ of a state $q \in S$ in A is defined as the smallest set that satisfies the following two conditions:*

- $q \in E_A(q)$
- $\forall p \in E_A(q) : \forall (s, a, t) \in \delta : (s = p \wedge a = \varepsilon) \Rightarrow t \in E_A(q)$.

A run of an NFA $A = (S, \Sigma, \delta, i, F)$ on a word $w = w_1, w_2, \dots, w_n$ is a finite sequence of states $r_0, r_1, \dots, r_n \in S^*$ that satisfies the following three conditions:

1. $r_0 \in E_A(i)$,
2. for all $0 \leq j < n$, there exists $r' \in S$ such that $r_{j+1} \in E_A(r')$ and $(r_j, w_{j+1}, r') \in \delta$,
3. $r_n \in F$.

An NFA A accepts a word w iff there exists a run of A on w . The language recognized by A is defined as $\mathcal{L}_*(A) \stackrel{\text{def}}{=} \{w \in \Sigma^* \mid A \text{ accepts } w\}$.

1.4.5 Büchi Automata

Büchi automata [Büc90, Far02, Saf88] are abstract machines that serve as acceptors of infinite words over alphabets. The syntax of Büchi automata without epsilon-moves as used in this thesis is similar to the syntax of NFAs:

Definition 1.3. A Büchi automaton (BA) is a tuple $(S, \Sigma, \delta, i, F)$ where

- S is a finite set of states,
- Σ is an alphabet,
- $\delta \subseteq S \times \Sigma \times S$ is a transition relation,
- $i \in S$ is an initial state, and
- $F \subseteq S$ is a set of final states.

A Büchi automaton $B = (S, \Sigma, \delta, i, F)$ accepts a word $w = w_1, w_2, \dots \in \Sigma^\infty$ iff there exists an infinite sequence of states $s_0, s_1, \dots \in S^\infty$ that satisfies the following conditions:

1. $s_0 = i$,
2. $(s_j, w_{j+1}, s_{j+1}) \in \delta$ for all $j \in \mathbb{N}$,
3. $s_j \in F$ for infinitely many $j \in \mathbb{N}$.

The language recognized by B is defined as $\mathcal{L}_\omega(B) \stackrel{\text{def}}{=} \{w \in \Sigma^\infty \mid B \text{ accepts } w\}$.

1.4.6 Graphs and Trees

A (directed) *graph* is a tuple (V, E) where V is a set of nodes and $E \subseteq V \times V$ is a set of edges. A *walk* in a graph (V, E) is a finite sequence of nodes v_0, \dots, v_k with $k \geq 0$ such that $(v_{i-1}, v_i) \in E$ for all $0 < i \leq k$. A walk v_0, \dots, v_k in a graph G is said to be a walk from the node s to the node t iff $v_0 = s$ and $v_k = t$. A walk v_0, \dots, v_k in a graph G is called a *path* iff $v_i \neq v_j$ for all $0 \leq i, j \leq k$ with $i \neq j$. A walk v_0, \dots, v_k in a graph G is called a *cycle* iff $k > 0$ and $v_0 = v_k$. A graph G is said to be *acyclic* iff there does not exist a cycle in G .

A rooted tree is a tuple (V, r, E) where (V, E) is an acyclic graph, $r \in V$ is the root node of the tree, and there exists exactly one path from the root r to every node $v \in V$ in (V, E) . A path v_0, \dots, v_k in a tree (V, r, E) is said to be rooted iff $v_0 = r$. A rooted tree (V, r, E) is said to be infinite iff V is infinite. An infinite branch of a tree (V, r, E) is an infinite sequence of nodes v_0, v_1, \dots such that $v_0 = r$ and $(v_i, v_{i+1}) \in E$ for all $i \in \mathbb{N}$. If $T = (V, r, E)$ is a rooted tree and $(s, t) \in E$, then s is said to be the parent of the node t in T and t is said to be a child of s in T . A rooted tree $T = (V, r, E)$ is said to be finitely branching iff every node $s \in V$ has finitely many children in T .

This thesis uses the special case of König's Lemma [Kön27] for rooted trees:

- Every finitely branching infinite rooted tree with a countable set of nodes contains an infinite branch.

1.5 Own Related Publications

The results of this thesis are grounded in multiple years of research. Hence, various parts have been published before this thesis. The previously published contents of this thesis that are presented in the main sections were developed by the author of this thesis. The following overviews the publications that are related to the topics of the thesis:

- The semantic differencing method for time-synchronous port automata is presented in [BKRW17]. A further advanced automaton variant based on time-synchronous port automata and the identification of an automaton class that enables efficient semantic differencing is presented in [BKRW19].
- The semantic differencing method for the feature diagram modeling language is presented in [DKMR19].
- The semantic differencing method for the activity diagram modeling language is presented in [KR18b].
- The method for automatic model repairs is grounded in the method presented in [KR18a] for repairing failed model refinements. The model repair framework is

a generalization of the refinement repair framework presented in [KR18a]. More specifically, a refinement repair problem is a model repair problem that uses the property refinement.

- A semantic differencing method for statecharts that model the behavior of object oriented systems is presented in [DEKR19]. The underlying semantics is based on mapping statecharts to finite stimulus/reaction traces.
- Achievements, failures, and a possible future for model-based software engineering are presented in [KRR18]. The publication emphasizes that models need a precise definition (including syntax and semantics) to be usable for amenable tooling (*e.g.*, tooling for semantic differencing).
- A translation from class diagrams to the model checking tool Alloy [Jac06] is presented in [KMRR17]. The described translation details the translation described in [MRR11d] and can be used for various semantic class diagram analyses, such as semantic differencing, as presented in [MRR11e].
- A pre-study on the usefulness of syntactic and semantic differencing operators for developers is presented in [DKMR20].

Chapter 2

A Generic Framework for Defining Modeling Languages

This chapter presents a generic framework for precisely defining modeling languages and model evolution possibilities.

A generic framework for defining modeling languages is useful for capturing common concepts that are fundamental components of each modeling language. It enables the development of concrete modeling language-independent methods for solving model analysis problems. For instance, the model repair framework introduced in Chapter 7, the refinement repair framework [KR18a], and the Diffuse framework [MR15, MR18] abstract from concrete modeling languages.

The central notion of the generic framework for defining modeling languages is the term “model“. This thesis considers each model to be an abstract representation of (parts of) a software system. In MDD, models are used constructively to automatically generate (parts of) an executable software system or are interpreted (executed) in the context of a running system [Sel03, Sch06, FR07, KRR18]. As sketched in [BC11], a precisely defined semantics of models is necessary to disable ambiguities in the implementations of code generation and interpretation procedures.

One major disadvantage of many MDD projects is the lack of precisely defined semantics for the used models. The lack of precisely defined semantics quickly leads to misconceptions among different stakeholders of the development project [BC11] and may ultimately result in incorrect system implementations. For example, the UML [BC11, BMMR12, OMG15] is a modeling language without a precisely defined semantics that leaves much room for interpretation. A precisely defined semantics not only eliminates ambiguities of the modeled concepts but also enables the development of methods and tools for performing semantic model analyses [GJK⁺13]. Thus, in this thesis, the semantics of models is regarded as a fundamental component of each modeling language.

Models continuously evolve due to iterative development methodologies, changing requirements, and bug fixes [MRR11a]. Therefore, a precise concept for capturing all model evolution possibilities is necessary for the development of precise model evolution analyses that incorporate the syntactic changes of models. For this reason, the generic framework constitutes the notion of change operation. Each change operation describes

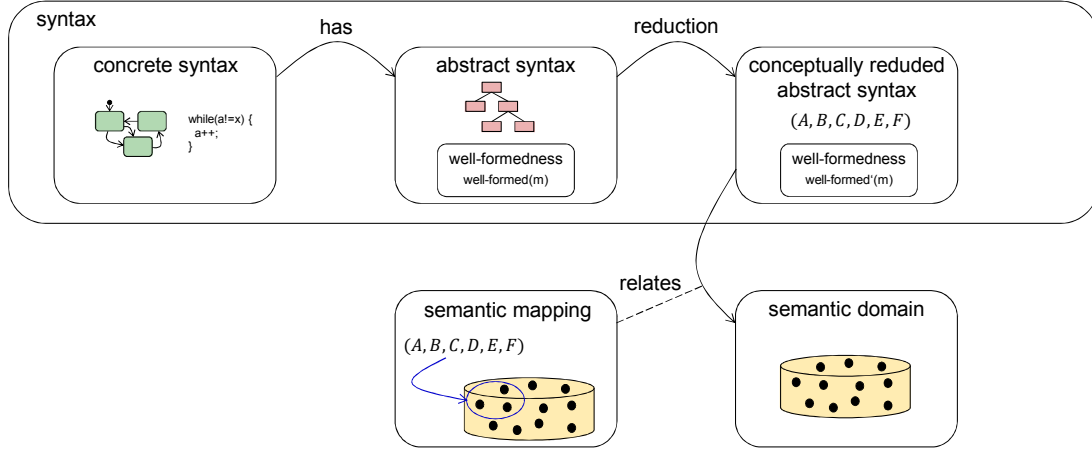


Figure 2.1: The conceptual parts of the complete definition of a modeling language inspired by [CGR09, Grö10].

(a part of) an evolution step from a predecessor model version to a successor version. A set of change operations can be used to describe all model evolution possibilities.

In the following, Section 2.1 introduces the framework for defining modeling languages. Subsequently, Section 2.2 introduces the notion of change operation. Section 2.3 discusses the usage of a uniform universe of names for syntactic model elements. Afterwards, Section 2.4 introduces a template for defining change operations. Finally, Section 2.5 presents related work.

2.1 Modeling Language

Figure 2.1 depicts the conceptual parts of a complete modeling language definition inspired by [CGR09, Grö10]. The concrete and abstract syntax are usually defined by grammars or metamodels [SBPM09, VBD⁺13, VC15, KRV10, HR17, Bet16]. Tools, such as EMF Ecore [SBPM09], MontiCore [KRV10, HR17], MPS [VBD⁺13], Neverlang [VC15], and Xtext [Bet16], facilitate the definition of the concrete or abstract syntax of models and often generate tooling for transforming models from their concrete syntax representation to their representation in the abstract syntax. The conceptually reduced abstract syntax is a simplified representation that can be derived from the abstract syntax by transforming complex syntactic constructs into simpler syntactic constructs such that the expressiveness of the modeling language is not reduced. The reduction enables defining the semantics of models containing complex syntactic constructs by transforming the constructs in the models into simpler constructs for which the semantics can be more easily defined directly. Thus, focusing on the reduced abstract syntax eases

the definition of the semantics of models. As sketched in Figure 2.1, the conceptually abstract syntaxes of the modeling languages presented in this thesis are defined mathematically using tuples. The semantic mapping relates each model to elements of the semantic domain.

The framework defined in this thesis abstracts from the internal syntactic properties of models and from the technology used to create the models' syntactic representation. Therefore, in the sense of this thesis, a precise definition of a modeling language consists of the definitions of the language's conceptually reduced abstract syntax, the semantic domain, and the semantic mapping [HR00, HR04, CGR09, Grö10, GR11].

Definition 2.1. *A modeling language \mathcal{L} is a tuple $\mathcal{L} = (M, Sem, sem)$ where*

- *M is a non-empty, countable set of models,*
- *Sem is a set, which is called semantic domain, and*
- *$sem: M \rightarrow \wp(Sem)$ is a semantic mapping.*

The definition abstracts from the internal details of models by simply assuming that the set M is a non-empty countable set of models. The set M represents the reduced abstract syntax of the modeling language. As argued in Section 2.3, the requirement for countability is not a limitation in practice. In this thesis, the countability assumption is exploited in Chapter 7 due to the use of König's Lemma [Kön27] (cf. Section 1.4).

The semantic domain Sem is a set of well-understood mathematical structures. For modeling languages that are used to describe the behavior of systems, such as statecharts, activity diagrams, and automata models, the semantic domain is usually a set that contains all traces of all executions that can be possibly modeled (*e.g.*, [Rum96, Rin14, BKRW17, KR18b, DEKR19]). The semantic domains for class diagram and object diagram modeling languages can be chosen as the set of all possible object structures (*e.g.*, [MRR11e, MRR11f, MRR11d, KMRR17]), which describe objects of a system and the links between them. A possible semantic domain for a feature diagram modeling language is the set of all possible feature configurations (*e.g.*, [AHC⁺12, DKMR19]), where each feature configuration is a finite set of feature names.

The semantic mapping sem maps each syntactically well-formed model $m \in M$ to its meaning $sem(m)$, which is a subset of the semantic domain. With this, a semantic mapping is a translation from an intuitive modeling notation (models) into mathematically well-understood entities. While models are typically finite in the sense that they are composed of finitely many modeling elements, the semantics of a model is usually an infinite set representing the model's possible realizations. While a model may be intuitively understandable for developers, the semantics of a model may be (and usually is) a complex and infinite set that is hard to understand on its own because it does not reveal the dependencies between its realizations.

As the semantic mapping is an axiomatic component of every modeling language, a semantic mapping cannot be proven to be right or wrong. It rather must be chosen in a way such that it adequately represents (an abstraction of) the real properties of the modeled system. Nevertheless, argumentations about the semantics, *i.e.*, whether a semantic mapping adequately represents the reality, are important. The semantics of the models must adequately represent the real system behaviors as realized by the used code generators or model interpreters. Otherwise, semantic model analyses may produce results that are incorrect with respect to the real properties of the system.

With the set-based semantic mapping, it is easy to model underspecification when interpreting each $r \in \text{sem}(m)$ as a possible realization of a model $m \in M$ [HKR⁺07]. As models are abstract representations of (parts of) a system, models are usually highly underspecified and thus permit multiple realizations. If a model does not permit any realization, then it is inconsistent [HKR⁺07]:

Definition 2.2. *Let $\mathcal{L} = (M, \text{Sem}, \text{sem})$ be a modeling language. A model $m \in M$ is called consistent in \mathcal{L} iff $\text{sem}(m) \neq \emptyset$.*

If \mathcal{L} is clear from the context, we say that a model m is consistent instead of saying that m is consistent in \mathcal{L} .

An inconsistent model contains some contradicting constraints in itself [HKR⁺07]. It can thus be interpreted to contain some bug, which needs to be fixed to make the model become meaningful.

The semantic difference from one model to another model is the set of elements in the semantics of the former model that are no elements in the semantics of the latter model [MRR11a]:

Definition 2.3. *Let $\mathcal{L} = (M, \text{Sem}, \text{sem})$ be a modeling language. The semantic difference from a model $m \in M$ to a model $m' \in M$ in \mathcal{L} is defined as $\delta_{\mathcal{L}}(m, m') \stackrel{\text{def}}{=} \text{sem}(m) \setminus \text{sem}(m')$.*

If \mathcal{L} is clear from the context, we write $\delta(m, m')$ instead of $\delta_{\mathcal{L}}(m, m')$ and say that $\delta(m, m')$ is the semantic difference from m to m' instead of saying that $\delta(m, m')$ is the semantic difference from m to m' in \mathcal{L} .

Each element $w \in \delta(m, m')$ is called a *diff witness* [MRR11a] (from m to m' in \mathcal{L}). Each diff witness represents a concrete proof for a semantic property of the model m that is no property of the model m' . It thus reveals a difference in the meaning of the two models. Due to changing requirements and the availability of additional information, all the information concerning the possible realizations of a system is usually not known a priori at the start of the development. The understanding of the system under development continuously increases. Therefore, developers start modeling highly underspecified models that capture their knowledge of a system at a certain stage during the development process and continuously change the models to incorporate new or

changed requirements. Semantic model differencing is an important task for developers to detect bugs and explore design alternatives. Specifically, developers are interested in the realizations of the successor model version that are no possible realizations of the predecessor version, and vice versa. Therefore, developers compare two model versions to detect the semantic difference from the successor version to the predecessor version and vice versa. When interpreting a model $m \in M$ as a successor version of a model $m' \in M$, then the set $\delta(m, m')$ contains exactly the elements added to the semantics of m' during the evolution to m . Vice versa, $\delta(m', m)$ contains exactly the realizations that have been removed from the semantics of m' during the evolution to m .

A *semantic differencing operator* [MRR11a] for a modeling language is an automatically executable method that takes two models of the language as input and outputs a finite set of diff witnesses contained in the semantic difference from one of the models to the other model if at least one exists. Developers can use semantic differencing operators and review the computed diff witnesses to increase their understanding of the semantic differences from one model to another model. Part II presents four semantic differencing operators for four modeling languages.

If additional information about a system under development becomes available, but no requirement changes, developers *refine* the models of the system by adequately changing the models to incorporate the newly obtained information. Then, the set of possible realizations of each refined model (capturing the available information) should be a subset of the possible realizations of the model's predecessor version. A model is a refinement of another model if the semantics of the former model is a subset of the semantics of the latter model [HKR⁺07]. Vice versa, if a model is a refinement of another model, then the latter model is a generalization of the former model. If two models are refinements of each other (*i.e.*, they are semantically equivalent), then the two models are called refactorings of each other.

Definition 2.4. Let $\mathcal{L} = (M, Sem, sem)$ be a modeling language and let $m, m' \in M$ be two models.

- m is called a *refinement* of m' in \mathcal{L} iff $sem(m) \subseteq sem(m')$.
- m is called a *generalization* of m' in \mathcal{L} iff $sem(m) \supseteq sem(m')$.
- m is called a *refactoring* of m' in \mathcal{L} iff $sem(m) = sem(m')$.

If \mathcal{L} is clear from the context, we say that a model m is a refinement (respectively generalization, refactoring) of a model m' instead of saying that m is a refinement (respectively generalization, refactoring) of m' in \mathcal{L} .

It is generally not the case that for each pair of models, one of the two models is a refinement, generalization, or refactoring of the other. In practice, most models are generally not related to each other via one of the three relations.

From the definition of semantic difference, it directly follows that a model is a refinement of another model iff the semantic difference from the model to the other model is empty, *i.e.*, it holds that $\text{sem}(m) \subseteq \text{sem}(m')$ iff $\delta(m, m') = \emptyset$.

If a model is a refinement of another model, all elements in the semantics of the former are included in the semantics of the latter. Thus, every predicate that holds for all realizations in the semantics of a model also holds for all realizations in the semantics of every refinement of the model: More formally, if $m \in M$ is a refinement of $m' \in M$, $P \subseteq \text{Sem}$ is a predicate on Sem , and it holds that $\text{sem}(m') \subseteq P$ (*i.e.*, all realizations of m' satisfy the predicate), then it holds that $\text{sem}(m) \subseteq P$ (*i.e.*, all realizations of m satisfy the predicate). With this interpretation, refinement is alternatively definable over the satisfaction of semantic predicates.

Proposition 2.1. *Let $\mathcal{L} = (M, \text{Sem}, \text{sem})$ be a modeling language and let $m, m' \in M$ be two models. Then, $\text{sem}(m) \subseteq \text{sem}(m')$ iff $\forall P \subseteq \text{Sem} : \text{sem}(m') \subseteq P \Rightarrow \text{sem}(m) \subseteq P$.*

Proof. Let \mathcal{L} , m , and m' be given as above.

" \Rightarrow ": Assume $\text{sem}(m) \subseteq \text{sem}(m')$ holds. Let $P \subseteq \text{Sem}$. If $\text{sem}(m') \subseteq P$ holds, then it also holds that $\text{sem}(m) \subseteq P$ because $\text{sem}(m) \subseteq \text{sem}(m')$.

" \Leftarrow ": Assume $\forall P \subseteq \text{Sem} : \text{sem}(m') \subseteq P \Rightarrow \text{sem}(m) \subseteq P$ holds. Define $P \stackrel{\text{def}}{=} \text{sem}(m')$. Using the assumption and $\text{sem}(m') \subseteq P$, we can derive $\text{sem}(m) \subseteq P = \text{sem}(m')$. \square

The alternative characterization of refinement can be exploited to analyze the evolution of models during system development. If a successor model version is a refinement of its predecessor version, then all semantic properties that hold for the predecessor version also hold for the successor version. Thus, every bug that can be represented by a semantic property of models that is absent in the predecessor version is guaranteed to be absent in the refined successor version.

The possibility to check whether a model refines another model is also useful on the receipt of new requirements. The addition of a new requirement to a set of existing requirements solely restricts the set of possible realizations: It might be the case that a new requirement becomes available and a model is changed to a successor version that is a refinement of its predecessor version and additionally satisfies the predicate corresponding to the new requirement. Then, the successor version is guaranteed to satisfy all previously available requirements too.

Vice versa, generalization is a useful property for reducing the examination of the satisfaction of a requirement by a more refined model to the examination of the satisfaction of the requirement by a more general model. If $P \subseteq \text{Sem}$ is a predicate on Sem that the realizations of a model m should satisfy and m' is known to be a generalization of the model m , then it suffices to check whether the realizations of the model m' satisfy the property to conclude that the realizations of m also satisfy the property. Analogously, if two models are refactorings of each other, then their realizations satisfy the same predicates on the semantic domain.

If the semantic difference from a model to another model is empty, then a semantic differencing operator outputs no diff witness. Therefore, if the semantic differencing operator always outputs a diff witness in the case at least one exists, then the semantic differencing operator can detect whether one model is a refinement, a generalization, or a refactoring of another model [MRR11a]. If requirements are represented by models, the semantic differencing operator can be used to automatically check whether a model satisfies the requirements.

The development of semantic differencing operators that can automatically detect refinement for Turing complete languages is impossible because, in general, language inclusion checking between Turing machines is undecidable [HU69]. In this thesis, we restrict our view to modeling languages for which the development of semantic differencing operators is possible.

2.2 Change Operations and Syntactic Differencing

Change operations describe how models can be changed. In this thesis, a change operation is a partial function from models to models.

Definition 2.5. Let $\mathcal{L} = (M, Sem, sem)$ be a modeling language. A change operation for \mathcal{L} is a partial function $o : M \rightarrow M$.

If \mathcal{L} is clear from the context, we say that o is a change operation instead of saying that o is a change operation for \mathcal{L} .

Change operations are partial functions because not all possible changes to a model are meaningful and a change may modify a model such that the result is not well-formed [KKT13, TELW14, GKLE13, MR15, MR18, KR18a]. For instance, adding a transition to an NFA that does not contain the transition's states results in a structure that is not a well-formed NFA.

Alternatively, it is also possible to model change operations by total functions as described in [Ste08] (where the term *edit* is used). There, if the application of a change to a model would result in an ill-formed model, then the corresponding change operation would leave the model unchanged. When modeling a change operation as a partial function, the domain of the change operation explicates the set of models for which the application of the change operation is meaningful.

A sequence of change operations is called *change sequence*. Change sequences can be used for describing steps for changing models. Change sequences can also be used for tracking model changes, for example, in a version control system.

Definition 2.6. Let $\mathcal{L} = (M, Sem, sem)$ be a modeling language and let O be a set of change operations for \mathcal{L} . For all models $m \in M$ and all change sequences $t \in O^*$, the

operator $\triangleright : M \times O^* \rightarrow M$ for applying change sequences is recursively defined as follows:

$$m \triangleright t = \begin{cases} m, & \text{if } t = \varepsilon \\ \perp, & \text{if } |t| \geq 1 \wedge t.0(m) = \perp \\ t.0(m) \triangleright rt(t), & \text{otherwise.} \end{cases}$$

We sometimes write $m \triangleright t \in M$ instead of $m \triangleright t \neq \perp$ or, equivalently, $(m, t) \in \text{dom}(\triangleright)$.

The application of a change sequence to a model iteratively applies the change operations contained in the sequence in order. If one of the change operations is not applicable to the model on which it is applied, then the complete change sequence is not applicable to the input model. The empty change sequence does not change the input model.

Definition 2.7. A countable set of change operations O is called *change operation suite* for \mathcal{L} iff each $o \in O$ is a change operation for \mathcal{L} .

If \mathcal{L} is clear from the context, we simply say that O is a change operation suite instead of saying that O is a change operation suite for \mathcal{L} .

Change operation suites are well suited to describe all evolution possibilities for the models of a modeling language. During development processes, it should be possible to change every model to any other model. Thus, meaningful change operation suites provide change operations such that every model can be changed to every other model by applying change sequences that consist of change operations of the change operation suite. These change operation suites are called *complete*.

Definition 2.8. A change operation suite O is called *complete* iff there exists a function $\Delta : M \times M \rightarrow O^*$ such that $\forall m, m' \in M : m \triangleright \Delta(m, m') = m'$.

The sequence $\Delta(m, m')$ (cf. Definition 2.8) is said to be a *syntactic difference* from m to m' . Each function satisfying the above property in Definition 2.8 is said to be a *syntactic differencing operator* for \mathcal{L} .

Models are usually encoded as finite structures. Hence, syntactic differencing operators often exist when using adequate change operations suites (cf. [AP03, CRP07, MR18, TELW14]). Usually, there is no unique syntactic differencing operator for a modeling language. Meaningful syntactic differencing operators map models to short, or even shortest, change sequences satisfying the property defined above.

The application of semantic differencing operators is usually computationally expensive. Therefore, when analyzing the semantic differences between two successive model versions, it is useful to be able to (partially) check whether the successor version refines its predecessor version by solely analyzing syntactic criteria [Rum96, BHP⁺98, PR97]. This automatic check is partially achievable with refinement calculi (e.g., [Rum96, BHP⁺98, PR97]). If every change operation in a change sequence changes every model (on which it is defined) to a refinement of the model, then the change sequence transforms every

model (on which it is defined) to a refinement of the model. This yields a sufficient, but not necessary, criterion for checking whether one model refines another model by analyzing a syntactic difference from the latter model to the former model. Analogous statements hold for generalizations and refactorings.

Definition 2.9. Let $\mathcal{L} = (M, Sem, sem)$ be a modeling language and let $o : M \rightarrow M$ be a change operation for \mathcal{L} .

- o is said to be *refining* iff $\forall m \in dom(o) : sem(o(m)) \subseteq sem(m)$.
- o is said to be *generalizing* iff $\forall m \in dom(o) : sem(m) \subseteq sem(o(m))$.
- o is said to be *refactoring* iff o is generalizing and refining.

Refining change operations are interpretable to be constructive realizations of the corresponding modeling language's refinement relation [Rum96]. Part II presents complete change operations suites for all modeling languages defined in this thesis. For each change operation, we analyze whether it is refining, generalizing, or refactoring. As described in Part III, the change operation properties defined in Definition 2.9 can also be used to achieve practical performance improvements for automatic model repairs.

Whether a change operation is refining, generalizing, or refactoring can be determined from the properties of one of its inverse operations:

Definition 2.10. A change operation o is called an *inverse* of a change operation p iff $\forall m \in dom(p) : p(m) \in dom(o) \wedge o(p(m)) = m$.

If an inverse of a change operation is refining, then the change operation is generalizing. Dually, if the inverse of a change operation is generalizing, then the change operation is refining. If an inverse of a change operation is refactoring, then the change operation is also refactoring.

Proposition 2.2. Let $\mathcal{L} = (M, Sem, sem)$ be a modeling language and let o, p be two change operations for \mathcal{L} such that o is an inverse of p .

1. If o is generalizing, then p is refining.
2. If o is refining, then p is generalizing.
3. If o is refactoring, then p is refactoring.

Proof. Let $\mathcal{L} = (M, Sem, sem)$, o , and p be given as above.

Proof of 1: Assume o is generalizing. Let $m \in dom(p)$ be a model. As o is an inverse of p , it holds that $p(m) \in dom(o)$ and $o(p(m)) = m$. As o is generalizing, it holds that $sem(p(m)) \subseteq sem(o(p(m))) = sem(m)$. Hence, p is refining.

Proof of 2: Assume o is refining. Let $m \in \text{dom}(p)$ be a model. As o is an inverse of p , it holds that $p(m) \in \text{dom}(o)$ and $o(p(m)) = m$. As o is refining, it holds that $\text{sem}(m) = \text{sem}(o(p(m))) \subseteq \text{sem}(p(m))$. Hence, p is generalizing.

Proof of 3: Assume o is refactoring. Let $m \in \text{dom}(p)$ be a model. As o is an inverse of p , it holds that $p(m) \in \text{dom}(o)$ and $o(p(m)) = m$. As o is refactoring, it holds that $\text{sem}(m) = \text{sem}(o(p(m))) = \text{sem}(p(m))$. Hence, p is refactoring. \square

2.3 Universe of Names

Models and change operations usually rely on names. For instance, automata-based modeling languages use names for states and transition labels. A uniform namespace enables the definition of meaningful change operations for modeling languages. For instance, the definition of a change operation for adding a state with a specific name to a finite automaton requires the knowledge that the name is a possible name for a state. For this reason, we assume the existence of an infinite set of names used in models. We use the same set of names for all modeling languages. We further require that the set of names is countable, which is of theoretical importance for the application of results in the context of model repairs (cf. Chapter 7) due to the use of König's Lemma [Kön27]. The requirement for countability is no restriction from a practical perspective: Developers introduce and reference model elements with names, which are usually finite strings over some fixed alphabet, such as the set of all alphanumerical letters ('a'-'Z', '0'-'9'). The set of all finite strings (words) over a finite alphabet is countably infinite (cf. Section 1.4). Thus, the set of all finite words over the set of all alphanumerical letters, for example, is an adequate universe of names. In the remainder of this thesis, let U_N denote this countably infinite set of names.

2.4 A Template for Describing Change Operations

The formal definitions of change operations for modeling languages with a complex syntax are sometimes cumbersome to define and difficult to understand. Therefore, this section introduces a notational convention for defining change operations by using an intuitive template. The remainder of this thesis uses this template for defining the change operations for the modeling languages presented in Part II.

Figure 2.2 illustrates the structure of the template. The change operation-specific parts are surrounded by curly brackets, consist of italic characters, and are highlighted in blue. Each template is divided into six sections.

The first section introduces the names of the described change operations and defines the signatures of the change operations. The second part contains assumptions and is used to specify the ranges of parameters used by the defined change operations. The third part consists of a short intuitive explanation of the change operation. The fourth

2.4 A TEMPLATE FOR DESCRIBING CHANGE OPERATIONS


Parameters	{Informal Change operation name} with signature {signature of the change operation}	
	{Assumptions that define the ranges of the parameters of the change operation}	
	{A short explanation of the change operation}	
	{Definition of the domain of the change operation}	
Application	{Representation of the input model of the change operation} {Formal identification of the change operation: \downarrow } {Representation of the output model of the change operation} where {definition of auxiliary values used in the representation of the output model}	
	<div style="display: flex; align-items: center; justify-content: space-around;"> <div style="border: 1px solid black; padding: 5px; text-align: center;">Example input Model</div> <div style="text-align: center;"> {Formal identification of the change operation}  </div> <div style="border: 1px solid black; padding: 5px; text-align: center;">Example output Model</div> </div>	

Figure 2.2: The template for defining change operations.

part defines the domain of the defined change operations. The fifth part defines the effect of applying the change operation. It contains a representation of the input model (in tuple notation) and a representation of the resulting model after applying the change operation (in tuple notation). A block introduced with the keyword *where* defines auxiliary values that are used in the representation of the resulting model. The sixth part illustrates an example application of the change operation. The input and output models are represented in an intuitive graphical notation.

Figure 2.3 depicts an example for the usage of the template where M_{NFA} denotes the set of all NFAs using state names and transition labels from the universe of names U_N . The first part states that the template defines change operations called transition-addition operations that have the signature $M_{NFA} \rightarrow M_{NFA}$. The second part introduces the assumption that $s, t \in U_N$ denote names of states and $l \in U_N$ denotes a transition label. The explanation states that the change operation $addT_{s,l,t}$ adds the transition (s, l, t) to an NFA that contains the states and uses the label. The fourth part defines the domain of the change operation $addT_{s,l,t}$ as the set of all NFAs that contain the states s and t in their set of states and the label l in their set of labels. The fifth part defines the effect from applying the change operation. In this case, if $A = (S, \Sigma, \delta, i, F) \in M_{NFA}$ is an NFA and $A \in dom(addT_{s,l,t})$, then $addT_{s,l,t}(A) = (S, \Sigma, \delta', i, F)$ where $\delta' = \delta \cup \{(s, l, t)\}$. The sixth part illustrates an example application of a transition-addition operation. The automata in the example are depicted in a graphical notation. In the example, the transition-addition operation $addT_{s,a,t}$ is applied to the automaton depicted on the left-hand side. The automaton on the right-hand side results from applying the change operation to the automaton that is depicted on the left-hand side.

Transition-addition operations with signature $M_{NFA} \rightarrow M_{NFA}$	
Parameters	Let $s, t \in U_N$ be two state names and let $l \in U_N$ be a transition label.
Explanation	The operation $addT_{s,l,t}$ adds the transition (s, l, t) to an NFA, if it contains the states and uses the label.
Domain	$(S, \Sigma, \delta, i, F) \in dom(addT_{s,l,t}) \Leftrightarrow s, t \in S \wedge l \in \Sigma$
Application	$ \begin{array}{c} (S, \Sigma, \delta, i, F) \\ \downarrow addT_{s,l,t} \\ (S, \Sigma, \delta', i, F) \text{ where } \delta' = \delta \cup \{(s, l, t)\} \end{array} $
Example	

Figure 2.3: An example template defining NFA change operations.

2.5 Related Work

This section presents related work on the definition of modeling languages (cf. Section 2.5.1), syntactic model differencing (cf. Section 2.5.2), and semantic model differencing (cf. Section 2.5.3).

2.5.1 Modeling Language Definition and Variability

The basic principle to represent a modeling language as a tuple of well-formed expressions (syntax), a semantic domain, and a semantic mapping is presented in [HR00, HR04]. The use of a set-based semantics for models is proposed in [HKR⁺07]. In this thesis, models also have a set-based semantics, which reflects by construction that models are usually highly underspecified.

Similar definitions of modeling language are used in the context of modeling language variability [CGR09, GR11, Grö10]. Modeling language variability is concerned with the possibility to define multiple modeling language variants, define common variants, and understand the differences and commonalities between different modeling languages. With this, modeling language variability at least concerns the abstract syntax, the semantic domain, and the semantic mapping of a modeling language. While this thesis abstracts from the concrete syntax of a modeling language, [CGR09, GR11, Grö10] additionally considers the concrete syntax of a modeling language as a variation point. Considering the concrete syntax of a modeling language is not necessary for this thesis. The language variability framework introduces a methodology to identify commonalities and differences between languages. This thesis focuses on model evolution analysis.

The description of syntactic modeling language variability is specialized to the case of the integrated definition of concrete and abstract syntax with grammars in [BEK⁺18, BEK⁺19]. This thesis abstracts from the technique used to define the abstract syntax and treats the available models as a possibly infinite set.

The abstract syntaxes of the modeling languages presented in this thesis are defined mathematically by using tuples. Change operations for the models of the languages are defined as partial mathematical functions taking a tuple representing a model as input and outputting a tuple representing the changed model. Alternatively, it could have been possible to define the syntaxes of the modeling languages with graph grammars or to define the change operations with graph transformation rules [Nag76, Nag79, AEH⁺99, Hec06, EEPT06]. The general mathematical notations using tuples and functions enable flexible and compact definitions of the abstract syntaxes and change operations. Graph grammars and graph transformation rules are special mathematical formalisms for defining languages of graphs and graph modifications. Thus, using general mathematical notations for defining the syntaxes and change operations enables defining at least as flexible and compact definitions as enabled by the use of the special formalisms. If this thesis used graph grammars or graph transformation rules, the definitions could have been less flexible and less compact. Less flexible and less compact definitions would have complicated several proofs that are part of this thesis. Nevertheless, the formalization of the languages and the transformations by means of graph grammars and graph transformations is interesting future work.

2.5.2 Syntactic Model Differencing and Change Operations

A modeling language-independent, generic procedure to determine a change operation suite for a modeling language and to compute the syntactic difference from one model to another model is described in [AP03]. The derived change operations represent the addition, deletion, and modification of model elements. The approach abstracts from the models' semantics and is neither concerned with refining nor generalizing nor refactoring change operations.

A method for determining the syntactic difference from a model to another model by using a model merging language is presented in [EPK06]. The method computes addition and deletion operations for adding and deleting model elements. Similarly, another approach where changes are represented by change operations for adding and removing model elements is presented in [MGH05].

An approach to syntactic differencing that computes change facts that are interpretable to represent syntactic differences in terms of operations for adding, deleting, moving, and renaming model elements is described in [XS07].

A method for traversing abstract syntax trees for computing the syntactic difference between models in terms of additions, deletions, and shifts of subtrees of the abstract syntax tree is presented in [OWK03].

An approach for the comparison of process models is presented in [KGE09, GKLE10, GLKE10]. The approach identifies dependencies and conflicts between change operations. It is tailored towards merging process models. The approach [GLKE10] presents a method for detecting semantically equivalent process model fragments via comparing their normal forms. However, it is neither tailored towards detecting two syntactically different but semantically equivalent (trace equivalent) process models nor towards semantic model differencing.

A survey of model versioning is presented in [BKL⁺12]. The survey includes syntactic model differencing and a comparison of various model versioning systems. A survey of model comparison approaches and their applications is presented in [SC13]. Most of the approaches presented in the survey focus on the calculation of syntactic similarities and differences of models.

The idea of refining change operations also appears in [Rum96]. The approach [Rum96] introduces a refinement calculus based on refining change operations for multiple modeling languages, which all use the *system model* [Rum96] as the semantic domain.

2.5.3 Semantic Model Differencing

Research in semantic model differencing focuses on the development of language-specific semantic differencing operators.

Approaches to semantic differencing are enumerative or non-enumerative [LMK14b]. Enumerative approaches compute a single witness or a finite set of witnesses as concrete proofs for semantic differences. Non-enumerative semantic differencing approaches do not compute witnesses. Instead, they aim at computing an aggregated description that summarizes some of the semantic differences (not necessarily all) from one model to another model. Non-enumerative approaches may present the differences by a model of the same modeling language or by using another notation.

Related work produced an enumerative semantic differencing operator for class diagrams [MRR11e]. The approach uses the set of all possible object structures as the semantic domain. The semantic mapping maps each class diagram to a (usually infinite) set of object structures representing the possible data states of systems as modeled in the class diagram. The semantic mapping used for the class diagram language is configurable by using the technique presented in [MRR11f]. The semantic differencing operator takes two class diagrams as input and returns a finite set of object structures that are instances of one of the class diagrams and not of the other class diagram. As presented in [MRR11d, KMRR17], the implementation is based on a reduction to Alloy [Jac06].

An enumerative semantic differencing operator for activity diagrams is presented in [MRR11b]. The set of models is the set of all valid activity diagrams. The semantic domain contains all possible finite execution traces over all possible action labels. The semantic mapping maps each activity diagram to the set of all execution traces of actions that are modeled by the activity diagram. The semantic differencing operator takes two

activity diagrams as input and returns a finite set of traces that are modeled in one of the activity diagrams and not modeled in the other activity diagram. The algorithm is based on a fixed point calculation inspired by symbolic model-checking [BCM⁺92].

Another enumerative semantic differencing operator for activity diagrams is presented in [KR18a]. The approach relies on translating activity diagrams to non-deterministic finite automata [HMU06]. The translation enables a reduction from semantic differencing of ADs to language inclusion checking between finite automata. The implementation of the differencing operator is based on the automaton language inclusion checking tool RABIT [ACC⁺11, www20].

An enumerative semantic differencing operator for a feature model language is introduced in [AHC⁺12]. The set of models is the set of all feature models. The semantic domain is the set of all possible configurations (sets of feature names). The semantic mapping maps each feature model to the set of all configurations described by the feature model. The approach is based on the common closed-world semantics where the configurations in the semantics of a feature model are required to contain features that are explicitly used in the feature model. The implementation is based on a translation to the boolean satisfiability problem. The semantic differencing operator takes two feature models as input and outputs configurations of one of the feature models that are not configurations of the other feature model.

A method that can be used for decreasing the computational complexity for semantic differencing of feature models using the closed-world semantics is introduced in [TBK09]. The method can be exploited to simplify the propositional logic formula used for semantic differencing based on common clauses of the formulas generated for the individual feature models. Based on the common clauses, it is possible to split the formula into multiple smaller formulas. If any of the smaller formulas is satisfiable, then the semantic difference is not empty.

Another enumerative semantic differencing operator for feature models is presented in [DKMR19]. As in the approach above, the set of models is the set of all feature models and the semantic domain is the set of all possible configurations. However, the semantic mapping of the approach presented in [DKMR19] differs from the semantic mapping used in the other approach [AHC⁺12]. In [DKMR19], the semantic mapping maps each feature model to the set of all configurations that are valid in the feature model. The approach assumes that a configuration in the semantics of a feature model may contain features that are not used in the feature model, *i.e.*, features that are not used in the feature model are considered to be unconstrained. Thus, the semantics of each feature model is an infinite set if the universe of possible features is infinite. This semantics is tailored towards feature model evolution analysis in early development stages [DKMR19]. The semantic differencing operator takes two feature models as input and returns a finite set of configurations that are valid in the first feature model and not in the second feature model. The approach also describes a possible implementation based on a translation to the boolean satisfiability problem.

An enumerative semantic differencing operator for UML/P statecharts [Rum16] is presented in [DEKR19]. UML/P statecharts are a statechart variant for describing the behavior of object-oriented systems [Rum16, Rum17]. The set of models is the set of all UML/P statecharts. The semantic domain contains all finite behaviors that describe stimulus/reaction traces [DEKR19]. The semantic mapping maps each statechart to the (usually infinite) set of all stimulus/reaction traces that it describes. The semantic differencing operator takes two UML/P statecharts as input and outputs a stimulus/reaction trace that is possible in one of the UML/P statecharts and not in the other UML/P statechart. The semantic differencing operator reduces semantic differencing for UML/P statecharts to language inclusion checking for finite automata [HMU06]. The reduction enables an implementation based on RABIT [ACC⁺11, www20].

A semantic differencing operator for finite time-synchronous port automata is presented in [BKRW17]. Finite time-synchronous port automata are an automaton variant for describing the behavior of interactive systems. The set of models is the set of all finite time-synchronous port automata. The semantic domain contains all infinite behaviors that represent infinite communication histories of messages sent via communication channels. The semantic mapping maps each automaton to the set of all possible communication histories that it describes. The semantic differencing procedure takes two automata as input and returns a finite set of communication histories of the first automaton that are not communication histories of the second automaton. The operator is based on a reduction to the language inclusion checking problem for Büchi automata [Büc90, Far02, Saf88]. The implementation is based on the language inclusion checking tool RABIT.

Similar automata are finite time-synchronous channel automata [BKRW19]. A semantic differencing procedure for these automata and subclasses that enable efficient semantic differencing are presented in [BKRW19].

A framework for semantic differencing based on behavioral semantics specifications is presented in [LMK14b]. The framework is instantiated with an activity diagram, a class diagram, and a Petri net language. However, from [LMK14b] it is not clear whether the semantic differencing operators always find at least one witness in case one exists. It is further unclear, whether the framework always detects whether one model is a refinement of another model in case refinement holds.

An approach to semantic differencing for combinatorial models of test designs is presented in [TM17]. The models are called combinatorial models. A combinatorial model consists of parameters, finite value sets (defining possible values for the parameters), and propositional constraints over the parameters. The semantic domain consists of tuples of assignments of values to parameters. Each element in the semantic domain is called a test. The semantics of a combinatorial model is the set of all tests assigning values to parameters such that the propositional constraints of the combinatorial model are satisfied. The approach presents semantic differences in the form of strongest exclusions, which are the smallest (in a special sense) parameter/value combinations that are excluded by

the constraints of a model. The set of all strongest exclusions can be interpreted to be an aggregated description of the semantic difference from one combinatorial model to another combinatorial model. The authors argue that the semantic difference could also be presented by complete tests [TM17]. However, they state that this presentation has semantic problems (as it might not be well-defined whether a test that is valid in one version is valid in another version) and that the number of possible tests in the semantic difference may be huge [TM17].

Non-enumerative semantic differencing approaches have been applied to feature models and automata [FLW11] as well as to class diagrams [FALW14]. Non-enumerative semantic differencing approaches present the semantic difference from a model to another model by a model of the respective same language. The approaches rely on composition operators. A differencing result is interpretable as a quotient resulting from dividing the one model by the other model. Composing the one input model with the quotient yields a model that has no semantic differences to the other input model.

Part II

Concrete Instantiations of the Generic Framework

Chapter 3

Finite Time-Synchronous Port Automata

Interactive systems consist of multiple software components that cooperate and exchange information with each other to perform a complex task [BS01, Rin14]. Cyber-physical systems are interactive systems that consist of software distributed on different computing devices interacting with each other as well as with physical processes via sensors and actuators [Lee08, Alu15]. This section focuses on a modeling language for specifying the behavior of interactive systems while focusing on modeling the software part. Interactive systems are developed in different domains, such as automotive, avionics, consumer electronics, or robotics. Due to the complexity of interactive systems, their development is difficult, costly, and error-prone [BS01].

FOCUS [BS01, Bro10, RR11] is a framework that provides formal foundations for modeling and specifying the semantics of interactive systems. In FOCUS, an interactive system is modeled with components that exchange messages via well-defined communication channels. The behavior of an interactive system is characterizable by the history of messages that are received and sent by the system. Stepwise refinement (*e.g.*, [BS01]) is a development methodology for the controlled evolution of software components. Each successor component version must be a refinement of its predecessor version. Thus, the development process starts with highly underspecified components that are iteratively refined until obtaining a correct-by-construction system implementation. Automated semantic differencing greatly facilitates stepwise refinement methodologies by enabling to automatically check whether a successor version is a refinement of its predecessor version [BKRW17, BKRW19].

A finite time-synchronous port automaton (TSPA) is a finite automaton variant semantically grounded in the FOCUS theory for describing the behavior of interactive components participating in logically timed interactive systems [BKRW17, BKRW19]. For instance, Figure 3.1 depicts the graphical notation of the TSPA `switch` inspired by a similar example from [Rin14]. As usual, states are represented as labeled circles. The initial state is marked with an arrow originating from a filled, black dot. The TSPA consists of the states `on` and `off`, where the state `on` is the initial state of the TSPA. It has exactly one input channel `btn` and one output channel `sig`. Both channels are of type $\{\xi, t, f\}$. Communicating the empty pseudo-message ξ via a channel during a time unit represents that no message is present on the channel during the time unit. The message

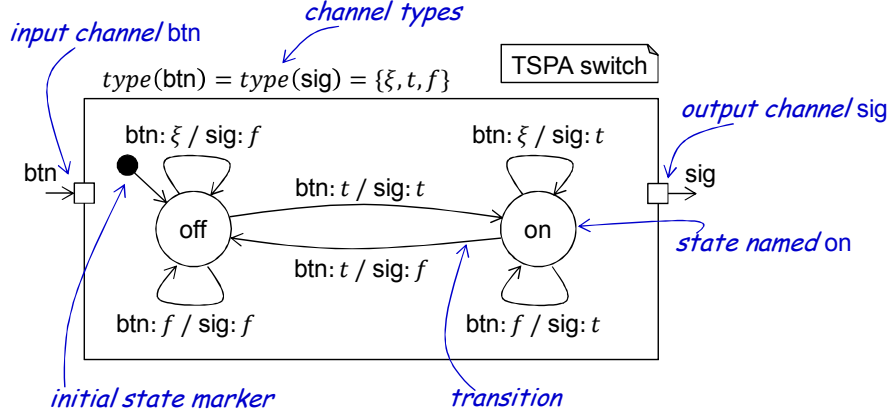


Figure 3.1: The TSPA switch models the behavior of a toggle switch.

t represents the value `true` and the message f represents the value `false`. Transitions are represented by directed arrows between states. Each transition is labeled with its channel valuation. The TSPA switch models the behavior of a toggle switch. The message t received via the channel `btn` represents that the button has been pressed. The message f received via the channel `btn` represents that the button has not been pressed and the message ξ represents that no message is available. The TSPA indicates whether the switch is turned on or off by sending messages via the channel `sig`. When the TSPA is in the state `off` and receives the message ξ or the message f , then it stays in state `off` and outputs the message f via channel `sig`. If the TSPA is in the state `off` and the button is pressed, then the TSPA switches to state `on` and outputs t via its output channel. The TSPA behaves analogously in the state `on`.

In time-synchronous [BS01, Bro10, RR11, BKRW17, BKRW19, GR95] systems, the execution of components is divided into logical time units. A time unit represents an abstract modeling concept. Component implementations may be unaware of time, perform their computations in local time units, or even synchronize their executions with each other. In each time unit, each component (TSPA) receives exactly one message on each of its input channels, executes finitely many computations, and eventually outputs exactly one message via each of its output channels. TSPAs are an adequate formal abstract syntax for finite, discrete, logically timed, interactive systems that can be modeled, for example, with AutoFocus [HF10, AVT⁺15], EmbeddedMontiArc [KRRvW17, KRSvW18], MontiArc [HRR12, Hab16], MontiArcAutomaton [Rin14, Wor16, BKRW17, BKRW19], and SysML's [OMG17] internal block diagrams.

The individual TSPAs in a system interact with each other via messages communicated on typed channels. This thesis focuses on syntactic and semantic differencing as well as the repair of single TSPAs (cf. Chapter 8). We refer to related work [GR95, BKRW17,

BKRW19] for a discussion on the syntactic composition of TSPA and the composition of the semantics of TSPAs. Semantic differencing operators for systems composed of multiple automata are presented in [BKRW17, BKRW19].

This chapter is based on previous work [BKRW17, BKRW19], where we introduced finite TSPAs and a semantic differencing operator for TSPAs. In the following, Section 3.1 introduces the syntax of TSPAs. Then, Section 3.2 defines the semantics of TSPAs. Section 3.3 presents a semantic differencing operator for TSPAs. Afterwards, Section 3.4 introduces change operations for TSPAs. Section 3.5 introduces the TSPA modeling language. Section 3.6 discusses related work.

3.1 Time-synchronous Port Automata Syntax

TSPAs interact with their environments by sending and receiving messages via typed channels. In the remainder of this section, let $\mathcal{M} \subseteq U_N$ be a fixed set of messages that contains a special element $\xi \in \mathcal{M}$. The special element $\xi \in \mathcal{M}$ is called *pseudo message* as it models the mathematical concept of the absence of a message.

A type is a set of messages that contains the empty message. Let *Type* denote a fixed non-empty set of data types where each type $t \in \text{Type}$ satisfies $t \subseteq \mathcal{M}$ and $t \in \xi$. Types facilitate restricting the possible messages that are allowed to be sent via a channel. For example, a type that contains the boolean values and the empty message can be defined by $\text{Bool} \stackrel{\text{def}}{=} \{\text{true}, \text{false}, \xi\}$.

A *channel* is a communication link between components. Each channel has a unique name. In the following, let $\mathcal{C} \subseteq U_N$ denote a set of channel names. The function $\text{type} : \mathcal{C} \rightarrow \text{Type}$ maps each channel $c \in \mathcal{C}$ to its type $\text{type}(c)$.

The input and output channels of a component are described by a channel signature. A *channel signature* is a tuple (I, O) where $I \subseteq \mathcal{C}$ and $O \subseteq \mathcal{C}$ with $I \cap O = \emptyset$ are non-empty, finite, disjoint sets of input and output channels.

A *channel assignment* $a \in B^\rightarrow$ over a set of channels $B \subseteq \mathcal{C}$ is an element of the set $B^\rightarrow \stackrel{\text{def}}{=} \{a : B \rightarrow \mathcal{M} \mid \forall c \in B : a(c) \in \text{type}(c)\}$. A channel assignment over a set of channels maps each channel to a message of its type. Channel assignments describe the messages communicated via a set of channels during a time unit. For example, if $a, b, c \in \mathcal{C}$ are channels that are typed with booleans, then $\text{type}(a) = \text{type}(b) = \text{type}(c) = \text{Bool}$. A channel assignment for the set of channels $\alpha = \{a, b, c\}$ is $\{a : \text{true}, b : \text{false}, c : \xi\}$. The channel assignment α maps the channel a to the message true , the channel b to the message false , and the channel c to the message ξ .

A TSPA specifies the behavior of (an excerpt of) an interactive system [BKRW17, BKRW19]. A TSPA can also model a component's implementation. We assume a white-box viewpoint on the components of an interactive system architecture [BKRW17, BKRW19] where the internals of component implementations and specifications are available. All component specifications and implementations are represented by TSPAs.

Definition 3.1. A (finite) TSPA is a tuple $A = (\Sigma, S, \iota, \delta)$ where

- $\Sigma = (I, O)$ is a channel signature,
- the type $\text{type}(c)$ of each channel $c \in I \cup O$ is finite,
- $S \subseteq U_N$ is a finite set of states,
- $\iota \in S$ is the initial state,
- $\delta \subseteq S \times (I \cup O)^\rightarrow \times S$ is the transition relation, and
- $\forall s \in S : \forall i \in I^\rightarrow : \exists (u, a, v) \in \delta : u = s \wedge a|_I = i$, i.e., the TSPA is reactive.

We denote by M_{PA} the set of all TSPAs. We sometimes write (I, O, S, ι, δ) instead of $((I, O), S, \iota, \delta)$ if it is clear from the context that $((I, O), S, \iota, \delta)$ is a TSPA. The states of a TSPA define its internal data states. A TSPA starts in its initial state. It receives inputs via its input channels and emits outputs via its output channels. The reactions of a TSPA are defined by its transition relation. The transition relation defines the possible outputs and internal state changes of the TSPA, depending on its current state and input. The last condition in Definition 3.1 requires TSPAs to be reactive. Reactivity states that for every state and every input, the TSPA defines at least one transition that is enabled for the input in the state. This condition ensures that TSPAs are adequate models for components [BKRW17, BKRW19]: A component must not block its environment and must be able to react to every possible input in every time unit [BKRW17, BKRW19].

For example, the TSPA `switch`, which is graphically depicted in Figure 3.1, can be formally defined by the tuple $(\Sigma, S, \iota, \delta)$ with

- the channel signature $\Sigma = (I, O)$ where $I = \{\text{btn}\}$ and $O = \{\text{sig}\}$,
- the set of states $S = \{\text{off}, \text{on}\}$,
- the initial state $\iota = \text{off}$, and
- the transition relation $\delta = (\text{off}, \{\text{btn} : \xi, \text{sig} : \xi\}, \text{off}), (\text{off}, \{\text{btn} : f, \text{sig} : \xi\}, \text{off}), (\text{off}, \{\text{btn} : t, \text{sig} : t\}, \text{on}), (\text{on}, \{\text{btn} : \xi, \text{sig} : t\}, \text{on}), (\text{on}, \{\text{btn} : f, \text{sig} : t\}, \text{on}), (\text{on}, \{\text{btn} : t, \text{sig} : \xi\}, \text{off})$ where
- $\text{type}(\text{btn}) = \text{type}(\text{sig}) = \{\xi, t, f\}$.

3.2 Time-synchronous Port Automata Semantics

This section defines a semantics for TSPAs based on sets of communication histories. Each communication history in the semantics of a TSPA corresponds to a complete

observation of the messages sent on the TSPA's channels during an execution. Alternatively, the semantics for TSPAs can be defined by sets of stream processing functions [GR95, BKRW17]. Each stream processing function in the semantics of a TSPA represents a deterministic implementation of the TSPA. Each stream processing function is an infinite mathematical structure that contains a communication history for every possible input. A single stream processing function is hardly presentable to users and is no easily comprehensible diff witness. In contrast, single communication histories clearly reveal a difference and are adequate diff witnesses. Hence, this thesis uses a semantic domain based on communication histories instead of a semantic domain based on stream processing functions. However, a semantic mapping based on stream processing functions is more fine-grained than a semantic mapping based on communication histories [RR11]. Refinement under the stream processing function semantics implies refinement under the communication history semantics but not vice versa. Thus, although a TSPA A may be a refinement of a TSPA B under the communication history semantics, the TSPA A might be no refinement of the other TSPA B under the stream processing function semantics. However, if a TSPA A is not a refinement of a TSPA B under the communication history semantics, then it is also guaranteed that A is not a refinement of B under the stream processing function semantics [BKRW17]. Thus, the communication history semantics can also be used as a sound heuristic for refinement checking under the stream processing function semantics.

In each time unit, a TSPA receives exactly one message via each of its input channels, performs one state change by executing one transition that is enabled by its inputs, and outputs exactly one message as defined by the transition via each of its output channels. Components continuously receive messages via their input channels and emit messages via their output channels. The history of messages processed by a channel of a component is thus describable by a stream that contains messages in the order of their transmission [BS01, Bro10, BKRW17]. A *stream* over a non-empty set M is an infinite sequence $s \in M^\infty$.

A *communication history* over a channel signature describes the streams of messages communicated via the signature's input and output channels. Each message sent via a channel must be well-typed, *i.e.*, an element of the channel's type.

Definition 3.2. A communication history over a channel signature $\Sigma = (I, O)$ is an infinite stream $((I \cup O)^\rightarrow)^\infty$.

For every channel signature Σ , we denote by Σ^Ω the set of all communication histories over Σ . Every communication history over a channel signature is a function that maps time units to channel assignments where every channel assignment maps the channels of the signature to messages of their types. If $\Sigma = (I, O)$ is a channel signature, then the set of communication histories Σ^Ω is isomorphic to the set $F = \{f : (I \cup O) \rightarrow \mathcal{M}^\infty \mid \forall c \in I \cup O : f(c) \in \text{type}(c)^\infty\}$, *i.e.*, the set of communication histories is isomorphic to the set of all functions mapping channels to streams of messages of the channels' types. For

example, let $\{a, b, c\} \in \mathcal{C}$ be three channels that are typed with the boolean values, *i.e.*, $\text{type}(a) = \text{type}(b) = \text{type}(c) = \{\text{true}, \text{false}, \xi\}$. Assume that $\Sigma = (\{a\}, \{b, c\})$ is a channel signature with the input channel a and the output channels b and c . Then, $(\{a : \text{true}, b : \text{false}, c : \xi\}, \{a : \text{false}, b : \text{false}, c : \text{false}\})^\infty$ is a communication history over Σ .

An execution of a TSPA starts in the TSPA's initial state. In each time unit, the TSPA receives one message on each of its input channels, performs one state change by taking a transition that is enabled by the inputs and outputs one message on each of its output channels as defined by the transition.

Definition 3.3. An execution $\sigma = s_0, \theta_0, s_1, \theta_1, \dots$ of a TSPA $A = (\Sigma, S, \iota, \delta)$ is an infinite alternating sequence of states and channel valuations such that $s_0 = \iota$ and $(s_i, \theta_i, s_{i+1}) \in \delta$ for all $i \in \mathbb{N}$.

The communication history $\text{his}(\sigma)$ produced by an execution $\sigma = s_0, \theta_0, s_1, \theta_1, \dots$ is defined as the communication history $\theta_0, \theta_1, \dots$ containing the channel assignments of σ .

We denote by $\text{execs}(A)$ the set of all executions of a TSPA A . The executions of a TSPA comprise its internal behaviors, which are invisible to its environment. Communication histories represent executions from a black-box viewpoint by abstracting from the internal state changes. The semantics of a TSPA are all communication histories produced by its executions.

Definition 3.4. Let A be a TSPA. The semantics $\llbracket A \rrbracket^{PA}$ of the TSPA A is defined as the set $\llbracket A \rrbracket^{PA} \stackrel{\text{def}}{=} \{\text{his}(\sigma) \mid \sigma \in \text{execs}(A)\}$ of all communication histories produced by the executions of A .

An example execution of the TSPA switch depicted in Figure 3.1 is given by $\sigma = (\text{off}, \{\text{btn} : t, \text{sig} : t\}, \text{on}, \{\text{btn} : t, \text{sig} : \xi\})^\infty$. The communication history produced by the execution is given by $(\{\text{btn} : t, \text{sig} : t\}, \{\text{btn} : t, \text{sig} : \xi\})^\infty$. The communication history is produced by the execution that occurs when the switch is continuously pressed in each time unit.

3.3 Semantic Differencing of Time-synchronous Port Automata

The semantic difference from a TSPA to another TSPA is the set of all communication histories produced by the executions of the former TSPA that are no communication histories produced by the executions of the latter TSPA. This section introduces a semantic differencing operator for TSPAs based on the procedure previously published in [BKRW17]. The procedure reduces semantic differencing of TSPA to the language inclusion checking problem for BAs, which is a well-known decidable problem [Büc90, Far02, Saf88]. To this effect, two TSPAs are translated to BAs, before it is checked whether the language recognized by one of the BAs is a subset of the language recognized by the other BA. The following defines the translation from TSPAs to BAs.

Definition 3.5. Let $A = (\Sigma, S, \iota, \delta)$ with $\Sigma = (I, O)$ be a TSPA. The BA $ba(A)$ associated with the TSPA A is defined as $ba(A) \stackrel{\text{def}}{=} (S, (I \cup O)^\rightarrow, \delta, \iota, S)$ with

- the set of states S ,
- the alphabet $(I \cup O)^\rightarrow$,
- the transition relation δ ,
- the initial state ι , and
- the set of accepting states S .

In Definition 3.5, the TSPA A is interpreted as a BA where all states are accepting. The BA $ba(A)$ is well-defined: The set of states S is finite. As I and O are finite, the set $I \cup O$ is finite. As further $\text{type}(c)$ is finite for each channel $c \in I \cup O$, it holds that $(I \cup O) \times \bigcup_{c \in I \cup O} \text{type}(c)$ is finite. Therefore, it especially holds that $(I \cup O)^\rightarrow$ is finite because $(I \cup O)^\rightarrow \subseteq (I \cup O) \times \bigcup_{c \in I \cup O} \text{type}(c)$. Further, as S and $(I \cup O)^\rightarrow$ are finite, the set $\delta \subseteq S \times (I \cup O)^\rightarrow \times S$ is also finite.

For every communication history in the semantics of a TSPA, there is a unique word that represents the communication history in the language recognized by the BA associated with the TSPA. Vice versa, for every word in the language recognized by a BA associated with a TSPA, there is a unique communication history representing the word.

Proposition 3.1. Let $A = (\Sigma, S, \iota, \delta)$ be a TSPA. Then, $\llbracket A \rrbracket^{PA} = \mathcal{L}_\omega(ba(A))$.

Proof. Let $A = (\Sigma, S, \iota, \delta)$ be a TSPA where $\Sigma = (I, O)$ and let $ba(A) = (S, (I \cup O)^\rightarrow, \delta, \iota, S)$ be the BA associated with A .

" \subseteq ": Let $\sigma = s_0, \theta_0, s_1, \theta_1, \dots$ be an execution of A . By definition of execution, it holds that $(s_i, \theta_i, s_{i+1} \in \delta)$ for all $i \in \mathbb{N}$ and $s_0 = \iota$. As further all states of $ba(A)$ are accepting, the BA $ba(A)$ accepts the word $\theta_0, \theta_1, \dots = \text{his}(\sigma)$. Thus, $\text{his}(\sigma) \in \mathcal{L}_\omega(ba(A))$.

" \supseteq ": Let $\sigma = \theta_0, \theta_1, \dots \in \mathcal{L}_\omega(ba(A))$. Then, there exists an infinite sequence of states $s_0, s_1, \dots \in S^\infty$ such that $s_0 = \iota$, $(s_j, w_j, s_{j+1}) \in \delta$ for all $j \in \mathbb{N}$, and $s_j \in S$ for infinitely many $j \in \mathbb{N}$. This directly implies that $e = s_0, \theta_0, s_1, \theta_1, \dots \in \text{execs}(A)$ is an execution of A . Thus, $\sigma = \text{his}(e) \in \llbracket A \rrbracket^{PA}$. \square

Proposition 3.1 enables reducing semantic differencing of TSPAs to language inclusion checking between BAs. A TSPA A is a refinement of a TSPA B iff the language recognized by the BA $ba(A)$ associated with the TSPA A is a subset of the language recognized by the BA $ba(B)$ associated with the TSPA B . As the language inclusion checking problem for BAs is a well-known decidable problem [Büc90, Far02, Saf88], this yields a semantic diff operator for the TSPA modeling language: Let A_1 and A_2 be two TSPAs. To check whether $\delta(A_1, A_2) = \emptyset$, we construct the BAs $ba(A_1)$ and $ba(A_2)$ and check whether $\mathcal{L}_\omega(ba(A_1)) \subseteq \mathcal{L}_\omega(ba(A_2))$. If the BA language inclusion procedure yields

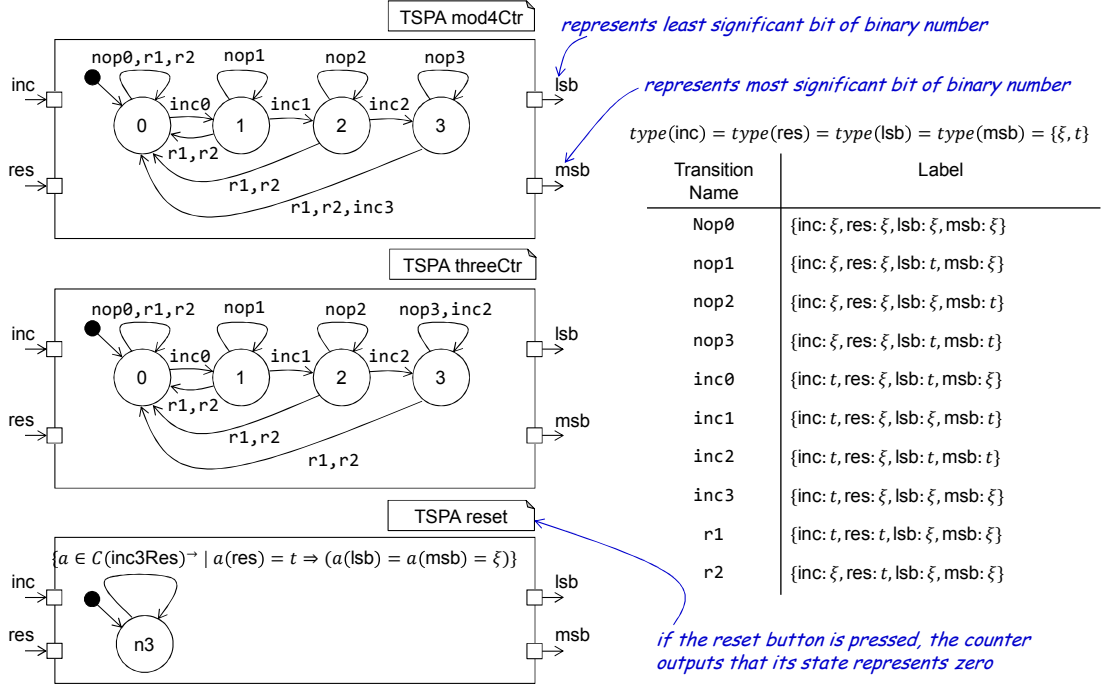


Figure 3.2: The three TSPAs mod4Ctr, threeCtr, and reset. The TSPAs mod4Ctr and threeCtr represent implementations with incomparable semantics. Both implementations are refinements of the TSPA reset modeling a highly underspecified specification.

a word $w \in \mathcal{L}_\omega(ba(A_1)) \setminus \mathcal{L}_\omega(ba(A_2))$ in case language inclusion does not hold, this word can be returned as a diff witness for the semantic difference from A_1 to A_2 .

Semantic differencing of TSPAs facilitates developers in understanding the evolution of the behaviors modeled by two TSPA versions. It can further be used as an automatic method to check whether a TSPA satisfies a specification modeled with another TSPA.

For instance, the left-hand side of Figure 3.2 depicts the three TSPAs mod4Ctr, threeCtr, and reset in their graphical notations inspired by the model from [Fuc95, BKRW19]. Each TSPA has the input channels inc and res as well as the output channels lsb and msb. All channels are of type $\{\xi, t\}$, where the empty pseudo-message ξ represents the absence of a message during a time unit and the message t (true) represents the presence of a message. The transition table on the right-hand side of Figure 3.2 defines the channel valuations of the transitions of the TSPAs mod4Ctr and threeCtr. For instance, the TSPA mod4Ctr has a transition labeled inc0 starting in the state 0 and ending in the state 1. The tuple $(0, \{inc: t, res: \xi, lsb: t, msb: \xi\}, 1)$ formally represents this transition.

The TSPA `mod4Ctr` represents the implementation of an interactive modulo-4-counter. Its states represent the counter's possible values. In each time unit, it outputs the counter's updated value in binary representation via the output channels `lsb` and `msb`. For example, the output channel valuation $\{\text{lsb} : \xi, \text{msb} : t\}$ represents the value *two* and the output channel valuation $\{\text{lsb} : t, \text{msb} : t\}$ represents the value *three*. Independent of the message the TSPA receives via the channel `inc`, it resets its value when it receives the message t via channel `res` and outputs the representation of its updated value *zero*. If the TSPA receives the message t via channel `inc` and the message ξ via channel `res`, then it updates the counter's value to its current value plus one modulo four and outputs the representation of the updated value. Otherwise, if it receives ξ via both of its input channels, it does not change its state and outputs the counted value.

The TSPA `threeCtr` behaves similar to the TSPA `mod4Ctr`. The difference is that it does not change its value when it is in the state 3 and receives the instruction to increment the counter. Semantic differencing reveals exactly this difference. Applying our semantic differencing operator reveals that the two TSPAs have incomparable semantics. On the one hand, semantic differencing reveals that $(\{\text{inc} : t, \text{res} : \xi, \text{lsb} : t, \text{msb} : \xi\}, \{\text{inc} : t, \text{res} : \xi, \text{lsb} : \xi, \text{msb} : t\}, \{\text{inc} : t, \text{res} : \xi, \text{lsb} : t, \text{msb} : t\}, \{\text{inc} : t, \text{res} : \xi, \text{lsb} : \xi, \text{msb} : \xi\})^\infty$ is a possible communication history of the TSPA `mod4Ctr` that is no possible communication history of the TSPA `threeCtr`. Vice versa, semantic differencing reveals that $\{\text{inc} : t, \text{res} : \xi, \text{lsb} : t, \text{msb} : \xi\}, \{\text{inc} : t, \text{res} : \xi, \text{lsb} : \xi, \text{msb} : t\} \& (\{\text{inc} : t, \text{res} : \xi, \text{lsb} : t, \text{msb} : t\})^\infty$ is a possible communication history of the TSPA `threeCtr` that is no possible communication history of the TSPA `mod4Ctr`.

The TSPAs `mod4Ctr` and `threeCtr` define exactly one transition for each state and input channel valuation combination. Thus, they are interpretable as deterministic implementations. In contrast, the TSPA `reset` is highly underspecified, as it defines multiple transitions with the same input channel valuation originating from its state. Its communication histories model the specification that the counter must output the representation of the value *zero* whenever it receives the message t via the input channel `res`. Its communication histories are all possible communication histories where the values on the output channels represent the value *zero* whenever the value on the channel `res` is equal to t . It is possible to check whether an implementation satisfies this specification by checking whether the semantic difference from the implementation to the specification is empty. Using our semantic differencing operator reveals that both, the semantic difference from the TSPA `mod4Ctr` to the TSPA `reset` and the semantic difference from the TSPA `threeCtr` to the TSPA `reset`, are empty. Therefore, the implementations modeled with the TSPAs `mod4Ctr` and `threeCtr` both satisfy the specification modeled with the TSPA `reset`.

TSPA	#States	#Transitions
aut1	2	4
aut2	2	4
impl	5	20
spec	2	98
mod4Ctr	4	13
threeCtr	4	13
reset	1	10

Figure 3.3: The number of states and transitions of the TSPAs used for the semantic differencing experiments.

Semantic Differencing Implementation and Experiments

We implemented the semantic differencing operator for TSPAs to perform experimental evaluations. The implementation is written in Java and uses the automaton language inclusion checking tool RABIT¹ [ACC⁺11] for BA language inclusion checking. The implementation of the semantic differencing operator takes two TSPAs as inputs. It translates the TSPAs into BAs according to the translation described above and outputs the BAs in the BA format, which is the input format of RABIT. Subsequently, the implementation uses the tool RABIT for language inclusion checking of the BAs. In case language inclusion does not hold, RABIT provides a counterexample, which is returned as a diff witness.

We performed experimental evaluations with seven example TSPAs. Appendix A presents the example TSPAs in detail. Figure 3.3 summarizes the sizes of the TSPAs in terms of the numbers of states and transitions of the TSPAs. By construction of the translation from TSPAs to BAs, each BA resulting from translating a TSPA has the same numbers of states and transitions.

We executed the semantic differencing operator for all pairs of example TSPAs that are thematically related. All experiments were executed on a laptop computer with an Intel Core i7-8650U CPU @ 1.90GHz processor, 16GB RAM, and a Samsung PM981 512GB SSD hard drive using Windows 10 and Java 1.8.0_192.

Figure 3.4 summarizes the computation times of the semantic differencing operator and presents the computed diff witnesses for the input pairs. If no witness exists, *i.e.*, refinements holds, then the corresponding cell in the table contains the symbol $-$. If Appendix A defines an abbreviation for a channel assignment, then the channel assignment is represented by its abbreviation in the witnesses to save space. For instance, the semantic differencing operator took 113ms to detect that the TSPA `threeCtr` is a refinement of the TSPA `reset`. The semantic differencing operator took 140ms to

¹<http://languageinclusion.org/>

3.4 TIME-SYNCHRONOUS PORT AUTOMATA CHANGE OPERATIONS

Difference	Time	Diff witness
$\delta(\text{aut1}, \text{aut1})$	125ms	–
$\delta(\text{aut1}, \text{aut2})$	127ms	$\{i : \xi, o : 1\}^\infty$
$\delta(\text{aut2}, \text{aut1})$	150ms	$\{i : \xi, o : \xi\}^\infty$
$\delta(\text{aut2}, \text{aut2})$	149ms	–
$\delta(\text{impl}, \text{impl})$	163ms	–
$\delta(\text{impl}, \text{spec})$	196ms	$(\text{emg1}, \text{emgOff2}) \& \text{fwd}^\infty$
$\delta(\text{spec}, \text{impl})$	176ms	$\{\text{lMot} : \text{FORWARD}, \text{rMot} : \text{FORWARD}, \text{bump} : \text{PRESSED}, \text{emgStp} : \text{PRESSED}\}^\infty$
$\delta(\text{spec}, \text{spec})$	140ms	–
$\delta(\text{mod4Ctr}, \text{mod4Ctr})$	126ms	–
$\delta(\text{mod4Ctr}, \text{threeCtr})$	140ms	$(\text{inc0}, \text{inc1}) \& (\text{inc2}, \text{inc3}, \text{inc0}, \text{inc1})^\infty$
$\delta(\text{mod4Ctr}, \text{reset})$	128ms	–
$\delta(\text{threeCtr}, \text{mod4Ctr})$	135ms	$(\text{inc0}, \text{inc1}) \& \text{inc2}^\infty$
$\delta(\text{threeCtr}, \text{threeCtr})$	119ms	–
$\delta(\text{threeCtr}, \text{reset})$	113ms	–
$\delta(\text{reset}, \text{mod4Ctr})$	118ms	nop1^∞
$\delta(\text{reset}, \text{threeCtr})$	111ms	nop1^∞
$\delta(\text{reset}, \text{reset})$	108ms	–

Figure 3.4: The time needed by the semantic differencing operator for semantic differencing of the pairs of example TSPAs.

compute the diff witness $(\text{inc0}, \text{inc1}) \& (\text{inc2}, \text{inc3}, \text{inc0}, \text{inc1})^\infty$ in the semantic difference from `mod4Ctr` to `threeCtr`.

For the examples, the computation times range from 108ms to 196ms. We conclude that the implementation handles the example TSPAs sufficiently quick. However, the example TSPAs are relatively small in terms of the numbers of states and transitions used in the TSPAs. Therefore, the results are not generalizable to large TSPAs and real world examples, especially because language inclusion checking between BAs is, in general, computationally hard.

3.4 Time-synchronous Port Automata Change Operations

This section defines the change operations for a complete TSPA change operation suite. Some of the change operations are neither refining nor generalizing. Although these change operations are irrelevant for developers to constructively refine or generalize models, the change operations are necessary to obtain a complete change operation suite. The completeness of the change operation suite is required for the model repair framework presented in Chapter 7 and the framework’s instantiation presented in Chapter 8. If a change operation is refining or generalizing, then it is possible to incorporate performance improvements into algorithms that compute solutions (cf. Section 7.5) for special model repair problems as introduced in Section 8.1.

Figure 3.5 overviews the different change operations. Adding a non-existing state (cf. No. 1) is a refactoring operation. State-deletion operations (cf. No. 2) are refining.

No.	Operation	Refining	Generalizing
1.	Adding a non-existing state	✓	✓
2.	Deleting a state, if reactivity is preserved	✓	✗
3.	Adding a transition with a specific label	✗	✓
4.	Deleting a transition with a specific label	✓	✗
5.	Adding an input channel	✗	✗
6.	Adding an output channel	✗	✗
7.	Deleting a channel	✗	✗
8.	Changing the initial state to another existing state	✗	✗

Figure 3.5: TSPA change operations and their properties.

The addition of a transition with a specific label (cf. No. 3) is a generalizing operation. Vice versa, transition-deletion (cf. No. 4) operations are refining. In general, the operations for adding and deleting channels (cf. No. 5, 6, 7) are neither refining nor generalizing because they can completely change the communication histories of a TSPA. Similarly, operations for changing the initial state (cf. No. 8) are neither refining nor generalizing because they may completely change the communication histories of a TSPA.

3.4.1 State-Addition Operations

State-addition operations with signature $M_{PA} \rightarrow M_{PA}$	
Parameters	Let $s \in U_N$ be a state name.
Explanation	The operation $addS_s$ adds the state s to a TSPA that does not contain the state.
Domain	$(I, O, S, \iota, \delta) \in \text{dom}(addS_s) \Leftrightarrow s \notin S$
Application	(I, O, S, ι, δ) \downarrow $(I, O, S', \iota, \delta')$ where $S' = S \cup \{s\}$ $\delta' = \delta \cup \{(s, a, s) \mid a \in (I \cup O)^{-*}\}$
Example	

 Figure 3.6: State-addition operations $addS_s$ for all state names $s \in U_N$.

Figure 3.6 defines the state-addition operations. Each state-addition operation $addS_s$ is parametrized with a name $s \in U_N$ representing a state name. On the application of

the state-addition operation $addS_s$ to a TSPA, the state s and a transition from the state to itself for each possible channel valuation of the TSPA's channels are added to the TSPA. To explicate that used states cannot be added, the operation is applicable to a TSPA iff the state s does not exist in the TSPA. Thus, an already defined state cannot be added. In the example application (Figure 3.6), the state s is added to the TSPA.

Adding a state to a TSPA yields a TSPA where the newly added state is isolated, *i.e.*, it cannot be reached by the initial state. Thus, the resulting TSPA has the same executions as the original TSPA. As this implies that the semantics of the TSAs contain the same behaviors, state-addition operations are refactoring:

Proposition 3.2. *Let $addS_s$ be a state-addition operation and let $A \in M_{PA}$ be a TSPA such that $addS_s$ is applicable to A . Then, $\llbracket addS_s(A) \rrbracket^{PA} = \llbracket A \rrbracket^{PA}$.*

Proof. Let $s \in U_N$, $addS_s$, and $A = (I, O, S, \iota, \delta) \in M_{PA}$ be given as above.

" \subseteq ": Let $\sigma \in \llbracket addS_s(A) \rrbracket^{PA}$ be a communication history. Then, there exists an execution $e = s_0, \theta_0, s_1, \theta_1, \dots \in execs(addS_s(A))$ such that $his(e) = \sigma$. By definition of execution, it holds that $s_0 = \iota$ and $(s_j, \theta_j, s_{j+1}) \in \delta \cup \{(s, a, s) \mid a \in (I \cup O)^\rightarrow\}$ for all $j \in \mathbb{N}$. As the state s is not reachable from the initial state of $addS_s(A)$, no state in the execution e is equal to s , *i.e.*, it holds that $s_j \neq s$ for all $j \in \mathbb{N}$. Therefore, it holds that $(s_j, \theta_j, s_{j+1}) \in \delta$ for all $j \in \mathbb{N}$. As further $s_0 = \iota$, we have that e is an execution of A . Therefore, $\sigma = his(e) \in \llbracket A \rrbracket^{PA}$.

" \supseteq ": Let $\sigma \in \llbracket A \rrbracket^{PA}$ be a communication history. Then, there exists an execution $e = s_0, \theta_0, s_1, \theta_1, \dots \in execs(A)$ such that $his(e) = \sigma$. By definition of execution it holds that $s_0 = \iota$ and $(s_j, \theta_j, s_{j+1}) \in \delta$ for all $j \in \mathbb{N}$. As each state of A is also a state of $addS_s(A)$ and each transition of A is also a transition of $addS_s(A)$, we can conclude that e is also an execution of $addS_s(A)$. Thus, $\sigma = his(e) \in \llbracket addS_s(A) \rrbracket^{PA}$. \square

3.4.2 State-Deletion Operations

Figure 3.7 defines the state-deletion operations. Each state-deletion operation $delS_d$ is parametrized with a name $d \in U_N$ representing a state name. On the application of the state-deletion operation $delS_d$ to a TSPA, the state d as well as all transitions using the state are removed from the TSPA. The operation is applicable to a TSPA iff the state to delete is not the initial state and deleting the state yields a TSPA that is reactive. This condition ensures the well-formedness of the resulting TSPA. In the example application (Figure 3.6), the state d is deleted from the TSPA.

The deletion of a state from a TSPA yields a TSPA with a state set that is a subset of the original TSPA's state set and a transition set that is a subset of the original TSPA's transition set. The initial state remains unchanged. Thus, each execution of the resulting TSPA is also an execution of the original TSPA. Therefore, state-deletion operations are refining:

	State-deletion operations with signature $M_{PA} \rightarrow M_{PA}$
Parameters	Let $d \in U_N$ be a state name.
Explanation	The operation $delS_d$ deletes the state d from a TSPA, if reactivity of the TSPA is preserved.
Domain	$(I, O, S, \iota, \delta) \in dom(delS_d) \Leftrightarrow d \in S \setminus \{i\} \wedge \forall s \in S \setminus \{d\}: \forall i \in I: \exists (u, a, v) \in \{(u, a, v) \in \delta \mid u \neq d \wedge v \neq d\}: u = s \wedge a _I = i$
Application	$delS_d \begin{array}{c} (I, O, S, \iota, \delta) \\ \downarrow \\ (I, O, S', \iota, \delta') \text{ where } S' = S \setminus \{d\} \\ \delta' = \{(u, a, v) \in \delta \mid u \neq d \wedge v \neq d\} \end{array}$
Example	

 Figure 3.7: State-deletion operations $delS_d$ for all state names $d \in U_N$.

Proposition 3.3. *Let $delS_d$ be a state-deletion operation and let $A \in M_{PA}$ be a TSPA such that $delS_d$ is applicable to A . Then, $\llbracket delS_d(A) \rrbracket^{PA} \subseteq \llbracket A \rrbracket^{PA}$.*

Proof. Let $d \in U_N$, $delS_d$, and $A = (I, O, S, \iota, \delta) \in M_{PA}$ be given as above. Let $\sigma \in \llbracket delS_d(A) \rrbracket^{PA}$ be a communication history. Then, there exists an execution $e = s_0, \theta_0, s_1, \theta_1, \dots \in execs(delS_d(A))$ of $delS_d(A)$ such that $his(e) = \sigma$. As $delS_d(A)$ and A share the same initial state and each state of $delS_d(A)$ is also a state of A and each transition of $delS_d(A)$ is also a transition of A , we can conclude that e is also an execution of A . Thus, $\sigma = his(e) \in \llbracket A \rrbracket^{PA}$. \square

3.4.3 Transition-Addition Operations

Figure 3.8 defines the transition-addition operations. Each transition-addition operation $addT_{s,t,a}$ is parametrized with two state names $s, t \in U_N$ and a channel assignment a . On the application of the transition-addition operation $addT_{s,t,a}$ to a TSPA, a transition from the state s to the state t with label a is added to the TSPA. The operation is applicable to a TSPA iff the states s and t exist in the TSPA and a is a valid channel assignment over the channels of the TSPA. This condition ensures the well-formedness of the resulting TSPA. Further, a transition-addition operation is not applicable to a TSPA if it already contains the transition. Thus, an already existing transition cannot be added. In the example application (Figure 3.8), the transition $(on, \{i : \xi, o : \xi\}, on)$ is added to the TSPA.

3.4 TIME-SYNCHRONOUS PORT AUTOMATA CHANGE OPERATIONS

Transition-addition operations with signature $M_{PA} \rightarrow M_{PA}$	
Parameters	Let $s, t \in U_N$ be two state names, and let a be a channel assignment.
Explanation	The operation $addT_{s,t,a}$ adds the transition (s, a, t) to a TSPA.
Domain	$(I, O, S, \iota, \delta) \in dom(addT_{s,t,a}) \Leftrightarrow s, t \in S \wedge a \in (I \cup O) \rightarrow \wedge (s, a, t) \notin \delta$
Application	(I, O, S, ι, δ) \downarrow $addT_{s,t,a}$ $(I, O, S, \iota, \delta')$ where $\delta' = \delta \cup \{(s, a, t)\}$
Example	

Figure 3.8: Transition-addition operations $addT_{s,t,a}$ for all state names $s, t \in U_N$ and channel assignments a .

Adding a transition to a TSPA yields a TSPA with a transition set that is a superset of the original TSPA's transition set. The set of states and the initial state remain unchanged. Thus, each execution of the original TSPA is also an execution of the resulting TSPA. Therefore, transition-addition operations are generalizing:

Proposition 3.4. *Let $addT_{s,t,a}$ be a transition-addition operation and let $A \in M_{PA}$ be a TSPA such that $addT_{s,t,a}$ is applicable to A . Then, $\llbracket A \rrbracket^{PA} \subseteq \llbracket addT_{s,t,a}(A) \rrbracket^{PA}$.*

Proof. Let $s, t \in U_N$, $a \in \mathcal{C} \rightarrow$, $addT_{s,t,a}$, and $A = (I, O, S, \iota, \delta) \in M_{PA}$ be given as above. Let $\sigma \in \llbracket A \rrbracket^{PA}$ be a communication history. Then, there exists an execution $e = s_0, \theta_0, s_1, \theta_1, \dots \in execs(A)$ such that $his(e) = \sigma$. By definition of execution it holds that $s_0 = \iota$ and $(s_j, \theta_j, s_{j+1}) \in \delta$ for all $j \in \mathbb{N}$. As every transition of A is also a transition of $addT_{s,t,a}(A)$, every state of A is also a state of $addT_{s,t,a}(A)$, and A as well as $addT_{s,t,a}(A)$ share the same initial state, we can conclude that $e \in execs(addT_{s,t,a}(A))$ is also an execution of $addT_{s,t,a}(A)$. Thus, $\sigma = his(e) \in \llbracket addT_{s,t,a}(A) \rrbracket^{PA}$. \square

3.4.4 Transition-Deletion Operations

Figure 3.9 defines the transition-deletion operations. Each transition-deletion operation $delT_{s,t,a}$ is parametrized with two names $s, t \in U_N$ representing state names and a channel assignment a . On the application of the transition-deletion operation $delT_{s,t,a}$ to a TSPA, the transition with source state s , labeled with the channel assignment a , and target state t is removed from the TSPA. The operation is only applicable to a

	Transition-deletion operations with signature $M_{PA} \rightarrow M_{PA}$	
Parameters	Let $s, t \in U_N$ be two state names, and let a be a channel assignment.	
Explanation	The operation $delT_{s,t,a}$ deletes the transition (s, a, t) from a TSPA containing the transition, if reactivity is preserved.	
Domain	$(I, O, S, \iota, \delta) \in dom(delT_{s,t,a}) \Leftrightarrow (s, a, t) \in \delta \wedge \forall i \in I^\rightarrow: \exists (u, a', v) \in \delta \setminus \{(s, a, t)\}: u = s \wedge a' _I = i$	
Application	(I, O, S, ι, δ) $\downarrow delT_{s,t,a}$ $(I, O, S, \iota, \delta') \text{ where } \delta' = \delta \setminus \{(s, a, t)\}$	
Example		

Figure 3.9: Transition-deletion operations $delT_{s,t,a}$ for all state names $s, t \in U_N$ and channel assignments a .

TSPA if the resulting TSPA obtained after deleting the transition is reactive. This condition ensures the well-formedness of the resulting TSPA. Further, the operation is only applicable to a TSPA, if the transition to delete is used by the TSPA. In the example application (Figure 3.9), the transition $(on, \{i : \xi, o : \xi\}, on)$ is deleted.

The deletion of a transition from a TSPA yields a TSPA with a set of transitions that is a subset of the original TSPA's set of transitions. The set of states and the initial state remain unchanged. Thus, each execution of the resulting TSPA is also an execution of the original TSPA. As this implies that each behaviors of the resulting TSPA is also a behavior of the original TSPA, transition-deletion operations are refining:

Proposition 3.5. *Let $delT_{s,t,a}$ be a transition-deletion operation and let $A \in M_{PA}$ be a TSPA such that $delT_{s,t,a}$ is applicable to A . Then, $\llbracket delT_{s,t,a}(A) \rrbracket^{PA} \subseteq \llbracket A \rrbracket^{PA}$.*

Proof. Let $s, t \in U_N$, $a \in \mathcal{C}^\rightarrow$, $delT_{s,t,a}$, and $A = (I, O, S, \iota, \delta) \in M_{PA}$ be given as above. Let $\sigma \in \llbracket delT_{s,t,a}(A) \rrbracket^{PA}$ be a communication history. Then, there exists an execution $e = s_0, \theta_0, s_1, \theta_1, \dots \in execs(delT_{s,t,a}(A))$ of $delT_{s,t,a}(A)$ such that $his(e) = \sigma$. As $delT_{s,t,a}(A)$ and A share the same initial state and each state of $delT_{s,t,a}(A)$ is also a state of A , and each transition of $delT_{s,t,a}(A)$ is also a transition of A , we can conclude that e is also an execution of A . Thus, $\sigma = his(e) \in \llbracket A \rrbracket^{PA}$. \square

	Input-channel-addition operations with signature $M_{PA} \mapsto M_{PA}$	
Parameters	Let $c \in \mathcal{C}$ be a channel name.	
Explanation	The operation $addIC_c$ adds the channel c as input channel to a TSPA not containing the channel.	
Domain	$(I, O, S, \iota, \delta) \in dom(addIC_c) \Leftrightarrow c \notin I \cup O$	
Application	$addIC_c$ (I, O, S, ι, δ) \downarrow $(I', O, S, \iota, \delta')$ where $I' = I \cup \{c\}$ $\delta' = \{(s, a, t) \in S \times (I' \cup O)^* \times S \mid (s, a _{(I \cup O)}, t) \in \delta\}$	
Example		

 Figure 3.10: Input-channel-addition operations $addIC_c$ for all channel names $c \in \mathcal{C}$.

3.4.5 Input-Channel-Addition Operations

Figure 3.10 defines the input-channel-addition operations. Each input-channel-addition operation $addIC_c$ is parametrized with a channel name $c \in \mathcal{C}$. On the application of the input-channel-addition operation $addIC_c$ to a TSPA, the channel c is added to the TSPA's input channel set. The operation is applicable to a TSPA iff the TSPA is not using the channel that is added by the operation. This explicates that adding already used channels is not possible. Adding an input channel to a TSPA completely changes the TSPA's set of transitions. The reactions to the inputs received via the channel are completely underspecified: For each transition of the original TSPA and each message in the type of the added input channel, the resulting TSPA contains a transition with the same source and target states as the original transition. The transition's assignment to the channels of the original transition remains unchanged and the assignment to the added input channel is equal to the message in the type of the added input channel. In the example application (Figure 3.10), the channel c is added to the TSPA. The type of the added channel c is $type(c) = \{\xi, 1\}$. As the original TSPA contains the transition $(\text{off}, \{i : \xi, o : \xi\}, \text{off})$, for example, the resulting TSPA contains the transitions $(\text{off}, \{i : \xi, o : \xi, c : \xi\}, \text{off})$ and $(\text{off}, \{i : \xi, o : \xi, c : 1\}, \text{off})$.

The addition of an input channel to a TSPA completely changes the TSPA's set of transitions. Therefore, input-channel-addition operations are neither refining nor

generalizing. For instance, every communication history of the TSPA depicted on the left-hand side in the example of Figure 3.10 is no communication history of the TSPA depicted on the right-hand side in the example of Figure 3.10.

3.4.6 Output-Channel-Addition Operations

Output-channel-addition operations with signature $M_{PA} \rightarrow M_{PA}$	
Parameters	Let $c \in \mathcal{C}$ be a channel name.
Explanation	The operation $addOC_c$ adds the channel c as output channel to a TSPA not containing the channel.
Domain	$(I, O, S, \iota, \delta) \in \text{dom}(addOC_c) \Leftrightarrow c \notin I \cup O$
Application	$ \begin{array}{c} (I, O, S, \iota, \delta) \\ \downarrow \\ (I, O', S, \iota, \delta') \text{ where } O' = O \cup \{c\} \\ \delta' = \{(s, a, t) \in S \times (I \cup O')^* \times S \mid (s, a _{(I \cup O)}, t) \in \delta\} \end{array} $
Example	

Figure 3.11: Output-channel-addition operations $addOC_c$ for all channel names $c \in \mathcal{C}$.

Figure 3.11 defines the output-channel-addition operations. The addition of an output channel has a similar effect as the addition of an input channel. Each output-channel-addition operation $addOC_c$ is parametrized with a channel name $c \in \mathcal{C}$. On the application of the output-channel-addition operation $addOC_c$ to a TSPA, the channel c is added to the TSPA's output channel set. The operation is applicable to a TSPA iff the TSPA is not using the channel that is added by the operation. Adding an output channel to a TSPA completely changes the TSPA's set of transitions. The reactions of the TSPA in terms of the messages sent via the output channel are completely underspecified: For each transition of the original TSPA and each message in the type of the added output channel, the resulting TSPA contains a transition with the same source and target states as the original transition. The transition's assignments to the channels of the original transition remain unchanged and the assignment to the added output channel is equal to the message in the type of the added output channel. In the example application (Figure 3.11), the channel c is added to the TSPA. The type of the added channel c is

$type(c) = \{\xi, 1\}$. As the original TSPA contains the transition $(\text{off}, \{i : \xi, o : \xi\}, \text{off})$, for example, the resulting TSPA contains the transitions $(\text{off}, \{i : \xi, o : \xi, c : \xi\}, \text{off})$ and $(\text{off}, \{i : \xi, o : \xi, c : 1\}, \text{off})$.

The addition of an output channel to a TSPA completely changes the TSPA's set of transitions. Therefore, output-channel-addition operations are neither refining nor generalizing change operations. For instance, every communication history in the semantics of the TSPA depicted on the left-hand side in the example of Figure 3.11 is no communication history in the semantics of the TSPA depicted on the right-hand side in the example of Figure 3.11.

3.4.7 Channel-Deletion Operations

Channel-deletion operations with signature $M_{PA} \rightarrow M_{PA}$	
Parameters	Let $c \in \mathcal{C}$ be a channel name.
Explanation	The operation $delC_c$ deletes the channel c from a TSPA that contains the channel.
Domain	$(I, O, S, \iota, \delta) \in dom(delC_c) \Leftrightarrow c \in I \cup O$
Application	$delC_c$ (I, O, S, ι, δ) \downarrow $(I', O', S, \iota, \delta')$ where $I' = I \setminus \{c\}$ $O' = O \setminus \{c\}$ $\delta' = \{(s, a _{(I \cup O) \setminus \{c\}}, t) \mid (s, a, t) \in \delta\}$
Example	

Figure 3.12: Channel deletion operations $delC_c$ for all channel names $c \in \mathcal{C}$.

Figure 3.12 defines the channel-deletion operations. Each channel-deletion operation $delC_c$ is parametrized with a channel name $c \in \mathcal{C}$. On the application of the channel-deletion-operation $delC_c$ to a TSPA, the channel c is deleted from the TSPA's sets of input and output channels. The operation is applicable to a TSPA iff the TSPA uses the channel that is to be deleted. Deleting a channel from a TSPA completely changes the TSPA's set of transitions. For each transition of the original TSPA, the resulting TSPA contains a transition that has the same source and target states as the original

transition. The transition's channel assignment maps the channels of the resulting TSPA to the same messages as the channel assignment of the original transition. In the example application (Figure 3.12), the channel c is deleted from the TSPA. As the original TSPA contains the transition $(\text{off}, \{i : \xi, c : \xi, o : \xi\}, \text{off})$, for example, the resulting TSPA contains the transition $(\text{off}, \{i : \xi, o : \xi\}, \text{off})$.

Deleting a channel from a TSPA completely changes the TSPA's set of transitions. Therefore, channel-deletion operations are neither refining nor generalizing operations. For instance, the TSPA depicted on the left-hand side in the example of Figure 3.12 does not share any communication history with the TSPA on the right-hand side in the example of Figure 3.12.

3.4.8 Initial-State-Change Operations

Parameters	Initial-state-change operations with signature $M_{PA} \rightarrow M_{PA}$	
	Let $s \in U_N$ be a state name.	
Explanation	The operation chngI_s changes the initial state in a TSPA to s , if the TSPA contains the state.	
Domain	$(I, O, S, \iota, \delta) \in \text{dom}(\text{chngI}_s) \Leftrightarrow s \in S$	
Application	$\begin{array}{c} (I, O, S, \iota, \delta) \\ \downarrow \text{chngI}_s \\ (I, O, S, \iota', \delta) \text{ where } \iota' = s \end{array}$	
Example		

Figure 3.13: Initial-state-change operations chngI_s for all state names $s \in C$.

Figure 3.13 defines the initial-state-change operations. Each initial-state-change operation chngI_s is parametrized with a state name $s \in U_N$. On the application of the initial-state-change operation chngI_s to a TSPA, the initial state of the TSPA is changed to the state s . The operation is applicable to a TSPA iff the TSPA contains the state that is made the initial state. This condition ensures the well-formedness of the resulting TSPA. In the example application (Figure 3.13), the initial state of the TSPA is changed to on .

Changing the initial state in a TSPA can completely change the TSPA's communication histories. Therefore, initial-state-change operations are, in general, neither refining

nor generalizing. For instance, every communication history of the TSPA depicted on the left-hand side in the example of Figure 3.13 is no communication history of the TSPA depicted on the right-hand side in the example of Figure 3.13.

3.5 Time-Synchronous Port Automaton Modeling Language

We define the change operation suite O_{PA} for TSPAs as the set of all TSPA change operations as defined in the previous sections. The change operation suite O_{PA} is complete. A simple algorithm for computing a change sequence to transform a TSPA A to another TSPA A' operates as follows:

1. Start with the empty sequence.
2. For each state of A' that is no state of A , append a change operation for adding the state.
3. If the initial state of A' is different from the initial state of A , then append a change operation for changing the initial state to the initial state of A' .
4. For each channel of A that is no channel of A' , append a change operation for deleting the channel.
5. For each channel of A' that is no channel of A , append a change operation for adding the channel.
6. For each transition of A' that is no transition of the TSPA obtained from applying the change sequence obtained after the first five steps to the TSPA A , append a transition-addition operation adding the transition.
7. For each state of A that is no state of A' , append a state-deletion operation deleting the state.
8. For each transition of the TSPA obtained from applying the change sequence obtained after the first seven steps to the TSPA A that is no transition of the TSPA A' , append a transition-deletion operation deleting the transition.

The algorithm sketched by the eight steps describes (disregarding the underspecification concerning the order in which the operations are appended in each step) a function $\Delta_{PA} : M_{PA} \times M_{PA} \rightarrow O_{PA}^*$ that takes two TSPAs as inputs and outputs a change sequence of TSPA change operations. The state-addition operations appended in the second step are applicable because they add states that do not exist in the input TSPA. The operation appended in the third step is applicable because the initial state must exist after applying the change operations computed in the second step. The channel-deletion

operations appended in the fourth step are applicable because the change sequence obtained after the first three steps does not change the sets of channels of the TSPA A . Similarly, the channel addition operations appended in the fifth step are applicable because the channels are not added by the change operations appended in the first four steps. The transition-addition operations appended in the sixth step are applicable because they add transitions that are not used by the TSPA. After applying the change sequence obtained after the first six steps to the TSPA A , the set of transitions of the resulting TSPA is a superset of the set of transitions of the TSPA A' . Further, the set of states of the resulting TSPA is a superset of the set of states of the TSPA A' . The state-deletion operations appended in the seventh step are applicable because they inherently preserve reactiveness: If the TSPA resulting from applying the change sequence obtained after the first seven steps were not reactive, then the TSPA A' would not be reactive. Similarly, the transition-deletion operations appended in the eighth step are applicable because they preserve reactivity as the TSPA A' is reactive by definition. Applying the computed change sequence to the TSPA A yields the TSPA A' . Thus, for the function Δ_{PA} , it holds that $\forall m, m' \in M_{PA} : m \triangleright \Delta_{PA}(m, m') = m'$. Therefore, the change operation suite O_{PA} is a complete change operation suite for TSPAs.

The TSPA modeling language is defined as $\mathcal{L}_{PA} = (M_{PA}, Sem_{PA}, \llbracket \cdot \rrbracket^{PA})$ where M_{PA} is the set of all TSPAs (cf. Section 3.1), the semantic domain is the set of all communication histories over all channel signatures $Sem_{PA} \stackrel{\text{def}}{=} \{\Sigma^\Omega \mid \Sigma \text{ is a channel signature}\}$, and $\llbracket \cdot \rrbracket^{PA}$ is the semantic mapping for TSPAs that maps each TSPA to all communication histories produced by the executions of the TSPA (cf. Section 3.2). The set O_{PA} is a complete change operation suite for the TSPA modeling language.

3.6 Related Work

This section presents related work on automata models for modeling interactive systems as well as related semantic differencing and model checking procedures.

The notion of finite TSPA is strongly inspired by the notions of port automata [GR95], I/O* automata [RR11, Rum96], and MAA_{ts} automata [Rin14].

Port automata and I/O* automata consume and produce time slices (finite streams of messages instead of a single message) containing finitely many input messages with every transition. In contrast, TSPAs and MAA_{ts} automata consume and produce one message (including the empty pseudo-message ξ) per channel in each time slice. By definition, TSPAs, as introduced in this thesis, have finitely many states. If the set of states and the channel types of an MAA_{ts} automaton are finite, then the automaton is guaranteed to have finitely many transitions. This does not hold for I/O* automata and port automata because both have to define a transition for each state and each possible input stream, *i.e.*, a possible transition for each state and each finite stream for each input channel. Even if the type of a channel is finite, the set of streams of the channel's

type is infinite. The finiteness of the sets of states and transitions enables the automatic semantic differencing procedure presented in Section 3.3. I/O^* automata and MAA_{ts} automata require initial outputs on all channels. Variables are explicitly modeled in the syntax of MAA_{ts} automata, whereas variables have to be represented implicitly in the state spaces of TSPAs [BKRW17]. MAA_{ts} automata distinguish between data and control states (*i.e.*, variables and (control) states). In contrast, TSPAs consist of control states, only. Data states are easily representable as control states. The time-synchronous channel automaton (TSCA) variant for the modeling of interactive systems originated from the notion of TSPA [BKRW19]. The major difference between TSCAs and TSPAs is that TSCAs solely consist of data states and enable the definition of a commutative and associative syntactic composition operator.

The π -ADL [Oqu04] is semantically grounded in the π -calculus [Mil99] and supports statistical model checking for verifying the properties of dynamic software architectures against properties specified in DynBLTL [CQT⁺16]. The approach constructs a statistical model of finite system executions and then checks whether the model satisfies a property within a confidential bound. The approach is tailored to dynamic architectures and only verifies finite traces. In contrast, the communication histories in the semantics of TSPAs encode infinite traces communicated via the TSPAs' channels, and the semantic differencing method guarantees full certainty.

The notion of refinement for timed I/O automata is discussed in [KLSV03]. Similar to the communication history semantics of TSPAs, the semantics of a timed automaton is a set of traces. Similar to our approach, a timed I/O automaton is a refinement of another timed I/O automaton if the semantics (the traces) of the former is a subset of the semantics (the traces) of the latter. However, timed I/O automata are only marked with one message per transition, whereas TSPAs allow modeling the receiving and sending of multiple messages via a single transition. Further, the timing concept of timed I/O automata is more powerful but also more complicated than the corresponding concepts of TSPAs [GR95]. A game-based extension of the timed I/O automaton framework that enables tool supported refinement checking is presented in [DLL⁺10].

Another approach that supports automated refinement checking based on the time-synchronous subset of FOCUS is described in [Rin14]. The semantic differencing procedure of [Rin14] is based on translating the semantics of components into WS1S formulas. The implementation of the semantic differencing procedure uses the model checker Mona [EKM98]. The approach suffers from drawbacks of the model checker Mona, which are grounded in the complexity of solving WS1S problems. In contrast, our approach is based on a translation to Büchi automata and a reduction to the language inclusion checking problem for Büchi automata.

The TSPA change operations are inspired by similar change operations defined in the contexts of refinement calculi for interactive systems [Rum96, PR97, PR99, RR11, RW18]. The previous works either abstract from the internal states of systems [PR97, PR99, RW18] or are tailored towards refinement calculi for infinite state systems [Rum96,

RW18, RR11]. In contrast, the change operations presented in this thesis are used for the definition of a complete change operation suite for components of finite state systems that are represented by TSPAs. Furthermore, this thesis not only focuses on refining change operations, but also examines whether the change operations are refactoring, generalizing, or whether the semantics of resulting TSPAs may be incomparable to the semantics of changed TSPAs.

While [Rum96, PR97, PR99, RR11, RW18] focus on infinite state systems and refinement calculi enabling the constructive application of refinement steps, we focus on finite state systems where semantic differencing can be performed fully automatically. Interesting future work is the application of abstraction techniques [BK08] for abstracting from irrelevant properties of infinite state systems to obtain finite state systems that enable the application of automated semantic differencing. One possible way to achieve this could be the partitioning of states into equivalent classes where each class depends on the property of interest.

Chapter 4

Feature Diagrams

Software product line engineering facilitates the development of similar software applications varying in distinct features [CN01, PBv05]. Feature diagrams [KLD02, SHTB06] are a widely used formalism for describing product lines of different software product configurations. A feature diagram (FD) describes constraints between features regarding their selection in configurations. With this, the primary purpose of FDs is to describe the valid feature combinations (so-called configurations) of a system. The valid configurations are the elements in the semantics of an FD.

The syntax of a FD describes the relationship between features in terms of mandatory and optional features, implies constraints, excludes constraints, and groups of features.

Figure 4.1 depicts the graphical representation of an FD inspired by a similar example from [MR18]. The FD contains all FD modeling elements. The features of an FD are organized in a tree. If a feature is selected and the feature is not the root of the tree, then the parent of the feature in the tree must also be selected. In the example FD, the feature `car` is the root of the tree. If the feature `locking` is selected, then the feature `car` must also be selected. All mandatory child features of a selected feature must also be selected. In the example FD, if the feature `car` is selected, then the feature `engine` must also be selected. If a feature is an optional child of its parent, then the feature does not need to be selected, although its parent is selected. For example, if the feature `car` is selected, then the feature `locking` does not need to be selected. If a feature is selected and there exists an implies constraints from the feature to another feature, then the other feature must also be selected. In the example FD, if the feature `fingerprint` is selected, then the feature `phone` must also be selected. Dually, if a feature is selected and there exists an excludes constraint between the feature and another feature, then the other feature must not be selected. For example, if the feature `keyless` is selected in the example FD, then the feature `phone` must not be selected. FDs may contain or- and xor-groups. A group consists of a parent feature and a subset of the parent feature's child features (called the group's participants). In the example FD, the feature `engine` is the parent of an xor-group that has the participants `electric`, `gas`, and `hybrid`. Similarly, `locking` is the parent of an or-group that has the participants `keyless`, `phone`, and `fingerprint`. If the parent feature of an or-group is selected, then at least one of the group's participants must also be selected. For example, if the

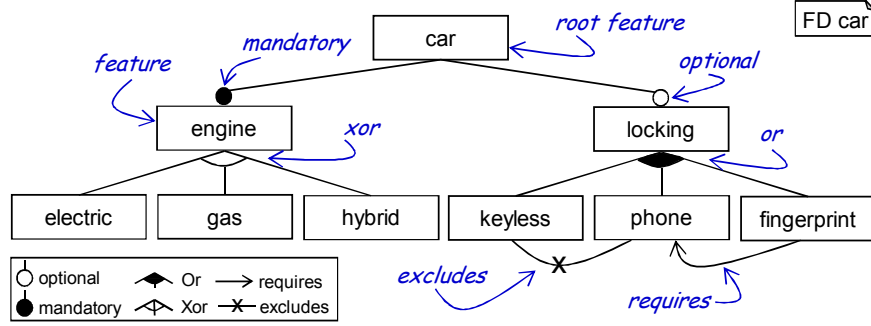


Figure 4.1: An FD containing all FD modeling elements inspired by a FD from [MR15].

feature `locking` is selected (cf. Figure 4.1), then at least one of the features `keyless`, `phone`, and `fingerprint` must also be selected. If the parent feature of an xor-group is selected, then exactly one of the group’s participants must also be selected. In the example FD, if the feature `engine` is selected, then exactly one of the features `electric`, `gas`, and `hybrid` must be selected.

Previous research developed various analyzes for FDs [AHC⁺12, BTRC05, BSRC10] under the consideration of the usual *closed-world* semantics. With the closed-world semantics, every configuration in the semantics of an FD must not contain features that are not used in the FD. This chapter presents a novel *open-world* semantics that is specifically tailored towards semantic evolution analysis in early development stages. With the open-world semantics, configurations in the semantics of an FD may contain features that are not used in the FD. This is interesting for comparing FDs developed in early development stages where it is clear that the original FD does not contain all available features. Section 4.6 further discusses the differences and the usefulness of the semantic mappings in different development stages. The contents of this chapter are based on our previously published work [DKMR19] that introduces the novel FD semantics and the semantic differencing operator.

In the following, Section 4.1 introduces the syntax of FDs. Afterwards, Section 4.2 defines the novel FD semantics. Then, Section 4.3 presents a semantic differencing operator for the novel semantics. Section 4.4 introduces FD change operations. Section 4.5 defines the FD modeling language and a complete change operation suite for the FD modeling language. Section 4.6 presents related work on FD analyses.

4.1 Feature Diagram Syntax

An FD consists of features organized in a tree. The child features of a feature are mandatory, optional, or organized in or- and xor-groups. Binary cross-tree constraints

between features can additionally be used to indicate whether the selection of a feature implies or excludes the selection of another feature. Inspired by [AHC⁺12], the syntax of FDs is formally defined as follows:

Definition 4.1. A feature diagram is a tuple $(F, E, r, M, Or, Xor, I, X)$ where

- $F \subseteq U_N$ is a finite set of features,
- (F, r, E) is a directed rooted tree with root $r \in F$,
- r is the root feature,
- $M \subseteq E$ is a set of edges that define the mandatory child features of parent features,
- $Or \subseteq F \times \wp(F)$ defines or-feature groups where for each tuple $(p, G) \in Or$, all features in the group G share the same parent p , i.e., $\forall f \in G : (p, f) \in E$,
- $Xor \subseteq F \times \wp(F)$ defines xor-feature groups where for each tuple $(p, G) \in Xor$, all features in the group G share the same parent p , i.e., $\forall f \in G : (p, f) \in E$,
- $I \subseteq F \times F$ is a set of implies constraints,
- $X \subseteq F \times F$ is a set of excludes constraints.

The following well-formedness rules apply:

- mandatory child features are not part of any feature group, i.e., $\forall (f, g) \in M : \forall (p, G) \in Or \cup Xor : f = p \Rightarrow g \notin G$.
- each feature is part of at most one or-group, i.e., $\forall f \in F : \forall (p, G), (p', G') \in Or : (p, G) \neq (p', G') \Rightarrow f \notin G \cap G'$,
- each feature is part of at most one xor-group, i.e., and $\forall f \in F : \forall (p, G), (p', G') \in Xor : (p, G) \neq (p', G') \Rightarrow f \notin G \cap G'$,
- each feature is not simultaneously part of an or-group and an xor-group, i.e., $\forall f \in F : \forall (p, G) \in Or : \forall (p', G') \in Xor : f \notin G \cap G'$,
- feature groups are not empty, i.e., $\forall (p, G) \in Or \cup Xor : G \neq \emptyset$.

Features that are neither the target of a mandatory edge nor participate in any feature group are optional child features of their parents.

For each FD $fd = (F, E, r, M, Or, Xor, I, X)$ and each feature $f \in F \setminus \{r\}$, we denote by $parent_{fd}(f) \in F$ the parent of f in the tree (F, r, E) , i.e., for all features $f \in F \setminus \{r\}$, it holds that $p = parent_{fd}(f)$ iff $(p, f) \in E$. If fd is clear from the context, we simply write $parent(f)$ instead of $parent_{fd}(f)$. In the remainder of this thesis, M_{FD} denotes the set of all FDs.

For instance, the FD `car` depicted in Figure 4.1 can be formally defined by `car = (F, E, r, M, Or, Xor, I, X)` with

- the set of features $F = \{\text{car}, \text{engine}, \text{locking}, \text{electric}, \text{gas}, \text{fingerprint}, \text{hybrid}, \text{keyless}, \text{phone}\}$,
- the set of edges $E = \{(\text{car}, \text{engine}), (\text{car}, \text{locking}), (\text{engine}, \text{electric}), (\text{engine}, \text{gas}), (\text{engine}, \text{hybrid}), (\text{locking}, \text{keyless}), (\text{locking}, \text{phone}), (\text{locking}, \text{fingerprint})\}$,
- the root feature $r = \text{car}$,
- the set of mandatory edges $M = \{(\text{car}, \text{engine})\}$,
- the set of or-groups $Or = \{(\text{locking}, \{\text{keyless}, \text{phone}, \text{fingerprint}\})\}$,
- the set of xor-groups $Xor = \{(\text{engine}, \{\text{electric}, \text{gas}, \text{hybrid}\})\}$,
- the set of implies constraints $I = \{(\text{fingerprint}, \text{phone})\}$, and
- the set of excludes constraints $X = \{(\text{phone}, \text{keyless})\}$.

The following section formalizes the semantics of FDs.

4.2 Feature Diagram Semantics

The semantics of FDs are defined as sets of feature configurations. A feature *configuration* C is a finite set of feature names $C \subseteq U_N$ that represents a set of selected features. Each FD describes a set of valid configurations. The modeling elements of an FD are interpreted as constraints on the set of all possible configurations.

Definition 4.2. Let $fd = (F, E, r, M, Or, Xor, I, X)$ be an FD. A configuration $C \subseteq U_F$ is valid in fd iff the following conditions are satisfied:

1. $r \in C$,
2. $\forall f \in C \cap F : f \neq r \Rightarrow \text{parent}(f) \in C$,
3. $\forall (f, g) \in M : f \in C \Rightarrow g \in C$,
4. $\forall (p, G) \in Or : p \in C \Rightarrow |C \cap G| \geq 1$,
5. $\forall (p, G) \in Xor : p \in C \Rightarrow |C \cap G| = 1$,
6. $\forall (f, g) \in I : f \in C \Rightarrow g \in C$,
7. $\forall (f, g) \in X : f \in C \Rightarrow g \notin C$.

The semantics $\llbracket fd \rrbracket^{FD}$ of fd is defined as the set of all configurations that are valid in fd .

The first constraint states that the root must be selected. The second constraint states that if a feature simultaneously exists in the configuration and in the FD, then the parent of the feature in the FD must also be included in the configuration. Thus, if a feature of the FD is selected, then the parent feature of the feature must also be selected. The third constraint states that all mandatory child features of a feature must be selected if the feature is selected. The fourth constraint states that at least one feature of an or-group must be selected in case the parent of the group is selected. The fifth constraint states that exactly one feature of an xor-group must be selected in case the parent of the group is selected. The sixth and seventh constraints require that implies and excludes constraints are respected.

For example, the configuration $\{\text{car}, \text{engine}, \text{gas}\}$ is valid in the FD car depicted in Figure 4.1. The configuration $\{\text{car}, \text{engine}, \text{gas}, \text{radio}\}$ is also valid in the FD car , although it contains the feature radio , which is not used in the FD car . As the feature radio is not used in the FD car , its selection is not constrained by the FD car . The configuration $\{\text{car}\}$ is not valid in the FD car because it contains the feature car , but does not contain the feature engine .

The usual semantics for FDs [AHC⁺12, Bat05, BTRC05, BSRC10, CW07, vdB12, SHTB07, ZZM04] are tailored towards a *closed-world*: Each configuration in the semantics of an FD must satisfy all the conditions stated in Definition 4.2 and, additionally, must not contain features that are not used in the FD. Therefore, every configuration in the closed-world semantics of an FD is also valid in the FD. In contrast, the semantics used in this thesis is more abstract than the usual closed-world semantics: Configurations in the semantics of an FD may contain features that are not used in the FD. The FD elements induce constraints over the features used in the FD but do not induce constraints on the features that are not used in the FD. Thus, if the set of possible features is infinite and an FD is consistent, the semantics of the FD also contains infinitely many configurations. In contrast, when using the usual closed-world semantics, the semantics of an FD is always a finite set.

From the above argumentation, it is intuitively clear that removing features that are not defined in an FD from a valid configuration of the FD again yields a valid configuration of the FD. This property holds because an FD does not constrain features that are not used in the FD. Similarly, adding features that are not used in an FD to a valid configuration of the FD again yields a valid configuration of the FD.

Proposition 4.1. *Let $fd = (F, E, r, M, Or, Xor, I, X)$ be an FD and let $C \subseteq U_N$ be a configuration. Then, $C \in \llbracket fd \rrbracket^{FD}$ iff $C \cap F \in \llbracket fd \rrbracket^{FD}$.*

Proof. Let fd and C be given as above.

" \Rightarrow ": Assume $C \in \llbracket fd \rrbracket^{FD}$. Then, all the seven conditions in Definition 4.2 are satisfied for fd and C .

As the first condition is satisfied for fd and C , it holds that $r \in C$. As $r \in F$ and $r \in C$, it holds that $r \in C \cap F$. Thus, the first condition is also satisfied for fd and

$C \cap F$. As the second condition is satisfied for fd and C and since $(C \cap F) \cap F = C \cap F$, the second condition is also satisfied for fd and $C \cap F$. As the third condition is satisfied for fd and C and as $M \subseteq F \times F$, the third condition is also satisfied for fd and $C \cap F$. As the fourth condition is satisfied for fd and C and as $\forall(p, G) \in Or : p \in F \wedge G \subseteq F$, the fourth condition is also satisfied for fd and $C \cap F$. As the fifth condition is satisfied for fd and C and as $\forall(p, G) \in Xor : p \in F \wedge G \subseteq F$, the fifth condition is also satisfied for fd and $C \cap F$. As the sixth condition is satisfied for fd and C and as $I \subseteq F \times F$, the sixth condition is also satisfied for fd and $C \cap F$. As the seventh condition is satisfied for fd and C and as $X \subseteq F \times F$, the seventh condition is also satisfied for fd and $C \cap F$.

Thus, all the seven conditions in Definition 4.2 are satisfied for fd and $C \cap F$. We can conclude that $C \cap F$ is valid in fd .

" \Leftarrow ": Assume $C \cap F \in \llbracket fd \rrbracket^{FD}$. Then, all the seven conditions in Definition 4.2 are satisfied for fd and $C \cap F$.

As the first condition is satisfied for fd and $C \cap F$, it holds that $r \in C$. Thus, it especially holds that $r \in C$. Thus, the first condition is also satisfied for fd and C . As the second condition is satisfied for fd and $C \cap F$ and since $(C \cap F) \cap F = C \cap F$, the second condition is also satisfied for fd and C . As the third condition is satisfied for fd and $C \cap F$, as $M \subseteq F \times F$, and as $C \cap F \subseteq C$, the third condition is also satisfied for fd and C . As the fourth condition is satisfied for fd and $C \cap F$, as $\forall(p, G) \in Or : p \in F \wedge G \subseteq F$, and as $C \cap F \subseteq C$, the fourth condition is also satisfied for fd and C . As the fifth condition is satisfied for fd and C , as $\forall(p, G) \in Xor : p \in F \wedge G \subseteq F$, and as $C \cap F \subseteq C$, the fifth condition is also satisfied for fd and C . As the sixth condition is satisfied for fd and C , as $I \subseteq F \times F$, and as $C \cap F \subseteq C$, the sixth condition is also satisfied for fd and C . As the seventh condition is satisfied for fd and C , as $X \subseteq F \times F$, and as $C \cap F \subseteq C$, the seventh condition is also satisfied for fd and C .

Thus, all the seven conditions in Definition 4.2 are satisfied for fd and C . We can conclude that C is valid in fd . \square

Using Proposition 4.1 enables to easily proof that an FD fd is consistent iff there exists a configuration that is valid in fd and solely contains features that are used in fd .

Proposition 4.2. *An FD $fd = (F, E, r, M, Or, Xor, I, X)$ is consistent iff there exists a configuration $C \subseteq F$ such that $C \in \llbracket fd \rrbracket^{FD}$.*

Proof. Let $fd = (F, E, r, M, Or, Xor, I, X)$ be an FD.

" \Rightarrow ": Assume fd is consistent, which is equivalent to $\llbracket fd \rrbracket^{FD} \neq \emptyset$. Thus, there exists $C \in \llbracket fd \rrbracket^{FD}$. Using Proposition 4.1, we obtain that $C \cap F \in \llbracket fd \rrbracket^{FD}$.

" \Leftarrow ": Assume there exists a configuration $C \subseteq F$ such that $C \in \llbracket fd \rrbracket^{FD}$. Then, $\llbracket fd \rrbracket^{FD} \neq \emptyset$. Thus, fd is consistent. \square

We further use Proposition 4.1 to proof the main property enabling automatic semantic differencing of FDs using the semantics as defined in this thesis.

4.3 Semantic Differencing of Feature Diagrams

The semantic difference $\delta(fd_1, fd_2)$ from the FD fd_1 to the FD fd_2 contains all configurations that are valid in the FD fd_1 and not valid in the FD fd_2 , *i.e.*, $\delta(fd_1, fd_2) = \llbracket fd_1 \rrbracket^{FD} \setminus \llbracket fd_2 \rrbracket^{FD}$. In case the FD fd_2 is a successor version of the FD fd_1 , the semantic difference $\delta(fd_1, fd_2)$ effectively reveals the configurations that have been removed during the evolution from fd_1 to fd_2 . Vice versa, the semantic difference $\delta(fd_2, fd_1)$ reveals the configurations that have been added during the evolution of fd_1 to fd_2 .

Previous work produced a semantic differencing operator for FDs using the usual closed-world semantics [AHC⁺12]. The semantics of an FD is always a finite set when using the usual closed-world semantics. Thus, it is possible to check whether there exists a configuration in the closed-world semantics of one FD that is not an element in the closed-world semantics of another FD by iteratively checking whether each element in the semantics of the former FD is contained in the semantics of the latter FD. The method presented in [AHC⁺12] relies on translating two FDs to a propositional formula, before checking whether the formula is satisfiable.

The following introduces a semantic differencing operator for FDs using the semantics introduced in Definition 4.2. The operator provides a method for automatically checking whether the semantic difference for any two FDs is empty and yields a witness in case the former FD is not a refinement of the other FD.

The semantic differencing method relies on the fact that it suffices to search a finite set of configurations for a configuration that is valid in one FD and not valid in another FD. It suffices to search the set of all possible configurations containing features that are used in at least one of the input FDs for a configuration that is valid in one FD and not valid in the other FD:

Proposition 4.3. *Let fd_1 and fd_2 be two feature diagrams. Then, $\llbracket fd_1 \rrbracket^{FD} \subseteq \llbracket fd_2 \rrbracket^{FD}$ iff $(\llbracket fd_1 \rrbracket^{FD} \cap \wp(F_1 \cup F_2)) \subseteq (\llbracket fd_2 \rrbracket^{FD} \cap \wp(F_1 \cup F_2))$.*

Proof. Let $fd_i = (F_i, E_i, r_i, M_i, Or_i, Xor_i, I_i, X_i)$ for $i \in \{1, 2\}$ be two FDs.

" \Rightarrow ": Assume $\llbracket fd_1 \rrbracket^{FD} \subseteq \llbracket fd_2 \rrbracket^{FD}$ holds. This directly implies $(\llbracket fd_1 \rrbracket^{FD} \cap \wp(F_1 \cup F_2)) \subseteq (\llbracket fd_2 \rrbracket^{FD} \cap \wp(F_1 \cup F_2))$.

" \Leftarrow ": Assume $(\llbracket fd_1 \rrbracket^{FD} \cap \wp(F_1 \cup F_2)) \subseteq (\llbracket fd_2 \rrbracket^{FD} \cap \wp(F_1 \cup F_2))$ holds. Let $C \in \llbracket fd_1 \rrbracket^{FD}$ be an arbitrary configuration that is valid in fd_1 . We define $C' \stackrel{\text{def}}{=} C \cap (F_1 \cup F_2)$. As $C \in \llbracket fd_1 \rrbracket^{FD}$, using Proposition 4.1, it is guaranteed that $C \cap F_1 \in \llbracket fd_1 \rrbracket^{FD}$. As $C \cap F_1 \in \llbracket fd_1 \rrbracket^{FD}$ and $C' \cap F_1 = (C \cap (F_1 \cup F_2)) \cap F_1 = C \cap F_1$, it holds that $C' \cap F_1 \in \llbracket fd_1 \rrbracket^{FD}$. Using Proposition 4.1, this implies $C' \in \llbracket fd_1 \rrbracket^{FD}$. Therefore, as $C' \subseteq F_1 \cup F_2$, we have that $C' \in \llbracket fd_1 \rrbracket^{FD} \cap \wp(F_1 \cup F_2)$. As by assumption $(\llbracket fd_1 \rrbracket^{FD} \cap \wp(F_1 \cup F_2)) \subseteq (\llbracket fd_2 \rrbracket^{FD} \cap \wp(F_1 \cup F_2))$, we obtain that $C' \in \llbracket fd_2 \rrbracket^{FD} \cap \wp(F_1 \cup F_2)$. Therefore, it especially holds that $C' \in \llbracket fd_2 \rrbracket^{FD}$. Using Proposition 4.1, this implies $C' \cap F_2 \in \llbracket fd_2 \rrbracket^{FD}$. As further $C' \cap F_2 = (C \cap (F_1 \cup F_2)) \cap F_2 = C \cap F_2$, we obtain that $C \cap F_2 \in \llbracket fd_2 \rrbracket^{FD}$. Using Proposition 4.1, this implies $C \in \llbracket fd_2 \rrbracket^{FD}$. \square

There exist well-known translations from FDs to propositional formulas such that the interpretations satisfying the formula obtained from translating an FD represent exactly the configurations of the FD using the closed-world semantics [BSRC10, BTRC05, CW07, ZZM04, AHC⁺12]. The existing method for semantic FD differencing under the closed-world semantics [AHC⁺12] uses such a translation for semantic FD differencing. Similarly, for two FDs fd_1 and fd_2 , it is possible to construct two formulas Φ_1 and Φ_2 with the following properties: For each feature in fd_1 , the formula Φ_1 contains a variable representing that feature. For each feature in fd_2 , the formula Φ_2 contains a variable representing that feature. Each satisfying interpretation of Φ_1 encodes a valid configuration of fd_1 . Each satisfying interpretation of Φ_2 encodes a valid configuration of fd_2 . If the features that are used in both FDs are encoded by the same variables in both formulas, then each satisfying interpretation of the formula $\Phi_1 \wedge \neg\Phi_2$ encodes a valid configuration of the FD fd_1 that is no valid configuration of the FD fd_2 . Thus, the formula $\Phi_1 \wedge \neg\Phi_2$ is satisfiable iff there exists a valid configuration of fd_1 that is no valid configuration of fd_2 and solely contains features used in fd_1 or fd_2 . Proposition 4.3 guarantees that the semantic difference from fd_1 to fd_2 is not empty iff the formula $\Phi_1 \wedge \neg\Phi_2$ is satisfiable.

Similarly, semantic differencing of the FDs using the closed-world semantics requires to check whether the following formula is satisfiable [AHC⁺12]: $(\Phi_1 \wedge (\bigwedge_{f \in F_2 \setminus F_1} \neg x_f)) \wedge \neg(\Phi_2 \wedge (\bigwedge_{f \in F_1 \setminus F_2} \neg x_f))$ where F_1 is the set of features of fd_1 , F_2 is the set of features of fd_2 , and x_f is the variable encoding the feature f .

Semantic differencing of FDs facilitates understanding the evolution of the configurations modeled by two FD versions and enables automatic FD refinement checking. For instance, Figure 4.2 depicts three versions of a FD modeling the configurations of a tablet computer. The FD `tablet1` is the initial version. The FD `tablet2` is the successor version of the FD `tablet1` and the FD `tablet3` is the successor version of the FD `tablet2`. A FD developer might be interested in the semantic differences from the successor versions to their respective predecessor versions and vice versa. Using semantic differencing automatically reveals that the FD `tablet2` is not a refinement of the FD `tablet1`. The semantic differencing operator outputs that $\{\text{tablet}, \text{display}, \text{memory}, \text{processor}, \text{dis12}, 64\text{GB}, \text{P100}\}$ is a configuration that is valid in `tablet2` and not valid in `tablet1`. This witness reveals exists since configurations that are valid in `tablet1` must either contain the feature `dis10` or the feature `dis11`. The semantic difference from `tablet3` to `tablet2` is not empty, either. Inter alia, the semantic differencing operator outputs that the configuration $\{\text{tablet}, \text{display}, \text{memory}, \text{processor}, \text{dis11}, 256\text{GB}, \text{P200}\}$ is valid in `tablet3` and not valid in `tablet2`. Vice versa, a developer might be interested in the semantic difference from `tablet2` to `tablet3`. The semantic differencing operator computes that the configuration $\{\text{tablet}, \text{display}, \text{memory}, \text{processor}, \text{dis12}, 64\text{GB}, \text{P100}\}$ is valid in `tablet2` and not valid in `tablet3`.

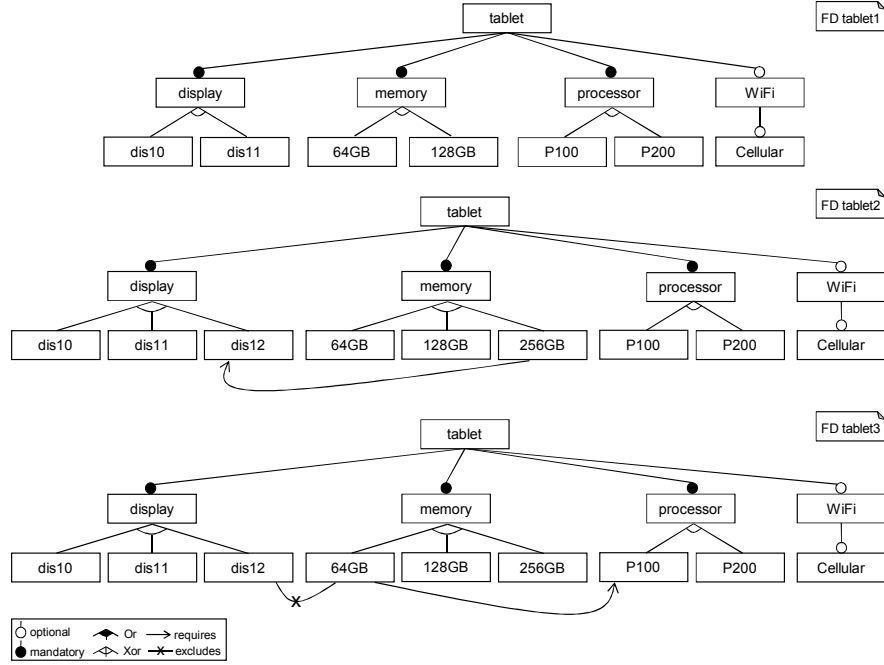


Figure 4.2: Three FDs modeling the valid configurations of a tablet computer.

Semantic Differencing Implementation and Experiments

We implemented the semantic differencing operator for FDs in Java to perform experimental evaluations. The implementation takes two FDs as input. It transforms the FDs to an Alloy module, an input model of the model checker Alloy¹ [Jac06]. An Alloy module resulting from translating the FDs contains a predicate modeling the propositional formula that can be checked for satisfiability to determine whether the semantic difference from one of the FDs to the other FD is empty. Subsequently, the implementation uses the Alloy analyzer to check whether there exists a legal instance of the predicate encoding the formula. A legal instance exists iff the corresponding propositional formula is satisfiable. In this case, the Alloy analyzer provides a legal instance, which encodes a configuration that is valid in the one FD and not valid in the other FD. If such a configuration is found, it is returned as a diff witness.

We choose Alloy for the implementation because it supports exchanging the used SAT solver for computing legal instances. Alternatively, implementations using other SAT solvers supporting the required expressiveness are possible. Although Alloy analyses are based on bounded scopes, the analyses for semantic differencing under the open-world

¹<https://alloytools.org/>

semantics are complete in case the bound is chosen equal to the sum of the numbers of features in the input FDs (cf. Proposition 4.3).

An Overview of the Required Alloy Elements

The sections briefly overviews the Alloy language [Jac06] elements required for the translation of FDs for semantic differencing.

The models of the Alloy language are called modules. The modules resulting from translating FDs consist of signatures, predicates, and a `run` command.

Each signature represents a set of objects. Signatures can be attached with multiplicities. For signatures, the `lone` multiplicity indicates that there is at most one object in the set of objects represented by the signature. Signatures can extend other signatures. If a signature s extends a signature t , then every object contained in the set represented by s is also an element of the set represented by t . Signatures can be marked as abstract. If a signature is marked as abstract, there cannot exist objects that are solely contained in the set represented by the abstract signature. However, if a signature s extends an abstract signature t , then there can exist an object in the set represented by s , which is then also an element of the set represented by t .

Predicates consist of constraints on the sets of objects represented by signatures. Inter alia, constraints enable to constrain the number of objects contained in the sets of objects represented by signatures. Constraints also enable to relate the sizes of the sets of objects represented by signatures.

Running a `run` command proceeded by a predicate makes the Alloy analyzer try to find an instance (sets of objects represented by the signatures) that satisfies the constraints of the predicate. To this effect, the Alloy analyzer translates Alloy modules into boolean constraints and then uses a satisfiability solver to check the constraints for satisfiability [Jac06].

Translation from Feature Diagrams to Alloy for Semantic Differencing

The translation from FDs to Alloy for semantic differencing takes two FDs as input. The resulting Alloy module always contains the abstract signature `feature`. For each feature contained in at least one of the input FDs, the module contains a signature declaration declaring a signature that has the multiplicity `lone`, is named as the feature, and extends the abstract signature `feature`. If the set of objects represented by the signature for a feature is not empty, then the feature is chosen as part of a configuration. Making every signature representing a feature extend the abstract signature `feature` enables a uniform handling of features in configurations in the Java implementation.

For each of the input FDs, the translation produces a predicate. For each input FD, the predicate encodes the constraints on the relationship between the features in the valid configurations of the FD as defined by the semantic mapping (cf. Definition 4.2).

The predicate resulting from translating a FD fd consists of the following constraints:

- If r is the root feature of fd , then the predicate contains the constraint $\#r = 1$. This constraint models that the root of fd must be selected.
- If p is the parent feature of a feature c in fd , then the predicate contains the constraint $\#p \geq \#c$. This constraint models that the parent must be selected if its child is selected.
- If c is a mandatory child of its parent feature p in fd , then the predicate contains the constraint $\#c \geq \#p$. This constraint models that the child must be selected if its parent is selected.
- If (p, G) is an or-group of fd with parent feature p and group participants $G = \{c_1, \dots, c_n\}$, then the predicate contains the constraint $\#p = 1 \text{ implies } \#(c_1 + \dots + c_n) \geq 1$. This constraint models that at least one of the group participants must be selected if the parent feature is selected.
- If (p, G) is an xor-group of fd with parent feature p and group participants $G = \{c_1, \dots, c_n\}$, then the predicate contains the constraint $\#p = 1 \text{ implies } \#(c_1 + \dots + c_n) = 1$. This constraint models that exactly one of the group participants must be selected if the parent feature is selected.
- If (f, g) is an implies constraint of fd , then the predicate contains the constraint $\#g \geq \#f$. This constraint models that the feature g must be selected in case the feature f is selected.
- If (f, g) is an excludes constraint of fd , then the predicate contains the constraints $\#f = 1 \text{ implies } \#g = 0$ and $\#g = 1 \text{ implies } \#f = 0$. These constraints model that at most one of the feature f and g may be selected.

For instance, the lower part of Figure 4.3 depicts the Alloy module that results from translating the FDs $fd1$ and $fd2$ that are depicted in the upper part of Figure 4.3. The module contains examples for all modeling elements of the Alloy language that are relevant for the semantic differencing implementation.

The module contains the abstract signature `Feature` (l. 1). For each of the features A, B, C, D, E that exist in at least one of the input FDs, the module declares a signature with multiplicity `1one` that has the same name as the feature. Each of these signatures extends the abstract signature `Feature` (l. 2). The predicate $fd1$ results from translating the FD $fd1$. It contains the constraint $\#A = 1$ because A is the root feature in $fd1$. The signature contains the constraints $\#A \geq \#B$ (l. 6), $\#A \geq \#C$ (l. 7), and $\#A \geq \#D$ (l. 8) because A is the parent feature of the features B, C , and D . The constraint $\#A = 1 \text{ implies } \#(B + C) \geq 1$ (l. 9) results from translating the or-group of $fd1$. The

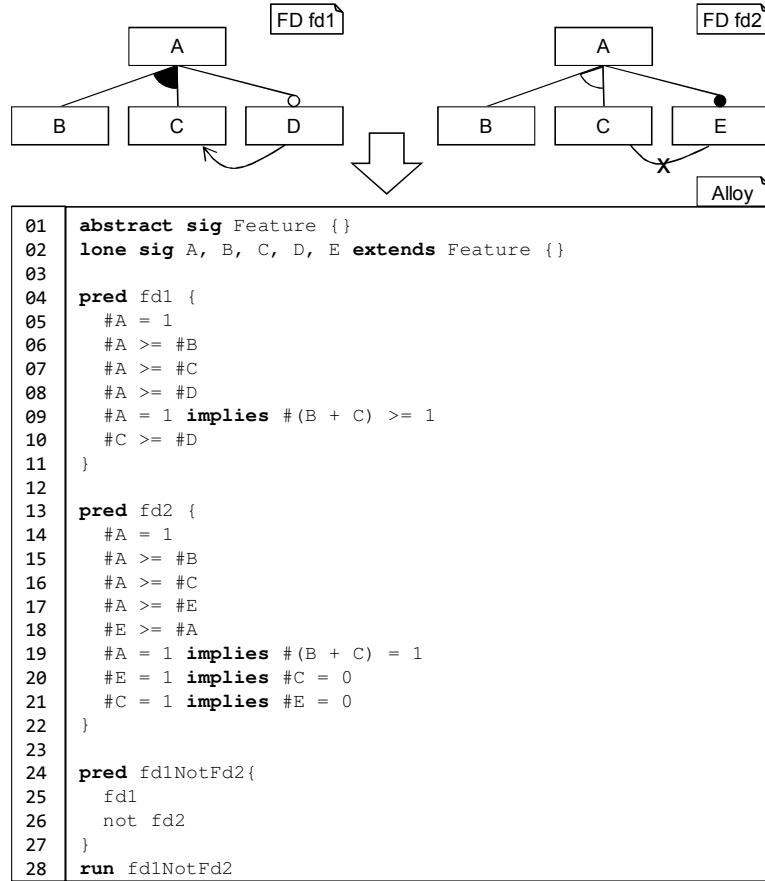


Figure 4.3: Two FDs and the Alloy module resulting from translating the FDs.

predicate contains the constraint $\#C \geq \#D$ (l. 10) because the FD contains an implies constraint from the feature D to the feature C.

The predicate `fd2` results from translating the FD `fd2`. The first four constraints (ll. 14-17) of the predicate `fd2` are constructed analogously as the first four constraints of the predicate `fd1` (ll. 5-8). The predicate `fd2` contains the constraint $\#E \geq \#A$ (l. 18) because E is a mandatory child of A in the FD `fd2`. The constraint $\#A = 1 \text{ implies } \#(B + C) = 1$ (l. 19) results from translating the xor-group of `fd2`. The predicate contains the constraints $\#E = 1 \text{ implies } \#C = 0$ and $\#C = 1 \text{ implies } \#E = 0$ (ll. 20-21) because `fd2` contains an excludes constraint between E and C.

The translation from FDs to Alloy modules always produces the predicate `fd1NotFd2` (ll. 24-27) and the run command depicted in Figure 4.3, l. 28. The predicate `fd1NotFd2` models all instances representing configurations that are valid in the FD represented by

4.3 SEMANTIC DIFFERENCING OF FEATURE DIAGRAMS

FD	#Features	#Constraints
car	9	15
car1	8	11
car2	9	12
tablet1	12	18
tablet2	14	21
tablet3	14	23
fd1	4	6
fd2	4	8
fd3	2	3
fd4	3	4

Figure 4.4: The number of features in the example FDs and the number of constraints in the Alloy predicates generated from the FDs.

the predicate `fd1` and not valid in the FD represented by the predicate `fd2`. Running the command (l. 28) yields instances iff the semantic difference is not empty.

Experiments

We performed experimental evaluations with the ten example FDs presented in Appendix B. Figure 4.4 summarizes the sizes of the FDs in terms of the numbers of their features and the number of constraints contained in the predicates resulting from translating the FDs. For example, the FD `tablet3` has 14 features and the predicate resulting from translating `tablet3` to Alloy consists of 23 constraints. All experiments were executed on a laptop computer with an Intel Core i7-8650U CPU @ 1.90GHz processor, 16GB RAM, and a Samsung PM981 512GB SSD hard drive using Windows 10, Java 1.8.0_192, and the Java API of Alloy 4.2 using the satisfiability solver SAT4J.

We executed the semantic differencing operator for all pairs of example FDs that are thematically related. Figure 4.5 summarizes the computation times of the semantic differencing operator implementation. Figure 4.5 also presents the (first) computed diff witnesses for the input pairs or indicates (by using the symbol `-`) that an FD is a refinement of the other FD. For example, the semantic differencing operator took 24ms to detect that the FD `car` is a refinement of the FD `car`. Similarly, computing the diff witness `{hybrid, phone, engine, car, locking, electric}` that is contained in the semantic difference from `car1` to `car` took 50ms. For the examples, the computation times range from 11ms to 112ms. We conclude that the implementation handles the example FDs sufficiently quick. However, the examples are relatively small in terms of the numbers of features of the FDs. Thus, the results cannot be generalized to large FDs and real world examples, especially because satisfiability checking is, in general, computationally hard.

CHAPTER 4 FEATURE DIAGRAMS

Difference	Time	Diff witness
$\delta(\text{car}, \text{car})$	24ms	-
$\delta(\text{car}, \text{car1})$	112ms	phone, engine, car, locking, electric, fingerprint
$\delta(\text{car}, \text{car2})$	60ms	-
$\delta(\text{car1}, \text{car})$	50ms	hybrid, phone, engine, car, locking, electric
$\delta(\text{car1}, \text{car1})$	29ms	-
$\delta(\text{car1}, \text{car2})$	36ms	hybrid, engine, car, locking, electric, gas, fingerprint
$\delta(\text{car2}, \text{car})$	40ms	-
$\delta(\text{car2}, \text{car1})$	85ms	phone, engine, car, locking, electric, fingerprint
$\delta(\text{car2}, \text{car2})$	29ms	-
$\delta(\text{tablet1}, \text{tablet1})$	43ms	-
$\delta(\text{tablet1}, \text{tablet2})$	69ms	tablet, wifi, 128GB, memory, 256GB, display, cellular, P200, dis11, processor, dis12
$\delta(\text{tablet1}, \text{tablet3})$	68ms	tablet, 128GB, memory, P100, 256GB, display, dis11, processor
$\delta(\text{tablet2}, \text{tablet1})$	59ms	tablet, memory, 256GB, display, P200, processor, dis12
$\delta(\text{tablet2}, \text{tablet2})$	39ms	-
$\delta(\text{tablet2}, \text{tablet3})$	47ms	tablet, wifi, memory, 64GB, display, cellular, P200, processor, dis12
$\delta(\text{tablet3}, \text{tablet1})$	42ms	tablet, memory, 256GB, display, P200, processor, dis12
$\delta(\text{tablet3}, \text{tablet2})$	45ms	tablet, memory, 256GB, display, P200, processor, dis10
$\delta(\text{tablet3}, \text{tablet3})$	28ms	-
$\delta(\text{fd1}, \text{fd1})$	19ms	-
$\delta(\text{fd1}, \text{fd2})$	19ms	A, B, C, D, E
$\delta(\text{fd1}, \text{fd3})$	21ms	A, C
$\delta(\text{fd1}, \text{fd4})$	24ms	A, B
$\delta(\text{fd2}, \text{fd1})$	18ms	A, B, D, E
$\delta(\text{fd2}, \text{fd2})$	17ms	-
$\delta(\text{fd2}, \text{fd3})$	13ms	-
$\delta(\text{fd2}, \text{fd4})$	13ms	A, B, E
$\delta(\text{fd3}, \text{fd1})$	14ms	A, B, D
$\delta(\text{fd3}, \text{fd2})$	11ms	A, B, C, E
$\delta(\text{fd3}, \text{fd3})$	12ms	-
$\delta(\text{fd3}, \text{fd4})$	16ms	A, B
$\delta(\text{fd4}, \text{fd1})$	15ms	-
$\delta(\text{fd4}, \text{fd2})$	15ms	A, B, C, E
$\delta(\text{fd4}, \text{fd3})$	12ms	A, C
$\delta(\text{fd4}, \text{fd4})$	14ms	-

Figure 4.5: The time needed by the semantic differencing operator for semantic differencing of the pairs of example FDs.

No.	Operation	Ref.	Gen.
1.	Adding a feature as optional child	✓	✗
2.	Deleting a non-root leaf feature	✗	✓
3.	Adding an implies constraint	✓	✗
4.	Deleting an implies constraint	✗	✓
5.	Adding an excludes constraint	✓	✗
6.	Deleting an excludes constraint	✗	✓
7.	Creating an or-group	✓	✗
8.	Change type of group to or	✗	✓
9.	Change type of group to xor	✓	✗
10.	Making a mandatory feature optional	✗	✓
11.	Making an optional feature mandatory	✓	✗
12.	Adding a child feature to a group of its parent feature	✗	✗
13.	Excluding a child feature from a group of its parent feature	✗	✗
14.	Renaming the root feature	✗	✗

Figure 4.6: Feature diagram change operation properties.

4.4 Feature Diagram Change Operations

This section presents FD change operations that are used to define a complete FD change operation suite. Some of the change operations are neither refining nor generalizing. Although these change operations are irrelevant for developers to constructively refine or generalize models, the change operations are necessary to obtain a complete change operation suite. The completeness of the change operation suite is required for the model repair framework presented in Chapter 7 and the framework’s instantiation presented in Chapter 8. If a change operation is refining or generalizing, then it is possible to incorporate performance improvements into algorithms that compute solutions (cf. Section 7.5) for special model repair problems as introduced in Section 8.1.

Figure 4.6 overviews the FD change operations, which are inspired by change operations from [TBK09, BKL⁺16]. The addition of a feature (cf. No. 1) adds an unused feature as an optional child of a used feature to the FD. Feature addition operations are refining but not generalizing. Vice versa, removing a non-root feature without children (cf. No. 2) is a generalizing change operation. Operations that add cross-tree constraints (cf. No. 3, 5) strengthen the constraints induced by the FD on its valid configurations and are, therefore, refining. Dually, operations for removing cross-tree constraints (cf. No. 4, 6) are generalizing. A change operation for creating an or-group (cf. No. 7) creates an or-group containing a single feature that is an optional child of its parent before applying the change operation. The parent of the feature contained in the group remains unchanged. Therefore, change operations for creating or-groups are refining. Converting

an xor-group to an or-group (cf. No. 8) is a generalizing change operation because or-groups induce weaker constraints than xor-groups. Dually, operations for converting an or-group to an xor-group (cf. No. 9) are refining change operation. Operations for making mandatory features optional (cf. No. 10) are generalizing. Operations for making optional features mandatory (cf. 11) are refining. Operations for adding features to groups (cf. No. 12) are neither refining nor generalizing. Similarly, operations for excluding features from groups (cf. No. 13) are neither refining nor generalizing, either. Operations for renaming the root feature (cf. No. 14) are neither refining nor generalizing.

4.4.1 Feature-Addition Operations

Feature-addition operations with signature $M_{FD} \rightarrow M_{FD}$	
Parameters	Let $f, g \in U_N$ be two names representing features.
Explanation	The operation $addF_{f,g}$ adds the unused feature g as optional child of the feature f .
Domain	$(F, E, r, M, Or, Xor, I, X) \in dom(addF_{f,g}) \Leftrightarrow f \in F \wedge g \notin F$
Application	$ \begin{array}{c} (F, E, r, M, Or, Xor, I, X) \\ \downarrow \\ (F', E', r, M, Or, Xor, I, X) \text{ where } F' = F \cup \{g\} \\ E' = E \cup \{(f, g)\} \end{array} $
Example	

Figure 4.7: Feature-addition operations $addF_{f,g}$ for all feature names $f, g \in U_N$.

Figure 4.7 defines the feature-addition operations. Each feature-addition operation $addF_{f,g}$ is parametrized with two names $f, g \in U_N$ representing feature. On the application of the feature-addition operation $addF_{f,g}$ to an FD, the feature g is added as an optional child to the feature f . The operation is applicable to an FD iff the feature g does not exist in the FD and the feature f exists in the FD. This condition ensures that the resulting FD is well-formed.

Thus, an already defined feature cannot be added and a feature can only be added as a child of an already existing feature. In the example application (Figure 4.7), the feature g is added as an optional child of the feature f .

Adding a feature as a child to another feature strengthens the constraints on the valid configurations modeled by the FD. Thus, feature-addition operations are refining:

Proposition 4.4. *Let $addF_{f,g}$ be a feature-addition operation and let $fd \in M_{FD}$ be an FD such that $addF_{f,g}$ is applicable to fd . Then, $\llbracket addF_{f,g}(fd) \rrbracket^{FD} \subseteq \llbracket fd \rrbracket^{FD}$.*

Proof. Let $f, g \in U_N$, $addF_{f,g}$, and $fd = (F, E, r, M, Or, Xor, I, X) \in M_{FD}$ be given as above. Let $C \in \llbracket addF_{f,g}(fd) \rrbracket^{FD}$ be a valid configuration of the FD $addF_{f,g}(fd)$. Then, all the seven conditions in Definition 4.2 are satisfied for $addF_{f,g}(fd)$ and C . As the application of the feature-addition operation $addF_{f,g}$ does not change the root and does not change the sets M , Or , Xor , I , and X , the first condition as well as the conditions 3.-7. are also satisfied for C and fd . As the second condition is satisfied for $addF_{f,g}(fd)$ and C , it holds that $\forall h \in C \cap (F \cup \{g\}) : h \neq r \Rightarrow p(h) \in C$. Thus, it especially holds that $\forall h \in C \cap F : h \neq r \Rightarrow p(h) \in C$. Thus, the second condition is satisfied for C and fd . We can conclude that C is valid in fd . \square

4.4.2 Feature-Deletion Operations

Feature-deletion operations with signature $M_{FD} \rightarrow M_{FD}$	
Parameters	Let $f \in U_N$ be a name representing a feature.
Explanation	The operation $delF_f$ deletes the non-root, optional, non-used, leaf feature f .
Domain	$(F, E, r, M, Or, Xor, I, X) \in dom(delF_f) \Leftrightarrow f \in F \setminus \{r\} \wedge$ $\forall (a, b) \in E : a \neq f \wedge$ $\forall (a, b) \in M : b \neq f \wedge$ $\forall (p, G) \in Or \cup Xor : p \neq f \wedge f \notin G \wedge$ $\forall (a, b) \in I \cup X : a \neq f \wedge b \neq f$
Application	$delF_f$ $(F, E, r, M, Or, Xor, I, X)$ \downarrow $(F', E', r, M, Or, Xor, I, X)$ where $F' = F \setminus \{f\}$ $E' = E \setminus \{(a, b) \in E \mid b = f\}$
Example	

Figure 4.8: Feature-deletion operations $delF_f$ for all feature names $f \in U_N$.

Figure 4.8 defines the feature-deletion operations. Each feature-deletion operation $delF_f$ is parametrized with a name $f \in U_N$ representing a feature. On the application of the feature-deletion operation $delF_f$ to an FD, the feature f is removed from the FD. The operation is applicable to an FD iff the feature f exists in the FD and is not the root feature, the feature f is a leaf feature in the FD, the feature f is not a mandatory child of its parent, the feature f is not part of any of its parent feature's groups, and the feature f is not used in any of the FD's implies and excludes constraints. These conditions ensure that the resulting FD is well-formed. Deleting the root is not possible because we require that an FD always contains at least one feature (the root feature). Thus, it is only possible to delete optional, non-root, leaf features. In the example application (Figure 4.8), the feature f is removed from the example FD.

Deleting a child feature weakens the constraints induced by an FD. Therefore, feature-deletion operations are generalizing:

Proposition 4.5. *Let $delF_f$ be a feature-deletion operation and let $fd \in M_{FD}$ be an fd such that $delF_f$ is applicable to fd . Then, $\llbracket fd \rrbracket^{FD} \subseteq \llbracket delF_f(fd) \rrbracket^{FD}$.*

Proof. Let $f \in U_N$, $delF_f$, and $fd = (F, E, r, M, Or, Xor, I, X) \in M_{FD}$ be given as above. Let $C \in \llbracket fd \rrbracket^{FD}$ be a valid configuration of the FD fd . Then, all the seven conditions in Definition 4.2 are satisfied for fd and C . As the application of the feature-deletion operation $delF_f$ does not change the root and does not change the sets M , Or , Xor , I , and X , the first conditions as well as the conditions 3.-7. are also satisfied for C and $delF_f(fd)$. As the second condition is satisfied for C and fd , it holds that $\forall h \in C \cap F : h \neq r \Rightarrow p(h) \in C$. Thus, it especially holds that $\forall h \in (C \setminus \{f\}) \cap F : h \neq r \Rightarrow p(h) \in C$. Thus, the second condition is satisfied for C and $delF_f(fd)$. From the above, we can conclude that C is valid in $delF_f(fd)$. \square

4.4.3 Implies-Constraint-Addition Operations

Implies-constraint-addition operations with signature $M_{FD} \rightarrow M_{FD}$	
Parameters	Let $f, g \in U_N$ be two names representing features.
Explanation	The operation $addI_{f,g}$ adds an implies constraint from the feature f to the feature g .
Domain	$(F, E, r, M, Or, Xor, I, X) \in \text{dom}(addI_{f,g}) \Leftrightarrow f, g \in F$
Application	$\begin{array}{c} (F, E, r, M, Or, Xor, I, X) \\ \downarrow \\ (F, E, r, M, Or, Xor, I', X) \text{ where } I' = I \cup \{(f, g)\} \end{array}$
Example	

Figure 4.9: Implies-constraint-addition operations $addI_{f,g}$ for feature names $f, g \in U_N$.

Figure 4.9 defines the implies-constraint-addition operations. Each implies-constraint-addition operation $addI_{f,g}$ is parametrized with two feature names $f, g \in U_N$. On the application of the implies-constraint-addition operation $addI_{f,g}$ to an FD, an implies constraint from the feature f to the feature g is added to the FD. The operation is applicable to an FD iff the features f and g exist in the FD. This condition ensures that the resulting FD is well-formed. Thus, it is only possible to add implies constraints between features that are defined in the FD. In the example application (Figure 4.9), an implies constraint from the feature f to the feature g is added.

Adding an implies constraint to an FD strengthens the constraints induced by the FD on the set of its valid configurations. Thus, implies-constraint-addition operations are refining change operations.

Proposition 4.6. *Let $\text{add}I_{f,g}$ be an implies-constraint-addition operation and let $fd \in M_{FD}$ be an FD such that $\text{add}I_{f,g}$ is applicable to fd . Then, $\llbracket \text{add}I_{f,g}(fd) \rrbracket^{FD} \subseteq \llbracket fd \rrbracket^{FD}$.*

Proof. Let $f, g \in U_N$, $\text{add}I_{f,g}$, and $fd = (F, E, r, M, Or, Xor, I, X) \in M_{FD}$ be given as above. Let $C \in \llbracket \text{add}I_{f,g}(fd) \rrbracket^{FD}$ be a valid configuration of the FD $\text{add}I_{f,g}(fd)$. Then, all the seven conditions in Definition 4.2 are satisfied for $\text{add}I_{f,g}(fd)$ and C . As the application of the feature addition operation $\text{add}I_{f,g}$ does not change the root and does not change the sets F , E , Or , Xor , and X , the conditions 1.-5. as well as 7. are also satisfied for C and fd . As the sixth condition is satisfied for $\text{add}I_{f,g}(fd)$ and C , it holds that $\forall (h, i) \in I \cup \{(f, g)\} : h \in C \Rightarrow i \in C$. Thus, it especially holds that $\forall (h, i) \in I : h \in C \Rightarrow g \in C$. Thus, the sixth condition is satisfied for C and fd . We can conclude that C is valid in fd . \square

4.4.4 Implies-Constraint-Deletion Operations

Implies-constraint-deletion operations with signature $M_{FD} \rightarrow M_{FD}$	
Parameters	Let $f, g \in U_N$ be two names representing features.
Explanation	The operation $\text{del}I_{f,g}$ deletes an implies constraint from the feature f to the feature g .
Domain	$(F, E, r, M, Or, Xor, I, X) \in \text{dom}(\text{del}I_{f,g}) \Leftrightarrow (f, g) \in I$
Application	$\begin{array}{c} (F, E, r, M, Or, Xor, I, X) \\ \downarrow \\ (F, E, r, M, Or, Xor, I', X) \text{ where } I' = I \setminus \{(f, g)\} \end{array}$
Example	

Figure 4.10: Implies-constraint-deletion operations $\text{del}I_{f,g}$ for feature names $f, g \in U_N$.

Figure 4.10 defines the implies-constraint-deletion operations. Each implies-constraint-deletion operation $\text{del}I_{f,g}$ is parametrized with two names $f, g \in U_N$ representing features. On the application of the implies-constraint-deletion operation $\text{del}I_f$ to an FD, the implies constraint from the feature f to the feature g is removed from the FD. The operation is applicable to an FD iff the features f and g both exist in the FD and the FD contains an implies constraint from the feature f to the feature g . This explicates that it is only possible to delete implies constraints that are defined in the FD. In the example application, an implies constraint from the feature f to the feature g is deleted.

Deleting an implies constraint from an FD relaxes the constraints induced by the FD on its valid configurations.

Proposition 4.7. *Let $delI_{f,g}$ be an implies-constraint-deletion operation and let $fd \in M_{FD}$ be an FD such that $delI_{f,g}$ is applicable to fd . Then, $\llbracket fd \rrbracket^{FD} \subseteq \llbracket delI_{f,g}(fd) \rrbracket^{FD}$.*

Proof. Let $f, g \in U_N$, $delI_{f,g}$, and $fd = (F, E, r, M, Or, Xor, I, X) \in M_{FD}$ be given as above. Let $C \in \llbracket fd \rrbracket^{FD}$ be a valid configuration of the FD fd . Then, all the seven conditions in Definition 4.2 are satisfied for fd and C . As the application of the implies-constraint-deletion operation $delI_{f,g}$ does not change the root and does not change the sets F , E , M , Or , Xor , and X , the conditions 1.-5. and 7. are also satisfied for C and $delI_{f,g}(fd)$. As the sixth condition is satisfied for fd and C , it holds that $\forall (h, i) \in I : h \in C \Rightarrow i \in C$. Thus, it especially holds that $\forall (h, i) \in I \setminus \{(f, g)\} : h \in C \Rightarrow i \in C$. Thus, the sixth condition is satisfied for C and $delI_{f,g}(fd)$. From the above, we can conclude that C is valid in $delI_{f,g}(fd)$. \square

4.4.5 Excludes-Constraint-Addition Operations

Excludes-constraint-addition operations with signature $M_{FD} \rightarrow M_{FD}$	
Parameters	Let $f, g \in U_N$ be two names representing features.
Explanation	The operation $addX_{f,g}$ adds an excludes constraint from the feature f to the feature g .
Domain	$(F, E, r, M, Or, Xor, I, X) \in dom(addX_{f,g}) \Leftrightarrow f, g \in F$
Application	$ \begin{array}{c} (F, E, r, M, Or, Xor, I, X) \\ \downarrow \\ addX_{f,g} \quad (F, E, r, M, Or, Xor, I, X') \text{ where } X' = X \cup \{(f, g)\} \end{array} $
Example	

Figure 4.11: Excludes-constraint-addition operations $addX_{f,g}$ for features $f, g \in U_N$.

Figure 4.11 defines the excludes-constraint-addition operations. Similar to implies constraints, excludes constraints can be added between two arbitrary defined features. Each excludes-constraint-addition operation $addX_{f,g}$ is parametrized with two names $f, g \in U_N$ representing feature names. On the application of the excludes-constraint-addition operation $addX_{f,g}$ to an FD, an excludes constraint from the feature f to the feature g is added to the FD. The operation is applicable to an FD iff the features f and g exist in the FD. Thus, it is only possible to add excludes constraints between features that are defined in the FD. In the example application, an excludes constraint from the feature f to the feature g is added.

Adding an excludes constraint to an FD strengthens the constraints induced by the FD on its valid configurations. Thus, excludes-constraint-addition operations are refining.

Proposition 4.8. *Let $addX_{f,g}$ be an excludes-constraint-addition operation and let $fd \in M_{FD}$ be an FD such that $addX_{f,g}$ is applicable to fd . Then, $\llbracket addX_{f,g}(fd) \rrbracket^{FD} \subseteq \llbracket fd \rrbracket^{FD}$.*

Proof. Let $f, g \in U_N$, $addX_{f,g}$, and $fd = (F, E, r, M, Or, Xor, I, X) \in M_{FD}$ be given as above. Let $C \in \llbracket addX_{f,g}(fd) \rrbracket^{FD}$ be a valid configuration of the FD $addX_{f,g}(fd)$. Then, all the seven conditions in Definition 4.2 are satisfied for $addX_{f,g}(fd)$ and C . As the application of the excludes-constraint-addition operation $addX_{f,g}$ does not change the root and does not change the sets F , E , Or , Xor , and I , the conditions 1-6. are also satisfied for C and fd . As the seventh condition is satisfied for $addX_{f,g}(fd)$ and C , it holds that $\forall (h, i) \in X \cup \{(f, g)\} : h \in C \Rightarrow i \notin C$. Thus, it especially holds that $\forall (h, i) \in X : h \in C \Rightarrow i \notin C$. Thus, the seventh condition is satisfied for C and fd . We can conclude that C is valid in fd . \square

4.4.6 Excludes-Constraint-Deletion Operations

Excludes-constraint-deletion operations with signature $M_{FD} \rightarrow M_{FD}$	
Parameters	Let $f, g \in U_N$ be two names representing features.
Explanation	The operation $delX_{f,g}$ deletes an excludes constraint from the feature f to the feature g .
Domain	$(F, E, r, M, Or, Xor, I, X) \in \text{dom}(delX_{f,g}) \Leftrightarrow (f, g) \in X$
Application	$\begin{array}{c} (F, E, r, M, Or, Xor, I, X) \\ \downarrow \\ delX_{f,g} \\ (F, E, r, M, Or, Xor, I, X') \text{ where } X' = X \setminus \{(f, g)\} \end{array}$
Example	

Figure 4.12: Excludes-constraint-deletion operations $delX_{f,g}$ for features $f, g \in U_N$.

Figure 4.12 defines the excludes-constraint-deletion operations. Each excludes-constraint-deletion operation is applicable to all FDs that contain the constraint to delete. Each excludes-constraint-deletion operation $delX_{f,g}$ is parametrized with two feature names $f, g \in U_N$. On the application of the excludes-constraint-deletion operation $delX_{f,g}$ to an FD, the excludes constraint from the feature f to the feature g is removed from the FD. The operation is applicable to an FD iff the features f and g exist in the FD and the FD contains an excludes constraint from the feature f to the feature g . This explicates that it is only possible to delete excludes constraints that are defined in the FD. In the example, an excludes constraint from the feature f to the feature g is deleted.

Deleting an excludes constraints from an FD relaxes the constraints induced by the FD on its valid configurations.

Proposition 4.9. *Let $delX_{f,g}$ be an excludes-constraint-deletion operation and let $fd \in M_{FD}$ be an FD such that $delX_{f,g}$ is applicable to fd . Then, $\llbracket fd \rrbracket^{FD} \subseteq \llbracket delX_{f,g}(fd) \rrbracket^{FD}$.*

Proof. Let $f, g \in U_N$, $delX_{f,g}$, and $fd = (F, E, r, M, Or, Xor, I, X) \in M_{FD}$ be given as above. Let $C \in \llbracket fd \rrbracket^{FD}$ be a valid configuration of the FD fd . Then, all the seven conditions in Definition 4.2 are satisfied for fd and C . As the application of the excludes-constraint-deletion operation $delX_{f,g}$ does not change the root and does not change the sets F , E , M , Or , Xor , and I , the conditions 1.-6. are also satisfied for C and $delX_{f,g}(fd)$. As the seventh condition is satisfied for fd and C , it holds that $\forall (h, i) \in X : h \in C \Rightarrow i \notin C$. Thus, it especially holds that $\forall (h, i) \in X \setminus \{(f, g)\} : h \in C \Rightarrow i \notin C$. Thus, the seventh condition is satisfied for C and $delX_{f,g}(fd)$. From the above, we can conclude that C is valid in $delX_{f,g}(fd)$. \square

4.4.7 Or-Group-Creation Operations

Or-group-creation operations with signature $M_{FD} \rightarrow M_{FD}$	
Parameters	Let $f \in U_N$ be a name representing a feature.
Explanation	The operation $createOr_f$ creates an or-group containing the feature f as group of the parent feature of f .
Domain	$(F, E, r, M, Or, Xor, I, X) \in \text{dom}(\text{addOr}_f) \Leftrightarrow$ $f \in F \setminus \{r\} \wedge$ $\forall (a, b) \in M : b \neq f \wedge$ $\forall (p, G) \in Or \cup Xor : f \notin G$
Application	$createOr_f(F, E, r, M, Or, Xor, I, X)$ \downarrow $(F, E, r, M, Or', Xor, I, X)$ where $Or' = Or \cup \{(parent(f), \{f\})\}$
Example	

Figure 4.13: Or-group-creation operations $createOr_f$ for all feature names $f \in U_N$.

Figure 4.13 defines the or-group-creation operations. Each or-group-creation operation $createOr_f$ is parametrized with a name $f \in U_N$ representing a feature name. The application of the or-group-creation operation $createOr_f$ to an FD creates an or-group containing the feature f as a subgroup of the parent feature of f . The operation is applicable to an FD iff the feature f exists in the FD and the feature f is an optional child feature of its parent. Thus, it is only possible create groups from optional child features. In the example application (Figure 4.13), the optional child feature f of its

parent feature r is added to an or-group that solely contains the feature f as subgroup of the feature r .

Or-group-creation operations create groups containing one feature. This feature is part of every configuration that contains the feature's parent feature. As the feature is optional before applying the operation, or-group-creation operations are refining.

Proposition 4.10. *Let $createOr_f$ be an or-group-creation operation and let $fd \in M_{FD}$ be an FD such that $createOr_f$ is applicable to fd . Then, $\llbracket createOr_f(fd) \rrbracket^{FD} \subseteq \llbracket fd \rrbracket^{FD}$.*

Proof. Let $f \in U_N$, $createOr_f$, and $fd = (F, E, r, M, Or, Xor, I, X) \in M_{FD}$ be given as above. Let $C \in \llbracket createOr_f(fd) \rrbracket^{FD}$ be a valid configuration of the FD $createOr_f(fd)$. Then, all the seven conditions in Definition 4.2 are satisfied for $createOr_f(fd)$ and C . As the application of the or-group-creation operation $createOr_f$ does not change the root and does not change the sets F , E , M , Xor , I , and X , the conditions 1.-3. as well as 5.-7. are also satisfied for C and fd . As the fourth condition is satisfied for $createOr_f(fd)$ and C , it holds that $\forall (p, G) \in Or \cup \{(parent(f), f)\} : p \in C \Rightarrow |C \cap G| \geq 1$. Thus, it especially holds that $\forall (p, G) \in Or : p \in C \Rightarrow |C \cap G| \geq 1$. Thus, the fourth condition is satisfied for C and fd . We can conclude that C is valid in fd . \square

4.4.8 Xor-to-Or-Conversion Operations

Xor-to-or-conversion operations with signature $M_{FD} \rightarrow M_{FD}$	
Parameters	Let $p \in U_N$ be a name representing a feature, $G \subseteq U_N$ be a finite set of names representing a set of features.
Explanation	The operation $xor2or_{p,G}$ changes the xor-group G with parent p to an or-group.
Domain	$(F, E, r, M, Or, Xor, I, X) \in dom(xor2or_{p,G}) \Leftrightarrow (p, G) \in Xor$
Application	$ \begin{array}{c} (F, E, r, M, Or, Xor, I, X) \\ \downarrow \\ xor2or_{p,G} \\ (F, E, r, M, Or', Xor', I, X) \text{ where } Or' = Or \cup \{(p, G)\} \\ Xor' = Xor \setminus \{(p, G)\} \end{array} $
Example	

Figure 4.14: Xor-to-or-conversion operations $xor2or_{p,G}$ for all feature names $p \in U_N$ and finite sets of feature names $G \subseteq U_N$.

Figure 4.14 defines the xor-to-or-conversion operations. Each xor-to-or-conversion operation $xor2or_{p,G}$ is parametrized with a feature name $p \in U_N$, and a finite set of names $G \subseteq U_N$ representing the names of the features participating in an xor-group. On the application of the xor-to-or-conversion operation $xor2or_{p,G}$ to an FD, the xor-group that is defined by the parent feature p and the features participating in the group

represented by the feature names in the set G is changed to an or-group. The operation is applicable to an FD iff the FD contains a xor-group with parent p where the group's participants are all the features contained in the set G . Thus, it is only possible to change the type of xor-groups that are defined in the FD. In the example application (Figure 4.14), the xor-group defined by $(r, \{f, g\})$ is changed to an or-group.

As the constraint induced by an xor-group is more constraining than the constraint induced by an or-group, xor-to-or-conversion operations are generalizing.

Proposition 4.11. *Let $xor2or_{p,G}$ be an xor-to-or-conversion operation and let fd be an FD such that $xor2or_{p,G}$ is applicable to fd . Then, $\llbracket fd \rrbracket^{FD} \subseteq \llbracket xor2or_{p,G}(fd) \rrbracket^{FD}$.*

Proof. Let $p \in U_N$, $G \subseteq U_N$, $xor2or_{p,G}$, and $fd = (F, E, r, M, Or, Xor, I, X) \in M_{FD}$ be given as above. Let $C \in \llbracket fd \rrbracket^{FD}$ be a valid configuration of the FD fd . Then, all the seven conditions in Definition 4.2 are satisfied for fd and C . As the application of the xor-to-or-conversion operation $xor2or_{p,G}$ does not change the root and does not change the sets F , E , M , I , and X , the conditions 1.-3. and 6.-7. are also satisfied for C and $xor2or_{p,G}(fd)$. As the fifth condition is satisfied for fd and C , it holds that $\forall (p', G') \in Xor : p' \in C \Rightarrow |C \cap G'| = 1$. Thus, it especially holds that $\forall (p', G') \in Xor \setminus \{(p, G)\} : p' \in C \Rightarrow |C \cap G'| = 1$. This implies that the fifth condition is satisfied for the FD $xor2or_{p,G}(fd)$ and C . This also especially implies that $p \in C \Rightarrow |C \cap G| = 1$ holds. Thus, it holds that $p \in C \Rightarrow |C \cap G| \geq 1$. As the fourth condition is satisfied for the FD fd and C , it holds that $\forall (p', G') \in Or : p' \in C \Rightarrow |C \cap G'| \geq 1$. From this and the previous statement, we can infer that $\forall (p', G') \in Or \cup \{(p, G)\} : p' \in C \Rightarrow |C \cap G'| \geq 1$ holds. Therefore, the fourth condition is satisfied for $xor2or_{p,G}(fd)$ and C . From the above, we can conclude that C is valid in $xor2or_{p,G}(fd)$. \square

4.4.9 Or-to-Xor-Conversion Operations

Figure 4.15 defines the or-to-xor-conversion operations. Each or-to-xor-conversion operation $or2xor_{p,G}$ is parametrized with a name $p \in U_N$ representing a feature and a finite set of names $G \subseteq U_N$ representing the names of the features participating in an or-group. On the application of the or-to-xor-conversion operation $or2xor_{p,G}$ to an FD, the or-group that is defined by the parent feature p and the features participating in the group represented by the feature names in the set G is changed to an xor-group. The operation is applicable to an FD iff the FD contains an or-group with parent p where the group's participants are all the features contained in the set G . With this, it is only possible to change the type of or-groups that are defined in the FD. In the example application (Figure 4.15), an or-group with the parent r and set of group participants $\{f, g\}$ is changed to an xor-group.

As the constraint induced by an xor-group is more constraining than the constraint induced by an or-group, or-to-xor-conversion operations are refining.

Or-to-xor-conversion operations with signature $M_{FD} \rightarrow M_{FD}$	
Parameters	Let $p \in U_N$ be a name representing a feature, $G \subseteq U_N$ be a finite set of names representing a set of features.
Explanation	The operation $or2xor_{p,G}$ changes the or-group G with parent p to an xor-group.
Domain	$(F, E, r, M, Or, Xor, I, X) \in dom(or2xor_{p,G}) \Leftrightarrow (p, G) \in Or$
Application	$ \begin{array}{c} (F, E, r, M, Or, Xor, I, X) \\ \downarrow \\ or2xor_{p,G} (F, E, r, M, Or', Xor', I, X) \text{ where } Or' = Or \setminus \{(p, G)\} \\ Xor' = Xor \cup \{(p, G)\} \end{array} $
Example	

Figure 4.15: Or-to-xor-conversion operations $or2xor_{p,G}$ for all feature names $p \in U_N$ and finite sets of feature names $G \subseteq U_N$.

Proposition 4.12. *Let $or2xor_{p,G}$ be an or-to-xor-conversion operation and let fd be an FD such that $or2xor_{p,G}$ is applicable to fd . Then, $\llbracket or2xor_{p,G}(fd) \rrbracket^{FD} \subseteq \llbracket fd \rrbracket^{FD}$.*

Proof. Let $p \in U_N$, $G \subseteq U_N$, $or2xor_{p,G}$, and $fd = (F, E, r, M, Or, Xor, I, X)$ be given as above. Let $C \in \llbracket or2xor_{p,G}(fd) \rrbracket^{FD}$ be a valid configuration of the FD $or2xor_{p,G}(fd)$. Then, all the seven conditions in Definition 4.2 are satisfied for $or2xor_{p,G}(fd)$ and C . As the application of the or-to-xor-conversion operation $or2xor_{p,G}$ does not change the root and does not change the sets F , E , M , I , and X , the conditions 1.-3. and 6.-7. are also satisfied for C and fd . As the fifth condition is satisfied for $or2xor_{p,G}(fd)$ and C , it holds that $\forall (p', G') \in Xor \cup \{(p, G)\} : p' \in C \Rightarrow |C \cap G'| = 1$. Thus, it especially holds that $\forall (p', G') \in Xor : p' \in C \Rightarrow |C \cap G'| = 1$. This implies that the fifth condition is satisfied for the FD fd and C . This also especially implies that $p \in C \Rightarrow |C \cap G| = 1$ holds. Thus, it holds that $p \in C \Rightarrow |C \cap G| \geq 1$. As the fourth condition is satisfied for the FD $or2xor_{p,G}(fd)$ and C , it holds that $\forall (p', G') \in Or \setminus \{(p, G)\} : p' \in C \Rightarrow |C \cap G'| \geq 1$. From this and the previous statement, we can infer that $\forall (p', G') \in (Or \setminus \{(p, G)\}) \cup \{(p, G)\} : p' \in C \Rightarrow |C \cap G'| \geq 1$ holds. This is equivalent to $\forall (p', G') \in Or \cup \{(p, G)\} : p' \in C \Rightarrow |C \cap G'| \geq 1$. Therefore, the fourth condition is satisfied for fd and C . We can conclude that C is valid in fd . \square

4.4.10 Mandatory-to-Optional-Conversion Operations

Figure 4.16 defines the mandatory-to-optional-conversion operations. Each mandatory-to-optional-conversion operation $man2opt_f$ is parametrized with a feature name $f \in U_N$. On the application of the mandatory-to-optional-conversion operation $man2opt_f$ to an FD, the mandatory child feature f is changed to an optional child feature of its parent. The operation is applicable to a FD iff the feature f exists in the FD and the feature f

Mandatory-to-optional-conversion operations with signature $M_{FD} \rightarrow M_{FD}$	
Parameters	Let $f \in U_N$ be a name representing a feature.
Explanation	The operation $man2opt_f$ makes the mandatory feature f an optional subfeature of its parent feature.
Domain	$(F, E, r, M, Or, Xor, I, X) \in dom(man2opt_f) \Leftrightarrow \exists (a, b) \in M : b = f$
Application	$\begin{array}{c} (F, E, r, M, Or, Xor, I, X) \\ \downarrow \\ (F, E, r, M', Or, Xor, I, X) \text{ where } M' = M \setminus \{(a, b) \in M \mid b = f\} \end{array}$
Example	

Figure 4.16: Mandatory feature to optional feature conversion operations $man2opt_f$ for all feature names $f \in U_N$.

is a mandatory child feature of its parent. In the example application (Figure 4.16), the mandatory child feature f of its parent feature r is changed to an optional child.

Making a mandatory feature optional in an FD relaxes the constraints induced by the FD on its valid configuration.

Proposition 4.13. *Let $man2opt_f$ be a mandatory-to-optional-conversion operation and let fd be an FD such that $man2opt_f$ is applicable to fd . Then, it holds that $\llbracket fd \rrbracket^{FD} \subseteq \llbracket man2opt_f(fd) \rrbracket^{FD}$.*

Proof. Let $f \in U_N$, $man2opt_f$, and $fd = (F, E, r, M, Or, Xor, I, X) \in M_{FD}$ be given as above. Let $C \in \llbracket fd \rrbracket^{FD}$ be a valid configuration of the FD fd . Then, all the seven conditions in Definition 4.2 are satisfied for fd and C . As the application of the mandatory feature to optional feature conversion operation $man2opt_f$ does not change the root and does not change the sets F , E , Or , Xor , I , and X , the conditions 1.-2. as well as 4.-7. are also satisfied for C and $man2opt_f(fd)$. As the third condition is satisfied for fd and C , it holds that $\forall (h, i) \in M : h \in C \Rightarrow i \in C$. Thus, it especially holds that $\forall (h, i) \in M \setminus \{(a, b) \in M \mid b = f\} : h \in C \Rightarrow i \in C$. Thus, the third condition is satisfied for C and $man2opt_f(fd)$. We can conclude that C is valid in $man2opt_f(fd)$. \square

4.4.11 Optional-to-Mandatory-Conversion Operations

Figure 4.17 defines the optional-to-mandatory-conversion operations. Each optional-to-mandatory-conversion operation $opt2man_f$ is parametrized with a feature name $f \in U_N$. On the application of the optional-to-mandatory-conversion operation $opt2man_f$ to an FD, the optional child feature f is changed to a mandatory child of its parent. The operation is applicable to an FD iff the feature f exists in the FD and the feature f is an optional child feature of its parent. Thus, it is only possible to change existing optional

Optional-to-mandatory-conversion operations with signature $M_{FD} \rightarrow M_{FD}$	
Parameters	Let $f \in U_N$ be a name representing a feature.
Explanation	The operation $opt2man_f$ makes the optional feature f a mandatory subfeature of its parent feature.
Domain	$(F, E, r, M, Or, Xor, I, X) \in dom(opt2man_f) \Leftrightarrow f \in F \setminus \{r\} \wedge$ $\forall (a, b) \in M: b \neq f \wedge$ $\forall (p, G) \in Or \cup Xor: f \notin G$
Application	$(F, E, r, M, Or, Xor, I, X)$ \downarrow $(F, E, r, M', Or, Xor, I, X)$ where $M' = M \cup \{(a, b) \in E \mid b = f\}$
Example	

Figure 4.17: Optional-to-mandatory-conversion operations $opt2man_f$ for all feature names $f \in U_N$.

child features to mandatory child features. In the example application, the optional child feature f of its parent feature r is changed to a mandatory child of its parent.

Making an optional feature mandatory in an FD strengthens the constraints induced by the FD on its valid configuration.

Proposition 4.14. *Let $opt2man_f$ be an optional-to-mandatory-conversion operation and let $fd \in M_{FD}$ be an FD such that $opt2man_f$ is applicable to fd . Then, it holds that $\llbracket opt2man_f(fd) \rrbracket^{FD} \subseteq \llbracket fd \rrbracket^{FD}$.*

Proof. Let $f \in U_N$, $opt2man_f$, and $fd = (F, E, r, M, Or, Xor, I, X) \in M_{FD}$ be given as above. Let $C \in \llbracket opt2man_f(fd) \rrbracket^{FD}$ be a valid configuration of the FD fd . Then, all the seven conditions in Definition 4.2 are satisfied for $opt2man_f(fd)$ and C . As the application of the optional-to-mandatory-conversion operation $opt2man_f$ does not change the root and does not change the sets F , E , Or , Xor , I , and X , the conditions 1.-2. as well as 4.-7. are also satisfied for C and fd . As the third condition is satisfied for $opt2man_f(fd)$ and C , it holds that $\forall (h, i) \in M \cup \{(a, b) \in E \mid b = f\} : h \in C \Rightarrow i \in C$. Thus, it especially holds that $\forall (h, i) \in M : h \in C \Rightarrow i \in C$. Thus, the third condition is satisfied for C and fd . We can conclude that C is valid in fd . \square

4.4.12 Feature-Group-Insertion Operations

Figure 4.18 defines the feature-group-insertion operations. Each of the feature-group-insertion operations $add2Grp_{f,G}$ is parametrized with a feature name $f \in U_N$ and a finite set of feature names $G \subseteq U_N$ representing the participants of a group. On the application of the feature-group-insertion operation $add2Grp_{f,G}$ to an FD, the feature f

Feature-group-insertion operations with signature $M_{FD} \rightarrow M_{FD}$	
Parameters	Let $f \in U_N$ be a name representing a feature, $G \subseteq U_N$ be a finite set of names representing a set of features.
Explanation	The operation $add2Grp_{f,G}$ adds the optional feature f to the group G of its parent feature.
Domain	$(F, E, r, M, Or, Xor, I, X) \in dom(add2Grp_{f,G}) \Leftrightarrow f \in F \setminus \{r\} \wedge$ $\forall (a, b) \in M: b \neq f \wedge$ $\forall (p, G') \in Or \cup Xor: f \notin G' \wedge$ $\exists (a, b) \in E: b = f \wedge (a, G) \in Or \cup Xor$
Application	$ \begin{array}{c} (F, E, r, M, Or, Xor, I, X) \\ \downarrow add2Grp_{f,G} \\ (F, E, r, M, Or', Xor', I, X) \text{ where } Or' = \{(p, G') \in Or \mid G' \neq G\} \cup \\ \{(p, G \cup \{f\}) \mid (p, f) \in E \wedge (p, G) \in Or\} \\ Xor' = \{(p, G') \in Xor \mid G' \neq G\} \cup \\ \{(p, G \cup \{f\}) \mid (p, f) \in E \wedge (p, G) \in Xor\} \end{array} $
Example	

Figure 4.18: Feature-group-insertion operations $add2Grp_{f,G}$ for all feature names $f \in U_N$, and finite sets of feature names $G \subseteq U_N$.

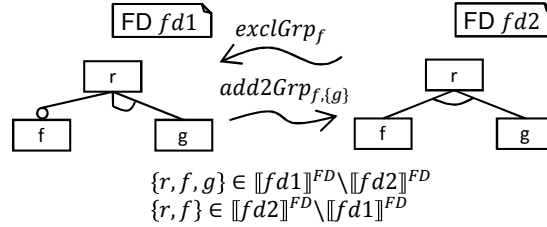


Figure 4.19: Example FDs illustrating that group-insertion operations and group-exclusion operations are neither refining nor generalizing.

is added to its parent feature's group defined by the set of features G . The operation is applicable to an FD iff the feature f is not the root feature, the feature f is an optional child feature of its parent feature, and the feature f and the group defined by the features in G share the same parent. Thus, it is only possible to add an optional feature to a group of its parent feature. In the example application (Figure 4.18), the optional child feature f of its parent feature r is added to the or-group of the parent feature r , which is defined by the set of feature $\{g, h\}$.

In general, adding an optional child feature to a group of its parent feature is neither a refining nor a generalizing operation. Figure 4.19 illustrates this. The FD $fd1$ is changed to the FD $fd2$: The feature f is added to an xor-group solely containing the feature g . The semantics of the original FD and the resulting FD are incomparable.

4.4.13 Feature-Group-Exclusion Operations

Feature-group-exclusion operations with signature $M_{FD} \rightarrow M_{FD}$	
Parameters	Let $f \in U_N$ be a name representing a feature.
Explanation	The operation $exclGrp_f$ excludes the feature f from its group and makes f optional.
Domain	$(F, E, r, M, Or, Xor, I, X) \in dom(exclGrp_f) \Leftrightarrow f \in F \setminus \{r\} \wedge \exists (a, G) \in Or \cup Xor: f \in G$
Application	$exclGrp_f \begin{matrix} (F, E, r, M, Or, Xor, I, X) \\ \downarrow \\ (F, E, r, M, Or', Xor', I, X) \end{matrix} \text{ where } Or' = \{(p, G \setminus \{f\}) \mid (p, G) \in Or\} \setminus \{(p, \emptyset) \mid p \in F\}$ $Xor' = \{(p, G \setminus \{f\}) \mid (p, G) \in Xor\} \setminus \{(p, \emptyset) \mid p \in F\}$
Example	

Figure 4.20: Feature-group-exclusion operations $exclGrp_f$ for all feature names $f \in U_N$.

Figure 4.20 defines the feature-group-exclusion operations. Each of the feature-group-exclusion operation $exclGrp_f$ is parametrized with a feature name $f \in U_N$. On the application of the feature-group-exclusion operation $exclGrp_f$ to an FD, the feature f is removed from the group that contains the feature and is made an optional child of its parent. The operation is applicable to an FD iff the feature f is part of a group. In the example application (Figure 4.20), the feature f is excluded from the or-group of its parent feature r and becomes an optional child feature of its parent.

Feature-group-exclusion operations are neither refining nor generalizing operations. Figure 4.19 illustrates this. The FD $fd1$ results from applying the feature-group-exclusion operation $exclGrp_f$ to the FD $fd2$. In this example, the feature f is removed from an xor-group. The semantics of each of the FDs contains a configuration that is not valid in the respective other FD.

4.4.14 Root-Rename Operations

Figure 4.21 defines the root-rename operations. Each root-rename operation $renRoot_f$ is parametrized with a feature name $f \in U_N$. The application of the root-rename operation operation $renRoot_f$ to an FD renames the root in the FD to f . The operation is applicable to an FD iff the feature f does not exist in the FD. This prevents that multiple nodes in the resulting FD are labeled with the same name. In the example application (Figure 4.21), the root r is renamed to f .

Root-rename operations are neither refining nor generalizing. The example application depicted in Figure 4.21 illustrates this. For example, $\{r, g\}$ is a configuration that is valid in the FD depicted on the left-hand side and not valid in the FD depicted on the right-

	Root-rename operations with signature $M_{FD} \rightarrow M_{FD}$
Parameters	Let $f \in U_N$ be a name representing a feature.
Explanation	The operation $rnRoot_f$ renames the root to the unused feature name f .
Domain	$(F, E, r, M, Or, Xor, I, X) \in dom(rnRoot_f) \Leftrightarrow f \notin F$
Application	$ \begin{array}{l} (F, E, r, M, Or, Xor, I, X) \\ \downarrow rnRoot_f \\ (F', E', r', M', Or', Xor', I', X') \text{ where } F' = (F \setminus \{r\}) \cup \{f\} \\ E' = \{(a, b) \in E \mid a \neq r\} \cup \{(f, b) \mid (r, b) \in E\} \\ r' = f \\ M' = \{(a, b) \in M \mid a \neq r\} \cup \{(f, b) \mid (r, b) \in M\} \\ Or' = \{(p, G) \in Or \mid p \neq r\} \cup \{(f, G) \mid (r, G) \in Or\} \\ Xor' = \{(p, G) \in Xor \mid p \neq r\} \cup \{(f, G) \mid (r, G) \in Xor\} \\ I' = \{(a, b) \in I \mid a, b \neq r\} \cup \{(f, b) \mid (r, b) \in I\} \cup \{(a, f) \mid (a, r) \in I\} \\ X' = \{(a, b) \in X \mid a, b \neq r\} \cup \{(f, b) \mid (r, b) \in X\} \cup \{(a, f) \mid (a, r) \in X\} \end{array} $
Example	

 Figure 4.21: Root-rename operations $rnRoot_f$ for all feature names $f \in U_N$.

hand side. Similarly, $\{f, g\}$ is valid in the FD depicted on the right-hand side and not valid in the FD depicted on the left-hand side.

4.5 Feature Diagram Modeling Language

The FD modeling language is defined as $\mathcal{L}_{FD} = (M_{FD}, Sem_{FD}, \llbracket \cdot \rrbracket^{FD})$ where M_{FD} is the set of all FDs (defined in Section 4.1), the semantic domain is defined as $Sem_{FD} \stackrel{\text{def}}{=} \{C \subseteq U_N \mid C \text{ is finite}\}$ the set of all possible feature configurations (defined in Section 4.2), and $\llbracket \cdot \rrbracket^{FD}$ maps each FD fd to the set $\llbracket fd \rrbracket^{FD}$ of all configurations that are valid in fd (defined in Section 4.2). We define the change operation suite O_{FD} for the FD modeling language as the set of all FD change operations, as defined in the previous sections. The change operation suite O_{FD} is complete. A simple algorithm for computing a change sequence to transform an FD fd to another FD fd' operates as follows:

1. Start with the empty sequence.
2. For each mandatory feature in fd , append an operation for making the mandatory feature optional.
3. For each implies constraint in fd , append an operation deleting the constraint.
4. For each excludes constraint in fd , append an operation deleting the constraint.

5. For each feature participating in a group in fd , append an operation for excluding the feature from its group.
6. Iteratively append leaf feature deletion operations to delete all features in fd except the root feature.
7. If the root of fd is different to the root in fd' , append a change operation for renaming the root of fd to the root in fd' .
8. Iteratively append feature addition operation to construct the feature tree of fd' .
9. For each mandatory feature of fd' , append a change operation for making the feature mandatory.
10. For each implies constraint of fd' , append a change operation for adding the implies constraint.
11. For each excludes constraint of fd' , append a change operation for adding the excludes constraint.
12. For each group of fd' , append an operation for creating an or-group that contains an arbitrary feature of the group.
13. For each feature participating in a group of fd' that has not been used for creating a group in the previous step, append a feature group addition operation for adding the feature to the group.
14. For each xor-group of fd' , append a change operation for transforming the or-group, which is defined by the xor-group's parent and the group's participants, to an xor-group.

The algorithm sketched by the fourteen steps describes (disregarding the underspecification concerning the order in which the operations are appended in each step) a function $\Delta_{FD} : M_{FD} \times M_{FD} \rightarrow O_{FD}^*$ that takes two FDs as inputs and outputs a change sequence of FD change operations. The operations appended in the second step are applicable because they make existing mandatory features optional. The constraint deletion operations appended in the third and fourth steps are applicable because they delete existing cross-tree constraints. The change operations appended in the fifth step are applicable because they remove existing group participants from their groups. After the application of the sequence obtained after appending the operations defined in the first five steps to the input FD fd , the resulting FD does not contain any mandatory features, groups, or cross-tree constraints. Thus, the application of the iteratively computed feature-deletion operations (sixth step) that delete all features except the root is possible. The seventh step changes the name of the root to the name of the root of the FD fd' . The application

of the change sequence obtained after appending the operations defined in the first seven steps to the input FD fd yields an FD without cross-tree constraints. The FD contains a single feature, which is the root feature of fd' . Applying the change sequence appended in the eighth step to this intermediate FD yields an FD that has the same feature tree as the FD fd' . All features in this FD are optional and the FD does neither contain groups nor cross-tree constraints. The change operations appended in the ninth step make all optional features mandatory that are also mandatory in fd' . The tenth and eleventh steps append change operations for adding the cross-tree constraints of fd' . The change operations appended in the twelfth step create the groups that exist in fd' . Afterwards, the change operations appended in the thirteenth step add all features to the groups that are also part of the corresponding groups in fd' . The change operations appended in the fourteenth step change all or-groups, which are xor-groups in fd' , to xor-groups. In summary, applying the computed change sequence to the FD fd yields the FD fd' . Thus, the function Δ_{FD} satisfies $\forall m, m' \in M_{FD} : m \triangleright \Delta_{FD}(m, m') = m'$. Therefore, the change operation suite O_{FD} is complete for \mathcal{L}_{FD} .

4.6 Related Work and Discussion

A semantic differencing operator for an FD modeling language based on the usual closed-world semantics is presented in [AHC⁺12]. Using the notations of this thesis, the closed-world semantics of an FD $fd = (F, E, r, M, Or, Xor, I, X)$ is defined as $\llbracket fd \rrbracket_{cw}^{FD} \stackrel{\text{def}}{=} \llbracket fd \rrbracket^{FD} \cap \wp(F)$, i.e., the set of configurations that are valid in fd and solely contain features that are used in fd . Semantic differencing with the closed-world semantics is useful when analyzing the semantic differences between two FDs in late development stages [DKMR19], such as the *application engineering* phase [BPvdL05] of feature-oriented development processes. In late development phases, all possible features of the domain of interest are usually identified and explicitly used in the FD that models the product line. Each added and each removed product should be detected and reviewed by product line engineers to ensure that the product line only permits meaningful realizations. In early development stages, however, such as the *domain engineering* phase [BPvdL05] in feature-oriented development processes, or when using agile methods for the product line development, requirements on the product line are permanently subject to change. This especially includes that new features are identified and added to the FD because the set of all possible features is not necessarily known at the beginning of the development of the product line. Then, product line developers have an *open-world* view on the product line and consider the addition of new features to FDs as refinements [DKMR19]. With the usual closed-world semantics, however, the addition of a feature usually yields an FD with a semantics that is incomparable to the semantics of the original FD.

An approach that suggests a method that can be used to decrease the computational

complexity for semantic differencing of feature models using the closed-world semantics is presented in [TBK09]. The idea of the method is to simplify the propositional logic formula used for semantic differencing based on common clauses of the formulas generated for the individual feature models, before splitting the formula into multiple formulas and performing distinct solver calls. The method could also decrease the computational complexity of semantic differencing operators using the open-world semantics.

Related work [BKL⁺16] defines the binary relations specialization, refactoring, and generalization for feature models. The approach uses the usual closed-world semantics. Tailored to our notation, a feature model is a specialization of another feature model iff the former feature model is a refinement of the latter feature model. The notions of refactoring and generalization defined in [BKL⁺16] coincide with our notions of generalization and refactoring, as defined in this thesis. The approach also defines change operations for feature models [BKL⁺16] and a method for syntactic differencing of feature models based on the change operations. The article [BKL⁺16] further presents sufficient conditions guaranteeing that a feature model is a specialization (respectively generalization, refactoring) of another feature model based on interpreting changes to feature models as changes to the sets of constraints induced by feature models.

Besides semantic differencing, there are many automated semantics-based FD analyses (*e.g.*, [TKB⁺14, BSRC10]). To the best of our knowledge, the existing analyses are based on the usual closed-world semantics. Basing the analyses on the semantics presented in this chapter often changes the perspectives of the analyses. The following discusses the differences between the perspectives using the different semantics in combination with various analyses.

There are translations to transform an FD to a propositional formula such that the satisfying interpretations of the formula represent exactly the products in the closed-world semantics of the FD [BSRC10, BTRC05, CW07, ZZM04, AHC⁺12]. As explained in Section 4.3, these translations are reusable for semantic differencing of FDs with the semantics presented in this chapter. Similarly, other translation-based techniques for reasoning about the products of an FD are reusable. Such techniques are, for example, based on constraint programming as presented in [TBD⁺08].

A feature is *dead* in an FD iff no configuration of the FD contains the feature [BSRC10]. Using the closed-world semantics, a feature is dead in an FD iff its instantiation is impossible due to contradicting constraints or it is not used by the FD. Using the semantics presented in this chapter, a feature not used in an FD is unconstrained by the FD. If a feature is used in an FD, then it is dead with the closed-world semantics iff it is dead with this chapter's semantics.

An FD is said to be *satisfiable* iff its semantics contain a configuration that is different from the empty set [BSRC10]. There are various automatic methods for checking whether an FD is satisfiable using the closed-world semantics [BSRC10, Bat05, TBD⁺08], for example, by checking whether the formula obtained from translating an FD is satisfiable. An FD is not satisfiable if it contains contradicting constraints. As the semantics

presented in this chapter assumes that the root of an FD is part of every valid configuration of the FD, an FD is satisfiable with the open-world semantics iff it is satisfiable with the closed-world semantics.

Operators to determine the number of the modeled products and the set of all possible products [BTRC05, BSRC10] reveal the size and degree of variability of the modeled product line. The results facilitate identifying required FD changes towards obtaining the intended product scope. With this chapter's semantics, when assuming an infinite universe of features, the number of valid configurations of an FD is usually infinite because every FD only constrains a finite set of features.

There are various syntactic FD composition operators [SHTB07, ACLF10, vdBGN10, vdB12, AHC⁺12]. Each composition operator takes two FDs as inputs and outputs another FD that satisfies a property with respect to the semantics of the original FDs. An intersection composition operator, for instance, composes two FDs to an FD such that the semantics of the resulting FD is equal to or a superset of the intersection of the semantics of the two original FDs [SHTB07, ACLF10, vdB12, AHC⁺12]. Existing syntactic FD composition operators are defined to obtain compounds that satisfy properties with respect to the closed-world semantics. Determining the relationship between the open-world semantics of the resulting FD and the semantics of the input FDs is interesting future work.

The FD synthesis problem [CW07] is concerned with translating a propositional formula, over variables representing features, to an FD such that the products of the FD (elements of the closed-world semantics) represent exactly the satisfying interpretations of the formula. The translation constructs an FD that respects all constraints between features modeled by the formula. As every element of an FD's closed-world semantics is also valid in the FD, existing methods solving the FD synthesis problem are well-reusable when using the open-world semantics.

Chapter 5

Sequence Diagrams

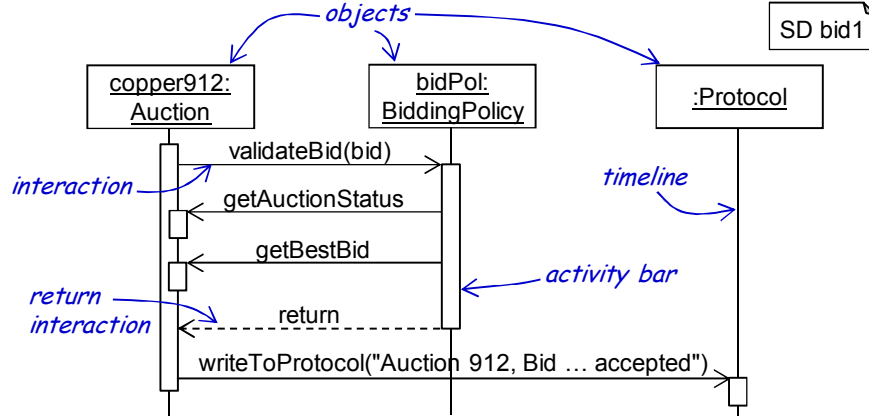
A sequence diagram (SD) represents an exemplary and possibly incomplete system run of a software system. This section uses the semantically relevant subset of the UML/P SD variant [Sch12, Rum16, Rum17], which is a simplified SD variant that is especially suited for requirements specification, modeling of tests, and system documentation.

An SD models the interactions between objects during possible system runs from a black-box viewpoint. It abstracts from the internal states of the individual objects. As SDs represent exemplary system run excerpts, the interactions between objects modeled by an SD may appear in a different order, multiple times, or even not at all in a real system run [Rum16]. The system run may also exhibit additional interactions in addition to the interactions modeled in the SD. Therefore, UML/P SDs are not suited for modeling the complete behavior of a system. They are primarily suited for requirements specification, test definition, and system documentation. We refer to [Sch12, Rum16, Rum17] for an introduction to the practical use of UML/P SDs in software development.

Figure 5.1 depicts the graphical representation of the UML/P SD `bid1` inspired by a similar example from [Rum16]. The objects in an SD are organized next to each other in a row. The objects in an SD have unique names. Names can be enriched with typing information. For example, the SD `bid1` contains an object with label `copper912:Auction`. The name `copper912` is the object's actual name and `Auction` describes the object's type. Similarly, the SD contains an object labeled `:Protocol`. This represents an anonymous object of type `Protocol`.

Each object has a timeline, which is represented by a vertical line starting at the object. The timeline of an object represents the progress of time for the object during system runs. Timelines abstract from real time, *i.e.*, the distances between interactions on a timeline do not faithfully represent the amount of real time that passes between the interactions. Therefore, the timeline only represents the chronological order of interactions, ordered from the top towards the bottom.

Activity bars are represented by white boxes that are placed on timelines or other activity bars. They indicate when objects are active during a system run. Recursive method calls are represented by activity bars laced on other activity bars. Activity bars may be omitted and are semantically irrelevant for the semantics used in this section. They are primarily used to increase the understandability for developers.

Figure 5.1: The SD `bid1` inspired by a similar SD from [Rum16].

Interactions between objects are modeled with horizontal, directed, and labeled arrows starting at the timeline of an object and ending at the timeline of an object. The label of an arrow representing an interaction models an action triggered by the object at the arrow's starting point on the object at the arrow's ending point. An action can represent asynchronous message transfer, synchronous method calls, returns in response to synchronous message calls, and exceptions. In the concrete syntax, synchronous message calls and asynchronous message transfers are indicated via arrows with solid lines representing the interaction. Returns and exceptions are indicated with dashed lines. The type of an action is solely important to increase the understandability of the SD for developers but is semantically irrelevant.

Objects can be tagged with the three stereotypes `<<complete>>`, `<<visible>>`, and `<<initial>>` [Rum16], which further constrain the interactions of objects in system runs. The stereotypes and their meanings are described in the following sections.

UML/P SDs can also contain guards specified in the object constraint language (OCL) for a detailed description of properties that hold during a system run [Rum16]. This thesis is not concerned with OCL guards used in SDs.

As an SD models exemplary and possibly incomplete system runs, the semantics of an SD is the infinite set of all system runs that exhibit the interactions specified by the SD. As stated above, this section is concerned with the semantically relevant syntax of UML/P SDs without OCL guards. Therefore, we abstract from activity bars. For simplicity, we only use arrows with solid lines for modeling interactions. For instance, Figure 5.2 depicts a representation of the SD `bid1` (cf. Figure 5.1) that only contains its semantically relevant syntactic modeling elements.

The contributions of this chapter are twofold. First, this chapter presents a semantic differencing operator for UML/P SDs based on a variant of the semantics presented

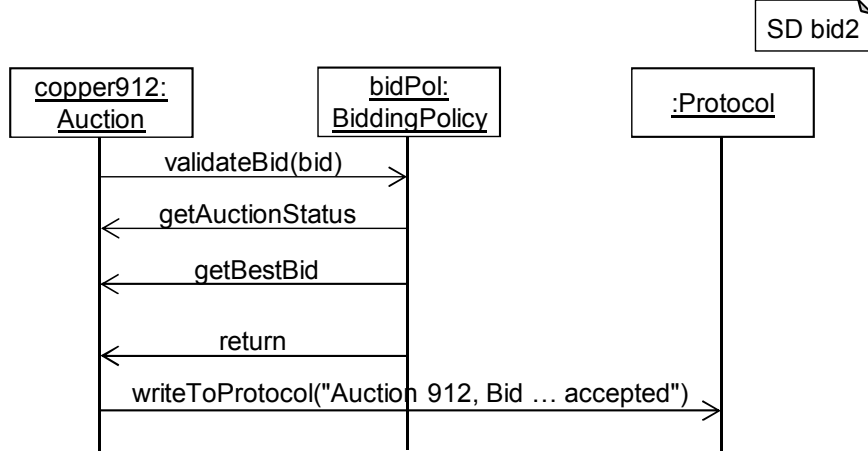


Figure 5.2: The SD `bid2` only contains the semantically relevant modeling elements of the SD `bid1` depicted in Figure 5.1.

in [Rum16]. Then, this chapter presents a complete change operation suite for SDs. For each change operation, it is examined whether the change operation is refining, generalizing, or refactoring.

In the following, Section 5.1 introduces the syntax of SDs. Afterwards, Section 5.2 defines the semantics of SDs. Based on this semantics, Section 5.3 presents the semantic differencing operator. Then, Section 5.4 introduces the SD change operations. Finally, Section 5.5 defines the SD modeling language and a complete SD change operation suite, before Section 5.6 discusses related work.

5.1 Sequence Diagram Syntax

SDs constrain the interactions between objects of a system. In the remainder, let $\mathcal{O} \subseteq U_N$ be an infinite set of objects and $\mathcal{A} \subseteq U_N$ be an infinite set of actions. The set of objects \mathcal{O} contains the possible names of objects in SDs and system runs. The set of actions \mathcal{A} contains the possible names of actions in SDs and system runs. Objects interact with each other via actions. An interaction triggered by an object $o \in \mathcal{O}$ on an object $o' \in \mathcal{O}$ with action $a \in \mathcal{A}$ is a tuple $(o, a, o') \in \mathcal{O} \times \mathcal{A} \times \mathcal{O}$. To avoid notational clutter, in the following, let $\mathcal{I} \stackrel{\text{def}}{=} \mathcal{O} \times \mathcal{A} \times \mathcal{O}$ denote the set of all possible interactions between objects. For an interaction $i = (src, act, trg) \in \mathcal{I}$, we denote by $src(i) \stackrel{\text{def}}{=} src$ the source of the interaction, $act(i) \stackrel{\text{def}}{=} act$ the action of the interaction, $trg(i) = trg$ the target of the interaction, and $obj(i) = \{src, trg\}$ the set containing the source object and the target object of the interaction. For instance,

the first interaction in the SD depicted in Figure 5.2 is represented by the interaction `(copper912:Auction, validateBid(bid), bidPol:BiddingPolicy)`.

The following defines the syntax of SDs:

Definition 5.1. *A sequence diagram is a tuple $sd = (O, O_c, O_v, O_i, A, d)$ where*

- $O \subseteq \mathcal{O}$ is a finite set of objects,
- $O_c \subseteq O$ is the subset of objects that are tagged as complete,
- $O_v \subseteq O$ is the subset of objects that are tagged as visible,
- $O_i \subseteq O$ is the subset of objects that are tagged as initial,
- $A \subseteq \mathcal{A}$ is a finite set of actions, and
- $d \in (O \times A \times O)^*$ is a finite sequence of diagram interactions.

In the following, M_{SD} denotes the set of all SDs. The set of objects O contains all objects that are used in the SD. The sequence d is the sequence of object interactions in the order as depicted from top to bottom in the graphical notation of SDs. The sequence models the interactions and the order of the interactions that must appear during a system run. The sets of objects O_c , O_v , and O_i contain the objects used in the SD that are tagged with one of the stereotypes `«complete»`, `«visible»`, and `«initial»`, respectively. Every object can be tagged with multiple of the different stereotypes. For example, the SD depicted in Figure 5.3 is mathematically defined as (O, O_c, O_v, O_i, A, d) with

- the objects $O = \{a, b, c, d\}$,
- the objects tagged as complete $O_c = \{b\}$,
- the objects tagged as visible $O_v = \{c\}$,
- the objects tagged as initial $O_i = \{d\}$,
- the actions $A = \{\text{foo}, \text{bar}\}$, and
- the sequence of diagram interactions $d = (a, \text{foo}, b), (b, \text{bar}, c), (d, \text{foo}, a)$.

Tagging an object with a stereotype further restricts the permitted interactions of the object during system runs. Generally, the interactions of objects before the occurrence of the first interaction and after the occurrence of the last interaction modeled in an SD are unconstrained. The tags restrict the possible interactions between the occurrences of the first and the last interactions modeled by the SD. If an object is tagged as complete, then the object must not interact with any object in between the first and the last interaction

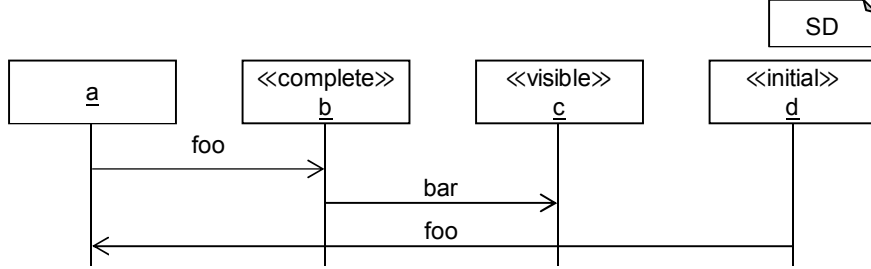


Figure 5.3: An SD that contains all SD modeling elements.

of the object differently from stated in the SD. Thus, the modeled interactions of the object are complete in between the first and the last modeled interactions of the object. If an object is tagged as visible, then the object must exactly interact with the objects of the SD as stated in the SD. It must not interact via further interactions with the objects used in the SD. The interactions of the object with other objects not used in the SD remain unconstrained. If an object is tagged as initial, then the object must not perform its interactions as stated in the SD before they are initially performed at the position at which they are stated. After the occurrence of the interaction at the stated position, performing the interaction is unconstrained. We refer to [Rum16] for further discussions on the implications of using the stereotypes. The following section precisely describes the meanings of the stereotypes through the semantic mapping for SDs.

5.2 Sequence Diagram Semantics

SDs represent possibly incomplete excerpts of system runs. Similar to an SD, a system run consists of a set of objects, a set of actions used by the objects to interact with each other, and a trace of interactions between the objects via the actions. Therefore, similar to SDs, system runs are defined as follows:

Definition 5.2. *A system run is a tuple $r = (Obj, Act, t)$ where*

- $Obj \subseteq \mathcal{O}$ *is a finite set of objects,*
- $Act \subseteq \mathcal{A}$ *is a finite set of actions,*
- $t \in (Obj \times Act \times Obj)^*$ *is a trace of interactions.*

$I(r) \stackrel{\text{def}}{=} Obj \times Act \times Obj$ *denotes the set of all possible interactions over the objects in* Obj *and the interactions in* Act *of the system run* r .

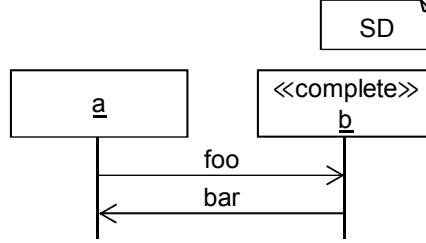


Figure 5.4: A simple SD containing two interactions and an object tagged as complete.

An SD models excerpts of system runs. A system run modeled by an SD may contain more objects and interactions than stated in the SD. Therefore, SDs are always exemplary abstractions of real system runs [Rum16]. The semantics of an SD is the set of all system runs that contain the objects and the interactions between the objects in the order stated by the SD while satisfying the constraints induced by the stereotypes.

Definition 5.3. A system run $r = (Obj, Act, t)$ is valid in an SD $sd = (O, O_c, O_v, O_i, A, d)$ where $d = d_1, \dots, d_n$ with $n \geq 0$ iff $t \in \mathcal{I}^* \& d_1 \& X_1^* \& d_2 \& X_2^* \& \dots \& X_{n-1}^* \& d_n \& \mathcal{I}^*$ where

- $X_k = \mathcal{I} \setminus (I_c \cup I_v \cup I_{i,k})$ for all $0 < k < n$,
- $I_c = \{i \in \mathcal{I} \mid obj(i) \cap O_c \neq \emptyset\}$,
- $I_v = \{i \in \mathcal{I} \mid obj(i) \cap O_v \neq \emptyset \wedge obj(i) \subseteq O\}$,
- $I_{i,k} = \{d_l \mid k < l \leq n \wedge obj(d_l) \cap O_i \neq \emptyset\}$ for all $0 < k < n$.

The semantics $\llbracket sd \rrbracket^{SD}$ of an SD sd is defined as the set of all system runs that are valid in sd .

According to Definition 5.3, the interactions of the system run are unconstrained before the occurrence of the first interaction stated in the SD. Then, the diagram interactions must occur in the specified order. In between two consecutive diagram interactions, further interactions may occur, which are the elements of the sets X_k . The stereotypes constrain the interactions that may occur between two consecutive diagram interactions. The set I_c contains the interactions that must not occur due to objects tagged as complete. The set I_v contains the interactions that must not occur due to objects tagged as visible. Similarly, the sets $I_{i,k}$ contain the actions that must not occur between the k -th and the $(k + 1)$ -th diagram interaction due to objects tagged as initial. After the occurrence of the last diagram interaction, the interactions that may occur in the system run are unconstrained, again.

For example, Figure 5.4 depicts a simple SD with two objects of which one is tagged as complete and two interactions. The system run $(\{a, b, c\}, \{\text{foo}, \text{bar}\}, t)$ with $t =$

$(c, \text{foo}, a), (a, \text{foo}, b), (c, \text{foo}, a), (b, \text{bar}, a), (c, \text{foo}, a)$ is valid in the SD. The system run $(\{a, b, c\}, \{\text{foo}, \text{bar}\}, t)$ with $t = (a, \text{foo}, b), (c, \text{foo}, b), (b, \text{bar}, a)$ is not valid in the SD because the object b is tagged as complete and the trace contains the interaction (c, foo, b) in between the two consecutive interactions of the sequence of diagram interactions. The system run $(\{a, b, c\}, \{\text{foo}, \text{bar}\}, t)$ with $t = (c, \text{foo}, b), (a, \text{foo}, b)$ is not valid in the SD because the trace of the system run does not contain the interaction (b, bar, a) after the interaction (a, foo, b) .

5.3 Semantic Differencing of Sequence Diagrams

The semantic difference from an SD to another SD is the set of all system runs that are valid in the former SD and not valid in the latter SD. This section presents a semantic differencing operator for the SD modeling language.

The following motivates the semantic differencing operator using three SDs inspired from similar SDs modeled in [ABH⁺17]: Figure 5.5 depicts the three SDs `rob1`, `rob2`, and `rob3`. The sequence diagrams model interactions between software components implementing the behavior of an autonomous service robot [ABH⁺17]. A developer initially developed the SD `rob1`. Later in the development process, the developer intends to abstract from the messages communicated with the `stateProvider` object. Thus, the developer changes the SD `rob1` to the SD `rob2`. The developer is interested in whether the changes indeed induce an abstraction in the sense that every system run of the original SD `rob1` is also a valid system run of the SD `rob2`. Thus, she uses the semantic differencing operator, which confirms that every valid system run of `rob1` is also a valid system run of `rob2`. Vice versa, the developer wants to check whether there are system runs of the new SD `rob2` that are no valid system runs of the SD `rob1`. Thus, she again uses the semantic differencing operator. The semantic differencing operator presents a valid system run of `rob2` that is not valid in `rob1`. With this information, she understands that `rob2` is not a refinement of `rob1`.

After a while, the developer recognizes that the `controller` object should not perform interactions different from the explicitly modeled interactions stated in the SD. Therefore, the developer changes the SD `rob2` to the SD `rob3` by tagging the object `controller` as complete. The developer expects the resulting SD `rob3` to be a refinement of its predecessor version `rob2`. To confirm this, she uses the semantic differencing operator, which confirms that every valid system run of `rob3` is also a valid system run of `rob2`. Vice versa, the developer wants to know whether the SD `rob3` is a refactoring of the SD `rob2`. Thus, she uses the semantic differencing operator to check whether the semantic difference from `rob2` to `rob3` is also empty. The semantic differencing operator automatically detects that the SD `rob3` is not a refactoring of `rob2` and presents a diff witness to the developer. The diff witness represents a concrete system run of the SD `rob2` that is no valid system run of the SD `rob3`.

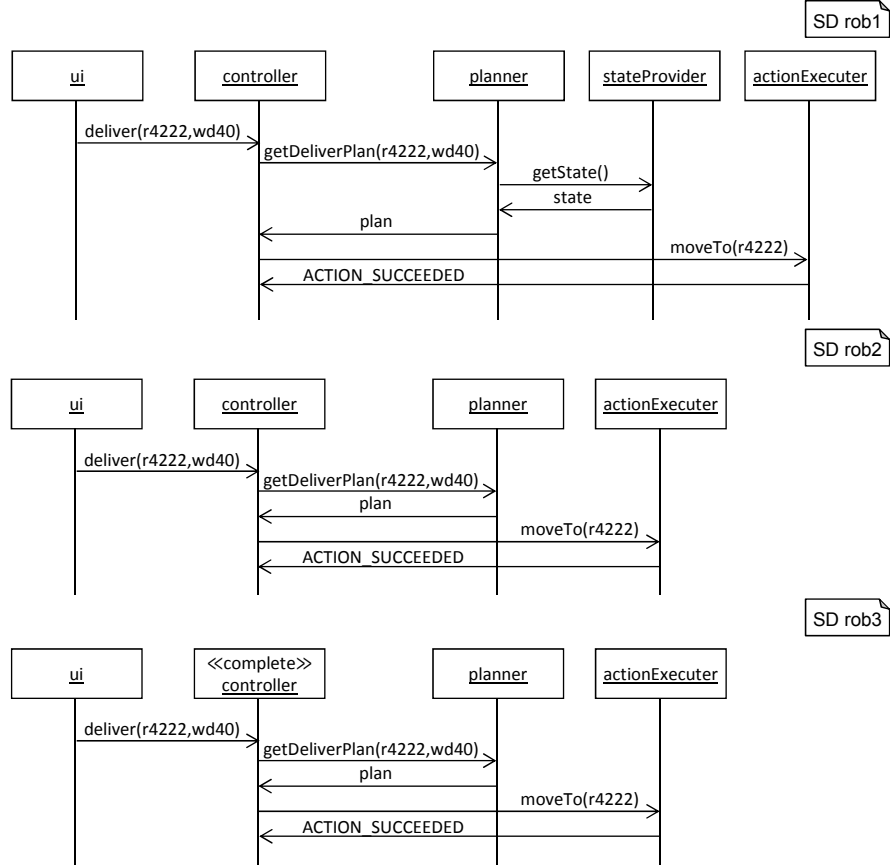


Figure 5.5: Consecutive SD versions rob1, rob2, and rob3 modeling systems runs of an autonomous service robot with pairwise different but not disjoint semantics.

The semantic differencing operator for SDs is based on a reduction to the language inclusion checking problem for finite automata. Under the assumption that the set \mathcal{I} of all possible interactions is infinite, the language \mathcal{I}^* is not regular, *i.e.*, there exists no NFA that accepts \mathcal{I}^* . As the expression \mathcal{I}^* is part of the validity condition in Definition 5.3, this directly implies that there does not exist an NFA that accepts exactly all the traces of all systems runs that are valid in an SD.

However, for determining whether there are semantic differences from an SD to another SD, it suffices to search for a diff witness in a set of all traces of all valid system runs of the former SD, where the set of all interactions used by the traces in the set is finite, *i.e.*, only finitely many different interactions need to be considered. The searched set of traces is a regular language. This ultimately enables reducing semantic differencing of SDs to language inclusion checking between NFAs.

The following Proposition 5.1 formally presents the construction of the set of traces. The proposition further states that it suffices to search this set for a trace of a diff witness. The proof of Proposition 5.1 is based on two observations: Every SD equally constrains the use of any two actions that are not used in the SD. Similarly, every SD equally constrains the interactions of any two objects not used in the SD. For semantic differencing of two SDs, this enables transforming every system run into a system run that solely uses the objects and actions of the SDs as well as one object and one action not used in both SDs. The latter system run is constructed by merging the objects not used in the SDs into a single object, redirecting the interactions accordingly, and relabeling the interactions having actions not used in the SDs. The constructed system run is contained in the semantic difference from one of the SDs to the other SD iff the original system run is contained in the semantic difference.

Proposition 5.1. *Let $sd = (O, O_c, O_v, O_i, A, d)$ and $sd' = (O', O'_c, O'_v, O'_i, A', d')$ be two sequence diagrams. Further let $o \in \mathcal{O} \setminus (O \cup O')$ be an object that is neither used by sd nor by sd' and let $a \in \mathcal{A} \setminus (A \cup A')$ be an action that is neither used by sd nor by sd' . Let $\omega \stackrel{\text{def}}{=} O \cup O' \cup \{o\}$ and $\alpha \stackrel{\text{def}}{=} A \cup A' \cup \{a\}$. The following statements are equivalent:*

1. $\llbracket sd \rrbracket^{SD} \not\subseteq \llbracket sd' \rrbracket^{SD}$.
2. There exists a system run $r = (Obj, Act, t)$ with $t \in (\omega \times \alpha \times \omega)^*$ such that $r \in \llbracket sd \rrbracket \wedge r \notin \llbracket sd' \rrbracket$.

Proof. Let sd, sd', o, a, ω , and α be given as above.

" \Leftarrow ": The existence of a system run $r = (Obj, Act, t)$ with $t \in (\omega \times \alpha \times \omega)^*$ such that $r \in \llbracket sd \rrbracket \wedge r \notin \llbracket sd' \rrbracket$ directly implies that $\llbracket sd \rrbracket^{SD} \not\subseteq \llbracket sd' \rrbracket^{SD}$.

" \Rightarrow ": Let $r = (Obj, Act, t)$ be an arbitrary system run. First, we fix our notations for this proof. In the following, we assume:

- $d = d_1, \dots, d_n$ where $n \geq 0$,
- X_1, \dots, X_{n-1} are the sets X_i for sd as defined in Definition 5.3,
- $I_c, I_v, I_{i,k}$ are the sets of interactions for sd as defined in Definition 5.3,
- $t = t_1, \dots, t_k$ where $k \geq 0$.

We define the system run $r' \stackrel{\text{def}}{=} (\omega, \alpha, t')$ where $|t'| = |t|$ with $t' = t'_1, \dots, t'_k$ and for all $0 < i \leq k$:

$$\begin{aligned} src(t'_i) &= \begin{cases} src(t_i) & , \text{ if } src(t_i) \in O_1 \cup O_2 \\ o & , \text{ otherwise.} \end{cases} \\ trg(t'_i) &= \begin{cases} trg(t_i) & , \text{ if } trg(t_i) \in O_1 \cup O_2 \\ o & , \text{ otherwise.} \end{cases} \end{aligned}$$

$$act(t'_i) = \begin{cases} act(t_i) & , \text{ if } act(t_i) \in A_1 \cup A_2 \\ a & , \text{ otherwise.} \end{cases}$$

Then, $t' \in (\omega \times \alpha \times \omega)^*$. Intuitively, the translation from r to r' leaves the objects, the actions, and the interactions between objects that are used in sd or sd' unchanged. All objects that are neither used by sd nor by sd' are merged into the object o , which is neither used by sd nor by sd' . Similarly, all actions that are neither used by sd nor by sd' are merged into the action a , which is neither used by sd nor by sd' .

In the following (at the end of this proof), we show that for the two sequence diagrams sd and sd' it holds that $r \in \llbracket sd \rrbracket^{SD}$ iff $r' \in \llbracket sd \rrbracket^{SD}$ as well as $r \in \llbracket sd' \rrbracket^{SD}$ iff $r' \in \llbracket sd' \rrbracket^{SD}$. Under the assumption that $\llbracket sd \rrbracket^{SD} \not\subseteq \llbracket sd' \rrbracket^{SD}$ holds, we can safely assume $r \in \llbracket sd \rrbracket^{SD}$ and $r \notin \llbracket sd' \rrbracket^{SD}$ because r is chosen arbitrarily. This then implies $r' \in \llbracket sd \rrbracket^{SD}$ and $r' \notin \llbracket sd' \rrbracket^{SD}$ and completes the proof because $t' \in (\omega \times \alpha \times \omega)^*$.

In the following, we show: $r \in \llbracket sd \rrbracket^{SD}$ iff $r' \in \llbracket sd \rrbracket^{SD}$. The proof for $r \in \llbracket sd' \rrbracket^{SD}$ iff $r' \in \llbracket sd' \rrbracket^{SD}$ is analogous. We first show the following two properties:

(P1) $t'_i = d_j$ iff $t_i = d_j$ for all $0 < i \leq k$ and $0 < j \leq n$.

(P2) $t'_i \in X_j$ iff $t_i \in X_j$ for all $0 < i \leq k$ and $0 < j < n$.

Proof of (P1):

Let $i, j \in \mathbb{N}$ with $0 < i \leq k$ and $0 < j \leq n$.

" \Rightarrow ": Assume $t'_i = d_j$. Then, $src(t'_i) \neq o$, $trg(t'_i) \neq o$, and $act(t'_i) \neq a$ because $d_j \in (O_1 \cup O_2) \times (A_1 \cup A_2) \times (O_1 \cup O_2)$, $o \notin O_1 \cup O_2$, and $a \notin A_1 \cup A_2$. Therefore, by definition of t' , it holds that $src(t'_i) = src(t_i)$, $trg(t'_i) = trg(t_i)$, and $act(t'_i) = act(t_i)$. From this and as $t'_i = d_j$, we obtain $t_i = d_j$.

" \Leftarrow ": Assume $t_i = d_j$. As $d_j \in (O_1 \cup O_2) \times (A_1 \cup A_2) \times (O_1 \cup O_2)$, by definition of t' , we have $t'_i = t_i$. Thus, $t'_i = t_i = d_j$.

Proof of (P2):

Let $i, j \in \mathbb{N}$ with $0 < i \leq k$ and $0 < j < n$.

" \Rightarrow ": Assume $t'_i \in X_j$. Then by definition of X_j , it holds that $t'_i \notin I_c$ and $t'_i \notin I_v$ and $t'_i \notin I_{i,j}$. We now show that this implies $t_i \notin I_c$ and $t_i \notin I_v$ and $t_i \notin I_{i,j}$, which implies that $t_i \in X_j$:

" $t_i \notin I_c$ ": As $t'_i \notin I_c$, it holds that $obj(t'_i) \cap O_c = \emptyset$. Suppose towards a contradiction $t_i \in I_c$. Then, $obj(t_i) \cap O_c \neq \emptyset$. Thus, $src(t_i) \in O_c$ or $trg(t_i) \in O_c$. If $src(t_i) \in O_c$, then $src(t_i) = src(t'_i)$ because $O_c \subseteq O$, which contradicts $obj(t'_i) \cap O_c = \emptyset$. If $trg(t_i) \in O_c$, then $trg(t_i) = trg(t'_i)$ because $O_c \subseteq O$, which contradicts $obj(t'_i) \cap O_c = \emptyset$. Thus, $t_i \notin I_c$.

" $t_i \notin I_v$ ": As $t'_i \notin I_v$, it holds that $obj(t'_i) \cap O_v = \emptyset \vee obj(t'_i) \not\subseteq O$. Suppose towards a contradiction that $t_i \in I_v$. Then, $obj(t_i) \cap O_v \neq \emptyset \wedge obj(t_i) \subseteq O$. Then, $src(t_i) \in O_v$ and $src(t_i), trg(t_i) \in O$ or $trg(t_i) \in O_v$ and $src(t_i), trg(t_i) \in O$. If $src(t_i) \in O_v$ and $src(t_i), trg(t_i) \in O$, then $src(t'_i) = src(t_i) \in O_v \subseteq O$ and $trg(t'_i) = trg(t_i) \in O$. This

contradicts that $t'_i \notin I_v$. If $trg(t_i) \in O_v$ and $src(t_i), trg(t_i) \in O$, then $trg(t'_i) = trg(t_i) \in O_v \subseteq O$ and $src(t'_i) = src(t_i) \in O$. This contradicts that $t'_i \notin I_v$. Thus, $t_i \notin I_v$.

" $t_i \notin I_{i,j}$ ": As $t'_i \notin I_{i,j}$, it holds that $t'_i \neq d_l$ for all $j < l \leq n$ with $obj(d_l) \cap O_i \neq \emptyset$. Suppose towards a contradiction that $t_i \in I_{i,j}$. Then, $t_i = d_l$ for some $j < l \leq n$ with $obj(d_l) \cap O_i \neq \emptyset$. As $t_i = d_l$ and $d_l \in O \times A \times O$, it holds that $t'_i = t_i$. This directly contradicts that $t'_i \notin I_{i,j}$. Thus, $t_i \notin I_{i,j}$.

" \Leftarrow ": Assume $t_i \in X_j$. Then, by definition of X_j , it holds that $t_i \notin I_c$ and $t_i \notin I_v$ and $t_i \notin I_{i,j}$. We now show that this implies $t'_i \notin I_c$ and $t'_i \notin I_v$ and $t'_i \notin I_{i,j}$, which implies that $t'_i \in X_j$.

" $t'_i \notin I_c$ ": As $t_i \notin I_c$, it holds that $obj(t_i) \cap O_c = \emptyset$. Suppose towards a contradiction that $t'_i \in I_c$. Then, $obj(t'_i) \cap O_c \neq \emptyset$. Thus, $src(t'_i) \in O_c$ or $trg(t'_i) \in O_c$. If $src(t'_i) \in O_c$, then $src(t'_i) \in O$ as well as $src(t'_i) \neq o$ and, therefore, $src(t_i) = src(t'_i) \in O_c$, which contradicts that $t_i \notin I_c$. If $trg(t'_i) \in O_c$, then $trg(t'_i) \in O$ as well as $trg(t'_i) \neq o$ and, therefore, $trg(t_i) = trg(t'_i) \in O_c$, which contradicts that $t_i \notin I_c$. Thus, $t'_i \notin I_c$.

" $t'_i \notin I_v$ ": As $t_i \notin I_v$, it holds that $obj(t_i) \cap O_v = \emptyset \vee obj(t_i) \not\subseteq O$. Suppose towards a contradiction that $t'_i \in I_v$. Then, $obj(t'_i) \cap O_v \neq \emptyset \wedge obj(t'_i) \subseteq O$. Then, $src(t'_i) \in O_v$ and $src(t'_i), trg(t'_i) \in O$ or $trg(t'_i) \in O_v$ and $src(t'_i), trg(t'_i) \in O$. If $src(t'_i) \in O_v$ and $src(t'_i), trg(t'_i) \in O$, then it especially holds that $src(t'_i) \neq o$ and $trg(t'_i) \neq o$. Therefore, $src(t_i) = src(t'_i) \in O_v \subseteq O$ and $trg(t_i) = trg(t'_i) \in O$. This contradicts that $t_i \notin I_v$. If $trg(t'_i) \in O_v$ and $src(t'_i), trg(t'_i) \in O$, then it especially holds that $src(t'_i) \neq o$ and $trg(t'_i) \neq o$. Therefore, $src(t_i) = src(t'_i) \in O$ and $trg(t_i) = trg(t'_i) \in O_v \subseteq O$. This contradicts that $t_i \notin I_v$. Thus, $t'_i \notin I_v$.

" $t'_i \notin I_{i,j}$ ": As $t_i \notin I_{i,j}$, it holds that $t_i \neq d_l$ for all $j < l \leq n$ with $obj(d_l) \cap O_i \neq \emptyset$. Suppose towards a contradiction that $t'_i \in I_{i,j}$. Then, $t'_i = d_l$ for some $j < l \leq n$ with $obj(d_l) \cap O_i \neq \emptyset$. As $t'_i = d_l$ and $d_l \in O \times A \times O$, it especially holds that $src(t'_i) \neq o$ and $trg(t'_i) \neq o$ and $act(t'_i) \neq a$. Thus, $t'_i = t_i$. This contradicts that $t_i \notin I_{i,j}$. Thus, $t'_i \notin I_{i,j}$.

This concludes the proof of (P2).

Combination of (P1) with (P2):

In the following, we combine (P1) and (P2) to show that $r \in \llbracket sd \rrbracket^{SD}$ iff $r' \in \llbracket sd \rrbracket^{SD}$:

" \Rightarrow ": Assume $r \in \llbracket sd \rrbracket^{SD}$. Then, $t \in \mathcal{I}^* d_1 X_1^* \dots X_{n-1}^* d_n \mathcal{I}^*$. Therefore, there exist $n+1$ sub-words $u_0, \dots, u_n \in \mathcal{I}^*$ of t such that $t = u_0 d_1 u_1 d_2 \dots d_n u_n$ and $u_j \in X_j^*$ for all $0 < j < n$. From the construction of t' , property (P1), and as $t \in \mathcal{I}^* d_1 X_1^* \dots X_{n-1}^* d_n \mathcal{I}^*$, it follows that there exist $n+1$ sub-words $v_0, \dots, v_n \in \mathcal{I}^*$ of t' such that $t' = v_0 d_1 v_1 d_2 \dots d_n v_n$ and $|u_j| = |v_j|$ for all $0 \leq j \leq n$. From the above, property (P2), and as $u_j \in X_j^*$ for all $0 < j < n$, it follows that $v_j \in X_j^*$ for all $0 < j < n$. This implies that $t' \in \mathcal{I}^* d_1 X_1^* \dots X_{n-1}^* d_n \mathcal{I}^*$, which again implies that $r' \in \llbracket sd \rrbracket$.

" \Leftarrow ": Assume $r' \in \llbracket sd \rrbracket^{SD}$. Then, $t' \in \mathcal{I}^* d_1 X_1^* \dots X_{n-1}^* d_n \mathcal{I}^*$. Therefore, there exist $n+1$ sub-words u_0, \dots, u_n of t' such that $t' = u_0 d_1 u_1 d_2 \dots d_n u_n$ and $u_j \in X_j^*$ for all $0 < j < n$. From the construction of t' , property (P1), and as $t' \in \mathcal{I}^* d_1 X_1^* \dots X_{n-1}^* d_n \mathcal{I}^*$, it follows that

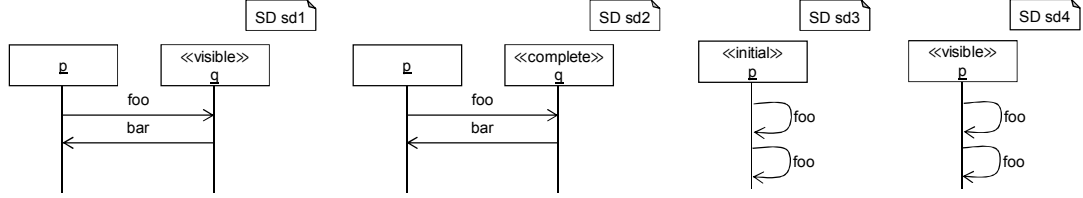


Figure 5.6: Four SDs sd1, sd2, sd3, and sd4 where sd1 is not a refinement of sd2 and sd3 is not a refinement of sd4.

there exist $n+1$ sub-words $v_0, \dots, v_n \in \mathcal{I}^*$ of t such that $t = v_0 d_1 v_1 d_2 \dots d_n v_n$ and $|u_j| = |v_j|$ for all $0 \leq j \leq n$. From the above, property (P2), and as $u_j \in X_j^*$ for all $0 < j < n$, it follows that $v_j \in X_j^*$ for all $0 < j < n$. This implies that $t \in \mathcal{I}^* d_1 X_1^* \dots X_{n-1}^* d_n \mathcal{I}^*$, which again implies that $r \in \llbracket sd \rrbracket^{SD}$. \square

Proposition 5.1 guarantees that searching the set of system runs with a trace over all the objects used in the SDs, one object that is not used in the SDs, the actions used in the SDs, and one action not used in the SDs for a diff witness suffices to determine whether the semantic difference from one of the SDs to the other SD is empty. Stated differently, an SD is a refinement of another SD iff all system runs with a trace over all the objects used in the SDs, one object that is not used in the SDs, the actions used in the SDs, and one action not used in the SDs of the former SD are also system runs of the latter SD.

Considering one object and one action that are not used by both of the SDs is necessary because of the constraints induced by objects that are tagged with stereotypes. For instance, Figure 5.6 depicts the four SDs sd1, sd2, sd3, and sd4. The SD sd1 is not a refinement of the SD sd2. For example, $w_1 \stackrel{\text{def}}{=} (\{p, q, o\}, \{foo, bar\}, t)$ with $t = (p, foo, q), (o, foo, q), (q, bar, p) \in \delta(sd1, sd2)$ is a system run contained in the semantic difference from sd1 to sd2. However, the set of all systems runs contained in the semantics of sd1 with a trace over all the objects and actions used in the SDs is a subset of the set of all system runs contained in the semantics of sd2 with a trace over all the objects and actions used in the SDs. In contrast, additionally considering an object not used in both SDs reveals the diff witness w_1 . Thus, considering an object not used in the SDs is necessary.

The SDs sd3 and sd4 illustrate why considering an action not used in the SDs is necessary. The SD sd3 is not a refinement of the SD sd4. For example, $w_2 \stackrel{\text{def}}{=} (\{p, q\}, \{foo, a\}, t)$ with $t = (p, foo, q), (p, a, p), (p, foo, p) \in \delta(sd3, sd4)$ is a system run contained in the semantic difference from sd3 to sd4. However, the set of all systems runs contained in the semantics of sd1 with a trace over all the objects and actions used in the SDs is a subset of the set of all system runs contained in the semantics of sd2 with a trace over all the objects and actions used in the SDs. It even holds that the set

of all systems runs contained in the semantics of $sd1$ with a trace over all the objects and actions used in the SDs and one object not used in the SDs is a subset of the set of all system runs contained in the semantics of $sd2$ with a trace over all the objects and actions used in the SDs and one object not used in the SDs. However, additionally considering an action not used in both SDs reveals the diff witness w_2 .

As the set of interactions used by an SD is always finite, the above enables a reduction from semantic SD differencing to language inclusion checking for NFAs. For each of the two SDs, it is possible to construct an NFA that exactly accepts all traces of all valid system runs with a trace over all the objects used in the SDs, one object that is not used in the SDs, the actions used in the SDs, and one action not used in the SDs. Proposition 5.1 guarantees that the semantic difference from one of the SDs to the other SD is not empty iff the language recognized by one of the NFAs contains a word that is not accepted by the other NFA. The following explicates the construction:

Let $sd = (O, O_c, O_v, O_i, A, d)$ and $sd' = (O', O'_c, O'_v, O'_i, A', d')$ be two sequence diagrams. Further, let $o \in \mathcal{O} \setminus (O \cup O')$ be an object that is neither used by sd nor by sd' and let $a \in \mathcal{A} \setminus (A \cup A')$ be an action that is neither used by sd nor by sd' . We define $\omega \stackrel{\text{def}}{=} O \cup O' \cup \{o\}$ and $\alpha \stackrel{\text{def}}{=} A \cup A' \cup \{a\}$. The set ω contains all objects used in sd or sd' as well as the object o that is neither used in sd nor in sd' . Analogously, the set α contains all actions used in sd or sd' as well as the action a that is neither used in sd nor in sd' . For the SD sd , we define the NFA A as $A \stackrel{\text{def}}{=} (S, \Sigma, \delta, i, F)$ where

- $S = \{s \in \mathbb{N} \mid s \leq |d|\}$,
- $\Sigma = \omega \times \alpha \times \omega$,
- $\delta = (\bigcup_{0 \leq k \leq |d|} L_k) \cup P_k$ with
 - $L_0 = \{(0, l, 0) \mid l \in \Sigma\}$,
 - $L_{|d|} = \{(|d|, l, |d|) \mid l \in \Sigma\}$,
 - $L_k = \{(k, l, k) \mid l \in \Sigma \cap X_k\}$ where X_k depends on sd and is defined as in Definition 5.3 for all $0 < k < |d|$,
 - $P_k = \{(k, d_{k+1}, k+1) \mid 0 \leq k < |d|\}$,
- $i = 0$, and
- $F = \{|d|\}$.

As d is a finite sequence and the set Σ is finite, the NFA A is well-defined. By construction, it directly follows that the language recognized by A is given by the regular set $\mathcal{L}_*(A) = (\mathcal{I}^* d_1 X_1^* d_2 x_2^* \dots X_{n-1}^* d_n \mathcal{I}^*) \cap (\omega \times \alpha \times \omega)^*$ where X_k depends on sd and is defined as in Definition 5.3 for all $0 < k < n$. From this and the definition of SD semantics (cf. Definition 5.3), it follows that a system run $r = (Obj, Act, t)$ with $t \in (\omega \times \alpha \times \omega)^*$ is valid in sd iff the trace t is accepted by A . Analogously, we can construct an NFA

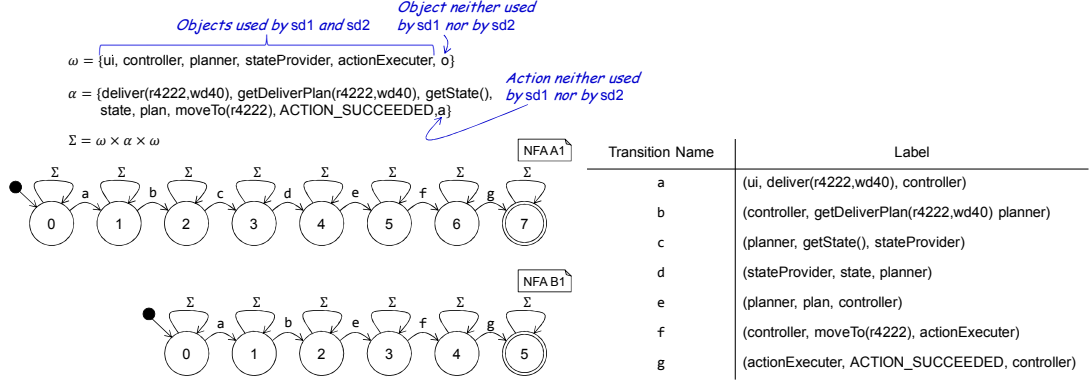


Figure 5.7: The automata constructed for semantic differencing of the SDs rob1 and rob2 depicted in Figure 5.5.

B such that a system run $r = (Obj, Act, t)$ with $t \in (\omega \times \alpha \times \omega)^*$ is valid in sd' iff the trace t is accepted by B . Then, every word accepted by A that is not accepted by B is the trace of a system run that is valid in A and not valid in B , i.e., a trace of an element of the semantic difference from sd to sd' . With Proposition 5.1, the semantic difference from sd to sd' is not empty iff there exists a word accepted by A that is not accepted by B . For checking whether there exists such a word, we can use standard constructions [RS59, HMU06, BK08] from automata theory to construct an automaton \overline{B} that accepts the complementary language of B (i.e., $\mathcal{L}_*(\overline{B}) = (\omega \times \alpha \times \omega)^* \setminus \mathcal{L}_*(B)$), construct an automaton C that recognizes the intersection of the languages recognized by A and \overline{B} (i.e., $\mathcal{L}_*(C) = \mathcal{L}_*(A) \cap \mathcal{L}_*(\overline{B})$), and check whether the language recognized by C is not empty. Each word recognized by C is then a trace of a diff witness in the semantic difference from sd to sd' .

The following illustrates the construction of the NFAs using the example SDs depicted in Figure 5.5. Figure 5.7 depicts the NFAs constructed for semantic differencing of rob1 and rob2. The NFA A1 is constructed from the SD rob1. The NFA B1 is constructed from the SD rob2. The alphabet of both NFAs is the set of all interactions with an action used in at least one of the two SDs and an action not used by both SDs between the objects used in at least one of the SDs and an object not used in both SDs. The transitions represent the interactions modeled in the SDs. The SD does not contain any object that is tagged with a stereotype. Therefore, for each interaction in the alphabet and for each state, the NFA contains a looping transition from the state to itself labeled with the interaction.

Figure 5.8 depicts the NFAs constructed for semantic differencing of the SDs rob2 and rob3 (cf. Figure 5.5). The NFA A2 is constructed from the SD rob2. The NFA B2 is constructed from the SD rob3. The alphabets of the NFAs are different from the

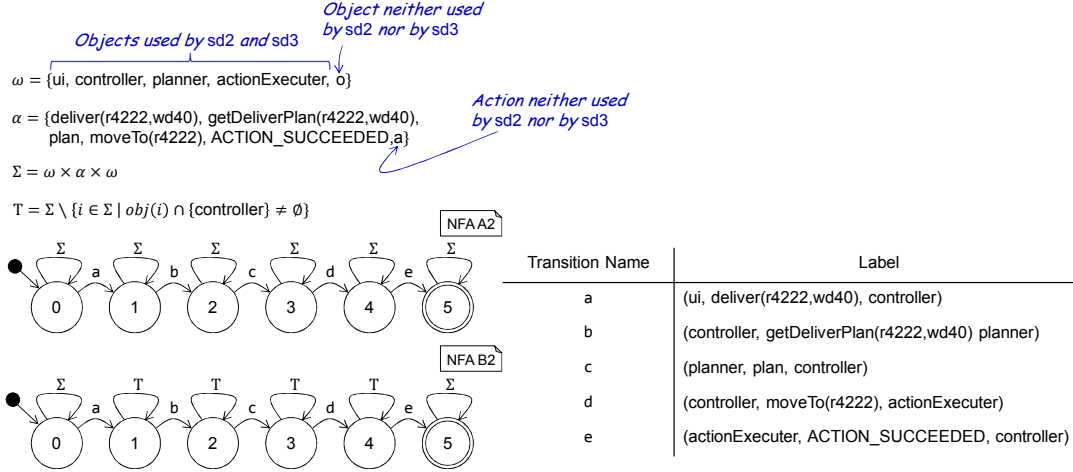


Figure 5.8: The automata constructed for semantic differencing of the SDs rob2 and rob3 depicted in Figure 5.5.

alphabets of the NFAs A1 and B1 because the SDs rob2 and rob3 do not use the object stateProvider, which is only used by rob1. The object controller is tagged as complete in rob3. Therefore, the NFA B2 does not contain a looping transition from each state to itself for each element of the alphabet.

Semantic Differencing Implementation and Experiments

We implemented the semantic differencing operator for SDs to perform experimental evaluations. The implementation is written in Java and uses the automaton language inclusion checking tool RABIT¹ [ACC⁺11] for NFA language inclusion checking. We choose the tool RABIT as it is easy to use and we already used it for the implementation of the semantic differencing operator for TSPAs (cf. Section 3.3). Alternatively, implementations using other tools for NFA language inclusion checking are possible. The implementation of the semantic differencing operator takes two SDs as inputs. It translates the SDs into NFAs according to the translation described above and outputs the NFAs in the BA format, which is the input format of RABIT. Subsequently, the implementation uses the tool RABIT for language inclusion checking of the NFAs. In case language inclusion does not hold, RABIT provides a counterexample, which is translated to a diff witness.

We performed experimental evaluations with the seven example SDs presented in Appendix C. Figure 5.9 summarizes the sizes of the SDs in terms of the numbers of states and transitions of the NFAs resulting from translating the SDs. As the number of

¹<http://languageinclusion.org/>

CHAPTER 5 SEQUENCE DIAGRAMS

Compared SDs sd_1, sd_2	#States $nfa(sd_1)$	#Trans. $nfa(sd_1)$	#States $nfa(sd_2)$	#Trans. $nfa(sd_2)$
sd1, sd1	4	147	4	147
sd1, sd2	4	147	4	144
sd2, sd2	4	144	4	144
rob1, rob1	8	2311	8	2311
rob1, rob2	8	2311	6	1733
rob1, rob3	8	2311	6	1381
rob1, rob4	8	2599	8	2599
rob1, rob5	8	2887	9	3248
rob2, rob2	6	905	6	905
rob2, rob3	6	905	6	689
rob2, rob4	6	1949	8	2599
rob2, rob5	6	2165	9	3248
rob3, rob3	6	689	6	689
rob3, rob4	6	1553	8	2599
rob3, rob5	6	1725	9	3248
rob4, rob4	8	2311	8	2311
rob4, rob5	8	2887	9	3248
rob5, rob5	9	2924	9	2924

Figure 5.9: The number of states and transitions of the NFAs constructed from the SDs for semantic differencing.

Action name	Abbreviation	Object name	Abbreviation
deliver(r4222, wd40)	d	ui	ui
getDeliverPlan(r4222, wd40)	gdp	controller	c
getState()	gs	planner	pl
state	s	stateProvider	sp
plan	p	actionExecuter	ae
moveTo(r4222)	mt	newObject1	no1
abortAction()	aa		
ACTION_SUCCEEDED	as		
ACTION_FAILED	af		
ACTION_ABORTED	ab		

Figure 5.10: Abbreviations of action and object names used for describing the diff witnesses presented in Figure 5.11.

states and transitions of the NFAs resulting from the translations depend on both input SDs, Figure 5.9 depicts the sizes of the NFAs for all pairs of input SDs. We executed the semantic differencing operator for all pairs of example SDs that are thematically related. All experiments were executed on a laptop computer with an Intel Core i7-8650U CPU @ 1.90GHz processor, 16GB RAM, and a Samsung PM981 512GB SSD hard drive using Windows 10 and Java 1.8.0_192.

Figure 5.11 summarizes the computation times of the semantic differencing operator and the computed diff witnesses for the input pairs. If no witness exists, *i.e.*, refinements holds, then the corresponding cell in the table contains the special symbol $-$. The object and action names of the SDs rob1, rob2, rob3, rob4, and rob5 are abbreviated in the presented witnesses to save space. Figure 5.10 depicts the re-

lation between the abbreviations and the original action and object names. For instance, the action name `plan` is abbreviated as `p` and the object name `controller` is abbreviated as `c`. The semantic differencing operator took 139ms to detect that the SD `sd1` is a refinement of itself (cf. Figure 5.11). Computing the diff witness $(ui, d, c), (c, gdp, pl), (pl, p, c), (c, mt, ae), (ae, as, c)$ contained in the semantic difference from `rob3` to `rob1` took 991ms.

For the examples, the computation times range from 139ms to 1976ms. We conclude that the implementation handles the example SDs sufficiently quick. However, the example SDs are relatively small in terms of the number of objects and interactions of the SDs. The results are not generalizable to large SDs and real world examples, especially because language inclusion checking between NFAs is, in general, computationally hard.

5.4 Sequence Diagram Change Operations

This section presents change operations for SDs. The change operations are used to define a complete change operation suite for the SD modeling language. Some of the change operations are neither refining nor generalizing. Although these change operations are irrelevant for developers to constructively refine or generalize models, the change operations are necessary to obtain a complete change operation suite. The completeness of the change operation suite is required for the model repair framework presented in Chapter 7 and the framework's instantiation presented in Chapter 8. If a change operation is refining or generalizing, then it is possible to incorporate performance improvements into algorithms that compute solutions (cf. Section 7.5) for special model repair problems as introduced in Section 8.1.

Figure 5.12 overviews the change operations and indicates whether they are refining or generalizing. Applying an object-addition operation (cf. No. 1) to an SD can produce an SD that induces further constraints caused by objects that are tagged as visible. Therefore, object-addition operations are refining. Vice versa, object-deletion operations (cf. No. 2) are generalizing. Adding an object to the sets of objects tagged as complete, visible, and initial, respectively, strengthens the constraints induced by the SD. Therefore, change operations for adding objects to the sets of objects marked as complete, visible, and initial (cf. No. 3, 5, 7) are refining. Vice versa, deleting an object from the sets of objects tagged as complete, visible, and initial, respectively, weakens the constraints induced by the SD. Therefore, change operations for deleting objects from the sets of objects marked as complete, visible, and initial (cf. No. 4, 6, 8) are generalizing. Adding an action to the set of actions (cf. No. 9) of an SD neither changes the diagram interactions nor the objects of the SD. Therefore, action-addition operations are refactoring. Only unused actions can be deleted from an SD. Therefore, action-deletion operations (cf. No. 10) are refactoring. Adding an interaction (cf. No. 11) at a specific position to the sequence of diagram interactions of an SD yields an SD that has an

Difference	Time	Diff witness
$\delta(\text{sd1}, \text{sd1})$	139ms	-
$\delta(\text{sd1}, \text{sd2})$	279ms	(a, foo, b), (b, bar, a), (a, baz, b)
$\delta(\text{sd2}, \text{sd1})$	339ms	(b, bar, a), (b, foo, a), (a, baz, b)
$\delta(\text{sd2}, \text{sd2})$	164ms	-
$\delta(\text{rob1}, \text{rob1})$	338ms	-
$\delta(\text{rob1}, \text{rob2})$	284ms	-
$\delta(\text{rob1}, \text{rob3})$	977ms	(ui, d, c), (no1, as, c), (c, gdp, pl), (pl, gs, sp), (sp, s, pl), (pl, p, c), (c, mt, ae), (ae, as, c)
$\delta(\text{rob1}, \text{rob4})$	1456ms	(ui, d, c), (c, gdp, pl), (pl, gs, sp), (sp, s, pl), (pl, p, c), (c, mt, ae), (ae, as, c)
$\delta(\text{rob1}, \text{rob5})$	1563ms	(ui, d, c), (c, gdp, pl), (pl, gs, sp), (sp, s, pl), (pl, p, c), (c, mt, ae), (ae, as, c)
$\delta(\text{rob2}, \text{rob1})$	1044ms	(ui, d, c), (c, gdp, pl), (pl, p, c), (c, mt, ae), (ae, as, c)
$\delta(\text{rob2}, \text{rob2})$	171ms	-
$\delta(\text{rob2}, \text{rob3})$	526ms	(ui, d, c), (no1, as, c), (c, gdp, pl), (pl, p, c), (c, mt, ae), (ae, as, c)
$\delta(\text{rob2}, \text{rob4})$	1116ms	(ui, d, c), (c, gdp, pl), (pl, p, c), (c, mt, ae), (ae, as, c)
$\delta(\text{rob2}, \text{rob5})$	1420ms	(ui, d, c), (c, gdp, pl), (pl, p, c), (c, mt, ae), (ae, as, c)
$\delta(\text{rob3}, \text{rob1})$	991ms	(ui, d, c), (c, gdp, pl), (pl, p, c), (c, mt, ae), (ae, as, c)
$\delta(\text{rob3}, \text{rob2})$	223ms	-
$\delta(\text{rob3}, \text{rob3})$	209ms	-
$\delta(\text{rob3}, \text{rob4})$	1359ms	(ui, d, c), (c, gdp, pl), (pl, p, c), (c, mt, ae), (ae, as, c)
$\delta(\text{rob3}, \text{rob5})$	1611ms	(ui, d, c), (c, gdp, pl), (pl, p, c), (c, mt, ae), (ae, as, c)
$\delta(\text{rob4}, \text{rob1})$	1734ms	(ui, d, c), (c, gdp, pl), (pl, gs, sp), (sp, s, pl), (pl, p, c), (c, mt, ae), (ae, af, c)
$\delta(\text{rob4}, \text{rob2})$	1469ms	(ui, d, c), (c, gdp, pl), (pl, gs, sp), (sp, s, pl), (pl, p, c), (c, mt, ae), (ae, af, c)
$\delta(\text{rob4}, \text{rob3})$	1455ms	(ui, d, c), (no1, af, c), (c, gdp, pl), (pl, gs, sp), (sp, s, pl), (pl, p, c), (c, mt, ae), (ae, af, c)
$\delta(\text{rob4}, \text{rob4})$	329ms	-
$\delta(\text{rob4}, \text{rob5})$	1942ms	(ui, d, c), (c, gdp, pl), (pl, gs, sp), (sp, s, pl), (pl, p, c), (c, mt, ae), (ae, af, c)
$\delta(\text{rob5}, \text{rob1})$	1976ms	(ui, d, c), (c, gdp, pl), (pl, gs, sp), (sp, s, pl), (pl, p, c), (ui, aa, ae), (c, mt, ae), (ae, ab, c)
$\delta(\text{rob5}, \text{rob2})$	1737ms	(ui, d, c), (c, gdp, pl), (pl, gs, sp), (sp, s, pl), (pl, p, c), (ui, aa, ae), (c, mt, ae), (ae, ab, c)
$\delta(\text{rob5}, \text{rob3})$	1570ms	(ui, d, c), (no1, ab, c), (c, gdp, pl), (pl, gs, sp), (sp, s, pl), (pl, p, c), (ui, aa, ae), (c, mt, ae), (ae, ab, c)
$\delta(\text{rob5}, \text{rob4})$	1935ms	(ui, d, c), (c, gdp, pl), (pl, gs, sp), (sp, s, pl), (pl, p, c), (ui, aa, ae), (c, mt, ae), (ae, ab, c)
$\delta(\text{rob5}, \text{rob5})$	373ms	-

Figure 5.11: The time needed by the semantic differencing operator for semantic differencing of the pairs of example SDs and the traces of the computed diff witnesses.

No.	Operation	Ref.	Gen.
1.	Adding an object	✓	✗
2.	Deleting an object	✗	✓
3.	Adding an object to the set of objects tagged as complete	✓	✗
4.	Deleting an object from the set of objects tagged as complete	✗	✓
5.	Adding an object to the set of objects tagged as visible	✓	✗
6.	Deleting an object from the set of objects tagged as complete	✗	✓
7.	Adding an object to the set of objects tagged as initial	✓	✗
8.	Deleting an object from the set of objects tagged as initial	✗	✓
9.	Adding an action	✓	✓
10.	Deleting an action	✓	✓
11.	Adding an interaction at a specific position	✗	✗
12.	Removing an interaction from a specific position	✗	✗

Figure 5.12: Sequence diagram change operation properties.

incomparable semantics to the semantics of the original SD. Vice versa, operations for removing an interaction at a specific position (cf. No. 12) from the sequence of diagram interactions are neither refining nor generalizing.

5.4.1 Object-Addition Operations

Object-addition operations with signature $M_{SD} \rightarrow M_{SD}$	
Parameters	Let $o \in \mathcal{O}$ be a name representing an object.
Explanation	The operation $addO_o$ adds the object o .
Domain	$(O, O_c, O_v, O_i, A, d) \in dom(addO_o) \Leftrightarrow o \notin O$
Application	$ \begin{array}{c} (O, O_c, O_v, O_i, A, d) \\ \downarrow \\ (O', O_c, O_v, O_i, A, d) \text{ where } O' = O \cup \{o\} \end{array} $
Example	

 Figure 5.13: Object-addition operations $addO_o$ for all object names $o \in \mathcal{O}$.

Figure 5.13 defines the object-addition operations. Each object-addition operation $addO_o$ is parametrized with an object name $o \in \mathcal{O}$. On the application of the object-addition operation $addO_o$ to an SD, the object o is added to the set of objects of the SD. The operation is applicable to an SD iff the object o does not exist in the SD. Thus, an

already defined object cannot be added. In the example application (cf. Figure 5.13), the object o is added to the SD depicted on the left-hand side.

Adding an object to an SD may change the sets of interactions that are prohibited due to objects tagged as visible. Therefore, object-addition operations are, in general, not generalizing. For example, Figure 5.14 depicts the two SDs $sd1$ and $sd2$ and a trace of a system run that is valid in $sd1$ and not valid in $sd2$. The SD $sd2$ is obtained from applying the object-addition operation $addO_o$ to $sd1$. The system run $(\{a, v, o\}, \{foo, baz, bar\}, t)$ where $t = (a, foo, v), (v, baz, o), (v, bar, a)$ is valid in $sd1$ and not valid in $sd2$. On the other hand, applying an object-addition operation to an SD causes that every interaction prohibited in the SD is also prohibited in the SD that results from the application. Therefore, object-addition operations are refining.

Proposition 5.2. *Let $addO_o$ be an object-addition operation and let $sd \in M_{SD}$ be an SD such that $addO_o$ is applicable to sd . Then, $\llbracket addO_o(sd) \rrbracket^{SD} \subseteq \llbracket sd \rrbracket^{SD}$.*

Proof. Let $o \in \mathcal{O}$, $addO_o$, and $sd = (O, O_c, O_v, O_i, A, d)$ where $d = d_1, \dots, d_n$ with $n \geq 0$ be given as above. Let $sd' = addO_o(sd) = (O', O'_c, O'_v, O'_i, A', d')$.

(1) By definition of $addO_o$, it holds that $O'_c = O_c$. Thus, $\{i \in \mathcal{I} \mid obj(i) \cap O_c \neq \emptyset\} = \{i \in \mathcal{I} \mid obj(i) \cap O'_c \neq \emptyset\}$, i.e., the set of interactions prohibited due to objects tagged as complete in sd is equal to the set of interactions prohibited due to objects tagged as complete in sd' .

(2) By definition of $addO_o$, it holds that $O \subset O'$ and $O'_v = O_v$. Therefore, $\{i \in \mathcal{I} \mid obj(i) \cap O_v \neq \emptyset \wedge obj(i) \subseteq O\} \subset \{i \in \mathcal{I} \mid obj(i) \cap O'_v \neq \emptyset \wedge obj(i) \subseteq O'\}$, i.e., the set of interactions prohibited due to objects tagged as visible in sd is a subset of the set of interactions prohibited due to objects tagged as visible in sd' .

(3) By definition of $addO_o$, it holds that $d = d'$ and $O'_i = O_i$. Therefore, $\{d_l \mid k < l \leq n \wedge obj(d_l) \cap O_i \neq \emptyset\} = \{d_l \mid k < l \leq n \wedge obj(d_l) \cap O'_i \neq \emptyset\}$ for all $0 < k < n$, i.e., the set of interactions prohibited due to objects tagged as initial in sd is equal to the set of interactions prohibited due to objects tagged as initial in sd' .

Using the definition of SD semantics and (1)-(3), we conclude that $\llbracket sd' \rrbracket^{SD} \subseteq \llbracket sd \rrbracket^{SD}$. \square

5.4.2 Object-Deletion Operations

Figure 5.15 defines the object-deletion operations. Each object-deletion operation $delO_o$ is parametrized with an object name $o \in \mathcal{O}$. On the application of the object-deletion operation $delO_o$ to an SD, the object o is removed from the SD. The operation is applicable to an SD iff the object o exists in the SD, no interaction in the SD uses the object, and the object is neither tagged as complete, nor as initial, nor as visible. In the example application depicted in Figure 5.15, the object o is deleted from the SD depicted on the left-hand side.

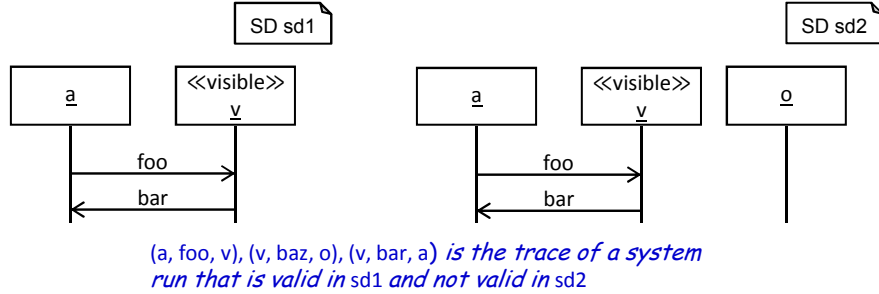


Figure 5.14: Object-addition operations are not generalizing.

Object-deletion operations with signature $M_{SD} \rightarrow M_{SD}$	
Parameters	Let $o \in \mathcal{O}$ be a name representing an object.
Explanation	The operation $delO_o$ deletes the object o .
Domain	$(O, O_c, O_v, O_i, A, d) \in dom(delO_o) \Leftrightarrow o \in O \wedge o \notin O_c \cup O_v \cup O_i \wedge \forall i \in \underline{d}: o \notin obj(d.i)$
Application	$ \begin{array}{c} (O, O_c, O_v, O_i, A, d) \\ \downarrow \\ (O', O_c, \bar{O}_v, O_i, A, d) \text{ where } O' = O \setminus \{o\} \end{array} $
Example	

 Figure 5.15: Object-deletion operations $delO_o$ for all object names $o \in \mathcal{O}$.

Deleting an object from an SD may change the sets of interactions that are prohibited due to objects tagged as visible. Therefore, object deletion operations are, in general, not refining. This is illustrated in Figure 5.14, which depicts the two SDs sd1 and sd2. It further depicts a trace of a system run that is valid in sd1 and not valid in sd2. The SD sd1 can be obtained by applying the object-deletion operation $delO_o$ to sd2. Object-deletion operations are generalizing as their application does not yield an SD that induces constraints on system runs that are not already induced by the original.

Proposition 5.3. *Let $delO_o$ be an object-deletion operation and let $sd \in M_{SD}$ be an SD such that $delO_o$ is applicable to sd . Then, $\llbracket sd \rrbracket^{SD} \subseteq \llbracket delO_o(sd) \rrbracket^{SD}$.*

Proof. In the following, we show that for each object-deletion operation, there exists an object-addition operation that is an inverse of the object-deletion operation. As object-addition operations are refining, this implies with Proposition 2.2 that object-deletion operations are generalizing.

Let $o \in \mathcal{O}$ be an object name. Let $sd = (O, O_c, O_v, O_i, A, d)$ be an SD such that

$sd \in \text{dom}(\text{del}O_o)$, i.e., the object-deletion operation $\text{del}O_o$ is applicable to sd . Let $sd' = (O', O_c, O_v, O_i, A, d)$ where $O' = O \setminus \{o\}$. By definition of $\text{del}O_o$, it holds that $\text{del}O_o(sd) = sd'$. As $o \notin O'$, it holds that $sd' \in \text{dom}(\text{add}O_o)$. We can derive $\text{add}O_o(sd') = ((O \setminus \{o\}) \cup \{o\}, O_c, O_v, O_i, A, d) = (O, O_c, O_v, O_i, A, d) = sd$. We can conclude that $\text{add}O_o$ is an inverse of $\text{del}O_o$. \square

5.4.3 Tag-Object-as-Complete Operations

Tag-object-as-complete operations with signature $M_{SD} \rightarrow M_{SD}$	
Parameters	Let $o \in \mathcal{O}$ be a name representing an object.
Explanation	The operation $\text{add}OC_o$ adds the object o to the set of objects tagged as complete.
Domain	$(O, O_c, O_v, O_i, A, d) \in \text{dom}(\text{add}OC_o) \Leftrightarrow o \in \mathcal{O} \wedge o \notin O_c$
Application	$\begin{array}{c} (O, O_c, O_v, O_i, A, d) \\ \downarrow \\ (O, O'_c, O_v, O_i, A, d) \text{ where } O'_c = O_c \cup \{o\} \end{array}$
Example	

Figure 5.16: Tag-object-as-complete operations $\text{add}OC_o$ for all object names $o \in \mathcal{O}$.

Figure 5.16 defines the tag-object-as-complete operations. Each tag-object-as-complete operation $\text{add}OC_o$ is parametrized with an object name $o \in \mathcal{O}$. On the application of the tag-object-as-complete operation $\text{add}OC_o$ to an SD, the object o is tagged as complete in the SD. The operation is applicable to an SD iff the object o exists in the SD and the object is not already tagged as complete. In the example application (cf. Figure 5.16), the object o of the SD depicted on the left-hand side is tagged as complete.

Tagging an object as complete may introduce additional constraints on interactions caused by the object that is tagged as complete. Thus, tag-object-as-complete operations are not generalizing. For instance, the system run $(\{a, b, o\}, \{foo, bar, baz\}, t)$ where $t = (a, foo, b), (a, foo, o), (b, bar, o), (b, baz, a)$ is valid in the original SD depicted on the left-hand side in the example of Figure 5.16 and is not valid in the SD depicted on the right-hand side. Tagging an object as complete in an SD never relaxes constraints induced by the SD. Therefore, tag-object-as-complete operations are refining.

Proposition 5.4. *Let $\text{add}OC_o$ be a tag-object-as-complete operation and let $sd \in M_{SD}$ be an SD such that $\text{add}OC_o$ is applicable to sd . Then, $\llbracket \text{add}OC_o(sd) \rrbracket^{SD} \subseteq \llbracket sd \rrbracket^{SD}$.*

Proof. Let $o \in \mathcal{O}$, $\text{add}OC_o$, and $sd = (O, O_c, O_v, O_i, A, d)$ where $d = d_1, \dots, d_n$ with $n \geq 0$ be given as above. Let $sd' = \text{add}OC_o(sd) = (O', O'_c, O'_v, O'_i, A', d')$.

(1) By definition of $addOC_o$, it holds that $O_c \subseteq O'_c$. Thus, $\{i \in \mathcal{I} \mid obj(i) \cap O_c \neq \emptyset\} \subseteq \{i \in \mathcal{I} \mid obj(i) \cap O'_c \neq \emptyset\}$, i.e., the set of interactions prohibited due to objects tagged as complete in sd is a subset of the interactions prohibited due to objects tagged as complete in sd' .

(2) By definition of $addOC_o$, it holds that $O = O'$ and $O'_v = O_v$. Therefore, $\{i \in \mathcal{I} \mid obj(i) \cap O_v \neq \emptyset \wedge obj(i) \subseteq O\} = \{i \in \mathcal{I} \mid obj(i) \cap O'_v \neq \emptyset \wedge obj(i) \subseteq O'\}$, i.e., the set of interactions prohibited due to objects tagged as visible in sd is equal to the set of interactions prohibited due to objects tagged as visible in sd' .

(3) By definition of $addOC_o$, it holds that $d = d'$ and $O'_i = O_i$. Therefore, $\{d_l \mid k < l \leq n \wedge obj(d_l) \cap O_i \neq \emptyset\} = \{d_l \mid k < l \leq n \wedge obj(d_l) \cap O'_i \neq \emptyset\}$ for all $0 < k < n$, i.e., the set of interactions prohibited due to objects tagged as initial in sd is equal to the set of interactions prohibited due to objects tagged as initial in sd' .

Using the definition of SD semantics and (1)-(3), we conclude $\llbracket sd' \rrbracket^{SD} \subseteq \llbracket sd \rrbracket^{SD}$. \square

5.4.4 Untag-Object-as-Complete Operations

Untag-object-as-complete operations with signature $M_{SD} \rightarrow M_{SD}$	
Parameters	Let $o \in \mathcal{O}$ be a name representing an object.
Explanation	The operation $delOC_o$ removes the object o from the set of objects tagged as complete.
Domain	$(O, O_c, O_v, O_i, A, d) \in dom(delOC_o) \Leftrightarrow o \in O_c$
Application	$ \begin{array}{c} (O, O_c, O_v, O_i, A, d) \\ \downarrow \\ (O, O'_c, O_v, O_i, A, d) \text{ where } O'_c = O_c \setminus \{o\} \end{array} $
Example	

Figure 5.17: Untag-object-as-complete operations $delOC_o$ for all object names $o \in \mathcal{O}$.

Figure 5.17 defines the untag-object-as-complete operations. Each untag-object-as-complete operation $delOC_o$ is parametrized with a name $o \in \mathcal{O}$ representing an object name. On the application of the untag-object-as-complete operation $delOC_o$ to an SD, the object o is removed from the set of objects tagged as complete in the SD. The operation is applicable to an SD iff the object o exists and is tagged as complete in the SD. In the example application (cf. Figure 5.17), the object o of the SD depicted on the left-hand side is untagged as complete.

Each untag-object-as-complete operation is the inverse of a tag-object-as-complete operation. Untagging an object as complete relaxes the constraints induced by the SD.

Thus, untag-object-as-complete operations are not refining. For instance, Figure 5.17 depicts two SDs where the SD on the right-hand side is obtained by applying an untag-object-as-complete operation to the SD depicted on the left-hand side. The system run $(\{a, b, o\}, \{foo, bar, baz\}, t)$ where $t = (a, foo, b), (a, foo, o), (b, bar, o), (b, baz, a)$ is not valid in the original SD depicted on the left-hand side and is valid in the SD depicted on the right-hand side. Untagging an object as complete never strengthens the constraints induced by the SD. Therefore, untag-object-as-complete operations are generalizing.

Proposition 5.5. *Let $delOC_o$ be an untag-object-as-complete operation and let $sd \in M_{SD}$ be an SD such that $delOC_o$ is applicable to sd . Then, $\llbracket sd \rrbracket^{SD} \subseteq \llbracket delOC_o(sd) \rrbracket^{SD}$.*

Proof. We show that for each untag-object-as-complete operation, there exists a tag-object-as-complete operation that is an inverse of the untag-object-as-complete operation. As tag-object-as-complete operations are refining, this implies with Proposition 2.2 that untag-object-as-complete operations are generalizing.

Let $o \in \mathcal{O}$ be an object name. Let $sd = (O, O_c, O_v, O_i, A, d)$ be an SD such that $sd \in dom(delOC_o)$, i.e., the untag-object-as-complete operation $delOC_o$ is applicable to sd . Then, $o \in O_c$. Let $sd' = (O, O'_c, O_v, O_i, A, d)$ where $O'_c = O_c \setminus \{o\}$. By definition of $delOC_o$, it holds that $delOC_o(sd) = sd'$. As $o \notin O'_c$, it holds that $sd' \in dom(addOC_o)$. We can derive $addOC_o(sd') = (O, (O_c \setminus \{o\}) \cup \{o\}, O_v, O_i, A, d) = (O, O_c, O_v, O_i, A, d) = sd$. We can conclude that $addOC_o$ is an inverse of $delOC_o$. \square

5.4.5 Tag-Object-as-Visible Operations

Tag-object-as-visible operations with signature $M_{SD} \rightarrow M_{SD}$	
Parameters	Let $o \in \mathcal{O}$ be a name representing an object.
Explanation	The operation $addOV_o$ adds the object o to the set of objects tagged as visible.
Domain	$(O, O_c, O_v, O_i, A, d) \in dom(addOV_o) \Leftrightarrow o \in \mathcal{O} \wedge o \notin O_v$
Application	$ \begin{array}{c} (O, O_c, O_v, O_i, A, d) \\ \downarrow \\ (O, O_c, O'_v, O_i, A, d) \text{ where } O'_v = O_v \cup \{o\} \end{array} $
Example	

Figure 5.18: Tag-object-as-visible operations $addOV_o$ for all object names $o \in \mathcal{O}$.

Figure 5.18 defines the tag-object-as-visible operations. Each tag-object-as-visible operation $addOV_o$ is parametrized with an object name $o \in \mathcal{O}$. On the application of

the tag-object-as-visible operation $addOV_o$ to an SD, the object o is tagged as visible in the SD. The operation is applicable to an SD iff the object o exists in the SD and the object is not already tagged as visible. In the example application (cf. Figure 5.18), the object o of the SD depicted on the left-hand side is tagged as visible.

Tagging an object as visible may introduce additional constraints on interactions caused by the object that is tagged as visible. Thus, tag-object-as-visible operations are not generalizing. For instance, the system run $(\{a, b, o\}, \{foo, bar, baz\}, t)$ where $t = (a, foo, b), (a, foo, o), (b, bar, o), (b, baz, a)$ is valid in the SD depicted on the left-hand side in the example of Figure 5.18 and is not valid in the resulting SD depicted on the right-hand side of Figure 5.18. Tagging an object as visible in an SD never relaxes constraints induced by the SD. Therefore, tag-object-as-visible operations are refining.

Proposition 5.6. *Let $addOV_o$ be a tag-object-as-visible operation and let $sd \in M_{SD}$ be an SD such that $addOV_o$ is applicable to sd . Then, $\llbracket addOV_o(sd) \rrbracket^{SD} \subseteq \llbracket sd \rrbracket^{SD}$.*

Proof. Let $o \in \mathcal{O}$, $addOV_o$, and $sd = (O, O_c, O_v, O_i, A, d)$ where $d = d_1, \dots, d_n$ with $n \geq 0$ be given as above. Let $sd' = addOV_o(sd) = (O', O'_c, O'_v, O'_i, A', d')$.

(1) By definition of $addOV_o$, it holds that $O_c = O'_c$. Thus, $\{i \in \mathcal{I} \mid obj(i) \cap O_c \neq \emptyset\} = \{i \in \mathcal{I} \mid obj(i) \cap O'_c \neq \emptyset\}$, i.e., the set of interactions prohibited due to objects tagged as complete in sd is equal to the interactions prohibited due to objects tagged as complete in sd' .

(2) By definition of $addOV_o$, it holds that $O = O'$ and $O_v \subseteq O'_v$. Therefore, $\{i \in \mathcal{I} \mid obj(i) \cap O_v \neq \emptyset \wedge obj(i) \subseteq O\} \subseteq \{i \in \mathcal{I} \mid obj(i) \cap O'_v \neq \emptyset \wedge obj(i) \subseteq O'\}$, i.e., the set of interactions prohibited due to objects tagged as visible in sd is a subset of the set of interactions prohibited due to objects tagged as visible in sd' .

(3) By definition of $addOV_o$, it holds that $d = d'$ and $O'_i = O_i$. Therefore, $\{d_l \mid k < l \leq n \wedge obj(d_l) \cap O_i \neq \emptyset\} = \{d_l \mid k < l \leq n \wedge obj(d_l) \cap O'_i \neq \emptyset\}$ for all $0 < k < n$, i.e., the set of interactions prohibited due to objects tagged as initial in sd is equal to the set of interactions prohibited due to objects tagged as visible in sd' .

Using the definition of SD semantics and (1)-(3), we conclude that $\llbracket sd' \rrbracket^{SD} \subseteq \llbracket sd \rrbracket^{SD}$. \square

5.4.6 Untag-Object-as-Visible Operations

Figure 5.19 defines the untag-object-as-visible operations. Each untag-object-as-visible operation $delOV_o$ is parametrized with an object name $o \in \mathcal{O}$. On the application of the untag-object-as-visible operation $delOV_o$ to an SD, the object o is removed from the set of objects tagged as visible in the SD. The operation is applicable to an SD iff the object o exists and is tagged as visible in the SD. In the example application (cf. Figure 5.17), the object o of the SD depicted on the left-hand side is untagged as visible.

Each untag-object-as-visible operation is the inverse of a tag-object-as-visible operation. Untagging an object as visible in an SD relaxes the constraints induced by

Untag-object-as-visible operations with signature $M_{SD} \rightarrow M_{SD}$	
Parameters	Let $o \in \mathcal{O}$ be a name representing an object.
Explanation	The operation $delOV_o$ removes the object o from the set of objects tagged as visible.
Domain	$(O, O_c, O_v, O_i, A, d) \in \text{dom}(delOV_o) \Leftrightarrow o \in O_v$
Application	$ \begin{array}{c} (O, O_c, O_v, O_i, A, d) \\ \downarrow \\ (O, O_c, O'_v, O_i, A, d) \text{ where } O'_v = O_v \setminus \{o\} \end{array} $
Example	

 Figure 5.19: Untag-object-as-visible operations $delOV_o$ for all object names $o \in \mathcal{O}$.

the SD. Thus, untag-object-as-visible operations are not refining. For instance, Figure 5.19 depicts two SDs where the SD on the right-hand side is obtained by applying an untag-object-as-visible operation to the SD on the left-hand side. The system run $(\{a, b, o\}, \{foo, bar, baz\}, t)$ where $t = (a, foo, b), (a, foo, o), (b, bar, o), (b, baz, a)$ is not valid in the SD depicted on the left-hand side and is valid in the SD depicted on the right-hand side. Untagging an object as visible in an SD never strengthens the constraints induced by the SD. Therefore, untag-object-as-visible operations are generalizing.

Proposition 5.7. *Let $delOV_o$ be an untag-object-as-visible operation and let $sd \in M_{SD}$ be an SD such that $delOV_o$ is applicable to sd . Then, $\llbracket sd \rrbracket^{SD} \subseteq \llbracket delOV_o(sd) \rrbracket^{SD}$.*

Proof. In the following, we show that for each untag-object-as-visible operation, there exists a tag-object-as-visible operation that is an inverse of the untag-object-as-visible operation. As tag-object-as-visible operations are refining, this implies with Proposition 2.2 that untag-object-as-visible operations are generalizing.

Let $o \in \mathcal{O}$ be an object name. Let $sd = (O, O_c, O_v, O_i, A, d)$ be an SD such that $sd \in \text{dom}(delOV_o)$, i.e., the untag-object-as-visible operation $delOV_o$ is applicable to sd . Then, $o \in O_v$. Let $sd' = (O, O_c, O'_v, O_i, A, d)$ where $O'_v = O_v \setminus \{o\}$. By definition of $delOV_o$, it holds that $delOV_o(sd) = sd'$. As $o \notin O'_v$, it holds that $sd' \in \text{dom}(addOV_o)$. We can derive $addOV_o(sd') = (O, O_c, (O_v \setminus \{o\}) \cup \{o\}, O_i, A, d) = (O, O_c, O_v, O_i, A, d) = sd$. We can conclude that $addOV_o$ is an inverse of $delOV_o$. \square

5.4.7 Tag-Object-as-Initial Operations

Figure 5.20 defines the tag-object-as-initial operations. Each tag-object-as-initial operation $addOI_o$ is parametrized with an object name $o \in \mathcal{O}$. On the application of the

Tag-object-as-initial operations with signature $M_{SD} \rightarrow M_{SD}$	
Parameters	Let $o \in \mathcal{O}$ be a name representing an object.
Explanation	The operation $addOI_o$ adds the object o to the set of objects tagged as initial.
Domain	$(O, O_c, O_v, O_i, A, d) \in \text{dom}(addOI_o) \Leftrightarrow o \in O \wedge o \notin O_i$
Application	$ \begin{array}{c} (O, O_c, O_v, O_i, A, d) \\ \downarrow \\ (O, O_c, O_v, O'_i, A, d) \text{ where } O'_i = O_i \cup \{o\} \end{array} $
Example	

 Figure 5.20: Tag-object-as-initial operations $addOI_o$ for all object names $o \in \mathcal{O}$.

tag-object-as-initial operation $addOI_o$ to an SD, the object o is tagged as initial in the SD. The operation is applicable to an SD iff the object o exists in the SD and the object is not already tagged as initial. In the example application depicted in Figure 5.20, the object o of the SD depicted on the left-hand side is tagged as initial.

Tagging an object as initial may introduce additional constraints on interactions caused by the object that is tagged as initial. Thus, tag-object-as-initial operations are not generalizing. For instance, Figure 5.20 depicts two SDs where the SD on the right-hand side is obtained by applying a tag-object-as-initial operation to the SD on the left-hand side. In this example, the system run $(\{a, b, o\}, \{foo, bar, baz\}, t)$ where $t = (a, foo, b), (o, foo, b), (b, bar, o), (b, baz, a), (o, foo, b)$ is valid in the original SD depicted on the left-hand side and is not valid in the resulting SD depicted on the right-hand side. Tagging an object as initial in an SD never relaxes constraints induced by the SD. Therefore, tag-object-as-initial operations are refining.

Proposition 5.8. *Let $addOI_o$ be a tag-object-as-initial operation and let $sd \in M_{SD}$ be an SD such that $addOI_o$ is applicable to sd . Then, $\llbracket addOI_o(sd) \rrbracket^{SD} \subseteq \llbracket sd \rrbracket^{SD}$.*

Proof. Let $o \in \mathcal{O}$, $addOI_o$, and $sd = (O, O_c, O_v, O_i, A, d)$ where $d = d_1, \dots, d_n$ with $n \geq 0$ be given as above. Let $sd' = addOI_o(sd) = (O', O'_c, O'_v, O'_i, A', d')$.

(1) By definition of $addOI_o$, it holds that $O_c = O'_c$. Thus, $\{i \in \mathcal{I} \mid obj(i) \cap O_c \neq \emptyset\} = \{i \in \mathcal{I} \mid obj(i) \cap O'_c \neq \emptyset\}$, i.e., the set of interactions prohibited due to objects tagged as complete in sd is equal to the interactions prohibited due to objects tagged as complete in sd' .

(2) By definition of $addOI_o$, it holds that $O = O'$ and $O_v = O'_v$. Therefore, $\{i \in \mathcal{I} \mid obj(i) \cap O_v \neq \emptyset \wedge obj(i) \subseteq O\} = \{i \in \mathcal{I} \mid obj(i) \cap O'_v \neq \emptyset \wedge obj(i) \subseteq O'\}$, i.e., the

set of interactions prohibited due to objects tagged as visible in sd is equal to the set of interactions prohibited due to objects tagged as visible for sd' .

(3) By definition of $addOI_o$, it holds that $d = d'$ and $O_i \subseteq O'_i$. Therefore, $\{d_l \mid k < l \leq n \wedge obj(d_l) \cap O_i \neq \emptyset\} \subseteq \{d_l \mid k < l \leq n \wedge obj(d_l) \cap O'_i \neq \emptyset\}$ for all $0 < k < n$, i.e., the set of interactions prohibited due to objects tagged as initial in sd is a subset of the set of interactions prohibited due to objects tagged as initial in sd' .

Using the definition of SD semantics and (1)-(3), we conclude that $\llbracket sd' \rrbracket^{SD} \subseteq \llbracket sd \rrbracket^{SD}$. \square

5.4.8 Untag-Object-as-Initial Operations

Untag-object-as-initial operations with signature $M_{SD} \rightarrow M_{SD}$	
Parameters	Let $o \in \mathcal{O}$ be a name representing an object.
Explanation	The operation $delOI_o$ removes the object o from the set of objects tagged as initial.
Domain	$(O, O_c, O_v, O_i, A, d) \in dom(delOI_o) \Leftrightarrow o \in O_i$
Application	$delOI_o \begin{array}{c} (O, O_c, O_v, O_i, A, d) \\ \downarrow \\ (O, O_c, O_v, O'_i, A, d) \text{ where } O'_i = O_i \setminus \{o\} \end{array}$
Example	

Figure 5.21: Untag-object-as-initial operations $delOI_o$ for all object names $o \in \mathcal{O}$.

Figure 5.21 defines the untag-object-as-initial operations. Each untag-object-as-initial operation $delOI_o$ is parametrized with an object name $o \in \mathcal{O}$. On the application of the untag-object-as-initial operation $delOI_o$ to an SD, the object o is removed from the set of objects tagged as initial in the SD. The operation is applicable to an SD iff the object o exists and is tagged as initial in the SD. In the example application (cf. Figure 5.21), the object o of the SD depicted on the left-hand side is untagged as initial.

Each untag-object-as-initial operation is the inverse of a tag-object-as-initial operation. Untagging an object as initial in an SD relaxes the constraints induced by the SD. Thus, untag-object-as-initial operations are not refining. For instance, Figure 5.21 depicts two SDs. The SD on the right-hand side is obtained by applying an untag-object-as-initial operation to the SD on the left-hand side. The system run $(\{a, b, o\}, \{foo, bar, baz\}, t)$ where $t = (a, foo, b), (o, foo, b), (b, bar, o), (b, baz, a), (o, foo, b)$ is not valid in the original SD depicted on the left-hand side, but is valid in the resulting SD depicted on the right-hand side. Untagging an object as initial never strengthens the constraints induced by

the SD. Therefore, untag-object-as-initial operations are generalizing.

Proposition 5.9. *Let $delOI_o$ be an untag-object-as-initial operation and let $sd \in M_{SD}$ be an SD such that $delOI_o$ is applicable to sd . Then, $\llbracket sd \rrbracket^{SD} \subseteq \llbracket delOI_o(sd) \rrbracket^{SD}$.*

Proof. In the following, we show that for each untag-object-as-initial operation, there exists a tag-object-as-initial operation that is an inverse of the untag-object-as-initial operation. As tag-object-as-initial operations are refining, this implies with Proposition 2.2 that untag-object-as-initial operations are generalizing.

Let $o \in \mathcal{O}$ be an object name. Let $sd = (O, O_c, O_v, O_i, A, d)$ be an SD such that $sd \in dom(delOI_o)$, i.e., the untag-object-as-initial operation $delOI_o$ is applicable to sd . Then, $o \in O_I$. Let $sd' = (O, O_c, O_v, O'_i, A, d)$ where $O'_i = O_i \setminus \{o\}$. By definition of $delOI_o$, it holds that $delOI_o(sd) = sd'$. As $o \notin O'_i$, it holds that $sd' \in dom(addOI_o)$. We can derive $addOI_o(sd') = (O, O_c, O_v, (O_i \setminus \{o\}) \cup \{o\}, A, d) = (O, O_c, O_v, O_i, A, d) = sd$. We can conclude that $addOI_o$ is an inverse of $delOI_o$. \square

5.4.9 Action-Addition Operations

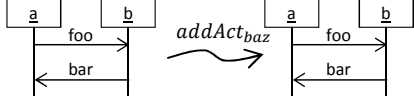
Action-addition operations with signature $M_{SD} \rightarrow M_{SD}$	
Parameters	Let $a \in \mathcal{A}$ be a name representing an action.
Explanation	The operation $addAct_a$ adds the action a to the set of possible actions.
Domain	$(O, O_c, O_v, O_i, A, d) \in dom(addAct_a) \Leftrightarrow a \notin A$
Application	$ \begin{array}{c} (O, O_c, O_v, O_i, A, d) \\ \downarrow \\ addAct_a \\ (O, O_c, O_v, O_i, A', d) \text{ where } A' = A \cup \{a\} \end{array} $
Example	 <p><i>the concrete graphical syntax does not reflect the addition of actions</i></p>

Figure 5.22: Action-addition operations $addAct_a$ for all action names $a \in \mathcal{A}$.

Figure 5.22 defines the action-addition change operations for SDs. Each action-addition operation $addAct_a$ is parametrized with an action name $a \in \mathcal{A}$. On the application of the action-addition operation $addAct_a$ to an SD, the action a is added to the set of possible actions of the SD. The operation is applicable to an SD iff the set of actions of the SD does not already contain the action a . In the example application (cf. Figure 5.22), the action a is added to the SD depicted on the left-hand side. The concrete syntax of the SD remains unchanged as the concrete syntax does not reflect the set of possible actions, unless they are used on interactions.

Adding an action to an SD neither changes the sequence of diagram interactions of the SD, nor the sets of objects tagged as complete, visible, or initial. Therefore, action-addition operations are refactoring.

Proposition 5.10. *Let addAct_a be an action-addition operation and let $sd \in M_{SD}$ be an SD such that addAct_a is applicable to sd . Then, $\llbracket \text{addAct}_a(sd) \rrbracket^{SD} = \llbracket sd \rrbracket^{SD}$.*

Proof. Let addAct_a be an action-addition operation and let $sd = (O, O_c, O_v, O_i, A, d)$ be an SD such that addAct_a is applicable to sd . Further, let $sd' = (O, O_c, O_v, O_i, A \cup \{a\}, d)$. By definition of addAct_a , it holds that $\text{addAct}_a(sd) = sd'$. By definition of SD semantics, it directly follows that $\llbracket sd \rrbracket^{SD} = \llbracket \text{addAct}_a(sd) \rrbracket^{SD}$. \square

5.4.10 Action-Deletion Operations

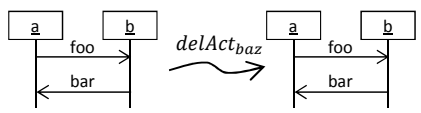
	Action-deletion operations with signature $M_{SD} \rightarrow M_{SD}$	
Parameters	Let $a \in \mathcal{A}$ be a name representing an action.	
Explanation	The operation delAct_a deletes the action a from the set of possible actions.	
Domain	$(O, O_c, O_v, O_i, A, d) \in \text{dom}(\text{delAct}_a) \Leftrightarrow a \in A \wedge \forall i \in \underline{d}: \text{act}(d.i) \neq a$	
Application	$\begin{array}{c} (O, O_c, O_v, O_i, A, d) \\ \downarrow \\ (O, O_c, O_v, O_i, A', d) \text{ where } A' = A \setminus \{a\} \end{array}$	
Example		

Figure 5.23: Action-deletion operations delAct_a for all action names $a \in \mathcal{A}$.

Figure 5.23 defines the action-deletion operations. Each action-deletion operation delAct_a is parametrized with an action name $a \in \mathcal{A}$. On the application of the action-deletion operation delAct_a to an SD, the action a is removed from the set of possible actions of the SD. The operation is applicable to an SD iff the set of actions of the SD contains the action a and the action is not used in any interaction of the SD's diagram interactions. In the example application, the action a is removed from the SD depicted on the left-hand side. The graphical syntax of the SD remains unchanged.

The application of an action-deletion operation to an SD neither changes the sequence of diagram interaction of the SD, nor the sets of objects tagged as complete, visible, or initial in the SD. Therefore, action-deletion operations are refactoring.

Proposition 5.11. *Let delAct_a be an action-deletion operation and let $sd \in M_{SD}$ be an SD such that delAct_a is applicable to sd . Then, $\llbracket \text{delAct}_a(sd) \rrbracket^{SD} = \llbracket sd \rrbracket^{SD}$.*

Proof. Let $delAct_a$ be an action-deletion operation and let $sd = (O, O_c, O_v, O_i, A, d)$ be an SD such that $delAct_a$ is applicable to sd . Further, let $sd' = (O, O_c, O_v, O_i, A \setminus \{a\}, d)$. By definition of $delAct_a$, it holds that $delAct_a(sd) = sd'$. By definition of SD semantics, it directly follows that $\llbracket sd \rrbracket^{SD} = \llbracket delAct_a(sd) \rrbracket^{SD}$. \square

5.4.11 Interaction-Addition Operations

Interaction-addition operations with signature $M_{SD} \rightarrow M_{SD}$	
Parameters	Let $i \in \mathbb{N}$, let $o, p \in \mathcal{O}$ be two names representing objects and let $a \in \mathcal{A}$ be a name representing an action.
Explanation	The operation $addIA_{i,o,a,p}$ adds the interaction (o, a, p) to the sequence of diagram interaction at position i .
Domain	$(O, O_c, O_v, O_i, A, d) \in dom(addIA_{i,o,a,p}) \Leftrightarrow o, p \in O \wedge a \in A \wedge i \leq d $
Application	(O, O_c, O_v, O_i, A, d) where $d = d_1, \dots, d_n$ with $n \geq 0$ \downarrow $(O, O_c, O_v, O_i, A, d')$ where $d' = d_1, \dots, d_i, (o, a, p), d_{i+1}, \dots, d_n$
Example	

Figure 5.24: Interaction-addition operations $addIA_{i,o,a,p}$ for all indices $i \in \mathbb{N}$, object names $o, p \in \mathcal{O}$, and action names $a \in \mathcal{A}$.

Figure 5.24 defines the interaction-addition operations. Each interaction-addition operation $addIA_{i,o,a,p}$ is parametrized with an index $i \in \mathbb{N}$ that defines the position where the interaction is inserted into the sequence of diagram interactions of an SD, an object $o \in \mathcal{O}$ representing the interaction's source object, the action of the interaction $a \in \mathcal{A}$, and an object $p \in \mathcal{O}$ representing the interaction's target object. On the application of the interaction-addition operation $addIA_{i,o,a,p}$ to an SD, the interaction (o, a, p) is inserted into the SD's sequence of diagram interactions at position i . The operation is applicable to an SD iff the index is smaller than or equal to the length of the SD's sequence of diagram interactions, the SD's set of actions contains the action a , and the objects o and p exist in the SD. In the example application (cf. Figure 5.24), the interaction (b, baz, o) is added at position 1 to the sequence of diagram interactions of the SD depicted on the left-and side.

Interaction-addition operations may change an SD such that the resulting SD's semantics is incomparable to the semantics of the original. Therefore, interaction-addition operations are neither refining nor generalizing. For instance, Figure 5.25 depicts the two SDs $sd1$ and $sd2$. The SD $sd2$ can be obtained by applying the interaction-addition operation $addIA_{1,a,baz,b}$ to the SD $sd1$. The system run $(\{a, b\}, \{foo, bar\}, t)$ where

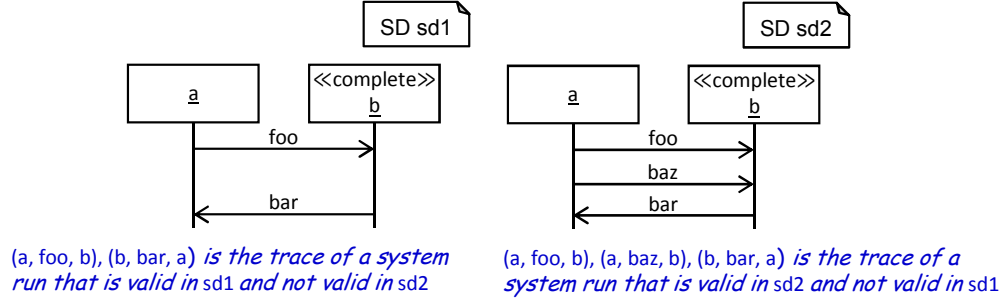


Figure 5.25: Interaction-addition and interaction-deletion operations are neither refining nor generalizing.

$t = (a, foo, b), (b, bar, a)$ is valid in sd1 and not valid in sd2. This illustrates that interaction-addition operations are, in general, not generalizing. On the other hand, the system run $(\{a, b\}, \{foo, bar, baz\}, t)$ where $t = (a, foo, b), (a, baz, b), (b, bar, a)$ is valid in the SD sd2 and not valid in the SD sd1, which shows that interaction-addition operations are, in general, not refining.

5.4.12 Interaction-Deletion Operations

Interaction-deletion operations with signature $M_{SD} \rightarrow M_{SD}$	
Parameters	Let $i \in \mathbb{N}$.
Explanation	The operation $delIA_i$ removes the interaction at position i from the sequence of diagram interaction.
Domain	$(O, O_c, O_v, O_i, A, d) \in dom(delIA_i) \Leftrightarrow i < d $
Application	(O, O_c, O_v, O_i, A, d) where $d = d_1, \dots, d_n$ with $n \geq 0$ \downarrow $(O, O_c, O_v, O_i, A, d')$ where $d' = d_1, \dots, d_i, d_{i+2}, \dots, d_n$
Example	<p>$d = foo, baz, bar$ with $d = 3$</p> <p>$d' = d_1, d_3 = foo, bar$ with $d' = 2$</p>

Figure 5.26: Interaction deletion operations $delIA_i$ for all indices $i \in \mathbb{N}$.

Figure 5.26 defines the interaction-deletion operations. Each interaction-deletion operation $delIA_i$ is parametrized with an index $i \in \mathbb{N}$, which defines the position of the interactions that is removed from an SD's sequence of diagram interactions. On the application of the interaction-deletion operation $delIA_i$ to an SD, the interaction at

position i in the sequence of diagram interactions of the SD is deleted from the SD's sequence of diagram interactions. The operation is applicable to an SD iff the index is smaller than the length of the SD's sequence of diagram interactions, *i.e.*, the index i is a valid position of the sequence.

In the example application (cf. Figure 5.26), the interaction at position 1 is deleted from the the SD depicted on the left-and side. Interaction-deletion operations may change an SD such that the resulting SD's semantics is incomparable to the semantics of the original. Therefore, interaction-deletion operations are neither refining nor generalizing. For instance, Figure 5.25 depicts the two SDs *sd1* and *sd2*. The SD *sd1* can be obtained by applying the interaction-deletion operation $delIA_1$ to the SD *sd2*. The system run $(\{a, b\}, \{foo, bar\}, t)$ where $t = (a, foo, b), (b, bar, a)$ is valid in *sd1* and not valid in *sd2*. This illustrates that interaction-addition operations are, in general, not refining. On the other hand, the system run $(\{a, b\}, \{foo, bar, baz\}, t)$ where $t = (a, foo, b), (a, baz, b), (b, bar, a)$ is valid in the SD *sd2* and not valid in the SD *sd1*, which shows that interaction-addition operations are, in general, not generalizing.

5.5 Sequence Diagram Modeling Language

The SD modeling language is defined as $\mathcal{L}_{SD} = (M_{SD}, Sem_{SD}, \llbracket \cdot \rrbracket^{SD})$ where M_{SD} is the set of all SDs (defined in Section 5.1), the semantic domain Sem_{SD} is defined as the set of all system runs, and $\llbracket \cdot \rrbracket^{SD}$ maps each SD *sd* to the set of all system runs that are valid in *sd* (defined in Section 5.2). We define the change operation suite O_{SD} for the SD modeling language as the set of all SD change operations defined in the previous sections. The following sketches an algorithm for computing a change sequence to transform an SD *sd* to another SD *sd'*:

1. Start with the empty sequence.
2. For each object that is tagged as complete in *sd* and not tagged as complete in *sd'*, append a change operation for untagging the object as complete.
3. For each object that is tagged as visible in *sd* and not tagged as visible in *sd'*, append a change operation for untagging the object as visible.
4. For each object that is tagged as initial in *sd* and not tagged as initial in *sd'*, append a change operation for untagging the object as initial.
5. For each interaction in *sd*, append a change operation for deleting the interaction.
6. For each object in *sd* that is no object in *sd'*, append a change operation for deleting the object.

7. For each object in sd' that is no object in sd , append a change operation for adding the object.
8. For each object that is tagged as complete in sd' and not tagged as complete in sd , append a change operation for tagging the object as complete.
9. For each object that is tagged as visible in sd' and not tagged as visible in sd , append a change operation for tagging the object as visible.
10. For each object that is tagged as initial in sd' and not tagged as initial in sd , append a change operation for tagging the object as initial.
11. Append operations for constructing the sequence of diagram interactions of sd' .

The algorithm sketched by the eleven steps describes (disregarding the underspecification concerning the order in which the operations are appended in each step) a function $\Delta_{SD} : M_{SD} \times M_{SD} \rightarrow O_{SD}^*$ that takes two SDs as inputs and outputs a change sequence of SD change operations. After applying the operations appended in the first four steps, every object that is tagged as complete (respectively visible and initial) in sd and not in sd' , is untagged as complete (respectively visible and initial). The operations appended in the fifth step delete all interactions used in sd . The operations appended in the sixth step delete all objects that are used in sd and not used in sd' . The operations are applicable because the operations appended in the fifth step delete all interactions between all objects of sd . Afterwards, step seven appends change operations for adding all objects used in sd' that are not used in sd . The operations appended in the steps eight to ten tag all objects according to the tags used in sd' . Thus, after the application of the change operations appended in the first ten steps, the resulting SD contains all objects that sd' contains. These objects are tagged in the same way as they are tagged in sd' . The sequence of diagram interactions of the SD is the empty sequence. The operations appended in the eleventh step add interaction-addition operations such that the sequence of diagram interactions of the resulting SD is equal to the sequence of diagram interactions of the SD sd' . Therefore, the eleven steps sketch a function for computing change sequences to change any SD to any other SD. From this, we can conclude that the SD change operation suite O_{SD} is complete for the SD modeling language \mathcal{L}_{SD} .

5.6 Related Work

This section focuses on related work concerning notions of refinement and procedures for refinement checking (semantic differencing) for SD variants with precisely defined semantics. We refer to [MW11] for a survey on different SD semantics.

Another trace semantics with another notion of refinement for an SD variant is proposed in [LK11, LK14]. The proposed semantics distinguishes between the sending and

the receiving of events. This leads to a more fine-grained semantics that includes interleaving of interactions, which is not possible with the UML/P semantics [Rum16]. In UML/P SDs, there is an implicit ordering of the receiving and sending events: The receiving event of an interaction always occurs directly after the corresponding sending event and the sending event of an interaction always directly follows the receiving event of the previous interaction. When assuming the absence of concurrency, the UML/P semantics is more adequate. The notion of refinement introduced in [LK11, LK14] is based on a simulation relation on traces and not on set inclusion as proposed in this thesis. The authors further present a conformance relation [LK14] that enables detecting whether a successor SD version is a refinement (in the sense of [LK11, LK14]) of its predecessor version in case lifelines and messages may have been renamed. The semantic SD differencing approach presented in this thesis can easily be adapted to incorporate conformance relations by applying an initial matching between object and message names, before applying the semantic differencing operator. The approach [LK11] is also concerned with an SD variant that includes alternative fragments, critical regions, optional fragments, and other fragment types. These are interesting extensions for UML/P SDs. In [LK11], SDs may contain guard conditions, which are encoded as elements of traces in the semantics of SDs. The above leads to a more complicated semantics of [LK11, LK14] in comparison to the semantics used in this thesis. Although introducing a notion of refinement, [LK11, LK14] present no automatic refinement checking (semantic differencing) procedure.

A safety-liveness semantics for UML 2.0 SDs is introduced in [GS05]. The approach is concerned with advanced SD modeling elements such as (bounded) high-level SDs and negative fragments. These are interesting extensions for UML/P SDs. The semantics assumes asynchronous communication and distinguishes between the sending and the receiving of messages. The approach translates SDs to Büchi automata capturing safety and liveness properties encoded by the SDs. Therefore, the approach is concerned with SDs modeling infinite traces, whereas the UML/P SD semantics is concerned with finite traces. The approach targets at modeling non-terminating, reactive systems [GS05]. Refinement is defined via set inclusion on the languages accepted by the Büchi automata. As set inclusion on the languages accepted by Büchi automata can be checked fully automatically, reuse of well-known algorithms for this task is possible for refinement checking and semantic differencing. Aside from stuttering and silent transitions, the alphabets of the automata resulting from applying the translation are equal to the sets of sending and receiving events of the underlying SDs. The approach assumes that the language encoded by an SD is (ω -)regular. In contrast, the semantics of a UML/P SD is not regular, as the set of possible interactions between objects is assumed to be infinite (cf. Definition 5.3). Therefore, in cases where two SDs model different events on their interactions, the semantics of [GS05] may be too restrictive for semantic differencing. In contrast, the UML/P semantics assumes interactions using events not used in an SD, between objects that are not constrained by stereotypes, to be unconstrained. With the

semantics of [GS05], such interactions must not occur.

The semantics defined in [Stö03b] for SDs without special fragments is similar to the UML/P semantics. In [Stö03b], SDs that correspond to the UML/P SD syntax without stereotypes are called plain interaction fragments. Each plain interaction fragment is defined by a partial order over a set of possible event occurrences. The semantics of such an interaction fragment is then defined as the set of all sequentialization of the partial order [Stö03b]. However, although concluding that the notion of refinement for SDs needs to be discussed [Stö03b], in contrast to this thesis, the approach presents no semantic differencing operator for SDs. The approach further defines a semantic mapping for SDs with advanced combined fragments, *e.g.*, for modeling parallelism, optional occurrences of events, and alternative choices. As stated above, the extension of UML/P SD modeling language with combined fragments is interesting future work. In general, according to [Stö03b], the semantics of an SD is a pair of sets of valid and invalid traces. In UML/P SDs, the set of invalid traces of an SD can be considered to be the set of all traces of all system runs that are not valid in the SD. Thus, an UML/P SD also implicitly defines a set of traces that are invalid in the SD.

In [Stö03a], the same author discusses the notion of refinement for SDs based on the semantics presented in [Stö03b]. Refinement is defined as set inclusion on the sets of valid and invalid traces modeled by SDs. In [Stö03b], an SD is a refinement of another SD if the set of traces unconstrained by the former SD (possible traces that are neither elements of the SD's set of valid traces nor of the set of the SD's invalid traces) is a subset of the traces unconstrained by the latter SD and every valid (respectively invalid) trace of the latter SD is also a valid (respectively invalid) trace of the former SD. In contrast, the UML/P semantics is only concerned with valid traces and not with invalid traces. Using the UML/P semantics, every trace corresponds to a system run that is valid in an SD or to a system run that is not valid in the SD. Refinement of SDs in the sense of this thesis corresponds to the transpose of the enrichment relation as defined in [Stö03a], *i.e.*, an SD is a refinement of another SD, if the valid traces in the semantics of the former is a subset of the valid traces in the semantics of the latter. The approach [Stö03a] does not present an automatic method for refinement checking or semantic differencing.

The semantics presented in [HHRS05] also distinguishes between the sending and the receiving of messages. In [HHRS05], the semantics of an SD is defined as a set of interaction obligations, where each interaction obligation is a tuple of two disjoint sets of positive and negative traces. Interaction obligations are used to describe the required non-determinism of implementations. An implementation of an SD must satisfy each interaction obligation in the semantics of the SD. Refinement is defined on two levels [HHRS05]: An interaction obligation refines another interaction obligation iff the set of positive (respectively negative) traces of the latter is a subset of the set of positive (respectively negative) traces of the former SD. A SD is a refinement of another SD iff each interaction obligation in the semantics of the latter SD refines at least one interaction obligation in the semantics of the former SD. Thus, refinement is defined

similar as in [Stö03a]. UML/P SDs are neither concerned with required non-determinism nor with negative traces. In contrast to this thesis, [HHR05] presents no semantic differencing operator.

Modal Sequence Diagrams (MSD) are introduced in [HM08]. The MSD language is a profile of the UML 2.0 SD language with different semantics. The MSD language adds modal interaction fragments. Modal interaction fragments allow denoting (parts of) an interaction as universal (hot) or existential (cold). With universal interaction fragments, it is possible to specify that messages must be sent or that conditions must evaluate to true under certain conditions (or in all runs). Existential fragments model that there are system runs that satisfy the fragment, but not necessarily all system runs must satisfy the fragment. The combination of existential and universal fragments inside a universal interaction allows specifying conditions concerning when certain fragments are mandatory. For example, the combination allows specifying that a certain fragment must occur in case another fragment occurred before. Considering universal and existential fragments in UML/P SDs is an interesting extension, which increases the expressiveness and enables modeling of more complex situations. The semantic domain of MSDs represents infinite system runs and their traces. The trace language of a universal MSD is defined using an alternating weak word automaton. In contrast, UML/P SDs model finite system runs and the semantic differencing operator constructs NFAs. MSDs also support constraints, which are boolean expressions over attributes associated with participants (objects), and the authors also describe the extension of MSDs with advanced constructs such as loops, alternatives, breaks, and nested fragments. Considering constraints and further advanced fragments is also an interesting extension for UML/P SDs.

The authors of [HM08] also introduce the notion of MSD specification. An MSD specification is a set of modal SDs. The semantics is given by system models satisfying an MSD specification, where a system model represents a set of possible runs. A system model satisfies an MSD specification if (1) every possible run of the system model satisfies every universal diagram in the specification and (2) every existential diagram in the specification is satisfied by at least one of the possible runs. This thesis does not introduce a notion of UML/P SD specification including sets of UML/P SDs and the semantic is purely based on system runs and not on system models. However, UML/P SDs can be interpreted to solely represent existential interaction fragments. In the notion of [HM08], there are at least two meaningful semantics of sets of UML/P SDs concerning the system models that satisfy the set. In some cases, it is meaningful to define that a system model satisfies a set of UML/P SDs if for every UML/P SD contained in the set there is a run of the system model that is valid in the SD. Thus, for each SD contained in the set, the system model has a run that is valid in the SD. This represents that the situation modeled in each SD can be achieved by at least one run of the system model. This corresponds to the satisfaction of a set of existential MSDs. In other cases, it is meaningful to define that a system model satisfies a set of UML/P if every run of the system model is valid in at least one of the SDs contained in the set. In this

case, the semantics of a set of UML/P SDs is defined as the union of the semantics of the SDs contained in the set. Then, the SDs contained in the set model all different alternatives concerning the possible system runs. Whether it is still possible to provide a fully automatic semantic differencing procedure when adding the features of MSDs to UML/P SDs is an interesting question for future work.

Chapter 6

Activity Diagrams

An activity diagram (AD) describes possible executions of an activity or a process [Stö05, Esh06, vdA99, GRR10, KG10, MRR11c, MRR11b, Sug16, KR18b]. This thesis uses an expressive variant of ADs with actions, and-fragments modeled with fork and join nodes, and xor- as well as cyclic-fragments modeled with decision and merge nodes [KR18b]. The AD variant and the semantic differencing operator for ADs are based on our previous work [KR18b]. As a notational convention, this thesis uses the standard graphical notations of the UML [OMG15].

Figure 6.1 depicts an AD that contains all AD modeling elements used in this thesis. An action represents a single task that needs to be executed as part of an activity or workflow. Control flow nodes are modeling elements for describing alternatives through xor-fragments (decision, merge nodes), repeated executions of the same actions through cyclic-fragments (decision, merge nodes), and for describing order-independent execution branches through and-fragments (fork, join nodes) within an activity. The start of an activity is marked with an initial node. A final node marks the end of an activity. Transitions between nodes describe the control flow of an AD, *i.e.*, how the execution of an AD proceeds after visiting a node [KR18b].

For example, the AD depicted Figure 6.1 models the executions of actions to be performed by employees of an insurance company in reaction to receiving a claim. The activity starts at the initial node and ends at the final node. First, the claim is recorded. Then, the claim is checked, which might require the employee to retrieve additional data. As soon as enough data is available, the claim is settled or rejected. The cyclic-fragment including the actions `Check Claim` and `Retrieve Add. Data` models the circumstance of iteratively checking the claim and retrieving additional data until enough data is available. If the claim is settled, the employee sends a confirmation, calculates the loss amount, recalculates the customer contribution, and initiates the payout. The execution of the action for calculating the loss amount is independent of the execution of the action for recalculating the customer contribution. The and-fragment starting at the inner fork node and ending at the inner join node models this circumstance. Every individual arrow originating from a fork node introduces an execution branch that is causally unrelated to the execution branches introduced by the other arrows originating from the fork node. The action for calculating the loss amount can be executed before

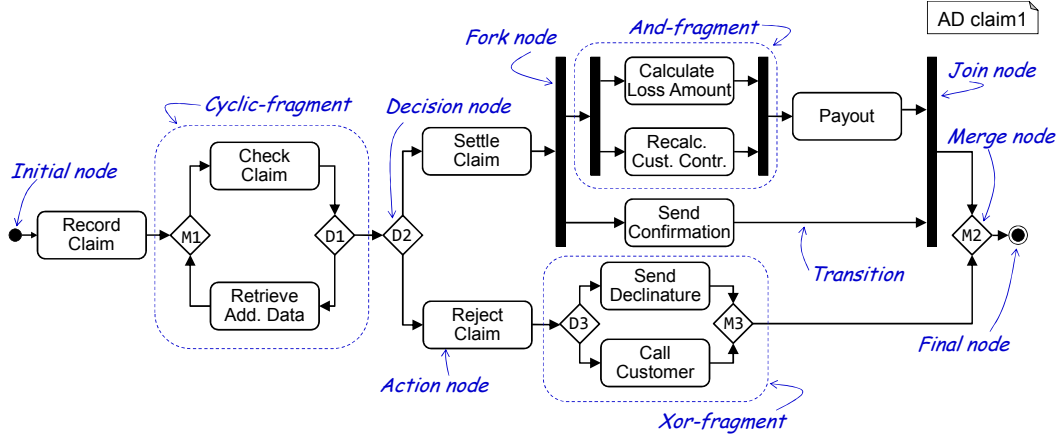


Figure 6.1: An AD adapted from [MR18] that contains all AD modeling elements.

the action for recalculating the customer contribution and vice versa. However, the join node cannot be traversed before every execution branch introduced by the fork node has been completely executed. If the claim is rejected, the employee either sends a declinature or calls the customer. This circumstance is modeled by the xor-fragment starting at the decision node D3 and ending at the merge node M3. When traversing a decision node, exactly one execution branch starting at the decision node must be executed. After either performing the actions for settling the claim or the actions for rejecting the claim, the AD traverses the merge node M2, before the activity ends at the final node.

This thesis formally defines the abstract syntax of ADs and an operational semantics for ADs via a translation to finite automata based on our previous work [KR18b]. To this effect, similar to related work [SO00, vdAHV02, VVL07], this thesis first introduces the syntax of activity graphs. An activity graph (AG) is interpretable to be a possibly unsound AD [vdAHV02, VVL07]. Subsequently, the set of ADs is defined as a subset of the set of all AGs. The semantics of an AG is the set of traces (finite sequences of action labels) that are described by the AG. For determining the traces, this thesis presents an operational semantics for AGs based on a translation to NFAs. The denotational semantics of an AG is then defined as the language recognized by the NFA obtained from translating the AG.

In the following, Section 6.1 describes the syntax of AGs. Then, Section 6.2 presents the semantics of AGs. Based on this, Section 6.3 presents a semantic differencing operator for AGs. Section 6.4 introduces AG change operations. Finally, Section 6.5 defines the AD modeling language with a syntax and a semantics based on AGs.

6.1 Activity Graph Syntax

The following first defines the syntax of activity graphs, which are interpretable as possibly unsound ADs [vdAHV02]. Every AD (defined in Section 6.5) is by construction an AG but not vice versa. Based on previous work for determining the soundness of ADs by means of single-entry-single-exit fragments [VVL07], we later define the syntax of the AD language (*i.e.*, the set of syntactically well-formed ADs, which are sound AGs by construction) recursively by stating the smallest AD (axiom AD) and rules for constructing more complex ADs via change operation applications.

An AG consists of action labels, nodes, action nodes, exactly one initial node, exactly one final node, fork nodes, join nodes, decision nodes, merge nodes, and-fragments, xor-fragments, cyclic-fragments, transitions, and a node labeling function. The abstract syntax of AGs is defined as follows:

Definition 6.1. *An AG is a tuple $(L, N, t, AND, XOR, C, T, l)$ where*

- $L \subseteq U_N$ is an alphabet of action labels,
- $N \subseteq U_N$ is a finite set of nodes,
- $t : N \rightarrow \{A, i, f, F, J, D, M\}$ is a node typing function,
- $AND \subseteq N \times N$ where $\forall (a, b) \in AND : t(a) = F \wedge t(b) = J$ contains tuples of fork and join nodes defining and-fragments,
- $XOR \subseteq D \times M$ where $\forall (a, b) \in XOR : t(a) = D \wedge t(b) = M$ contains tuples of decision and merge nodes defining xor-fragments,
- $C \subseteq D \times M$ where $\forall (a, b) \in C : t(a) = D \wedge t(b) = M$ contains tuples of decision and merge nodes defining cyclic-fragments,
- $T \subseteq N \times N$ is the transition relation,
- $l : N \rightarrow L \cup \{\varepsilon\}$ is the node labeling function that is required to map exactly the control-flow nodes to the empty word, *i.e.*, $\forall n \in N : l(n) = \varepsilon \Leftrightarrow t(n) \neq A$.

The following well-formedness rules apply:

- $\exists a, b \in N : a \neq b \wedge t(a) = i \wedge t(b) = f$, *i.e.*, an initial and a final node exist.
- $\forall a, b \in N : (t(a) = i \wedge a \neq b) \Rightarrow t(b) \neq i$, *i.e.*, the initial node is unique.
- $\forall a, b \in N : (t(a) = f \wedge a \neq b) \Rightarrow t(b) \neq f$, *i.e.*, the final node is unique.
- $|\{(n, a) \in T \mid t(a) = i\}| = 0$, *i.e.*, the initial node has no incoming transitions.
- $|\{(a, n) \in T \mid t(a) = i\}| = 1$, *i.e.*, the initial node has one outgoing transition.

- $|\{(a, n) \in T \mid t(a) = f\}| = 0$, i.e., the final node has no outgoing transitions.
- $|\{(n, a) \in T \mid t(a) = f\}| = 1$, i.e., the final node has one incoming transition.

In the following, AG denotes the set of all AGs. As notational convention, for an AG $ag = (L, N, t, AND, XOR, C, T, l)$ and all nodes $n \in N$,

- $\delta_{ag}^+(n) \stackrel{\text{def}}{=} \{(n, x) \in T \mid x \in N\}$ denotes the set of transitions starting in n ,
- $\delta_{ag}^-(n) \stackrel{\text{def}}{=} \{(x, n) \in T \mid x \in N\}$ denotes the set of transitions ending in n .
- $F(ag) \stackrel{\text{def}}{=} \{n \in N \mid t(n) = F\}$ denotes the set of fork nodes of ag .
- $J(ag) \stackrel{\text{def}}{=} \{n \in N \mid t(n) = J\}$ denotes the set of join nodes of ag .
- $D(ag) \stackrel{\text{def}}{=} \{n \in N \mid t(n) = D\}$ denotes the set of decision nodes of ag .
- $M(ag) \stackrel{\text{def}}{=} \{n \in N \mid t(n) = M\}$ denotes the set of merge nodes of ag .
- $A(ag) \stackrel{\text{def}}{=} \{n \in N \mid t(n) = A\}$ denotes the set of action nodes of ag .

The AD `hire1` depicted in Figure 6.2, for instance, can be formally defined by $\text{hire1} \stackrel{\text{def}}{=} (L, N, t, AND, XOR, C, T, l)$ with

- labels $L = \{\text{Register, Assign to Project, Get Wel. Package, Add to Website, Manager Interview, Manager Report, Authorize Payment}\}$,
- nodes $N = \{R, ATP1, GWP, ATP2, ATW, MI, MR, AP, D1, M1, F1, J1, i, f\}$,
- node typing function $t : N \rightarrow \{A, i, f, F, J, D, M\}$ with $t(n) = A$ for all action nodes $n \in \{R, ATP1, GWP, ATP2, ATW, MI, MR, AP\}$, $t(i) = i$ for the initial node i , $t(f) = f$ for the final node f , $t(n) = F$ for all fork nodes $n \in \{F1\}$, $t(n) = J$ for all join nodes $n \in \{J1\}$, $t(n) = D$ for all decision nodes $n \in \{D1\}$, $t(n) = M$ for all merge nodes $n \in \{M1\}$,
- parallel fragments $AND = \{(F1, J1)\}$,
- xor-fragments $XOR = \{(D1, M1)\}$,
- cyclic-fragments $C = \emptyset$,
- the transition relation $T = \{(i, R), (R, D1), (D1, ATP1), (ATP1, M1), (D1, GWP), (GWP, F1), (F1, ATP2), (F1, ATW), (ATP2, J1), (ATW, J1), (J1, MI), (MI, MR), (MR, M1), (M1, AP), (AP, f)\}$, and
- labeling function l with $l(R) = \text{Register}$, $l(ATP1) = \text{Assign to Project}$, $l(GWP) = \text{Get Wel. Package}$, $l(ATP2) = \text{Assign to Project}$, $l(ATW) = \text{Add to Website}$, $l(MI) = \text{Manager Interview}$, $l(MR) = \text{Manager Report}$, $l(AP) = \text{Authorize Payment}$, and $l(D1) = l(M1) = l(F1) = l(J1) = l(i) = l(f) = \varepsilon$.

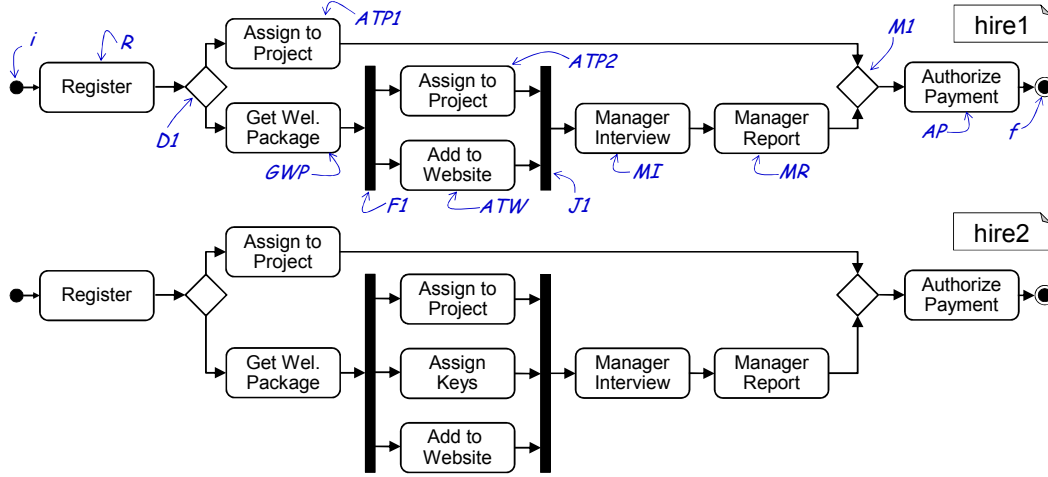


Figure 6.2: Two activity diagrams adapted from [MRR11b].

6.2 Activity Graph Trace Semantics

This section defines a translation from AGs to NFAs. The NFA obtained from translating an AG encodes the traces of actions modeled by the AG under the assumption that exactly one action can be performed at a point in time. Multiple actions cannot be performed simultaneously. Thus, and-fragments solely model independent execution paths. Based on this, the trace semantics of an AG is defined as the language recognized by the NFA obtained from translating the AG. If the possibility to execute multiple actions simultaneously is required, defining and using a more fine-grained semantics might be more adequate.

The actions of ADs naturally correspond to the transitions of NFAs because the actions in ADs and the transitions in NFAs describe the behavior. Consequently, although rather counterintuitive, the transitions of ADs correspond to states in NFAs.

The set of states of the NFA is the powerset of the set of transitions of the AG. Using the powerset is necessary as AGs can reside in several transitions simultaneously (a consequence of traversing a fork node). However, the reachable number of states is often much smaller as the powerset and can be algorithmically easily computed. The set of transition labels of the NFA is equal to the set of node labels of the AG. The initial state of the NFA is the singleton set containing the transition that has the initial AG node as the source node. The set of final states is the singleton set containing the transition that has the final AG node as the target node.

The transition relation of the NFA is defined as the smallest set of transitions satisfying two conditions. The first condition represents the AG's behavior in case it executes a node that is neither a join nor a fork node. If an action node is executed, then the AG

moves through the executed action via the action's outgoing transition to the next node while performing the action node's label. The AG's execution state with respect to other nodes remains unchanged. Similarly, if the AG proceeds through a decision or merge node, its state regarding the other nodes remains unchanged and no action is performed (encoded in the NFA by the empty word ε). The second condition represents the AG's behavior in case it executes a fork or a join node: An AG can only proceed through a join or fork node if its current state contains all nodes that have a transition to the node. When the AG proceeds through a fork or a join node, it leaves all nodes that proceed the node and enters all its successor nodes.

Definition 6.2. Let $ag = (L, N, t, AND, XOR, C, T, l)$ be an AG. The NFA associated with ag is defined as $nfa(ag) \stackrel{\text{def}}{=} (Q, \Sigma, \delta, q_0, F)$ where

- $Q = \wp(T)$,
- $\Sigma = L$,
- $q_0 = \{(a, n) \in T \mid t(a) = i\}$,
- $F = \{\{(n, a)\} \subseteq T \mid t(a) = f\}$, and
- $\delta \subseteq Q \times (\Sigma \cup \{\varepsilon\}) \times Q$ is the smallest set satisfying the following conditions:
 - Move other than fork or join:

$$\forall X \subseteq T : \forall n_1, n_2, n_3 \in N : ((n_1, n_2) \in X \wedge (n_2, n_3) \in T \wedge t(n_2) \notin \{F, J\}) \Rightarrow (X, l(n_2), (X \setminus \{(n_1, n_2)\}) \cup \{(n_2, n_3)\}) \in \delta$$
 - Move fork or join:

$$\forall X \subseteq T : \forall n \in F(ag) \cup J(ag) : (\delta_{ag}^-(n) \subseteq X) \Rightarrow (X, \varepsilon, (X \setminus \delta_{ag}^-(n)) \cup \delta_{ag}^+(n)) \in \delta$$

The trace semantics $traces(ag)$ of an AG ag is defined as the language recognized by the NFA associated with the AG ag , i.e., $traces(ag) \stackrel{\text{def}}{=} \mathcal{L}_*(nfa(ag))$.

For example, Figure 6.3 depicts the trimmed NFA obtained from translating the AG `hire1`, which is graphically illustrated in Figure 6.2 and formally defined in Section 6.1. Figure 6.3 only depicts the states and transitions of the NFA that are reachable from the initial state of the NFA. A possible trace of the AG is, for instance, the word **Register, Assign to Project, Authorize Payment** $\in traces(\text{hire1})$.

Algorithm 1 is a simple procedure for translating AGs to NFAs. The algorithm takes an AG as input and outputs the reachable part of the NFA associated with the AG. The initial state of the automaton is initialized as the singleton set containing the transition originating from the initial node (l. 1). The set of states is initialized as the singleton set containing the initial state (l. 2). The transition relation and the final states are initialized as empty sets (ll. 3-4). Then, the algorithm initializes the variable *processedStates*

Algorithm 1 Computing the reachable part of the NFA associated with an AG.

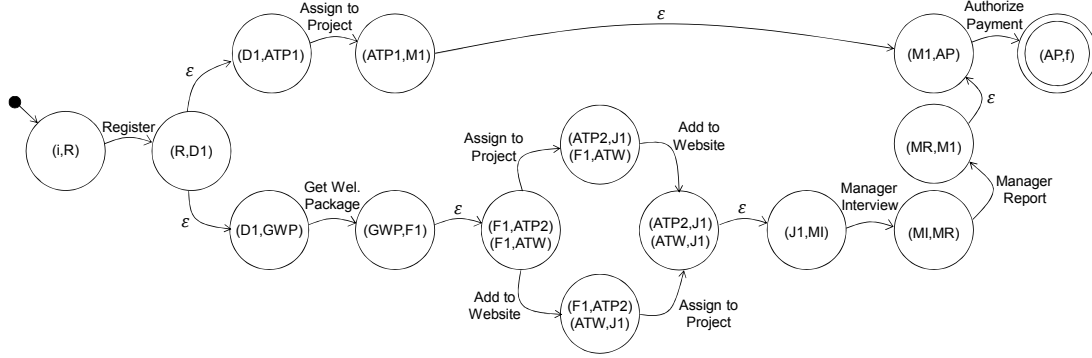
Input: AG $ag = (L, N, t, AND, XOR, C, T, l)$.

Output: The NFA $nfa(ag)$ associated with the input AG ag .

```

1: define  $q_0 \leftarrow \{(s, t) \in T \mid t(s) = i\}$  as subset of  $T$ 
2: define  $Q \leftarrow \{q_0\}$  as subset of  $\wp(T)$ 
3: define  $\delta \leftarrow \emptyset$  as subset of  $\wp(T) \times (L \cup \{\varepsilon\}) \times \wp(T)$ 
4: define  $F \leftarrow \emptyset$  as subset of  $\wp(T)$ 
5: define  $processedStates \leftarrow \emptyset$  as subset of  $\wp(T)$ 
6: define  $statesToProcess$  as empty stack of  $\wp(T)$ 
7:  $statesToProcess.push(q_0)$ 
8: while  $statesToProcess$  not empty do
9:   define  $currentState \leftarrow statesToProcess.pop()$ 
10:   $processedStates \leftarrow processedStates \cup \{currentState\}$ 
11:  if  $currentState \in \{(n, a) \mid t(a) = f\}$  then
12:     $F \leftarrow F \cup \{currentState\}$ 
13:  end if
14:  for all  $(n_1, n_2) \in currentState$  do
15:    for all  $n_3 \in N$  do
16:      if  $(n_2, n_3) \in T \wedge t(n_2) \notin \{F \cup J\}$  then
17:        define  $trgState \leftarrow (currentState \setminus \{(n_1, n_2)\}) \cup \{(n_2, n_3)\}$ 
18:         $Q \leftarrow Q \cup \{trgState\}$ 
19:         $\delta \leftarrow \delta \cup \{(currentState, l(n_2), trgState)\}$ 
20:        if  $trgState \notin processedStates$  then
21:           $statesToProcess.push(trgState)$ 
22:        end if
23:      end if
24:    end for
25:  end for
26:  for all  $n \in F(ag) \cup J(ag)$  do
27:    if  $\delta_{ag}^-(n) \subseteq currentState$  then
28:      define  $trgState \leftarrow (currentState \setminus \delta_{ag}^-(n)) \cup \delta_{ag}^+(n)$ 
29:       $Q \leftarrow Q \cup \{trgState\}$ 
30:       $\delta \leftarrow \delta \cup \{(currentState, \varepsilon, trgState)\}$ 
31:      if  $trgState \notin processedStates$  then
32:         $statesToProcess.push(trgState)$ 
33:      end if
34:    end if
35:  end for
36: end while
37: return  $(Q, L, \delta, q_0, F)$ 

```


 Figure 6.3: Reachable part of the NFA associated with `hire1` (cf. Figure 6.2).

as a set for storing already processed states (l. 5). The variable *statesToProcess* is a stack of states storing the states that still need to be processed (l. 6). First, the initial state is pushed on the stack (l. 7). Then, while the stack of states to process is not empty (l. 8), the algorithm pops the top element of the stack, stores it in variable *currentState* (l. 9) and adds it to the set *processedStates* (l. 10). If the *currentState* is the singleton set containing the transition to the final state, then the algorithm adds the state to the set of final NFA states (ll. 11-13).

In the following for-loop (ll. 14-25), the algorithm constructs the NFA transitions that correspond to the AG transitions that do not target fork or join nodes. These transitions are added to the set of NFA transitions. For each transition contained in the set represented by the *currentState* (l. 14) and for each node of the AG (l. 15), the algorithm checks whether there exists an AG transition from the target state of the transition contained in the state's set to the AG node and if the AG node is neither a fork nor a join node (l. 16). If the conditions are satisfied, then the algorithm adds the corresponding transition as defined in Definition 6.2 (ll. 17-19) and adds the transition's target state to the set of NFA states. If the target state has not been processed by a previous iteration of the algorithm, then the target state is pushed on the stack of states to process (ll. 20-22). The following for-loop (ll. 26-35) constructs the NFA transitions that correspond to the AG transitions that target fork or join nodes. The constructed transitions are added to the transition relation of the NFA. For each fork and each join node (l. 26), the algorithm checks whether all transitions targeting the node are contained in the set of transitions represented by the currently handled state (l. 27). If this is the case, then the corresponding transition is added to the set of NFA transitions as defined in Definition 6.2 (ll. 28-30). The target state is also added to the set of NFA states (l. 29). If the target state has not been processed by a previous algorithm iteration, then it is pushed on the stack of states that are still to be processed (ll. 31-33). Finally, the algorithm returns the computed NFA (l. 37). Figure 6.3 depicts the NFA obtained from

applying Algorithm 1 the AG `hire1`, which is graphically illustrated in Figure 6.2 and formally defined in Section 6.1.

6.3 Semantic Differencing of Activity Graphs

The semantic difference $\delta(ag_1, ag_2)$ from an AG ag_1 to an AG ag_2 is defined as the set of traces of the AG ag_1 that are no traces of the AG ag_2 , *i.e.*, $\delta(ag_1, ag_2) = \text{traces}(ag_1) \setminus \text{traces}(ag_2)$. With the explicit mapping from AGs to NFAs, the reuse of well-known constructions from automata theory [HMU06] is possible. It holds that $\delta(ag_1, ag_2) = \emptyset$ iff $\text{traces}(ag_1) \subseteq \text{traces}(ag_2)$, which is again equivalent to $\mathcal{L}_*(nfa(ag_1)) \subseteq \mathcal{L}_*(nfa(ag_2))$. We can thus reuse well-known techniques for language inclusion checking and counterexample generation for NFAs, which are two well-studied decidable problems.

For example, Figure 6.2 depicts two AGs from [MRR11b]. The AGs describe workflows of a company to be executed when hiring new employees. The AG `hire1` describes the company's original workflow. The AG `hire2` describes a successor version. In the original workflow, the employee is first registered. Then, the employee either directly gets assigned a project before her payment is authorized, or the employee gets a welcome package, before her contact data is added to the company website, she is assigned a project, she is interviewed by the manager, and she gets a manager report, before her payment is finally authorized. The actions for adding the employee to the company website and assigning the employee to a project can be executed independently of each other, *i.e.*, there is no strict order in which the two actions have to be executed. The company decides to explicate that new employees must receive keys to enter the building. The company thus changes the workflow `hire1` by adding the action labeled `Assign Keys` as depicted in the bottom part of Figure 6.2. A manager wants to understand how the syntactic changes impact the AG's possible execution traces. Using our method for semantic differencing reveals that there are traces of `hire1` that are no traces of `hire2` and vice versa. Thus, execution traces have been removed and new execution traces have been added. The semantic differencing operator outputs that `Register, Get Wel. Package, Assign to Project, Add to Website, Manager Interview, Manager Report, Authorize Payment` is a possible execution trace of `hire1` that is no execution trace of `hire2`. This execution trace has been removed during the evolution of the workflow. Vice versa, the semantic differencing operators presents an execution trace that includes the action `Assign Keys`, which is possible in `hire2` and not possible in `hire1`. This execution trace has been added during the evolution from `hire1` to `hire2`.

Figure 6.1 and Figure 6.4 depict the AGs `claim1` and `claim2` inspired by ADs from [MR18]. The AGs model workflows to be performed in an insurance company in response to an incoming claim. The AD `claim1` models the company's original workflow. After some time, the company decides to modify the workflow to `claim2`. A manager is interested in the semantic difference from the new workflow to the original

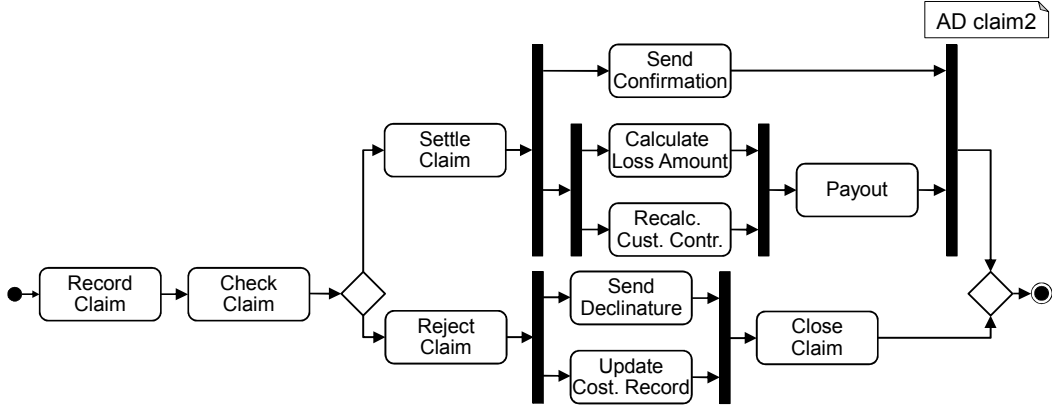


Figure 6.4: An AD adapted from [MR18].

AG	#AG nodes	#AG trans.	#NFA states	#NFA trans.
ad1	8	8	8	8
ad2	6	6	6	6
hire1	14	15	15	16
hire2	15	17	19	24
claim1	23	27	32	42
claim2	21	24	29	38
claim3	31	38	53	79
claim4	27	32	41	57

Figure 6.5: The number of nodes and transitions of the AGs and the number of states and transitions of the NFAs constructed from the AGs for semantic differencing.

workflow and vice versa. She thus uses our framework for semantic AG differencing. The semantic differencing operator outputs that **Record Claim**, **Check Claim**, **Reject Claim**, **Send Declinature** is a possible execution trace of the new workflow that is not possible in the original workflow. Vice versa, the manager is interested if execution traces have been removed during the evolution from `claim1` to `claim2`. She thus uses our framework again. The semantic differencing operator outputs that **Record Claim**, **Check Claim**, **Reject Claim**, **Send Declinature**, **Update Cust. Record**, **Close Claim** is a possible execution trace of `claim1` that is not possible in `claim2`.

Semantic Differencing Implementation and Experiments

We implemented the semantic differencing operator for AGs to perform experimental evaluations. The implementation is written in Java and uses the automaton language

inclusion checking tool RABIT¹ [ACC⁺11] for NFA language inclusion checking.

The implementation takes two AGs as inputs. It translates the AGs into NFAs according to the translation described above. Then, it transforms the NFAs to equivalent NFAs without ε -transitions by using a standard construction for eliminating ε -transitions [HMU06]. Subsequently, it outputs the resulting NFAs in the BA format, which is the input format of RABIT. Finally, the implementation uses the tool RABIT for language inclusion checking of the NFAs. In case language inclusion does not hold, RABIT provides a counterexample. The counterexample is returned as a diff witness.

We performed experimental evaluations with eight example AGs. Appendix D presents the example AGs in detail. Figure 6.5 summarizes the sizes of the AGs in terms of the numbers of their nodes and transitions as well as the numbers of states and transitions of the NFAs resulting from translating the AGs. We executed the semantic differencing operator for all pairs of example AGs that are thematically related to each other. All experiments were executed on a laptop computer with an Intel Core i7-8650U CPU @ 1.90GHz processor, 16GB RAM, and a Samsung PM981 512GB SSD hard drive using Windows 10 and Java 1.8.0_192.

Figure 6.6 summarizes the computation times of the semantic differencing operator and the computed diff witnesses for the input pairs. If no witness exists, *i.e.*, refinements holds, then the corresponding cell in the table contains the special symbol $-$. For instance, the semantic differencing operator took 145ms to detect that the AG `claim2` is a refinement of the AG `claim3`. The semantic differencing operator implementation took 127ms to compute the diff witness `C, A` in the semantic difference from the AG `ad1` to the AG `ad2`.

For the examples, the computation times range from 108ms to 283ms. We conclude that the implementation handles the example AGs sufficiently quick. However, the example AGs are relatively small in terms of the numbers of nodes and transitions in the AGs. Therefore, the results are not generalizable to large AGs and real world examples, especially because language inclusion checking between NFAs is, in general, computationally hard.

6.4 Activity Graph Change Operations

This section presents change operations for AGs. Based on these, the following section defines the syntax of the AD language and AD change operations. Most of the change operations are neither refining nor generalizing. Although these change operations are irrelevant for developers to constructively refine or generalize models, the change operations are necessary to obtain a complete change operation suite. The completeness of the change operation suite is required for the model repair framework presented in Chapter 7

¹<http://languageinclusion.org/>

Difference	Time	Diff witness
$\delta(ad1, ad1)$	108ms	-
$\delta(ad1, ad2)$	127ms	C, A
$\delta(ad2, ad1)$	128ms	C, B
$\delta(ad2, ad2)$	142ms	-
$\delta(hire1, hire1)$	160ms	-
$\delta(hire1, hire2)$	238ms	Register, Get Wel., Package, Add to Website, Assign to Project, Manager Interview, Manager Report, Authorize Payment
$\delta(hire2, hire1)$	249ms	Register, Get Wel., Package, Add to Website, Assign Keys, Assign to Project, Manager Interview, Manager Report, Authorize Payment
$\delta(hire2, hire2)$	145ms	-
$\delta(claim1, claim1)$	166ms	-
$\delta(claim1, claim2)$	216ms	Record Claim, Check Claim, Reject Claim, Call Customer
$\delta(claim1, claim3)$	180ms	Record Claim, Check Claim, Reject Claim, Call Customer
$\delta(claim1, claim4)$	197ms	Record Claim, Check Claim, Reject Claim, Call Customer
$\delta(claim2, claim1)$	174ms	Record Claim, Check Claim, Reject Claim, Update Cust., Record, Send Declination, Close Claim
$\delta(claim2, claim2)$	126ms	-
$\delta(claim2, claim3)$	145ms	-
$\delta(claim2, claim4)$	202ms	Record Claim, Check Claim, Settle Claim, Calculate Loss Amount, Send Confirmation, Recalc. Cust. Contr., Payout
$\delta(claim3, claim1)$	231ms	Record Claim, Check Claim, Reject Claim, Update Cust., Record, Call Customer, Close Claim
$\delta(claim3, claim2)$	283ms	Record Claim, Check Claim, Retrieve Add. Data, Check Claim, Reject Claim, Call Customer, Update Cust., Record, Close Claim
$\delta(claim3, claim3)$	150ms	-
$\delta(claim3, claim4)$	260ms	Record Claim, Check Claim, Reject Claim, Call Customer, Send Declination, Update Cust., Record, Close Claim
$\delta(claim4, claim1)$	232ms	Record Claim, Check Claim, Settle Claim, Calculate Loss Amount, Payout, Recalc. Cust. Contr., Send Confirmation
$\delta(claim4, claim2)$	243ms	Record Claim, Check Claim, Retrieve Add. Data, Check Claim, Reject Claim, Call Customer, Update Cust., Record, Close Claim
$\delta(claim4, claim3)$	216ms	Record Claim, Check Claim, Settle Claim, Calculate Loss Amount, Payout, Recalc. Cust. Contr., Send Confirmation
$\delta(claim4, claim4)$	135ms	-

Figure 6.6: The time needed by the semantic differencing operator for semantic differencing of the pairs of example AGs.

No.	Operation	Ref.	Gen.
1.	Adding a label to an AG	✓	✓
2.	Deleting an unused label from an AG	✓	✓
3.	Inserting an action node between two succeeding nodes	✗	✗
4.	deleting an action node and reconnecting the control flow	✗	✗
5.	Inserting an xor-fragment between two succeeding nodes	✗	✗
6.	Deleting an xor-fragment	✗	✗
7.	Inserting an and-fragment between two succeeding nodes	✗	✗
8.	Deleting an and-fragment	✗	✗
9.	Inserting a cyclic-fragment between two succeeding nodes	✗	✗
10.	Deleting a cyclic-fragment	✗	✗
11.	Inserting a branch into a fragment	✗	✗
12.	Deleting a branch from a fragment	✗	✗

Figure 6.7: Activity graph change operation properties.

and the framework’s instantiation presented in Chapter 8. If a change operation is refining or generalizing, then it is possible to incorporate performance improvements into algorithms that compute solutions (cf. Section 7.5) for special model repair problems as introduced in Section 8.1.

Figure 6.7 overviews the change operations. The different change operations are inspired by previously existing change operations for ADs [MR15, MR18, KR18a] and business process models [KGFE08, KGE09]. Adding a label to an AG (cf. No. 1) adds an unused label to the set of labels of the AG. Label-addition operations are refactoring as they neither add actions nor transitions to AGs. Deleting an unused label from an AG (cf. No. 2) removes the label from the set of labels of the AG. As the label is required to be unused in the AG, label-deletion operations are refactoring. Inserting an action between a preceding and a succeeding node (cf. No. 3) reconnects the control flow from the preceding node to the inserted node and from the inserted node to the succeeding node. Inserting an action between two succeeding nodes can completely change the AG’s modeled traces. Similarly, deleting an action node (cf. No. 4) reconnects the control flow from the deleted node’s predecessor node to the deleted node’s successor node. Operations that delete action nodes between succeeding nodes can completely change the traces modeled by an AG. Inserting an xor-fragment between two succeeding nodes (cf. No. 5) inserts an xor-fragment containing exactly one action between the inserted decision node and the inserted merge node. Therefore, xor-fragment insertion operations are neither refining nor generalizing. The operations for deleting xor-fragments (cf. No. 6) are applicable iff there is exactly one action between the fragment’s decision and merge nodes. Deleting the xor-fragment deletes the action, decision, and merge nodes. The control flow is reconnected from the xor-fragment’s preceding

node to the xor-fragment's succeeding node. Thus, xor-fragment deletion operations are neither refining nor generalizing. The operations for adding and-fragments (cf. No. 7) and cyclic-fragments (cf. No. 9) are defined analogously to the change operations for adding xor-fragments. Thus, they are neither refining nor generalizing. Similarly, the operations for deleting and-fragments (cf. No. 8) and cyclic-fragments (cf. No. 10) are defined analogously to the change operations for deleting xor-fragments. Therefore, they are neither refining nor generalizing. Inserting an action into a fragment (cf. No. 11) adds exactly one action that is connected to the fragment's starting node and the fragment's ending node. This operation can completely change the traces of an AG. Analogously, deleting an action from a fragment (cf. No. 12) deletes an action that is connected to the fragment's starting node and the fragment's ending node. The transitions connecting the deleted action are also deleted. As this operation can completely change the traces of an AG, deleting an action from a fragment is neither refining nor generalizing.

6.4.1 Label-Addition Operations


Label-addition operations with signature $AG \rightarrow AG$	
Parameters	Let $k \in U_N$ be a name representing an action label.
Explanation	The operation $addL_k$ adds the label k to the set of possible labels.
Domain	$(L, N, t, AND, XOR, C, T, l) \in dom(addL_k) \Leftrightarrow k \notin L$
Application	$ \begin{array}{c} (L, N, t, AND, XOR, C, T, l) \\ \downarrow \\ addL_k \\ (L', N, t, AND, XOR, C, T, l) \text{ where } L' = L \cup \{k\} \end{array} $
Example	 <p><i>label-addition operation applications have no effect on the graphical concrete syntax</i></p>

Figure 6.8: Label-addition operations $addL_k$ for all action labels $k \in U_N$.

Figure 6.8 defines the label-addition operations. Each label-addition operation $addL_k$ is parametrized with a name $k \in U_N$ representing an action label. A label-addition operation is applicable to an AG iff the AG does not use the label k . The application of the operation $addL_k$ to an AG adds the action label k to the set of labels of the AG. In the example application (Figure 6.8), the label k is added to the AG.

As illustrated in Figure 6.8, the addition of a label to an AG does neither change the set of nodes nor the set of transitions of the AG. Therefore, label-addition operations are refactoring.

Proposition 6.1. *Let $addL_k$ be a label-addition operation and let ag be an AG such that $addL_k$ is applicable to ag . Then, $traces(addL_k(ag)) = traces(ag)$.*

Proof. Let ag be an AG and let $addL_k$ be a label-addition operation that is applicable to ag . Then, every transition of $addL_k(ag)$ is also a transition of ag and vice versa. Further, every node of $addL_k(ag)$ is also a node of ag and vice versa. Thus, every word accepted by $nfa(addL_k(ag))$ is also accepted by $nfa(ag)$ and vice versa. Therefore, $traces(addL_k(ag)) = traces(ag)$. \square

6.4.2 Label-Deletion Operations

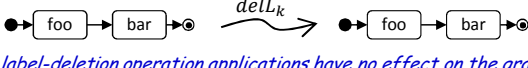
Label-deletion operations with signature $AG \rightarrow AG$	
Parameters	Let $k \in U_N$ be a name representing an action label.
Explanation	The operation $delL_k$ deletes the label k from the set of possible labels.
Domain	$(L, N, t, AND, XOR, C, T, l) \in dom(delL_k) \Leftrightarrow k \in L \wedge \forall n \in A: l(n) \neq k$
Application	$ \begin{array}{c} (L, N, t, AND, XOR, C, T, l) \\ \downarrow \\ (L', N, t, AND, XOR, C, T, l) \text{ where } L' = L \setminus \{k\} \end{array} $
Example	 <p><i>label-deletion operation applications have no effect on the graphical concrete syntax</i></p>

Figure 6.9: Label-deletion operations $delL_k$ for all action labels $k \in U_N$.

Figure 6.9 defines the label-deletion operations. Each label-deletion operation $delL_k$ is parametrized with a name $k \in U_N$ representing an action label. A label-deletion operation is applicable to an AG iff the set of labels of the AG contains the label and no action of the AG is labeled with the label. The application of the operation $delL_k$ to an AG removes the action label k from the set of labels of the AG. In the example application in Figure 6.8, the label k is deleted from the AG.

As illustrated in Figure 6.9, the deletion of a label from an AG neither changes the sets of nodes nor the set of transitions of the AG. Therefore, label-deletion operations are refactoring.

Proposition 6.2. *Let $delL_k$ be a label-deletion operation and let ag be an AG such that $delL_k$ is applicable to ag . Then, $traces(delL_k(ag)) = traces(ag)$.*

Proof. Let ag be an AG and let $delL_k$ be a label-deletion operation that is applicable to ag . Then, every transition of $delL_k(ag)$ is also a transition of ag and vice versa. Further, every node of $delL_k(ag)$ is also a node of ag and vice versa. Thus, every word accepted by $nfa(delL_k(ag))$ is also accepted by $nfa(ag)$ and vice versa. Therefore, $traces(delL_k(ag)) = traces(ag)$. \square

6.4.3 Action-Insertion Operations

Action-insertion operations with signature $AG \rightarrow AG$	
Parameters	Let $x, y, a \in U_N$ be three names representing nodes and $k \in U_N$ be a name representing an action label.
Explanation	The operation $addA_{x,y,a,k}$ adds the action node a with label k between the node x and its succeeding node y .
Domain	$(L, N, t, AND, XOR, C, T, l) \in dom(addA_{x,y,a,k}) \Leftrightarrow (x, y) \in T \wedge k \in L \wedge a \notin N$
Application	$ \begin{array}{c} (L, N, t, AND, XOR, C, T, l) \\ \downarrow \\ addA_{x,y,a,k} (L, N', t', AND, XOR, C, T', l') \text{ where } N' = N \cup \{a\} \\ t' = A \cup \{a: A\} \\ T' = (T \setminus \{x, y\}) \cup \{(x, a), (a, y)\} \\ l' = l \cup \{a: k\} \end{array} $
Example	

Figure 6.10: Action-insertion operations $addA_{x,y,a,k}$ for all node names $x, y, a \in U_N$ and action labels $k \in U_N$.

Figure 6.10 depicts the definition of the action-insertion operations. Each action-insertion operation $addA_{x,y,a,k}$ is parametrized with three names $x, y, a \in U_N$ representing node names and a name $k \in U_N$ representing an action label. An action-insertion operation is applicable to an AG iff the AG contains the nodes x and y , a transition from the node x to the node y , the action label k , and the AG does not contain the node a . The application of the operation $addA_{x,y,a,k}$ inserts the action node a labeled with k between the nodes x and y . To this effect, the operation adds a transition from the node x to the node a and a transition from the node a to the node y . The transition from the node x to the node y is removed from the AG. In the example application (Figure 6.10), the node a with label k is inserted between the nodes x and y .

As illustrated in Figure 6.10, the insertion of an action can completely change the traces of an AG. For instance, `foo, bar` is a trace of the AG depicted on the left-hand side in the example of Figure 6.10, but no trace of the AG depicted on the right-hand side. The trace `foo, k, bar` is a trace of the AG depicted on the right-hand side, but no trace of the AG depicted on the left-hand side. The semantics of the AGs are incomparable.

6.4.4 Action-Deletion Operations

Figure 6.11 defines the action-deletion operations. Each action-deletion operation $delA_a$ is parametrized with a name $a \in U_N$ representing an action node. The action-deletion operation is applicable to an AG iff the AG contains the action node a . The application of the operation $delA_a$ deletes the action node a from the AG and reconnects the control

Action-deletion operations with signature $AG \rightarrow AG$	
Parameters	Let $a \in U_N$ be a name representing an action node.
Explanation	The operation $delA_a$ deletes the action node a .
Domain	$(L, N, t, AND, XOR, C, T, l) \in dom(delA_a) \Leftrightarrow t(a) = A$
Application	$ \begin{aligned} &(L, N, t, AND, XOR, C, T, l) \\ &\quad \downarrow delA_a \\ &(L, N', t', AND, XOR, C, T', l') \text{ where } N' = N \setminus \{a\} \\ &\quad t' = t \setminus \{a: A\} \\ &\quad T' = (T \cap (N' \times N')) \cup \{(n, m) \in N \times N \mid (n, a) \in T \wedge (a, m) \in T\} \\ &\quad l' = \{n: l(n) \mid n \in N'\} \end{aligned} $
Example	

 Figure 6.11: Action-deletion operations $delA_a$ for all action node names $a \in U_N$.

flow from the action node's predecessor nodes to the action node's successor nodes. To this effect, the operation deletes the transitions originating from and leading to the node a and adds a transition from the predecessor nodes of a to the successor nodes of a . In the example application (Figure 6.11), the node a is deleted from the AG.

As illustrated in Figure 6.11, the deletion of an action from an AG can completely change the traces of the AG. For instance, **foo, k, bar** is a trace of the AG depicted on the left-hand side in the example of Figure 6.11, but no trace of the AG depicted on the right-hand side. The trace **foo, bar** is a trace of the AG depicted on the right-hand side, but no trace of the AG depicted on the left-hand side.

6.4.5 Xor-Fragment-Insertion Operations

Figure 6.12 defines the xor-fragment-insertion operations. Each xor-fragment-insertion operation $addXor_{x,y,d,m,a,k}$ is parametrized with five node names $x, y, d, m, a \in U_N$. The node names x and y represent arbitrary connected nodes. The node name d represents a decision node, the node name m represents a merge node, and the node name a represents an action node. The parameter k represents an action label. An xor-fragment-insertion operation is applicable to an AG iff the AG contains the nodes x and y , there exists a transition from the node x to node y , the node names d, m, a are not used by the AG, and the label k is contained in the set of labels of the AG. The application of the operation $addXor_{x,y,d,m,a,k}$ inserts an xor-fragment with the decision node d and the merge node m between the nodes x and y . The operation further adds the action node a with label k , adds a transition from the decision node d to the action node a , and adds a transition from the action node a to the merge node m . The operation adds a transition from the

Xor-fragment-insertion operations with signature $AG \rightarrow AG$	
Parameters	Let $x, y, d, m, a \in U_N$ be node names with $d \neq m, d \neq a, m \neq a$ and let $k \in U_N$ be an action label.
Explanation	The operation $addXor_{x,y,d,m,a,k}$ inserts the xor-fragment (d, m) with action a labeled k between x and y .
Domain	$(L, N, t, AND, XOR, C, T, l) \in dom(addXor_{x,y,d,m,a,k}) \Leftrightarrow (x, y) \in T \wedge d, m, a \notin N \wedge k \in L$
Application	$ \begin{array}{c} (L, t, AND, XOR, C, T, l) \\ \downarrow \\ (L, N', t', AND, XOR', C, T', l') \text{ where } N' = N \cup \{d, m, a\} \\ t' = t \cup \{a: A, d: D, m: M\} \\ XOR' = XOR \cup \{(d, m)\} \\ T' = (T \setminus \{(x, y)\}) \cup \{(x, d), (d, a), (a, m), (m, y)\} \\ l' = l \cup \{d: \varepsilon, m: \varepsilon, a: k\} \end{array} $
Example	

Figure 6.12: Xor-fragment-insertion operations $addXor_{x,y,d,m,a,k}$ for node names $x, y, d, m, a \in U_N$ and action labels $k \in U_N$.

node x to the node d and a transition from the node m to the node y to reconnect the control flow. The transition from the node x to the node y is removed from the AG. In the example application (Figure 6.12), an xor-fragment defined by the decision node d and the merge node m with action a labeled k is added between the nodes x and y .

As illustrated in Figure 6.12, xor-fragment-insertion operations are neither refining nor generalizing. For instance, `foo, bar` is a trace of the AG depicted on the left-hand side in the example, but no trace of the AG depicted on the right-hand side. The trace `foo, k, bar` is a trace of the AG depicted on the right-hand side, but no trace of the AG depicted on the left-hand side. The sets of traces of the AGs are incomparable.

6.4.6 Xor-Fragment-Deletion Operations

Figure 6.13 defines the xor-fragment-deletion operations. Each xor-fragment-deletion operation $delXor_{d,m,a}$ is parametrized with a node name $d \in U_N$ representing a decision node, a node name $m \in U_N$ representing a merge node, and a node name $a \in U_N$ representing an action node. An xor-fragment-deletion operation is applicable to an AG iff the AG contains the xor-fragment defined by the decision node d and the merge node m and the fragment exactly contains the action node a , connected via transitions to the nodes d and m . The application of the operation $delXor_{d,m,a}$ deletes the xor-fragment defined by the decision node d and the merge node m and reconnects the control flow from the decision node's predecessor node to the merge node's successor node. To achieve this, the operation removes d, m , and a from the sets of nodes, removes the fragment from the AG's xor-fragments, removes the transition from the decision node's predecessor node

Xor-fragment-deletion operations with signature $AG \rightarrow AG$	
Parameters	Let $d, m, a \in U_N$ be names representing nodes.
Explanation	The operation $delXor_{d,m,a}$ deletes the xor-fragment (d, m) with action a .
Domain	$(L, N, t, AND, XOR, C, T, l) \in dom(delXor_{d,m,a}) \Leftrightarrow$ $(d, a), (a, m) \in T \wedge t(a) = A \wedge$ $(d, m) \in XOR \wedge$ $(\forall x \in N: (d, x) \in T \Rightarrow x = a) \wedge$ $(\forall x \in N: (x, m) \in T \Rightarrow x = a)$
Application	$delXor_{d,m,a} \begin{pmatrix} (L, N, t, AND, XOR, C, T, l) \\ \downarrow \\ (L, N', t', AND, XOR', C, T', l') \end{pmatrix}$ where $N' = N \setminus \{d, m, a\}$ $l' = l \setminus \{a: A, d: D, m: M\}$ $XOR' = XOR \setminus \{(d, m)\}$ $T' = (T \cup \{(x, y) \mid (x, d), (m, y) \in T\}) \cap N' \times N'$ $l' = \{x: l(x) \mid x \in N'\}$
Example	

 Figure 6.13: Xor-fragment-deletion operations $delXor_{d,m,a}$ for node names $d, m, a \in U_N$.

to the decision node, removes the transition from the decision node to the action node, removes the transition from the action node to the merge node, removes the transition from the merge node to its successor node, and adds a transition from the decision node's predecessor node to the merge node's successor node. In the example application (Figure 6.13), the xor-fragment defined by the decision node d and the merge node m containing the action node a is removed and the control flow is reconnected, accordingly.

As illustrated in Figure 6.13, xor-fragment-deletion operations are neither refining nor generalizing. For instance, `foo, k, bar` is a trace of the AG depicted on the left-hand side in the example, but no trace of the AG depicted on the right-hand side. The trace `foo, bar` is a trace of the AG depicted on the right-hand side, but no trace of the AG depicted on the left-hand side. The sets of traces of the AGs are incomparable.

6.4.7 And-Fragment-Insertion Operations

Figure 6.14 defines the and-fragment-insertion change operations. Each and-fragment-insertion operation $addAnd_{x,y,f,j,a,k}$ is parametrized with five node names $x, y, f, j, a \in U_N$ and an action label $k \in U_N$. The node names x and y represent arbitrary connected nodes. The node name f represents a fork node name, the node name j represents a join node name, and the node name a represents an action node name. An and-fragment-insertion operation is applicable to an AG iff the AG contains the nodes x and y , there exists a transition from the node x to node y , the node names f, j , and a are not used in the AG, and the label k is contained in the set of labels of the AG. Applying the operation $addAnd_{x,y,f,j,a,k}$ inserts an and-fragment with the fork node f and the join

And-fragment-insertion operations with signature $AG \rightarrow AG$	
Parameters	Let $x, y, f, j, a \in U_N$ be node names with $f \neq j$, $f \neq a$, $j \neq a$ and let $k \in U_N$ be an action label.
Explanation	The operation $addAnd_{x,y,f,j,a,k}$ inserts the and-fragment (f, j) with action a labeled k between x and y .
Domain	$(L, N, t, AND, XOR, C, T, l) \in dom(addAnd_{x,y,f,j,a,k}) \Leftrightarrow (x, y) \in T \wedge f, j, a \notin N \wedge k \in L$
Application	$ \begin{array}{c} (L, N, t, AND, XOR, C, T, l) \\ \downarrow addAnd_{x,y,f,j,a,k} \\ (L, N', t', AND', XOR, C, T', l') \text{ where } N' = N \cup \{d, m, a\} \\ t' = t \cup \{a: A, f: F, j: J\} \\ AND' = AND \cup \{(f, j)\} \\ T' = (T \setminus \{(x, y)\}) \cup \{(x, f), (f, a), (a, j), (j, y)\} \\ l' = l \cup \{f: \varepsilon, j: \varepsilon, a: k\} \end{array} $
Example	

Figure 6.14: And-fragment-insertion operations $addAnd_{x,y,f,j,a,k}$ for node names $x, y, f, j, a \in U_N$ and action labels $k \in U_N$.

node j containing the action a labeled k between the nodes x and y . To this effect, the operation adds the fork node f , the join node j , the and-fragment defined by the fork and the join node, the action node a labeled k , a transition from the node x to the fork node f , a transition from the fork node f to the action node a , a transition from the action node a to the join node j , and a transition from the join node j to the node y . The transition from the node x to the node y is removed from the AG. In the example application (Figure 6.14), an and-fragment defined by the fork node f and the join node j containing the action a labeled k is added between the nodes x and y .

And-fragment-insertion operations are neither refining nor generalizing. For instance, `foo, bar` is a trace of the AG depicted on the left-hand side in Figure 6.14, but no trace of the AG depicted on the right-hand side. The trace `foo, k, bar` is a trace of the AG depicted on the right-hand side, but no trace of the AG depicted on the left-hand side.

6.4.8 And-Fragment-Deletion Operations

Figure 6.15 defines the and-fragment-deletion operations. Each and-fragment-deletion operation $delAnd_{f,j,a}$ is parametrized with a fork node name $f \in U_N$, a join node name $j \in U_N$, and an action node name $a \in U_N$. An and-fragment-deletion operation is applicable to an AG iff the AG contains the and-fragment defined by the fork node f and the join node j , and the fragment contains exactly the action node a . The application of the operation $delAnd_{f,j,a}$ deletes the and-fragment defined by the fork node f and the join node j and reconnects the control flow from the fork node's predecessor node to the join node's successor node. To this effect, the operation removes f , j , and a

And-fragment-deletion operations with signature $AG \rightarrow AG$	
Parameters	Let $f, j, a \in U_N$ be names representing nodes.
Explanation	The operation $delAnd_{f,j,a}$ deletes the and-fragment (f, j) with action a .
Domain	$(L, N, t, AND, XOR, C, T, l) \in dom(delAnd_{f,j,a}) \Leftrightarrow (f, a), (a, j) \in T \wedge t(a) = A \wedge$ $(f, j) \in AND \wedge$ $(\forall x \in N: (f, x) \in T \Rightarrow x = a) \wedge$ $(\forall x \in N: (x, j) \in T \Rightarrow x = a)$
Application	$delAnd_{f,j,a} \begin{pmatrix} (L, N, t, AND, XOR, C, T, l) \\ \downarrow \\ (L, N', t', AND', XOR, C, T', l') \end{pmatrix}$ where $N' = N \setminus \{f, j, a\}$ $t' = t \setminus \{a: A, f: F, j: J\}$ $AND' = AND \setminus \{(f, j)\}$ $T' = (T \cup \{(x, y) \mid (x, f), (j, y) \in T\}) \cap N' \times N'$ $l' = \{x: l(x) \mid x \in N'\}$
Example	

 Figure 6.15: And-fragment-deletion operations $delAnd_{f,j,a}$ for node names $f, j, a \in U_N$.

from the sets of nodes, removes the fragment from the AG's and-fragments, removes the transition from the fork node's predecessor node to the fork node, removes the transition from the fork node to the action node, removes the transition from the action node to the join node, removes the transition from the join node to its successor node, and adds transitions from the fork node's predecessor nodes to the join node's successor nodes. In the example application (Figure 6.13), the and-fragment defined by the fork node f and the join node j containing the action a is removed.

As illustrated in the example of Figure 6.15, and-fragment-deletion operations are neither refining nor generalizing. For instance, **foo**, **k**, **bar** is a trace of the AG depicted on the left-hand side in the example, but no trace of the AD depicted on the right-hand side. The trace **foo**, **bar** is a trace of the AG depicted on the right-hand side, but no trace of the AG depicted on the left-hand side.

6.4.9 Cyclic-Fragment-Insertion Operations

Figure 6.16 depicts the definition of the cyclic-fragment-insertion operations. Each cyclic-fragment-insertion operation $addC_{x,y,d,m,a,k}$ is parametrized with five node names $x, y, d, m, a \in U_N$ and an action label $k \in U_N$. The node names x and y represent arbitrary connected nodes. The node name d represents a decision node name, the node name m represents a merge node name, and the node name a represents an action node name. A cyclic-fragment-insertion operation is applicable to an AG iff the AG contains the nodes x, y , there exists a transition from the node x to the node y , the node names

Cyclic-fragment-insertion operations with signature $AG \rightarrow AG$	
Parameters	Let $x, y, d, m, a \in U_N$ be node names with $d \neq m$, $d \neq a$, $m \neq a$ and let $k \in U_N$ be an action label.
Explanation	The operation $addC_{x,y,d,m,a,k}$ inserts the cyclic-fragment (d, m) with action a labeled k between x and y .
Domain	$(L, N, t, AND, XOR, C, T, l) \in dom(addC_{x,y,d,m,a,k}) \Leftrightarrow (x, y) \in T \wedge d, m, a \notin N \wedge k \in L$
Application	$ \begin{array}{c} (L, N, t, AND, XOR, C, T, l) \\ \downarrow \\ (L, N', t', AND, XOR, C', T', l') \text{ where } N' = N \cup \{d, m, a\} \\ t' = t \cup \{a: A, d: D, m: M\} \\ C' = C \cup \{(d, m)\} \\ T' = (T \setminus \{(x, y)\}) \cup \{(x, m), (m, a), (a, d), (d, m), (d, y)\} \\ l' = l \cup \{d: \varepsilon, m: \varepsilon, a: k\} \end{array} $
Example	

Figure 6.16: Cyclic-fragment-insertion operations $addC_{x,y,d,m,a,k}$ for node names $x, y, d, m, a \in U_N$ and action labels $k \in U_N$.

d , m , and a are not used in the AG, and the label k is contained in the set of labels of the AG. Applying the operation $addC_{x,y,d,m,a,k}$ inserts a cyclic-fragment with the merge node m and the decision node d containing the action a labeled k between the nodes x and y . The operation adds the merge node m , the decision node d , the cyclic-fragment defined by the merge and the decision node, the action node a , a transition from the node x to the merge node m , a transition from the merge node m to the action node a , a transition from the action node a to the decision node d , a transition from the decision node d to the merge node m , and a transition from the decision node d to the node y . The transition from the node x to the node y is removed from the AG. In the example application depicted in Figure 6.16, a cyclic-fragment defined by the merge node m and the decision node d containing the action a labeled k is added between the node x and the node y .

As illustrated in Figure 6.16, cyclic-fragment-insertion operations are neither refining nor generalizing. For instance, **foo**, **bar** is a trace of the AG depicted on the left-hand side in the example, but no trace of the AG depicted on the right-hand side. The trace **foo**, k , **bar** is a trace of the AG depicted on the right-hand side, but no trace of the AG depicted on the left-hand side. The sets of traces of the AGs are incomparable.

6.4.10 Cyclic-Fragment-Deletion Operations

Figure 6.17 depicts the definition of the cyclic-fragment-deletion operations. Each cyclic-fragment-deletion operation $delC_{d,m,a}$ is parametrized with a decision node name $d \in U_N$, a merge node name $m \in U_N$, and an action node name $a \in U_N$. A cyclic-fragment-

Cyclic-fragment-deletion operations with signature $AG \rightarrow AG$	
Parameters	Let $d, m, a \in U_N$ be names representing nodes.
Explanation	The operation $delC_{d,m,a}$ deletes the cyclic-fragment (d, m) with action a .
Domain	$(L, N, t, AND, XOR, C, T, l) \in dom(delC_{d,m,a}) \Leftrightarrow (m, a), (a, d), (d, m) \in T \wedge t(a) = A \wedge$ $(d, m) \in C \wedge$ $ \{(x, y) \in T \mid y = m\} = 2 \wedge$ $ \{(x, y) \in T \mid x = m\} = 1 \wedge$ $ \{(x, y) \in T \mid y = d\} = 1 \wedge$ $ \{(x, y) \in T \mid x = d\} = 2$
Application	$delC_{d,m,a} (L, N, t, AND, XOR, C, T, l)$ \downarrow $(L, N', t', AND, XOR, C', T', l')$ where $N' = N \setminus \{d, m, a\}$ $t' = t \setminus \{a: A, d: D, m: M\}$ $C' = C \setminus \{(d, m)\}$ $T' = (T \cup \{(x, y) \mid (x, m), (d, y) \in T\}) \cap N' \times N'$ $l' = \{x: l(x) \mid x \in N'\}$
Example	

 Figure 6.17: Cyclic-fragment-deletion operations $delC_{d,m,a}$ for node names $d, m, a \in U_N$.

deletion operation is applicable to an AG iff the AG contains the cyclic-fragment defined by the decision node d and the merge node m and the fragment exactly contains the action a . This condition is ensured by requiring that d and m define a cyclic fragment, the AD contains a transition from the merge node m to the action node a , a transition from the action node a to the decision node d , a transition from the decision node d to the merge node m , by requiring that the merge node has exactly two incoming transition as well as exactly one outgoing transition, and by requiring that the decision node has exactly one incoming transition and exactly two outgoing transitions. The application of the operation $delC_{d,m,a}$ deletes the cyclic-fragment defined by the decision node d and the merge node m and reconnects the control flow from the merge node's predecessor node to the decision node's successor node. The operation removes d , m , and a from the sets of nodes, removes the fragment from the AG's cyclic-fragments, removes the transition from the merge node's predecessor node to the merge node, removes the transition from the merge node to the action node, removes the transition from the action node to the decision node, removes the transition from the decision node to the merge node, removes the transitions from the merge node to its successor nodes, and adds transitions from the merge node's predecessor nodes to the decision node's successor nodes.

In the example application depicted in Figure 6.17, the cyclic-fragment defined by the decision node d and the merge node m containing the action a is removed. As illustrated in the example of Figure 6.17, cyclic-fragment-deletion operations are neither refining

nor generalizing. For instance, `foo, k, bar` is a trace of the AG depicted on the left-hand side in the example, but no trace of the AG depicted on the right-hand side. The trace `foo, bar` is a trace of the AG depicted on the right-hand side, but no trace of the AG depicted on the left-hand side.

6.4.11 Fragment-Branch-Insertion Operations

Fragment-branch-insertion operations with signature $AG \rightarrow AG$	
Parameters	Let $n, m \in U_N$ be node names.
Explanation	The operation $addFB_{n,m}$ adds a transition from n to m if (n, m) defines an xor-, an and-, or a cyclic-fragment.
Domain	$(L, N, t, AND, XOR, C, T, l) \in dom(addFB_{n,m}) \Leftrightarrow (n, m) \in XOR \cup C \cup AND \wedge (n, m) \notin T$
Application	$ \begin{array}{c} (L, N, t, AND, XOR, C, T, l) \\ \downarrow \\ (L, N, t, AND, XOR, C, T', l) \text{ where } T' = T \cup \{(n, m)\} \end{array} $
Example	

Figure 6.18: Fragment-branch-insertion operations $addFB_{n,m}$ for nodes $n, m \in U_N$.

Figure 6.18 defines the fragment-branch-insertion operations. Each fragment-branch-insertion operation $addFB_{n,m}$ is parametrized with two node names $n, m \in U_N$. A fragment-branch-insertion operation $addFB_{n,m}$ is applicable to an AG iff the AG contains a fragment (n, m) and the AG contains no transitions from n to m . The application of the operation adds the transition (n, m) to the AG. In the example depicted in Figure 6.18, the change operation inserts the transition (n, m) into the xor-fragment (n, m) .

Fragment-branch-insertion operations are neither refining nor generalizing. For example, `foo, bar` is a trace of the AG depicted on the right-hand side in the example of Figure 6.18, but no trace of the AG depicted on the left-hand side. Thus, fragment-branch-insertion operations are not refining. The trace `foo, baz, bar` is a trace of the AG depicted on the left-hand side, but no trace of the other AG. The trace `foo` is a trace of the AG `ag1` depicted on the left-hand side of Figure 6.19 and no trace of the AG `ag2` depicted on the right-hand side. The AG `ag2` can be obtained from the AG `ag1` by applying a fragment-branch-insertion operation. Thus, fragment-branch-insertion operations are not generalizing.

6.4.12 Fragment-Branch-Deletion Operations

Figure 6.20 defines the fragment-branch-deletion operations. Each fragment-branch-deletion operation $delFB_{n,m}$ is parametrized with two node names $n, m \in U_N$. The node

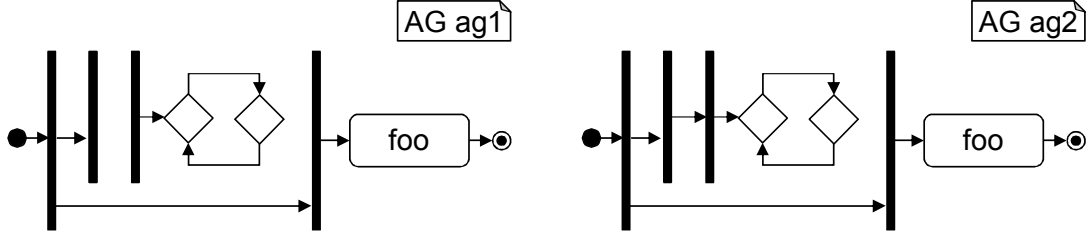


Figure 6.19: The AG *ag2* can be obtained from the AG *ag1* by applying a fragment-branch-insertion operation. The AG *ag1* can be obtained from the AG *ag2* by applying a fragment-branch-deletion operation.

Fragment-branch-deletion operations with signature $AG \rightarrow AG$	
Parameters	Let $n, m, \in U_N$ be node names.
Explanation	The operation $delFB_{n,m}$ deletes the transition (n, m) if (n, m) defines an xor-, an and-, or a cyclic-fragment.
Domain	$(L, N, t, AND, XOR, C, T, l) \in dom(delFB_{n,m}) \Leftrightarrow (n, m) \in XOR \cup C \cup AND \wedge (n, m) \in T$
Application	$ \begin{array}{c} (L, N, t, AND, XOR, C, T, l) \\ \downarrow \\ delFB_{n,m} \\ (L, N, t, AND, XOR, C, T', l) \text{ where } T' = T \setminus \{(n, m)\} \end{array} $
Example	

Figure 6.20: Fragment-branch-deletion operations $delFB_{n,m}$ for node names $n, m \in U_N$.

names n and m represent a fragment defined by (n, m) . The fragment-branch-deletion operation $delFB_{n,m}$ is applicable to an AG iff the AG contains a fragment defined by (n, m) and a transition from n to m . The application of the operation $delFB_{n,m}$ deletes the the transition (n, m) . In the example depicted in Figure 6.20, the change operation deletes the transition (n, m) .

Fragment-branch-deletion operations are neither refining nor generalizing. For example, **foo, bar** is a trace of the AG depicted on the left-hand side in the example of Figure 6.20, but no trace of the AG depicted on the right-hand side. Thus, fragment-branch-deletion operations are not generalizing. The AG *ag2* depicted on the right-hand side of Figure 6.19 is inconsistent. The AG *ag1* depicted on the left-hand side is consistent. The AG *ag1* can be obtained from the AG *ag2* by applying a fragment-branch-deletion operation. Thus, fragment-branch-insertion operations are not refining.

6.5 Activity Diagram Modeling Language

This section defines the AD language $\mathcal{L}_{AD} = (M_{AD}, Sem_{AD}, \llbracket \cdot \rrbracket^{AD})$ with a syntax based on AGs as introduced in Section 6.1, a semantic mapping based on the AG semantics presented in Section 6.2, and a complete change operation suite O_{AD} based on the change operations introduced in Section 6.4.

Let $i, f \in U_N$ with $i \neq f$ be two arbitrary but fixed node names. The names are used as unique identifiers for the initial and final nodes of ADs. The set of ADs M_{AD} is recursively defined as follows:

- The AG $(\emptyset, \{i, f\}, t, \emptyset, \emptyset, \emptyset, \{(i, f)\}, \{i : \varepsilon, f : \varepsilon\}) \in M_{AD}$ with $t(i) = i$ and $t(f) = f$ is an AD.
- If $ad \in M_{AD}$ is an AD and o is a label-addition-operation that is applicable to ad , then $o(ad) \in M_{AD}$ is an AD.
- If $ad \in M_{AD}$ is an AD and o is an action-insertion operation that is applicable to ad , then $o(ad) \in M_{AD}$ is an AD.
- If $ad \in M_{AD}$ is an AD and o is a xor-fragment-insertion operation that is applicable to ad , then $o(ad) \in M_{AD}$ is an AD.
- If $ad \in M_{AD}$ is an AD and o is an and-fragment-insertion operation that is applicable to ad , then $o(ad) \in M_{AD}$ is an AD.
- If $ad \in M_{AD}$ is an AD and o is a cyclic-fragment-insertion operation that is applicable to ad , then $o(ad) \in M_{AD}$ is an AD.
- If $ad \in M_{AD}$ is an AD and o is a fragment-branch-insertion operation that is applicable to ad , then $o(ad) \in M_{AD}$ is an AD.
- If $ad \in M_{AD}$ is an AD and o is a fragment-branch-deletion operation that is applicable to ad , then $o(ad) \in M_{AD}$ is an AD.

The semantic domain of the AD modeling language is defined as $Sem_{AD} \stackrel{\text{def}}{=} U_N^*$ the set of all possible action traces. The semantics of an AD $ad \in M_{AD}$ is defined as the set $\llbracket ad \rrbracket^{AD} \stackrel{\text{def}}{=} traces(ag)$ of its traces. The set O_{AD} of AD change operations is defined as the smallest set that satisfies: if o is an AG change operation (cf. Section 6.4), then $p \in O_{AD}$ is an AD change operation, where p is defined as follows:

$$p : M_{AD} \rightarrow M_{AD}, ad : \begin{cases} o(ad), & \text{if } ad \in dom(o) \wedge o(ad) \in M_{AD} \\ \perp, & \text{otherwise .} \end{cases}$$

The change operation suite O_{AD} is complete for the AD modeling language \mathcal{L}_{AD} because every AD change operation used in the recursive syntax definition for ADs has an inverse:

Proposition 6.3. *The change operation suite O_{AD} is complete for the AD modeling language \mathcal{L}_{AD} .*

Proof. We first show that every AD change operation used in the recursive definition of the AD syntax has an inverse, *i.e.*, for each AD ad and each AD change operation o used in the recursive definition of the AD syntax such that o is applicable to ad , there exists an AD change operation $p \in O_{AD}$ such that p is applicable to $o(ad)$ and $p(o(ad)) = ad$. Let $ad \in M_{AD}$ be an AD.

Let $addL_k$ with $k \in U_N$ be a label-addition operation that is applicable to ad . Then, for the label-deletion operation $delL_k$, it holds by the definitions of the operations that $addL_k(ad) \in \text{dom}(delL_k)$ and $delL_k(addL_k(ad)) = ad$.

Let $addA_{x,y,a,k}$ with $x, y, a, k \in U_N$ be an action-insertion operation that is applicable to ad . Then, for the action-deletion operation $delA_a$, it holds by the definitions of the operations that $addA_{x,y,a,k}(ad) \in \text{dom}(delA_a)$ and $delA_a(addA_{x,y,a,k}(ad)) = ad$.

Let $addXor_{x,y,d,m,a,k}$ with $x, y, d, m, a, k \in U_N$ such that $d \neq m$, $d \neq a$, and $m \neq a$ be a xor-fragment-insertion operation that is applicable to ad . Then, for the xor-fragment-deletion operation $delXor_{d,m,a}$, it holds by the definitions of the operations that $addXor_{x,y,d,m,a,k}(ad) \in \text{dom}(delXor_{d,m,a})$ and $delXor_{d,m,a}(addXor_{x,y,d,m,a,k}(ad)) = ad$.

Let $addAnd_{x,y,f,j,a,k}$ with $x, y, f, j, a, k \in U_N$ such that $f \neq j$, $f \neq a$, and $j \neq a$ be an and-fragment-insertion operation that is applicable to ad . For the and-fragment-deletion operation $delAnd_{f,j,a}$, it follows from the definitions of the operations that $addAnd_{x,y,f,j,a,k}(ad) \in \text{dom}(delAnd_{f,j,a})$ and $delAnd_{f,j,a}(addAnd_{x,y,f,j,a,k}(ad)) = ad$.

Let $addC_{x,y,d,m,a,k}$ with $x, y, d, m, a, k \in U_N$ such that $d \neq m$, $d \neq a$, and $m \neq a$ be a cyclic-fragment-insertion operation that is applicable to ad . Then, for the cyclic-fragment-deletion operation $delC_{d,m,a}$, it holds by the definitions of the operations that $addC_{x,y,d,m,a,k}(ad) \in \text{dom}(delC_{d,m,a})$ and $delC_{d,m,a}(addC_{x,y,d,m,a,k}(ad)) = ad$.

Let $addFB_{n,m}$ with $n, m \in U_N$ be a fragment-branch-insertion operation that is applicable to ad . Then, for the fragment-branch-deletion operation $delFB_{n,m}$, it holds that $addFB_{n,m}(ad) \in \text{dom}(delFB_{n,m})$ and $delFB_{n,m}(addFB_{n,m}(ad)) = ad$.

Let $delFB_{n,m}$ with $n, m \in U_N$ be a fragment-branch-deletion operation that is applicable to ad . Then, for the fragment-branch-insertion operation $addFB_{n,m}$, it holds that $delFB_{n,m}(ad) \in \text{dom}(addFB_{n,m})$ and $addFB_{n,m}(delFB_{n,m}(ad)) = ad$.

The above enables showing that the change operation suite O_{AD} is complete for the AD modeling language \mathcal{L}_{AD} : Let $S \stackrel{\text{def}}{=} (\emptyset, \{i, f\}, t, \emptyset, \emptyset, \emptyset, \{(i, f)\}, \{i : \varepsilon, f : \varepsilon\}) \in M_{AD}$ where $i, f \in U_N$ denote the smallest AD. Let $ad, ad' \in M_{AD}$ be two arbitrary ADs. Then, by definition of the syntax of ADs, there exist sequences of change operations $t = o_1, \dots, o_n \in O_{AD}$ with $n \geq 0$ and $u = p_1, \dots, p_m \in O_{AD}$ with $m \geq 0$ such that $S \triangleright t = ad$, $S \triangleright u = ad'$, and o_j as well as p_k are label-addition operations, action-insertion

operations, xor-fragment-insertion operations, and-fragment-insertion operations, cyclic-fragment-insertion operations, or fragment-branch-insertion operations for all $0 < j \leq n$ and $0 < k \leq m$. As each change operation o_j for $0 < j \leq n$ has an inverse operation, there exists a change sequence $v = q_1, \dots, q_n$ with $|v| = |t|$ such that $(S \triangleright t) \triangleright v = S$. From the above, we conclude that $ad \triangleright vu = ((S \triangleright t) \triangleright v) \triangleright u = S \triangleright u = ad'$. As ad and ad' are chosen arbitrarily, this implies that there exists at least one function $\Delta_{AD} : M_{AD} \times M_{AD} \rightarrow O_{AD}^*$ such that $\forall ad, ad' \in M_{AD} : ad \triangleright \Delta(ad, ad') = ad'$. \square

6.6 Related Work

In contrast to previous works on semantic differencing and model checking of ADs [Esh06, MRR11b, MRR11c], the method presented in this thesis explicitly maps ADs to finite automata. This thesis does not define an implicit mapping via a translation from ADs to models in the input format of a model checker, which again encode finite automata having a semantics based on recognized words. Through the explicit mapping, our translation is less complex, which reduces implementation efforts.

Semantic differencing of ADs is introduced in [MRR11b]. In this approach, the input ADs are translated to models in the input language of the SMV model checker [McM93] using the translation described in [MRR11c]. These models then encode finite automata that correspond to the ADs. The translation from ADs to SMV models is rather complicated. Further, the translation steps from ADs to SMV models and from SMV models to finite automata causes additional overhead compared to directly translating ADs to finite automata. In contrast, the translation from ADs to finite automata presented in this thesis is direct and simple. However, in contrast to our translation, the approach presented in [MRR11b] supports immutable input variables and local variables over finite domains as well as assignment expressions for changing the values of local variables on actions and guards on transitions leaving decision nodes. Thus, although the translation of [MRR11b] is more complex, the language supported by [MRR11b] is more expressive as it supports variables. This thesis focuses on a direct and simple translation without incorporating variables, although extending the approach of this thesis to support variables is straight forward: The algorithm for the translation from ADs to NFAs must include a value assignment for the variables in the states of the resulting NFAs. Whenever a transition that enters an action node is processed, the algorithm must evaluate the assignment expressions of the action given the variable assignment of the currently processed state as input and incorporate the resulting variable assignment in the target state of the added NFA transition. Whenever a transition that leaves a decision node is processed, the algorithm must evaluate the boolean guard of the transition and only add the corresponding NFA transition and the corresponding NFA state if the guard evaluates to true when given the variable assignment of the currently processed state as input. Using variables increases the state space of the resulting NFA. Interesting

future work is extending and implementing the translation of this thesis with variable assignments and comparing the runtime of the resulting semantic differencing operator with the runtime of the semantic differencing operator presented in [MRR11b].

The translation from ADs to SMV models of [MRR11b] is similar to a translation defined in [Esh06]. The scope of [Esh06] is symbolic model checking of ADs, whereas this thesis focuses on semantic differencing.

The framework presented in [MRR11b] has been extended in [MRR11g] for summarizing elements in the semantic difference from one AD to another AD based on equivalence classes defined on the set of possible traces. The idea is to present only one representative of an equivalence class to a user. The summarization technique is easily integrable into the framework of this thesis. The framework presented in [MR18] can be used to detect which syntactic changes between two different versions of an AD induce a concrete witness. The application to ADs, as presented in [MR18], is based on the semantic differencing operator of [MRR11b]. Using the techniques of [MR18] with the semantic differencing method presented in this thesis is directly possible.

Other direct translations from ADs and business process models to finite automata focus on deadlock detection [Sug16, TSJ10]. However, the translations do not result in automata that represent the set of execution traces of the input ADs. Further, the translations are more complex than our translation. With minor adjustments, the translation of this thesis can also be used for deadlock detection by changing the accepting states of the automaton resulting from translating an AD. Instead of choosing the final automaton states as the automaton states that contain a transition with an accepting AD state, one needs to choose the set of final automaton states as all states without outgoing transitions that do not contain an AD transition that involves a final AD node. Then, a run of the automaton ending in an accepting state represents an AD execution that cannot proceed to an accepting state.

Other semantics definitions for ADs are based on Petri nets [Stö05], on the system model [GRR10] for characterizing object-oriented systems as defined in [BCGR09], or on the notion of *step* [KG10] inspired by the popular STATEMATE semantics for statecharts [HN96]. In contrast, this thesis defines an operational semantics based on a mapping to NFAs and a denotational semantics based on the languages recognized by the resulting NFAs.

Part III

Automatic Model Repairs

Chapter 7

A Framework for Automatic Model Repairs

Analyzing the evolution of models is an important task in MDD. Many existing analyses consider syntactic model evolution (e.g. [AP03, KKT11, KKT13, KGE09, KGFE08, TELW14, TBK09]) and focus on detecting the syntactic difference from a model to another model. Semantic differencing approaches analyze the evolution of models with respect to the changes in the meanings of models (e.g. [AHC⁺12, LMK14a, MRR11e, LMK14b, MRR11b, BKRW17, FLW11, FALW14]). A few approaches combine syntactic and semantic model evolution analyses [MR15, MR18, KR18b]. Combined approaches focus on relating syntactic to semantic model differences.

There are well-accepted generic approaches for syntactic differencing that abstract from concrete modeling languages (e.g., [AP03]). There are only a few generic approaches for analyzing the semantic differences of models that abstract from concrete modeling languages [FLW11, LMK14a, LMK14b]. Similarly, generic approaches that abstract from concrete modeling languages and relate syntactic to semantic model differences rarely exist [MR15, MR18, KR18b].

The syntax and the semantic mapping of each modeling language are usually tailored towards a specific application domain. Developing new approaches to evolution management for each newly emerging modeling language is expensive [KR18b]. Therefore, the main advantage of generic approaches abstracting from concrete modeling languages is that they provide general results that can be applied to multiple modeling languages.

Employing a general definition for the constituents of modeling languages and syntactic changes (cf. Chapter 2) enables the development of generic analyses: Stating generic assumptions on a modeling language and its change operations enables developing generic model evolution analyses for all concrete modeling languages satisfying the assumptions [KR18b].

Models naturally evolve during their development due to changing requirements and bug fixes. A model is changed with the intention to obtain an updated model satisfying a specific property that the original model does not satisfy. A *model evolution step* is a process of applying syntactic changes to a model such that the resulting model satisfies a well-defined property. Refinement steps are typical examples for special model evolution steps: A *model refinement step* is a process of applying syntactic changes to a model such that the semantics of the original model subsumes the resulting model's seman-

tics. Refinement steps effectively remove underspecification from a model by eliminating possible realizations. Model refinement steps are naturally performed when additional information becomes available during the development process. Performing a refinement step is, in general, error-prone and thus needs automated and meaningful support for repair in case an intended refinement step yields an incorrect result, *i.e.*, when the resulting model contains bugs.

This chapter introduces sufficient conditions on modeling languages that enable the fully automatic calculation of syntactic changes that transform a model to another model satisfying a well-defined property (such as refinement, refactoring, or generalization). In contrast to previous work, this chapter’s approach is

- independent of a concrete modeling language,
- independent of a concrete model property,
- computes shortest change sequences to maintain the developer’s intention behind the model as much as possible, and
- does not assume that powerful model composition operators are available.

The method relies on partitioning the syntactic change operations applicable to each model in equivalence classes and on excluding syntactic changes that are not part of a shortest change leading to a model satisfying the property.

The idea of the partitioning is grounded in the intuition that the application of change operations introducing names (elements of U_N) that are not used in a model often have similar effects on the properties satisfied by the model. The application of one of the change operations to the model yields a model that can be changed to a model satisfying the property with equally many change operations as a model obtained from applying another of the change operations to the model. This often enables to reduce the search space for change sequences changing the model to a model that satisfies the property. As models are usually finite structures using finitely many names, the sets of equivalent change operations are often infinite, which often enables reducing an infinitely branching search space to a finitely branching search space by only considering one change operation of each equivalence class. The main result is a generic and fully automatic method to repair failed model evolution steps under intuitive assumptions. The assumptions require the availability of an automatic procedure to check the satisfaction of the model property and the possibility to partition change operations into model-specific and property-specific equivalence classes.

In the remainder of this chapter (if not stated otherwise), let $\mathcal{L} = (M, S, sem)$ be an arbitrary modeling language and let O be a complete (cf. Definition 2.8, p. 20) change operation suite for \mathcal{L} .

This chapter builds upon the results of previously published work [KR18a] that is specifically concerned with repairing failed model refinement steps. In the following,

Section 7.1 recaps the problem of repairing failed model refinements and presents motivating examples for automatic model repairs in the context of repairing failed model refinements and the concrete modeling languages presented in Part II. Afterwards, Section 7.2 presents the general model repair problem that abstracts from a concrete modeling language and a concrete property. Then, Section 7.3 introduces two properties for change operations. Based on this, Section 7.4 formalizes the assumptions that enable the computation of repairing change sequences and presents algorithms for their computation. Section 7.6 discusses whether the assumptions are reasonable with respect to their applicability to concrete modeling languages. Section 7.8 presents related work.

7.1 Motivating Examples in Context of Repairing Refinement

This section motivates the usefulness of automatically repairing models by examples in the context of repairing refinement.

Two different models may be syntactically very similar but semantically very different. Vice versa, two models may be semantically equivalent but syntactically very different. Syntactic differencing operators detect the syntactic differences of models but do not compare the semantics of models. On the other hand, semantic differencing operators detect whether there are semantic differences from a model to another model but do not detect the syntactic model elements causing the existence of the semantic differences. The state of the art provides little support for detecting which syntactic model elements cause the existence of semantic differences from the model to another model.

A model refinement step is the evolution process of changing a model such that the successor model version is a refinement of the predecessor version. Thus, applying a refinement step removes underspecification concerning the system under development by changing the model such that each realization of the successor version is also a realization of the predecessor version. Refinement steps are naturally performed during development processes in reaction to the receipt of additional information, such as a new requirement or new insights obtained through the communication between different stakeholders. At each stage during the development of a system, the available models encode the information that is currently available. Every time new information becomes available, the models are refined until, ultimately, a correct system implementation is obtained. In general, refinement steps are error-prone. The state of the art provides little support for repairing failed refinement steps:

- Syntactic differencing solely reveals the syntactic differences between models in the form of change operations but does not provide information about semantic properties such as refinement.
- Refinement calculi (*e.g.*, in the contexts of interactive systems [Rum96, PR97, PR99], feature models [BKL⁺16], or the modeling languages and their change

operations presented in Part II) often define refining change operations and can detect whether a model refines another model by checking whether there exists a change sequence that transforms the former model to the other model and solely contains refining change operations. Such calculi are usually sound but incomplete, *i.e.*, they never provide wrong answers, but they may fail to give an answer at all.

- Semantic differencing reveals whether there exist elements in the semantics of one model that are not elements of the other model’s semantics and usually provide diff witnesses. If a diff witness exists, the former model is no refinement of the latter model. The provided witnesses facilitate developers in detecting the syntactic model elements causing that the former model is not a refinement of the latter model. However, the task of finding the syntactic elements causing the witness is still manually performed by developers.

Applying automatic model repair in the context of refinement enables the automatic computation of a change sequence that transforms one model to a refinement of another model. On the one hand, the change sequence can be directly applied to the model to obtain a refinement of the other model. On the other hand, the change sequence provides valuable information for developers in terms of the syntactic model elements that it affects. The affected elements can be investigated to identify the syntactic model elements causing the semantic model differences: As the application of the sequence leads to a refinement, some of the affected syntactic model elements must be “responsible” for the existence of semantic differences. Thus, the sequence can also provide additional information to developers for manually repairing failed model refinement steps. Even if the model is not intended to be a refinement of the other model, automatic model repair facilitates developers in detecting the syntactic elements of the model that cause the existence of semantic differences to the other model. This can increase developers’ understandings of model evolution steps.

The following sections motivate automatic repair of refinement by example using the modeling languages presented in Part II.

7.1.1 Shortest Repair of a Failed Activity Diagram Refinement Step

Figure 7.1 depicts two ADs taken from [KR18a] and inspired by [KGE09, KGFE08]. A manager of an insurance company wants to improve the efficiency of processing incoming claims. Therefore, the manager models the workflow that needs to be executed by employees on the receipt of incoming claims with the AD `claim3`. This AD represents the executions that are reasonable from the manager’s perspective. It is highly underspecified as it, for example, permits multiple different possibilities regarding the execution of actions after rejecting a claim. The manager hands the AD over to an employee. The task of the employee is to refine the workflow to exclude executions that are not reasonable from her perspective.

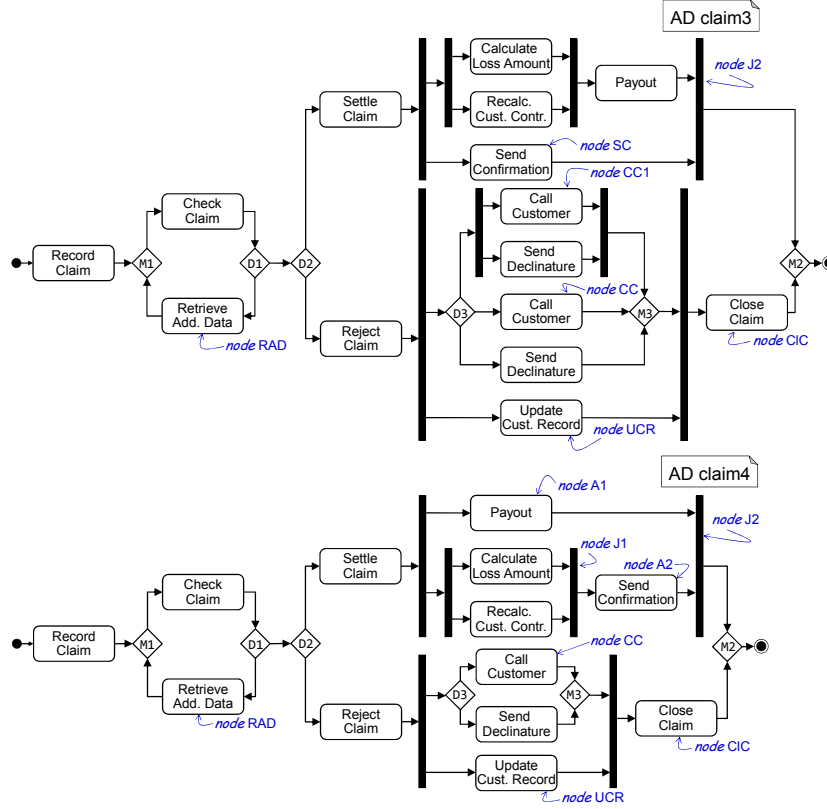


Figure 7.1: Two ADs taken from [KR18a] and inspired by [KGE09, KGFE08] modeling workflows in the context of an insurance company.

The employee edits the workflow modeled with the AD `claim3` and obtains the AD `claim4`. She informs the manager about the changes. The manager wants to verify the correctness of the changes. To this effect, she uses semantic differencing (cf. Section 6.3). The semantic differencing operator presents a witness proving that the successor version `claim4` is not a refinement of its predecessor version `claim3`. The manager decides to identify the error causing non-refinement for preparing a suggestion to repair the model. For this task, she uses automatic model repair as presented in this thesis. The model repair framework outputs that the application of at least two change operations (cf. Section 6.5) is required to change the AD `claim4` to a refinement of the AD `claim3`. More detailedly, the framework outputs that removing the action node `A1` labeled `Payout` and afterwards adding an action node labeled `Payout` between the nodes `J1` and the action node `A2` labeled `Send Confirmation` in the AD `claim4` yields a consistent AD that refines the AD `claim3`. The change sequence $delA_{A1}, addA_{J1,A2,A1,Payout}$ containing the two AD change operations (cf. Section 6.5) that encode the changes is a shortest change

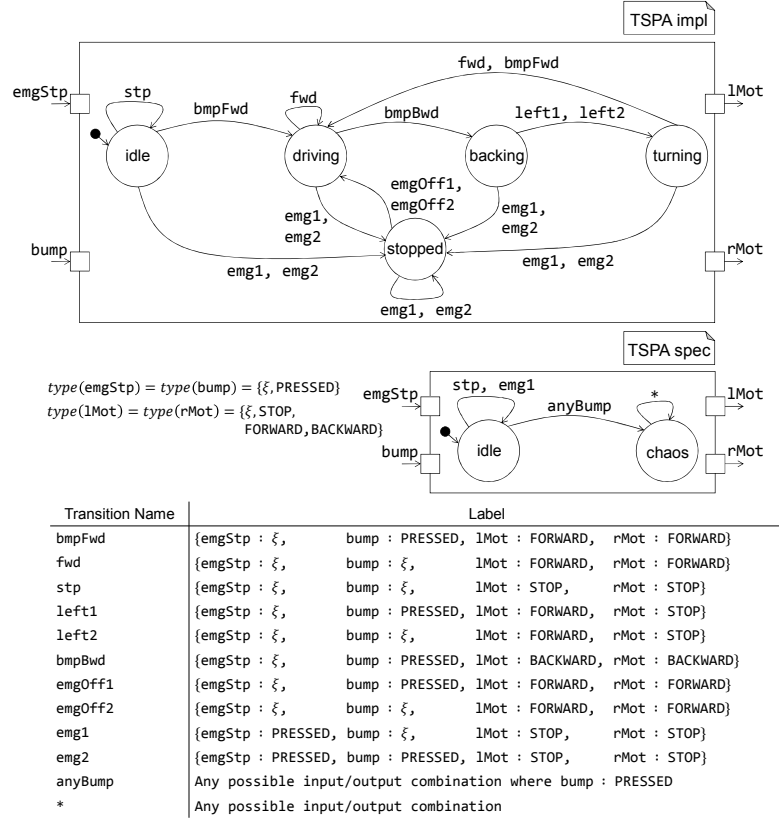


Figure 7.2: Two TSPAs adapted from similar automata presented in [KR18b] that initially appeared in [Rin14].

sequence that repairs the AD `claim4` towards refining the AD `claim3`. The manager considers the changes to be correct and consults the employee. It turns out that payouts should be definitely executed after calculating the exact loss amount and recalculating the customer contribution. Finally, the manager applies the fully automatically calculated change sequence to `claim4` and obtains the final AD, which is a refinement of the AD `claim3`.

7.1.2 Shortest Repair of a Time-Synchronous Port Automaton to Achieve the Satisfaction of a Requirement

Figure 7.2 depicts two TSPAs that are inspired by similar automata presented in [Rin14, KR18b]. The TSPAs `impl` and `spec` model behaviors of a simple mobile robot. The robot is equipped with a motor for steering its two left wheels and a motor for steering its two right wheels. It is further equipped with an emergency button and a button

(bump sensor) that indicates whether it hit a wall. Each of the two TSPAs has the two input channels `emgStp` and `bump` as well as the two output channels `lMot` and `rMot`. Messages received via channel `emgStp` indicate whether the emergency button is pressed. Similarly, messages received via the channel `bump` indicate whether the bump sensor is pressed. The input channels are of type $\{\xi, \text{PRESSED}\}$. The message ξ indicates that a button is not pressed, *i.e.*, no signal is present. The message `PRESSED` indicates that a button is pressed. The robot steers its motors by sending messages via its output channels `lMot` and `rMot`. Both channels are of type $\{\xi, \text{STOP}, \text{FORWARD}, \text{BACKWARD}\}$. Sending the message `FORWARD` via one of the output channels indicates that the motor should make its corresponding wheels drive forward. Vice versa, sending the message `BACKWARD` via one of the output channels indicates that the motor should make its corresponding wheels drive backward. The robot's task is to drive forward until hitting a wall. When hitting a wall, the robot should drive a little backward, turn in any direction, and drive forward again. As long as the emergency button is pressed, the robot should stop moving. A developer initially modeled the robot's behaviors with the TSPA `impl` (cf. Figure 7.2).

The developer receives a new requirement during the development process: When the robot is turned on, the robot should not start moving until its bump sensor has been pressed. The developer decides to check whether the implementation already satisfies the new specification. Thus, the developer models the specification with the TSPA `spec` (cf. Figure 7.2). The semantics of the TSPA `spec` contains all possible behaviors where the robot does not start moving until the bump sensor has been pressed. Therefore, the implementation satisfies the specification iff the TSPA `impl` is a refinement of the TSPA `spec`. To check this property, the developer uses semantic differencing (cf. Section 3.3). The semantic differencing operator outputs that there exist behaviors of the TSPA `impl` that are no behaviors of the TSPA `spec`. Thus, the implementation does not satisfy the specification. Therefore, the developer uses our framework to repair the implementation towards satisfying the specification automatically. Our framework calculates a shortest change sequence that transforms the TSPA `impl` to a TSPA that satisfies the specification modeled with the TSPA `spec`. It outputs that first adding a transition looping in state `idle` labeled `emg1` and then removing the transition from state `idle` to state `stopped` labeled `emg1` are changes that transform the TSPA `impl` to a TSPA that refines the TSPA `spec`. More precisely, the framework outputs that the change sequence $\text{add}T_{\text{idle}, \text{idle}, \text{emg1}}, \text{del}T_{\text{idle}, \text{stopped}, \text{emg1}}$ containing two TSPA change operations (cf. Section 3.5) is a shortest change sequence that changes the TSPA `impl` to a TSPA that refines the TSPA `spec`. The developer applies the fully automatically calculated change sequence to the TSPA `impl` and obtains a TSPA that is a correct implementation with respect to the specification modeled with the TSPA `spec`.

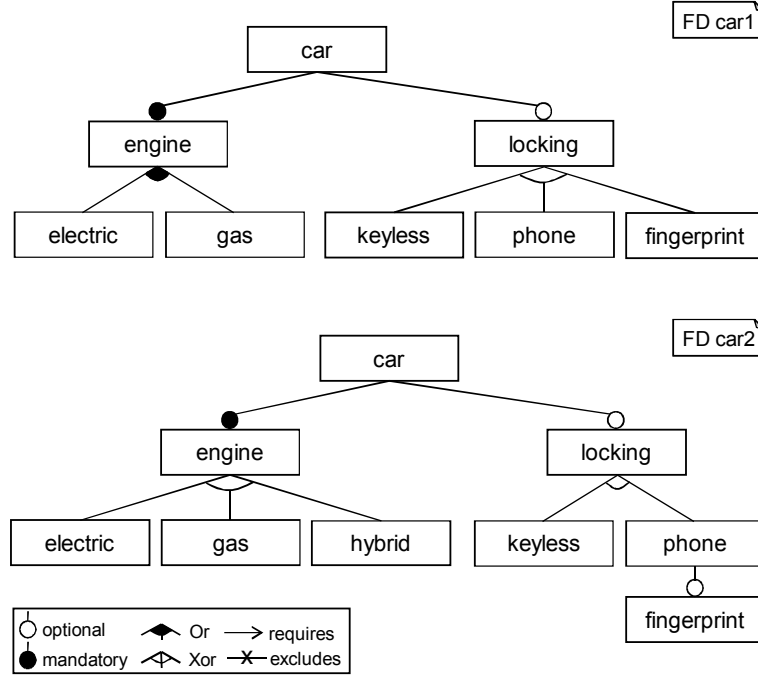


Figure 7.3: Two FDs adapted from [MR18, KR18b] and inspired by a similar FD example from [MR15].

7.1.3 Understanding a Feature Diagram Evolution Step

Figure 7.3 depicts two FDs that are adapted from [MR18, KR18b] and are inspired by a similar example from [CW07]. The FDs model the possible configurations of the engine and locking systems of a car. The FD *car1* is the original version modeled by the car development team.

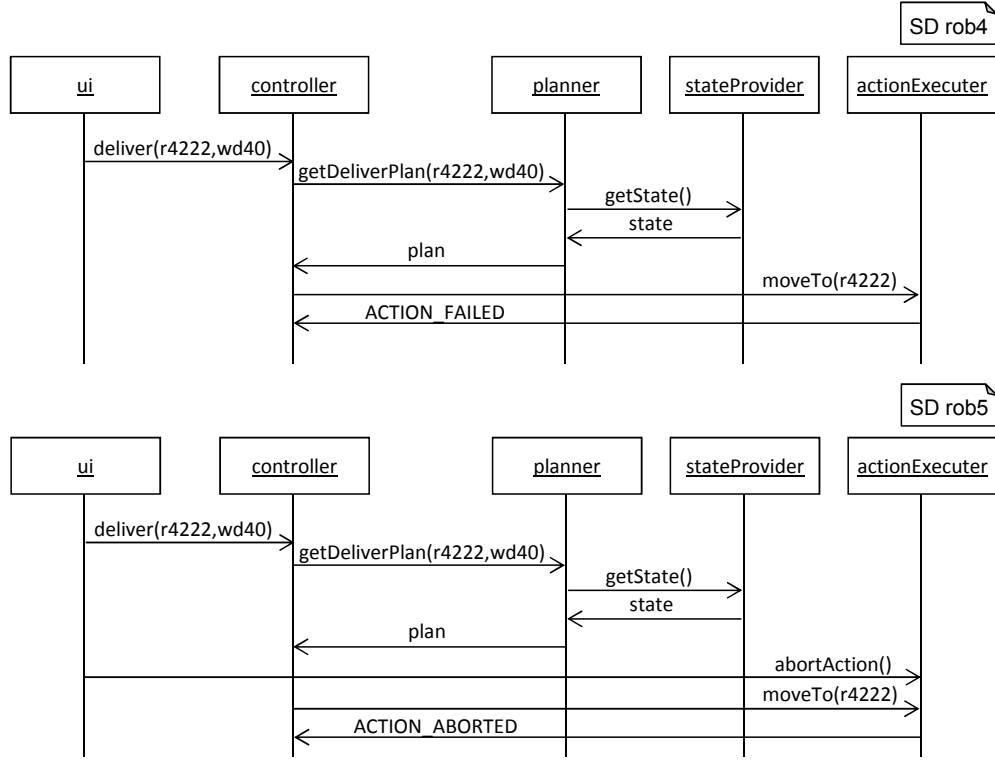
During the car development process, the team receives a new requirement and additionally decides to change the model to increase its understandability. The development team changes the FD *car1* to the FD *car2*. The change increasing the model’s understandability is the change of the type of the group of the feature engine. In *car1*, the feature engine has an or-group that represents the possible engine configurations. Simultaneously selecting both features of the group represents a hybrid engine. In *car2*, the feature hybrid is explicitly added to the group and the type of the group is changed to xor. Therefore, in *car2*, it must be explicitly selected whether a configuration of the car consists of an electric engine, a gas engine, or a hybrid engine. The changed requirements state that the phone and fingerprint locking systems can be chosen simultaneously and that the phone locking system must be selected when the fingerprint locking system is selected.

Another engineer, who has not been involved in the change process, wants to understand the changes performed by the development team. She uses semantic differencing. The semantic differencing operator outputs that there are configurations of the FD `car2` containing the features `hybrid` and `fingerprint` that are no configurations of the FD `car1`. The configurations `{car,engine,gas,locking,phone,fingerprint}` and `{car,engine,hybrid}`, for instance, are valid in `car2` and not valid in `car1`. The engineer is interested in which syntactic elements of the FD `car2` cause the semantic differences. She thus uses our framework for automatic model repairs. The framework outputs that excluding the feature `hybrid` from its group and adding an excludes constraint between the features `phone` and `fingerprint` are changes that change the FD `car2` to a consistent FD that has no semantic differences to the FD `car1`. More precisely, the framework outputs that $exclGrp_{hybrid}, addX_{phone,fingerprint}$ is a shortest change sequence of FD change operations that transforms the FD `car2` to a consistent FD that is a refinement of the FD `car1`. With this information, the engineer understands that the changes affecting the features `hybrid`, `phone`, and `fingerprint` are responsible for the existence of semantic differences from the FD `car2` to the FD `car1`. With this information, she examines the FDs and understands that configurations representing hybrid engine systems must now explicitly contain the feature `hybrid` and that the `phone` and `fingerprint` locking systems can now be chosen simultaneously.

7.1.4 Understanding the Semantic Differences between Sequence Diagrams

Figure 7.4 depicts two SDs inspired by similar SDs from [ABH⁺17]. The SDs model excerpts of the communication between objects in a software system implementing a service robot. The situations modeled by the SDs are different. In the situation modeled with the SD `rob4`, the robot should deliver the item `wd40` to the room `r4222`. After computing a plan for performing the task, the object `actionExecutor` is instructed to make the robot move to the room `r4222` via action `moveTo(r4222)`. The execution of the action fails, which is indicated by the interaction between the `actionExecutor` object and the `controller` object via the action `ACTION_FAILED`. The SD `rob5` models a situation where the execution of an action is aborted by a user.

A developer wants to understand if there are differences concerning the system runs that are valid in the SDs modeling the different situations. Thus, she uses semantic differencing. The semantic differencing operators outputs that there are system runs that are valid in `rob4` and not valid in `rob5`. Vice versa, the semantic differencing operator outputs that there are system runs that are valid in `rob5` and not valid in `rob4`. The developer is interested in the syntactic model elements of the SD `rob5` that cause the semantic differences to the SD `rob4`. She uses our framework for automatic model repairs. The framework outputs that adding an interaction from the object `actionExecutor` to the object `controller` with action `ACTION_FAILED` after the interaction with the action `moveTo(r4222)` changes the SD `rob5` to an SD that re-


 Figure 7.4: Two SDs adapted inspired by similar SDs from [ABH⁺17].

fines the SD `rob4`. More precisely, the framework outputs that the change sequence `addIA7,actionExecuter,ACTION_FAILED,controller` containing exactly one SD change operation is a shortest change sequence that transforms the SD `rob5` to a refinement of the SD `rob4`. With this information, she understands that the interactions between the `actionExecuter` and the `controller` objects for indicating that the action execution is not successful are different in the two scenarios. Vice versa, the developer is interested in the syntactic model elements of the SD `rob4` that cause the semantic differences to the SD `rob5`. She thus uses our framework for automatic model repairs. The framework outputs that at least two changes are required to change the SD `rob4` to a refinement of the SD `rob5`. The first change is the addition of the interaction `(ui,abortAction(),actionExecuter)` between the interactions with the actions `plan` and `moveTo(r4222)`. The second change is the addition of the interaction `(actionExecuter,ACTION_ABORTED,controller)` after the last interaction modeled in the SD. More precisely, the framework outputs that the change sequence `addIA5,ui,abortAction(),actionExecuter, addIA8,actionExecuter,ACTION_ABORTED,controller` is a shortest change sequence containing two SD change operations that changes the

SD `rob4` to an SD that is a refinement of the SD `rob5`. With this information, the developer understands that users must initiate action abortions.

7.2 Model Repair Problems

This section introduces the notion of *model repair problem*, which is independent of a concrete modeling language, a concrete change operation suite, and a concrete property.

A model property is a characteristic of models that each model either satisfies or not. Thus, we represent properties over the models of a modeling language as a subset of the set of models. With this encoding, a model satisfies the property iff it is an element of the subset representing the property. A property is accomplishable iff the set representing the property is not empty, *i.e.*, there exists at least one model that satisfies the property.

Definition 7.1. *A property P of the models of \mathcal{L} is a set $P \subseteq M$. The property P is said to be accomplishable iff $P \neq \emptyset$. The complement property of P is denoted \overline{P} and defined as $\overline{P} \stackrel{\text{def}}{=} M \setminus P$.*

If \mathcal{L} is clear from the context, we simply say that P is a model property instead of saying that P is a property of the models of \mathcal{L} . For each model $m \in M$, we write $P(m)$ and say that the model m satisfies the property P iff $m \in P$.

An example model property is refinement of a concrete model. For instance, let $\mathcal{L} = (M, S, \text{sem})$ be a modeling language. Then, for every $m \in M$, the model property $P_m = \{n \in M \mid \emptyset \neq \text{sem}(n) \subseteq \text{sem}(m)\}$ contains all consistent models of the modeling language \mathcal{L} that are refinements of the model m . Stated differently, a model $n \in M$ satisfies the property P_m iff n is consistent and n is a refinement of m .

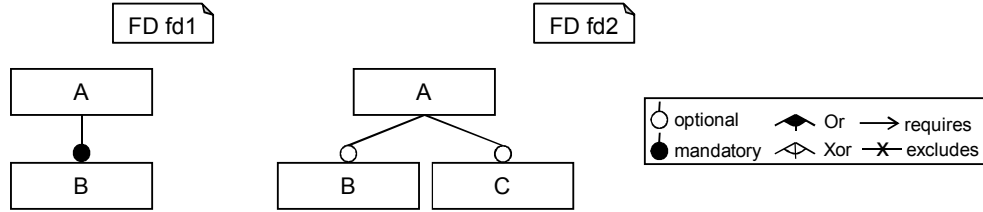
For a model and a model property, we study sufficient conditions that enable the computation of a change sequence such that applying the sequence to the model results in a model that satisfies the property. This motivates the notion of change sequence that repairs a model towards satisfying a property:

Definition 7.2. *A change sequence $t \in O^*$ repairs a model $m \in M$ towards satisfying a model property $P \subseteq M$ iff $m \triangleright t \in M \wedge P(m \triangleright t)$.*

Intuitively, the application of a change sequence that repairs a model towards satisfying a property to the model results in a model that satisfies the property.

For example, Figure 7.5 depicts the two FDs `fd1` and `fd2`. For the remainder of this chapter, we define the model property $R \stackrel{\text{def}}{=} \{m \in M_{FD} \mid \emptyset \neq \llbracket m \rrbracket^{FD} \subseteq \llbracket \text{fd1} \rrbracket^{FD}\}$. The model property R contains all consistent FDs that refine the FD `fd1`. In this example, the following holds:

- The change sequence ε does not repair the FD `fd2` towards satisfying R because $\text{fd2} \triangleright \varepsilon = \text{fd2}$ and `fd2` is not a refinement of `fd1`.


 Figure 7.5: The FD $fd2$ does not refine the FD $fd1$.

- The change sequence $t = delF_B, opt2man_C$ does not repair the FD $fd2$ towards satisfying R because the FD $fd2 \triangleright t$ does not satisfy the property R .
- The change sequence $addX_{A,A}$ does not repair $fd2$ towards satisfying R because $fd2 \triangleright addX_{A,A}$ is inconsistent.
- The change sequence $delF_X$ does not repair $fd2$ towards satisfying R because $fd2 \triangleright delF_X \notin M$.
- The change sequence $u = opt2man_B, delF_C$ repairs the FD $fd2$ towards satisfying R because the FD $fd2 \triangleright u$ satisfies the property R .
- Another change sequence that repairs the FD $fd2$ towards satisfying R is $v = opt2man_B$ because $fd2 \triangleright v$ satisfies the property R .

Developers usually define a model with a precise intuition and syntactic elements that support to understand the model's meaning in the context in which the model is used. For a repairing change sequence to be useful, the application of the change sequence should result in a model that keeps this intention as much as possible. This motivates the notion of shortest repairing change sequence:

Definition 7.3. A change sequence $t \in O^*$ is a shortest change sequence that repairs a model $m \in M$ towards satisfying a property on models $P \subseteq M$ iff the following two conditions are satisfied:

1. $m \triangleright t \in M \wedge P(m \triangleright t)$ and
2. $\forall u \in O^* : (m \triangleright u \in M \wedge P(m \triangleright u)) \Rightarrow |t| \leq |u|$.

The first condition in Definition 7.3 states that t must be a sequence that repairs the model towards satisfying the property. The second condition states that the length of every change sequence that repairs the model towards satisfying the property must be greater than or equal to the length of the change sequence t . There must not exist a shorter change sequence that repairs the model towards satisfying the property.

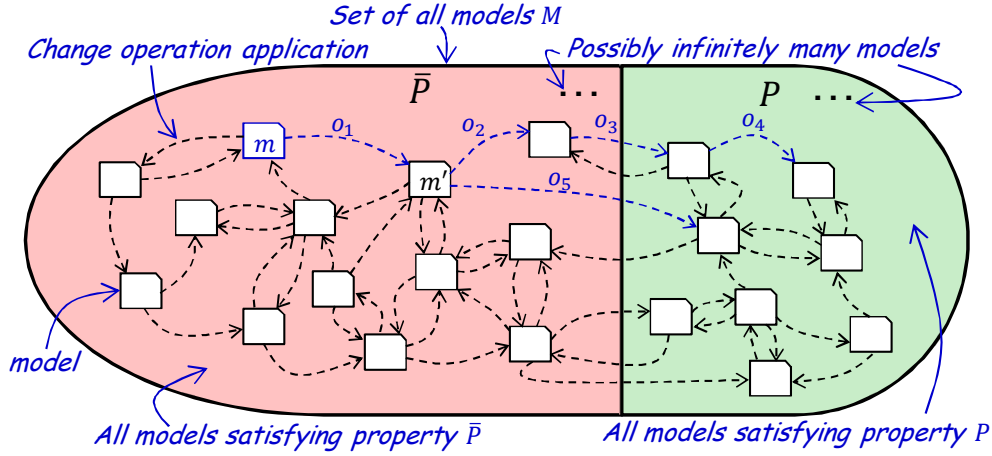


Figure 7.6: Schematic illustration of the relations between models, model properties, and change operations.

Figure 7.6 schematically illustrates the relations between models, model properties, and change operations. The properties P and its complement property \bar{P} partition the set of all models into two sets. The nodes in the graph depicted in Figure 7.6 represent the models contained in the sets representing the properties. Dashed lines between the nodes represent the effects of applying change operations. For instance, applying the change operation o_1 to the model m yields the model m' . In Figure 7.6, The change sequence o_1, o_2, o_3 is a change sequence that repairs the model m towards satisfying the property P . Similarly, o_1, o_2, o_3, o_4 is also a change sequence that repairs the model m towards satisfying the property P . However, both of these change sequences are not shortest change sequences that repair m towards satisfying P because the change sequence o_1, o_5 is a shorter change sequence that repairs m towards satisfying P . The change sequence o_1, o_2 is not a change sequence that repairs the model m towards satisfying the property P because $m \triangleright o_1, o_2 \notin P$.

In the FD example (cf. Figure 7.5), the following holds:

- The change sequence ε is not a shortest change sequence that repairs the FD fd2 (cf. Figure 7.5) towards satisfying the model property R because ε is not a change sequence that repairs fd2 towards satisfying R .
- Although $u = \text{opt2man}_B, \text{del}F_C$ is a change sequence that repairs fd2 towards satisfying R , the change sequence is not a shortest change sequence that repairs fd2 towards satisfying R .
- The change sequence opt2man_B is a shortest change sequence that repairs fd2

towards satisfying R because it repairs fd2 towards satisfying R and there exists no shorter change sequence that repairs fd2 towards satisfying R .

If the change operation suite for a modeling language is complete, then a change sequence that repairs a model towards satisfying a property exists iff the property is accomplishable, *i.e.*, there exists at least one model that satisfies the property. Thus, if a change operation suite is complete and a property is accomplishable, then it is always possible to find a change sequence that repairs any model towards satisfying the property.

Proposition 7.1. *Let $m \in M$ be a model and $P \subseteq M$ be a property on models. There exists a sequence $t \in O^*$ that repairs m towards satisfying P iff P is accomplishable.*

Proof. Let $m \in M$ be a model and $P \subseteq M$ be a property on models.

" \Rightarrow ": Assume there exists a change sequence $t \in O^*$ that repairs m towards satisfying P . Then, by definition of repairing change sequence, it holds for $n \stackrel{\text{def}}{=} m \triangleright t$ that $n \in M$ and $P(n)$. Thus, $n \in P$, which implies that P is accomplishable.

" \Leftarrow ": Assume P is accomplishable. Then, there exists a model $r \in M$ with $P(r)$. As O is complete, there exists a change sequence $t \in O^*$ such that $m \triangleright t = r$. As by assumption $P(r)$ and as $m \triangleright t = r$, it holds that t repairs m towards satisfying P . \square

Proposition 7.1 reveals that a repairing sequence always exists if there exists a model that satisfies the property and the underlying change operation suite is complete. Vice versa, the proposition also reveals that repairing sequences never exist if the property is not accomplishable. The following formally defines the notions of model repair problem, model repair problem instance, solution of a model repair problem, and shortest solution of a model repair problem:

Definition 7.4. *A model repair problem is a tuple $\mathcal{P} = (\mathcal{L}, O, P)$ where*

- $\mathcal{L} = (M, S, \text{sem})$ is a modeling language,
- O is a complete change operation suite for \mathcal{L} , and
- $P \subseteq M$ is an accomplishable model property.

A model repair problem instance is a tuple (\mathcal{P}, m) where $\mathcal{P} = (\mathcal{L}, O, P)$ with $\mathcal{L} = (M, S, \text{sem})$ is a model repair problem and $m \in M$ is a model. A change sequence $t \in O^$ is called a solution for a model repair problem instance (\mathcal{P}, m) with $\mathcal{P} = (\mathcal{L}, O, P)$ iff t is a change sequence that repairs the model m towards satisfying the property P . A change sequence $t \in O^*$ is called a shortest solution for a model repair problem instance (\mathcal{P}, m) with $\mathcal{P} = (\mathcal{L}, O, P)$ iff t is a shortest change sequence that repairs the model m towards satisfying the property P .*

For instance, in the FD example, the following holds:

- $Q \stackrel{\text{def}}{=} (\mathcal{L}_{FD}, O_{FD}, R)$ is a model repair problem for the FD modeling language \mathcal{L}_{FD} with the change operation suite O_{FD} and the property R containing all consistent FDs that refine the FD fd1 (cf. Figure 7.5).
- $(Q, \text{fd2})$ is an instance of the model repair problem Q .
- As $t = \text{opt2man}_B$ is a shortest change sequence that repairs fd2 towards satisfying the property R , the change sequence t is a shortest solution for the model repair problem instance $(Q, \text{fd2})$.

Definition 7.4 requires that the properties of model repair problems are accomplishable because Proposition 7.1 revealed that model repair problems with unaccomplishable properties are not meaningful: If the property of a model repair problem were not accomplishable, then there would not exist a solution for any instance of the model repair problem. Vice versa, the requirement for realizability guarantees that there exist solutions for all instances of the model repair problem:

Proposition 7.2. *There exists a solution for every model repair problem instance.*

Proof. Let $I = (\mathcal{P}, m)$ be model repair problem instance where $\mathcal{P} = (\mathcal{L}, O, P)$. By definition P is accomplishable and O is complete. Using Proposition 7.1, this implies that there exists a change sequence t that repairs m towards satisfying P . By the definition of solutions for model repair problem instances, t is a solution for I . \square

The overall goal is the development of an algorithm that is capable of computing shortest solutions for all instances of an arbitrary but fixed model repair problem. There may exist multiple shortest solutions for a model repair problem instance. However, it directly follows by definition that all shortest solutions have the same lengths.

Proposition 7.3. *Let $I = (\mathcal{P}, m)$ with $\mathcal{P} = (\mathcal{L}, O, P)$ be a model repair problem instance. If $t, u \in O^*$ are two shortest solutions for I , then $|t| = |u|$.*

Proof. Let $I = (\mathcal{P}, m)$ with $\mathcal{P} = (\mathcal{L}, O, P)$ be a model repair problem instance. Suppose towards a contradiction there exist two shortest solutions $t, u \in O^*$ for I with $|t| \neq |u|$. Without loss of generality, assume $|t| > |u|$. This contradicts the assumption that t is a shortest solution for I because u is also a solution for I and $|u| < |t|$. \square

In the following, to reduce notational overhead, for each model repair problem \mathcal{P} , we introduce the function $d_{\mathcal{P}} : M \rightarrow \mathbb{N}$, which maps each model $m \in M$ to the length $d_{\mathcal{P}}(m)$ of all shortest solutions for the model repair problem instance (\mathcal{P}, m) . The function $d_{\mathcal{P}}$ is well-defined because all instances of each model repair problem have a solution (cf. Proposition 7.2) and all shortest solutions for each instance of the model repair problem have the same lengths (cf. Proposition 7.3).

For instance, for the FD example, the following holds:

- $d_Q(\text{fd2}) = 1$ because $t = \text{opt2man}_B$ with $|t| = 1$ is a shortest solution for the model repair problem instance $(Q, \text{fd2})$.
- $d_Q(\text{fd1}) = 0$ because ε with $|\varepsilon| = 0$ is a shortest solution for the model repair problem instance $(Q, \text{fd1})$.

7.3 Change Operation Properties

Usually, infinitely many change operations are applicable to each model, which hampers the computation of solutions for model repair problems. To tackle this challenge, this section introduces a method for partitioning all change operations applicable to a model into equivalence classes. Section 7.4 shows that automatically computing a shortest solution is possible if the set of equivalence classes is finite for each model.

Syntactically very similar models may satisfy very different properties. Vice versa, syntactically very different models may satisfy the same property. For instance, let fd' be an arbitrary FD. Further, let fd be an FD that is a refinement of fd' . Applying a single root-rename operation to fd may yield an FD that is syntactically very similar to fd , but not a refinement of fd . In contrast, every FD obtained from applying arbitrary many feature-addition operations to fd is also a refinement of FD because feature-addition operations are refining. Applying many feature-addition operations to fd yields an FD that still refines the FD fd' and is syntactically very different from fd .

As many different models may satisfy the same property, there may exist multiple (even infinitely many) solutions for a model repair problem instance. There are usually infinitely many different change operations applicable to each model of a modeling language. This hampers the computability of shortest solutions for model repair problem instances as individually testing whether the application of any of the infinitely many applicable change operations yields a model satisfying the property might not terminate. Therefore, reducing the search space for shortest solutions is necessary.

Let $I = (\mathcal{P}, m)$ be a model repair problem instance. In many cases, it is possible to determine from the definition of two change operations o, p , whether for each shortest solution for $(\mathcal{P}, o(m))$, there exists a solution of the same length for $(\mathcal{P}, p(m))$ and vice versa. Then, there exists a shortest solution for (\mathcal{P}, m) that starts with o iff there exists a shortest solution for (\mathcal{P}, m) that starts with p because otherwise the lengths of the shortest solutions for $(\mathcal{P}, o(m))$ would be different to the lengths of the shortest solutions for $(\mathcal{P}, p(m))$. This property can be used to reduce the search space for shortest solutions: It suffices to determine whether a shortest solution starts either with o or with p to conclude whether there exists a shortest solution that starts with o or p .

For instance, let (Q, fd) be an FD refinement repair problem instance where fd is an arbitrary FD, and let $o = \text{addF}_{p,f}$ as well as $p = \text{addF}_{p,g}$ be two feature-addition operations that are applicable to fd where f, g are two features not used in fd and p is a feature that is used in fd . Then, for every shortest solution t for $(Q, o(fd))$,

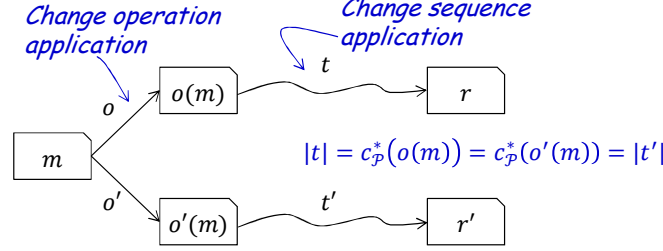


Figure 7.7: Schematic representation of the relation between two change operations o, o' that induce an equally long shortest solution for a model repair problem instance using the model m .

there exists a solution u for $(Q, p(fd))$ with $|t| = |u|$: Let t be a shortest solution for $(Q, o(fd))$. The change sequence u is constructed by exchanging every change operation in t affecting the feature f (respectively g) by a change operation of the same type that affects the feature g (respectively f) instead of the feature f (respectively g). For instance, a feature-deletion operation $delF_f$ is replaced by the feature-deletion operation $delF_g$ and an implies constraint addition operation $addI_{f,g}$ is replaced by the operation $addI_{g,f}$. Then, the FD $o(fd) \triangleright t$ can be obtained from the FD $p(fd) \triangleright u$ by replacing the feature f (if it exists) by the feature g and replacing the feature g (if it exists) by the feature f in $p(fd) \triangleright u$. Therefore, the valid configurations of $o(fd) \triangleright t$ can be obtained from the valid configurations of $p(fd) \triangleright u$ by replacing the feature f (if it exists) in each valid configuration of $p(fd) \triangleright u$ by the feature g and replacing the feature g (if it exists) in each valid configuration of $p(fd) \triangleright u$ by the feature f . Let C be a configuration that is valid in $p(fd) \triangleright u$. From the above, it follows that the configuration C' obtained from replacing the feature f (if it exists) in C by the feature g and replacing the feature g (if it exists) in C by the feature f is valid in $o(fd) \triangleright t$. As $o(fd) \triangleright t$ is a refinement of fd , the configuration C' is also valid in fd . As the features f and g are not used in fd , this implies with Proposition 4.1 that the configuration C is valid in fd . We can conclude that every valid configuration of $p(fd) \triangleright u$ is also a valid configuration of fd . Analogously, it is possible to show that for every shortest solution u for $(Q, p(fd))$, there exists a solution t for $(Q, o(fd))$ with $|t| = |u|$. The two change operations induce an equally long shortest solution for the corresponding model repair problem.

Two change operations o, o' induce an equally long shortest solution for a model repair problem instance (\mathcal{P}, m) iff the lengths of the shortest solutions for the model repair problem $(\mathcal{P}, o(m))$ are equal to the lengths of the shortest solutions for the model repair problem $(\mathcal{P}, o'(m))$. This situation is sketched in Figure 7.7. The two change operations o and o' may change the model m to different models. However, for every shortest change sequence t that repairs the model obtained from applying o to m , there exists a shortest change sequence t' that repairs the model obtained from applying o' to m such that

$|t| = |t'|$ and vice versa. The relation is formally captured by the following definition:

Definition 7.5. Let $I = (\mathcal{P}, m)$ where $\mathcal{P} = (\mathcal{L}, O, P)$ with $\mathcal{L} = (M, S, sem)$ be a model repair problem instance. Two change operations $o, o' \in O$ induce an equally long shortest solution for I iff $o(m), o'(m) \notin M$ or $o(m), o'(m) \in M \wedge d_{\mathcal{P}}(o(m)) = d_{\mathcal{P}}(o'(m))$.

We write $o \sim_I o'$ iff $o, o' \in O$ induce an equally long shortest solution for the model repair problem instance I . For $o, o' \in O$, we write $o \not\sim_I o'$ iff $o \sim_I o'$ is not satisfied.

Every two change operations that are not applicable to a model induce an equally long shortest solution for the model repair problem instance. If o is a change operation that is applicable to a model and o' is a change operation that is not applicable to the model, then o and o' do not induce an equally long shortest solution of the corresponding model repair problem instance. Two change operations $o, o' \in O$ that are applicable to the model $m \in M$ of a model repair problem instance (\mathcal{P}, m) induce an equally long shortest solution for the instance iff the length $d_{\mathcal{P}}(o(m))$ of the shortest solutions for repairing the model $o(m)$ is equal to the length $d_{\mathcal{P}}(o'(m))$ of the shortest solutions for repairing the model $o'(m)$. Therefore, for each shortest change sequence that repairs the model $o(m)$, it is possible to find a change sequence of equal length that repairs the model $o'(m)$ and vice versa.

For example, for the model repair problem instance $I = (Q, fd2)$, the following holds:

- $opt2man_B \sim_I addI_{A,B}$ because $d_Q(opt2man_B(fd2)) = 0 = d_Q(addI_{A,B}(fd2))$.
- $opt2man_B \not\sim_I delF_C$ because $0 = d_Q(opt2man_B(fd2)) \neq d_Q(delF_C(fd2)) = 1$.
- $addF_{C,f} \sim_I addF_{C,g}$ where f, g are two features neither used in $fd1$ nor in $fd2$ because of the argumentation above.

Each induce equally long shortest solution relation depends on a repair problem instance. Thus, for every model repair problem for a modeling language with an infinite set of models, there are infinitely many induce equally long shortest solution relations, one for each instance of the problem. Each of these relations is an equivalence relation:

Proposition 7.4. Let I be a model repair problem instance. The relation \sim_I is an equivalence relation.

Proof. Let $I = (\mathcal{P}, m)$ where $\mathcal{P} = (\mathcal{L}, O, P)$ with $\mathcal{L} = (M, S, sem)$ be a model repair problem instance.

Reflexivity: Let $o \in O$. If $o(m) \notin M$, then it directly follows by Definition 7.5 that $o \sim_I o$. If $o(m) \in M$, then $d_{\mathcal{P}}(o(m)) = d_{\mathcal{P}}(o(m))$ and, thus, $o \sim_I o$.

Symmetry: Let $o, o' \in O$. Assume $o \sim_I o'$ holds. Then, it holds that $o(m), o'(m) \notin M$ or $o(m), o'(m) \in M \wedge d_{\mathcal{P}}(o(m)) = d_{\mathcal{P}}(o'(m))$. Rearrangement yields the equivalent formulation $o'(m), o(m) \notin M$ or $o'(m), o(m) \in M \wedge d_{\mathcal{P}}(o'(m)) = d_{\mathcal{P}}(o(m))$. Using Definition 7.5, it holds that $o' \sim_I o$.

Transitivity: Let $o, o', o'' \in O$. Assume it holds that $o \sim_I o'$ and $o' \sim_I o''$. Then, the following two statements hold:

1. $o(m), o'(m) \notin M$ or $o(m), o'(m) \in M \wedge d_{\mathcal{P}}(o(m)) = d_{\mathcal{P}}(o'(m))$ and
2. $o'(m), o''(m) \notin M$ or $o'(m), o''(m) \in M \wedge d_{\mathcal{P}}(o'(m)) = d_{\mathcal{P}}(o''(m))$.

It holds that $o(m) \notin M \Leftrightarrow o'(m) \notin M$ and $o'(m) \notin M \Leftrightarrow o''(m) \notin M$. Thus, if $o(m) \notin M$ or $o'(m) \notin M$ or $o''(m) \notin M$, then $o(m), o'(m), o''(m) \notin M$, which implies with Definition 7.5 that $o \sim_I o''$. Assume $o(m), o'(m), o''(m) \in M$. Then, it holds that $d_{\mathcal{P}}(o(m)) = d_{\mathcal{P}}(o'(m))$ and $d_{\mathcal{P}}(o'(m)) = d_{\mathcal{P}}(o''(m))$. This implies that $d_{\mathcal{P}}(o(m)) = d_{\mathcal{P}}(o''(m))$. As further $o(m), o''(m) \in M$, we obtain with Definition 7.5 that $o \sim_I o''$. \square

Let $I = (\mathcal{P}, m)$ where $\mathcal{P} = (\mathcal{L}, O, P)$ be a model repair problem instance and let $o \in O$ be a change operation. We denote by $[o]_I \stackrel{\text{def}}{=} \{o' \in O \mid o \sim_I o'\}$ the equivalence class of o under \sim_I . Further, for each set of change operations $Q \subseteq O$, we denote by $Q / \sim_I \stackrel{\text{def}}{=} \{[o]_I \mid o \in Q\}$ the quotient of Q under \sim_I .

For instance, in the FD example with the model repair problem instance $I = (Q, \text{fd2})$, the following holds:

- $\text{opt2man}_B, \text{addI}_{A,B} \in [\text{opt2man}_B]_I = [\text{addI}_{A,B}]_I$.
- $[\text{opt2man}_B]_I \neq [\text{delF}_C]_I$.
- $\{[\text{opt2man}_B]_I\} = \{\text{opt2man}_B\} / \sim_I = \{\text{addI}_{A,B}\} / \sim_I = \{\text{opt2man}_B, \text{addI}_{A,B}\} / \sim_I$.

As each induce equally long shortest solution relation is an equivalence relation, the relation's equivalence classes partition the set of all change operations. Figure 7.8 schematically illustrates the partitioning. The equivalence classes are pairwise disjoint. Each equivalence class may contain infinitely many change operations. The set of partitions can also be infinite.

As all change operations in the same class induce an equally long shortest solution, it is possible to reduce the search space for shortest solutions for model repair problem instances. All operations in the same class induce shortest solutions of the same lengths. Thus, it suffices to check whether a single operation of an equivalence class is the first element of a shortest solution to determine whether all operations of the equivalence class are the first elements of a shortest solution. The partitioning into the equivalence classes is especially useful as the set of operations in the same equivalence class may be infinite, whereas there may be only finitely many different classes. The following formally shows that considering a single representative of each equivalence class suffices:

Proposition 7.5. *Let $I = (\mathcal{P}, m)$ where $\mathcal{P} = (\mathcal{L}, O, P)$ be a model repair problem instance and let $o, o' \in O$ be two change operations with $o \sim_I o'$. Then, there exists a shortest solution t for I with $|t| > 0$ and $t.0 = o$ iff there exists a shortest solution u for I with $|u| > 0$ and $u.0 = o'$.*

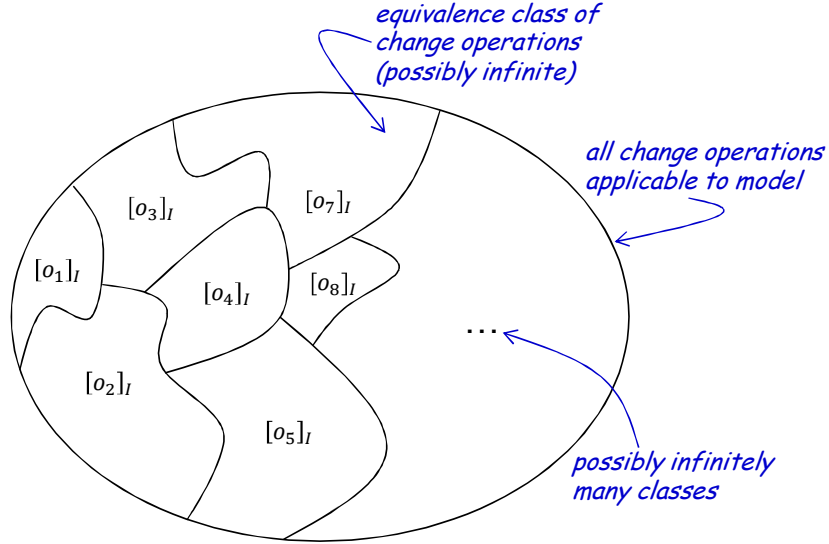


Figure 7.8: Schematic representation of the quotient of the set of all change operations under the equivalence relation \sim_I where I is a model repair problem instance.

Proof. Let $I = (\mathcal{P}, m)$ where $\mathcal{P} = (\mathcal{L}, O, P)$ be a model repair problem instance and let $o, o' \in O$ be two change operations with $o \sim_I o'$. As \sim_I is symmetric, it suffices to show one direction of the proposition. Assume there exists a shortest solution t for I with $|t| > 0$ and $t.0 = o$. As $o \sim_I o'$, there exists a change sequence u such that $|u| = |t| - 1$ and $m \triangleright (o' : u) \in M$ and $m \triangleright (o' : u) \in P$. As $|o' : u| = 1 + |u| = |t|$ and $|t|$ is a shortest solution for I , Proposition 7.3 guarantees that $o' : u$ is also a shortest solution for I . \square

A change operation may never be part of any shortest solution for a model repair problem instance. It is often possible to determine whether a change operation is not part of a shortest solution from the type of the change operation.

For instance, the FD consistency property contains all FDs that are consistent. In this case, we can already determine from the property that shortest solutions for repair problem instances using the property never start with operations that add an excludes constraint from the root feature to itself: If the operation is applied, one of the following operations must remove the excludes constraint because otherwise the resulting model would not be consistent. Thus, every solution for the instance starting with the change operation that adds the excludes constraint can be shortened by removing the change operation adding the excludes constraint and the change operation that removes the excludes constraint. Thus, change sequences starting with the operation adding the excludes constraint are never shortest solutions for the repair problem instances using the consistency property. The operation *delays* the solution for the instance.

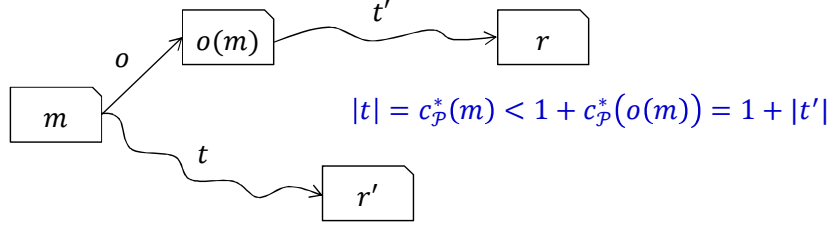


Figure 7.9: Schematic representation of a change operation that delays the solution for a model repair problem instance using the model m .

This situation is schematically illustrated in Figure 7.9. The length of a shortest change sequence t that repairs the original model m is smaller than or equal to the length of a shortest change sequence that repairs the model $o(m)$ obtained from applying the change operation o . Thus, shortest change sequences that repair the model m never start with the change operation o .

Definition 7.6. Let $I = (\mathcal{P}, m)$ where $\mathcal{P} = (\mathcal{L}, O, P)$ be a model repair problem instance. A change operation $o \in O$ delays the solution for the instance I iff for all shortest solutions $t \in O^*$ for I it holds that $|t| > 0 \Rightarrow t.0 \neq o$.

The set of all change operations that delay the solution for a model repair problem instance I is denoted by Del_I .

Intuitively, a change operation delays the solution for a model repair problem instance iff no shortest solution for the instance starts with the change operation. Stated differently, every change sequence that starts with a change operation that delays the solution for the instance is never a shortest solution for the instance. Therefore, on the one hand, change operations that delay the solution for a model repair problem instance can be safely ignored during an iterative approach to computing a shortest solution for the instance. On the other hand, change operations that do not delay the solution for a model repair problem instance are always part of a shortest solution for the instance.

For instance, for the model repair problem instance $I = (Q, \text{fm2})$ in the FD example, the following holds:

- $delF_C \in Del_I$ delays the solution for I because $d_Q(delF_C) > 0$ and the change sequence $t = \text{opt2man}_B$ with $|t| = 1$ is a shortest solution for I .
- $delF_X \in Del_I$ delays the solution for I because $delF_X$ is not applicable to fm2 , which implies that there exists no shortest solution for I that starts with $delF_X$.
- $\text{opt2man}_B \notin Del_I$ does not delay the solution for I because the change sequence solely containing the change operation opt2man_B is a shortest solution for I .

All change operations that do not delay the solution for a model repair problem instance induce an equally long shortest solution for the instance:

Proposition 7.6. *Let $I = (\mathcal{P}, m)$ where $\mathcal{P} = (\mathcal{L}, O, P)$ be a model repair problem instance. If $o, o' \in O$ with $o, o' \in O \setminus Del_I$ are two change operations that do not delay the solution for the instance I , then $o \sim_I o'$.*

Proof. Let $I = (\mathcal{P}, m)$ where $\mathcal{P} = (\mathcal{L}, O, P)$ be a model repair problem instance. Assume $o, o' \in O$ with $o, o' \in O \setminus Del_I$ are two change operations that do not delay the solution for the instance I . As the change operations $o, o' \in O \setminus Del_I$ do not delay the solution for the instance I , there exist shortest solutions $t, u \in O^*$ for I such that $|t| > 0$ and $t.0 = o$ and $|u| > 0$ and $u.0 = o'$. This especially implies that $o(m), o'(m) \in M$. Suppose towards a contradiction that $o \not\sim_I o'$ holds. This implies that $d_{\mathcal{P}}(o(m)) \neq d_{\mathcal{P}}(o'(m))$. Without loss of generality, assume $d_{\mathcal{P}}(o(m)) > d_{\mathcal{P}}(o'(m))$. As t is a shortest solution for I and $t.0 = o$, it must especially hold that $|t| = 1 + d_{\mathcal{P}}(o(m))$ because otherwise $|t|$ would be no shortest solution for I . Analogously, as u is a shortest solution for I and $u.0 = o'$, it must especially hold that $|u| = 1 + d_{\mathcal{P}}(o'(m))$ because otherwise $|u|$ would be no shortest solution for I . From this, we can derive $|t| = 1 + d_{\mathcal{P}}(o(m)) > d_{\mathcal{P}}(o'(m)) + 1 = |u|$. Thus, $|t| > |u|$, which contradicts that t and u are both shortest solutions for I because Proposition 7.3 guarantees that all shortest solutions for I have the same lengths. \square

The equivalence class of change operations that do not delay the solution for a model repair problem instance is exactly the equivalence class of all change operations that are the first elements of at least one shortest solution for the instance. Thus, if a change operation does not delay the solution for an instance, then it is not equivalent to any change operation that delays the solution for the instance.

Proposition 7.7. *Let $I = (\mathcal{P}, m)$ where $\mathcal{P} = (\mathcal{L}, O, P)$ be a model repair problem instance. If $o, o' \in O$ are change operations with $o \in Del_I$ and $o' \notin Del_I$, then $o \not\sim_I o'$.*

Proof. Let $I = (\mathcal{P}, m)$ where $\mathcal{P} = (\mathcal{L}, O, P)$ be a model repair problem instance. Assume $o, o' \in O$ are change operations with $o \in Del_I$ and $o' \notin Del_I$. As $o' \notin Del_I$, there exists a shortest solution $t \in O^*$ for I such that $|t| > 0 \wedge t.0 = o'$. This especially implies that $o'(m) \in M$. Suppose towards a contradiction that $o \sim_I o'$ holds. Then, by definition of \sim_I and as $o'(m) \in M$, it holds that $o(m) \in M$ and $d_{\mathcal{P}}(o(m)) = d_{\mathcal{P}}(o'(m))$. As t is a shortest solution for I and $t.0 = o'$, it holds that $d_{\mathcal{P}}(o'(m)) = |t| - 1$ because otherwise t would be no shortest solution for I . With $d_{\mathcal{P}}(o'(m)) = d_{\mathcal{P}}(o(m))$, the above implies that there exists a change sequence u with $|u| = d_{\mathcal{P}}(o(m)) = |t| - 1$ such that $o : u$ is a solution for I . As $|o : u| = 1 + |t| - 1 = |t|$ and as t is a shortest solution for I , it holds that $o : u$ is also a shortest solution for I . This contradicts the assumption that $o \in Del_I$ delays the solution for I . \square

7.4 Computing Shortest Repairing Change Sequences

This section presents two assumptions for model repair problems. If a model repair problem satisfies the assumptions, then automatically computing a shortest solution for each instance of the model repair problem is possible.

Computing a shortest solution is reduced to applying a finite search in a finitely branching, infinite, rooted tree. Based on the assumptions, this section presents algorithms for the computation of shortest solutions. As the computational complexity of computing shortest solutions is high (as discussed in the following Section 7.5), several performance improvements and algorithm variants are presented and discussed.

The first assumption is obvious: An automatic method to check whether a model of the model repair problem's modeling language satisfies the model property $P \subseteq M$ under consideration must be available (*i.e.*, the property P must be decidable). If there did not exist such an automatic method, then it would not be possible to automatically check whether a change sequence is a solution for an instance of the model repair problem.

The second assumption demands the availability of an algorithm that computes a function mapping models to finite sets of change operations. The function is required to map each model to a finite set of change operations containing at least one representative of the equivalence class of a change operation that does not delay the solution for the model repair problem instance corresponding to the model. From a methodological viewpoint, the function has to be implemented by a developer and is used as input for the procedures that compute shortest solutions for model repair problems.

This assumption is equivalent to the assumption that the function maps each model to a finite set of change operations containing at least one change operation that does not delay the solution for the model repair problem instance corresponding to the model. However, determining an operation that does not delay the solution for the model repair problem instance is methodologically more complicated: If computing an operation that does not delay the solution for the model repair problem instance was easily possible, then solutions for model repair problem instances would be interpretable to be known a priori and, thus, would be effectively computable. In most cases, it is methodologically easier to determine the change operations that delay solutions for model repair problem instances, before partitioning the other change operations into the equivalence classes of change operations that induce equally long shortest solutions.

Determining a change operation that does not delay the solution for a model repair problem instance requires a priori knowledge of a definitive solution for the model repair problem instance. In contrast, determining a subset of the change operations that delay the solution for a model repair problem instance and partitioning the other applicable change operations into the equivalence classes only requires knowledge of the solution space and not of a definitive solution. The second assumption demands the availability of a repair-representative function:

Definition 7.7. Let $\mathcal{P} = (\mathcal{L}, O, P)$ be a model repair problem. A \mathcal{P} -repair-representative function is a function $\mathcal{R} : M \rightarrow \wp_{fin}(O)$ satisfying $(\mathcal{R}(m)/\sim_I) \cup (Del_I/\sim_I) = O/\sim_I$ for all instances $I = (\mathcal{P}, m)$ of \mathcal{P} .

For all models $m \in M$, the set $\mathcal{R}(m)$ contains at least one representative of the equivalence class of change operations that do not delay the repair of the model m towards satisfying the property P . Although the set of change operations O is usually infinite in practice, the finite sets $\mathcal{R}(m)$ can often be analytically determined because of symmetries induced by the syntax of modeling languages. For instance, two models may be syntactically different (but in some sense symmetric concerning their syntax) and semantically equivalent.

An alternative characterization of repair-representative functions is the following: If the set of change operations that do not delay the solution for a model repair problem instance is not empty, then the function maps the model of the instance to a finite set containing at least one change operation that does not delay the solution for the instance.

Proposition 7.8. Let $\mathcal{P} = (\mathcal{L}, O, P)$ be a model repair problem. A function $\mathcal{R} : M \rightarrow \wp_{fin}(O)$ is a \mathcal{P} -repair-representative function iff $Del_I \neq O \Rightarrow \mathcal{R}(m) \cap (O \setminus Del_I) \neq \emptyset$ for every instance $I = (\mathcal{P}, m)$ of \mathcal{P} .

Proof. Let $\mathcal{P} = (\mathcal{L}, O, P)$ be a model repair problem and let $\mathcal{R} : M \rightarrow \wp_{fin}(O)$ be a function.

" \Rightarrow ": Let $I = (\mathcal{P}, m)$ be an instance of \mathcal{P} and assume \mathcal{R} is a \mathcal{P} -repair-representative function. Then, it holds that $(\mathcal{R}(m)/\sim_I) \cup (Del_I/\sim_I) = O/\sim_I$. Assume it holds that $Del_I \neq O$. This implies that there exists a change operation $o \in (O \setminus Del_I)$. By Proposition 7.7, this implies $o \not\sim_I d$ for all $d \in Del_I$. Thus, $[o]_I \notin Del_I/\sim_I$. Therefore, as $o \in O$ and $(\mathcal{R}(m)/\sim_I) \cup (Del_I/\sim_I) = O/\sim_I$ and $[o]_I \notin Del_I/\sim_I$, there must exist a change operation $o' \in \mathcal{R}(m)$ such that $[o']_I = [o]_I$. As \sim_I is transitive and $o' \sim_I o$ and $o \not\sim_I d$ for all $d \in Del_I$, it holds that $o' \not\sim_I d$ for all $d \in Del_I$ because otherwise if $o' \sim_I d$ hold for some $d \in Del_I$, then $o \sim_I d$ would also hold. As $o' \not\sim_I d$ for all $d \in Del_I$, it especially holds that $o' \notin Del_I$ because \sim_I is reflexive. Therefore, $o' \in O \setminus Del_I$. As $o' \in \mathcal{R}(m)$ and $o' \in O \setminus Del_I$, we can conclude that $\mathcal{R}(m) \cap (O \setminus Del_I) \neq \emptyset$.

" \Leftarrow ": Assume $Del_I \neq O \Rightarrow \mathcal{R}(m) \cap (O \setminus Del_I) \neq \emptyset$ holds for every instance $I = (\mathcal{P}, m)$ of \mathcal{P} . Let $I = (\mathcal{P}, m)$ be an instance of \mathcal{P} . We need to show that $(\mathcal{R}(m)/\sim_I) \cup (Del_I/\sim_I) = O/\sim_I$ holds. If $Del_I = O$, then $(\mathcal{R}(m)/\sim_I) \cup (Del_I/\sim_I) = (\mathcal{R}(m)/\sim_I) \cup (O/\sim_I) = O/\sim_I$ is satisfied. Assume $Del_I \neq O$. Then, by assumption $\mathcal{R}(m) \cap (O \setminus Del_I) \neq \emptyset$ holds. Let $o \in \mathcal{R}(m) \cap (O \setminus Del_I)$ be a change operation. Then, as $o \in \mathcal{R}(m)$, it holds that $\mathcal{R}(m)/\sim_I \supseteq \{[o]_I\}$. Further, as $o \in O \setminus Del_I$ and all operations that do not delay the solution for I are in the same equivalence class (cf. Proposition 7.6), it holds that $\{[o]_I\} = (O \setminus Del_I)/\sim_I$. From the above, we can derive $O/\sim_I \supseteq \mathcal{R}(m)/\sim_I \cup Del_I/\sim_I \supseteq \{[o]_I\} \cup Del_I/\sim_I = (O \setminus Del_I)/\sim_I \cup Del_I/\sim_I = ((O \setminus Del_I) \cup Del_I)/\sim_I = O/\sim_I$. As $O/\sim_I \supseteq \mathcal{R}(m)/\sim_I \cup Del_I/\sim_I \supseteq O/\sim_I$, we can conclude that $\mathcal{R}(m)/\sim_I \cup Del_I/\sim_I = O/\sim_I$. \square

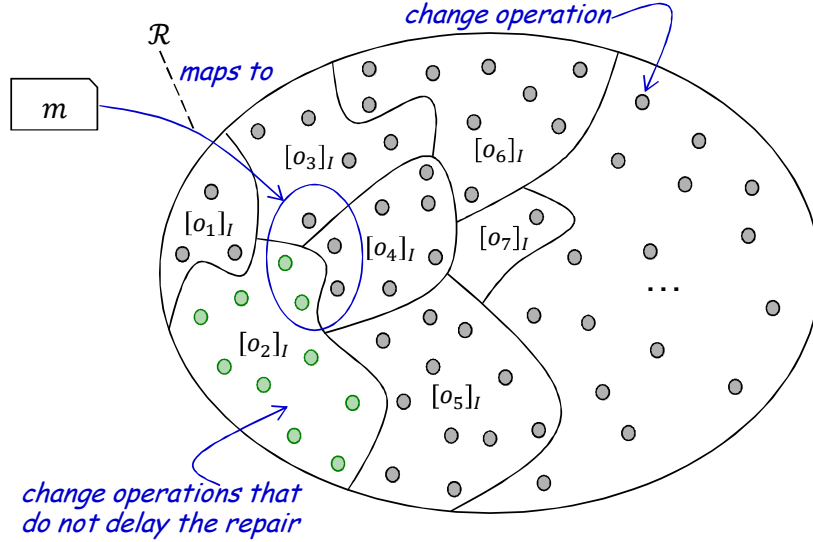


Figure 7.10: Schematic illustration of a repair-representative function mapping the model m to a set of change operations.

Thus, if the set of change operations that do not delay the solution for an instance is not empty, then a repair-representative function maps the model of the instance to a set that contains at least one change operation that does not delay the solution for the instance. Vice versa, every function from models to finite sets of change operations that maps every model to a set containing at least one change operation that does not delay the solution for the corresponding model repair problem instance, if one exists, is a repair-representative function for the corresponding model repair problem.

Figure 7.8 depicts the quotient of the set of all change operations under the \sim_I equivalence relation, where $I = (\mathcal{P}, m)$ is a model repair problem instance. The set of equivalence classes may be infinite. In this case, the repair-representative function cannot map the model m to at least one representative of each equivalence class because repair-representative functions must map models to finite sets of change operations. However, the repair-representative function does not necessarily map the model to a set containing at least one representative of each equivalence class. In case the model satisfies the property, then the repair-representative function may map the model to any finite set of change operations. Otherwise, it suffices that the function maps the model to a set containing at least one element of the equivalence class of change operations that do not delay the solution for the instance I . Figure 7.10 illustrates this condition. The repair-representative function maps the model m to a finite set of change operations that contains at least one representative of the equivalence class containing the change operations that do not delay the solution for the model repair problem instance.

The following reveals that it requires to consider change operations from the sets $\mathcal{R}(m)$ for computing shortest solutions for model repair problem instances.

Proposition 7.9. *Let $I = (\mathcal{P}, m)$ where $\mathcal{P} = (\mathcal{L}, O, P)$ be a model repair problem instance and let \mathcal{R} be a \mathcal{P} -repair-representative function. There exists a shortest solution $t \in O^*$ for I with $t.i \in \mathcal{R}(m \triangleright t \downarrow i)$ for all $i \in \mathbb{N}$ with $0 \leq i < |t|$.*

Proof. Let $\mathcal{P} = (\mathcal{L}, O, P)$ be a model repair problem and let \mathcal{R} be a \mathcal{P} -repair-representative function.

We show a more general property: For all $d \in \mathbb{N}$ and for all instances (\mathcal{P}, m) of \mathcal{P} , if there exists a shortest solution t for I with $|t| = d$, then there exists a shortest solution u for I with $|u| = |t|$ and $u.i \in \mathcal{R}(m \triangleright u \downarrow i)$ for all $i \in \mathbb{N}$ with $0 \leq i < |u|$. The proof is by induction over the lengths of shortest solutions for model repair problem instances:

$d = 0$: Let $I = (\mathcal{P}, m)$ be an instance of \mathcal{P} . Assume there exists a solution t for I with $|t| = 0$. As $t = \varepsilon$, the statement is satisfied for $u = \varepsilon$.

Induction hypothesis: Let $n \in \mathbb{N}$. Assume the statement holds for all change sequences t with $|t| \leq n$.

$d = n + 1$: Let $I = (\mathcal{P}, m)$ be an instance of \mathcal{P} . Assume there exists a shortest solution t for I with $|t| = n + 1$. As t is a shortest solution for I and $|t| > 0$, using Definition 7.6, we have that $t.0 \notin \text{Del}_I$. Thus, $[t.0]_I \notin \text{Del}_I / \sim_I$. Therefore, $[t.0]_I \in \mathcal{R}(m) / \sim_I$ because $[t.0]_I \in O / \sim_I$ and $\mathcal{R}(m) / \sim_I \cup \text{Del}_I / \sim_I = O / \sim_I$. Now let $o \in \mathcal{R}(m)$ such that $[o]_I = [t.0]_I$. As $t.0$ and o induce an equally long shortest solution for I and $|t|$ is a shortest solution for I , there exists a shortest change sequence v that repairs $o(m)$ towards satisfying P with $|v| = |t| - 1$. Thus, v is a shortest solution for the model repair problem instance $(\mathcal{P}, o(m))$ with $|v| = |t| - 1$.

Using the induction hypothesis, we obtain that there exists a shortest solution w for $(\mathcal{P}, o(m))$ with $|w| = |v|$ and $w.i \in \mathcal{R}(o(m) \triangleright w \downarrow i)$ for all $i \in \mathbb{N}$ with $0 \leq i < |w|$.

In conclusion, we have that $|o \& w| = 1 + |w| = 1 + |v| = |t|$ and $(o \& w).i \in \mathcal{R}(m \triangleright (o \& w) \downarrow i)$ for all $i \in \mathbb{N}$ with $0 \leq i < |o \& w|$. As further $o \& w$ is a solution for I with $|o \& w| = |t|$ and as t is a shortest solution for I , $o \& w$ is also a shortest solution for I . \square

Using Proposition 7.9, it is possible to reduce the search of shortest solution for a model repair problem instance to a search in a rooted tree:

Definition 7.8. *Let $I = (\mathcal{P}, m)$ where $\mathcal{P} = (\mathcal{L}, O, P)$ be a model repair problem instance and let \mathcal{R} be a \mathcal{P} -repair-representative function. The change sequence search tree $T(\mathcal{R}, I)$ induced by \mathcal{R} and I is defined as $T(\mathcal{R}, I) \stackrel{\text{def}}{=} (V, r, E)$ with*

- nodes $V = \{t \in O^* \mid m \triangleright t \in M \wedge t.i \in \mathcal{R}(m \triangleright t \downarrow i) \text{ for all } i \in \mathbb{N} \text{ with } 0 \leq i < |t|\}$,
- root $r = \varepsilon$, and
- edges $E = \{(v, w) \in V \times V \mid \exists o \in \mathcal{R}(m \triangleright v) : v \& o = w\}$.

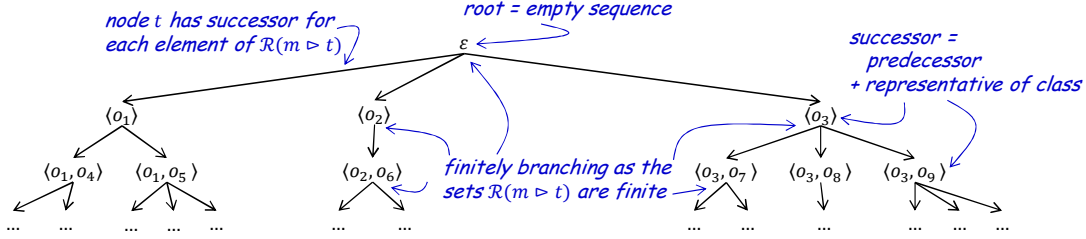


Figure 7.11: Schematic representation of a change sequence search tree for a repair problem instance that uses the model m .

The nodes of the tree represent change sequences that are obtained from concatenating change operations that are elements of the sets in the image of the repair-representative function. The change sequence represented by a node must be applicable to the model repair problem instance's model. The root node is the empty sequence. Two nodes $v, w \in V$ are connected in the tree iff the concatenation of the sequence v with a change operation contained in $\mathcal{R}(m \triangleright v)$ is equal to w .

Figure 7.11 schematically illustrates a change sequence search tree for a model repair problem instance that uses the model m . The nodes of the tree are change sequences. The root of the tree is the empty sequence. The children of a node t are exactly the change sequences obtained from appending a change operation contained in the set $\mathcal{R}(m \triangleright t)$ to the change sequence t . Thus, for every element contained in $\mathcal{R}(m \triangleright t)$, the node t has one child node. As the repair-representative function \mathcal{R} is required to map every model to a finite set, every node has finitely many children.

The tree $T(\mathcal{R}, I)$ is guaranteed to contain a finite path from the root to a node that encodes a shortest solution for the model repair problem instance I :

Proposition 7.10. *Let $I = (\mathcal{P}, m)$ where $\mathcal{P} = (\mathcal{L}, O, P)$ be a model repair problem instance and let \mathcal{R} be a \mathcal{P} -repair-representative function. The tree $T(\mathcal{R}, I) = (V, r, E)$ contains a finite path from the root r to a node $t \in V$ that is a shortest solution for I .*

Proof. Let $I = (\mathcal{P}, m)$ where $\mathcal{P} = (\mathcal{L}, O, P)$ be a model repair problem instance and let \mathcal{R} be a \mathcal{P} -repair-representative function. Let $T(\mathcal{R}, I) = (V, r, E)$ be the change sequence search tree induced by \mathcal{R} and I .

Then, Proposition 7.9 guarantees that there exists a shortest solution $t \in O^*$ for I with $t.i \in \mathcal{R}(m \triangleright t \downarrow i)$ for all $i \in \mathbb{N}$ with $0 \leq i < |t|$. As t is a solution for I , it holds that $m \triangleright t \in M$ and, thus, it especially holds that $m \triangleright t \downarrow i \in M$ for all $i \in \mathbb{N}$ with $0 \leq i \leq |t|$. Using Definition 7.8, the above implies that $m \triangleright t \downarrow i \in V$ for all $i \in \mathbb{N}$ with $0 \leq i \leq |t|$.

As $t.i \in \mathcal{R}(m \triangleright t \downarrow i)$ for all $i \in \mathbb{N}$ with $0 \leq i < |t|$ and $m \triangleright t \downarrow i \in V$ for all $i \in \mathbb{N}$ with $0 \leq i \leq |t|$, using Definition 7.8, we obtain that $(t \downarrow i, t \downarrow (i + 1)) \in E$ for all $0 \leq i < |t|$. Therefore, there exists a path from the root $r = \varepsilon$ to the node t in $T(\mathcal{R}, I)$. \square

The length of a shortest solution for a model repair problem instance is an upper bound for the length of a rooted finite path in the tree to a node that encodes a shortest solution for the instance. Nevertheless, there could be infinitely many finite rooted paths in the tree with a length shorter than or equal to the length of a shortest solution. This would hamper the computation of shortest solutions. In the following, we show that the number of finite rooted paths in the tree with a length shorter than or equal to an arbitrary but fixed bound is always finite. Change sequence search trees are always finitely branching:

Proposition 7.11. *Let $I = (\mathcal{P}, m)$ be a model repair problem instance and let \mathcal{R} be a \mathcal{P} -repair-representative function. The tree $T(\mathcal{R}, I)$ is finitely branching.*

Proof. Let $I = (\mathcal{P}, m)$ where $\mathcal{P} = (\mathcal{L}, O, P)$ be a model repair problem instance and let \mathcal{R} be a \mathcal{P} -repair-representative function. Let $T(\mathcal{R}, I) = (V, r, E)$ be the change sequence search tree induced by \mathcal{R} and I . Let $v \in V$ be a node of the tree $T(\mathcal{R}, I)$. By Definition 7.8, the set of nodes connected via an edge with source node v is given by $out = \{w \in V \mid \exists o \in \mathcal{R}(m \triangleright v) : v \&o = w\}$. As the set $\mathcal{R}(x)$ is finite for all $x \in M$, the set out is also finite. We can conclude that $T(\mathcal{R}, I)$ is finitely branching. \square

As change operation suites are required to be countable, change sequence search trees always have a countable number of nodes.

Proposition 7.12. *Let $I = (\mathcal{P}, m)$ be a model repair problem instance and let \mathcal{R} be a \mathcal{P} -repair-representative function. The tree $T(\mathcal{R}, I)$ has a countable number of nodes.*

Proof. Let $I = (\mathcal{P}, m)$ where $\mathcal{P} = (\mathcal{L}, O, P)$ be a model repair problem instance, let \mathcal{R} be a \mathcal{P} -repair-representative function and let $T(\mathcal{R}, m) = (V, r, E)$ be the change sequence search tree induced by \mathcal{R} and I . By assumption, every change operation suite for every modeling language is countable (cf. Section 2.2). Therefore, especially the change operation suite O is countable. As the set of finite sequences over a countable set is countable, the set O^* of all change sequences over O is countable. As every subset of a countable set is countable and as by construction $V \subseteq O^*$, we can conclude that V is countable. \square

As change sequence search trees are always finitely branching and always have a countable number of nodes, the number of paths in the tree $T(\mathcal{R}_P, m)$ with a length shorter than or equal to an arbitrary but fixed natural number is finite.

Proposition 7.13. *Let $I = (\mathcal{P}, m)$ be a model repair problem instance and let \mathcal{R} be a \mathcal{P} -repair-representative function. For all $d \in \mathbb{N}$, the number of rooted paths of length at most d in $T(\mathcal{R}, m)$ is finite.*

Proof. Let $I = (\mathcal{P}, m)$ where $\mathcal{P} = (\mathcal{L}, O, P)$ be a model repair problem instance, let \mathcal{R} be a \mathcal{P} -repair-representative function. Further, let $d \in \mathbb{N}$ and $T(\mathcal{R}, m) = (V, r, E)$ be

the change sequence search tree induced by \mathcal{R} and I . We define the subtree S of the tree $T(\mathcal{R}, m)$ as follows: $S \stackrel{\text{def}}{=} (V', r, E')$ where $V' = \{t \in V \mid |t| \leq d\}$ and $E' = E \cap (V' \times V')$. By construction, the set of rooted paths in the tree S is equal to the set of rooted paths of length at most d in $T(\mathcal{R}, m)$. Suppose towards a contradiction that the set of rooted paths in $T(\mathcal{R}, m)$ of length at most d is infinite. This implies that the tree S is infinite. However, the tree S is finitely branching because $E' \subseteq E$ and $T(\mathcal{R}, m)$ is finitely branching (cf. Proposition 7.13). Further, the set of nodes of S is countable because $V' \subseteq V$ and V is countable. (cf. Proposition 7.12). Hence, by König's Lemma [Kön27], the tree S contains an infinite branch. Let $p = v_0, v_1, \dots$ be an infinite branch of the tree S . By definition of the set E' , it holds that $v_i \sqsubset v_{i+1}$ for all $i \in \mathbb{N}$. Thus, $|v_i| < |v_{i+1}|$ for all $i \in \mathbb{N}$. As p is infinite and the length of successor nodes in p is monotonically increasing, it holds that $|v_j| > d$ for all $j \in \mathbb{N}$ with $j > d$. This contradicts that p is an infinite branch of S because $|v_{d+1}| > d$ and, therefore, $v_{d+1} \notin V' = \{t \in V \mid |t| \leq d\}$. \square

As the number of rooted paths of every arbitrary but fixed length in every change sequence search tree is finite (cf. Proposition 7.13), it is possible to enumerate all change sequences encoded by the nodes of the tree up to every arbitrary but fixed length. The search tree is guaranteed to contain a solution (cf. Proposition 7.10). Therefore, we can iteratively enumerate all change sequences in the search tree in increasing lengths and eventually find a shortest solution. The first solution that is found in this way is guaranteed to be a shortest solution. If it were not a shortest solution, then the sketched procedure would have found a shorter solution in a previous iteration.

7.5 Algorithms for Computing Shortest Solutions

Algorithm 2 depicts an algorithm for computing shortest solutions for model repair problem instances. It takes a model repair problem instance and a repair-representative function for the corresponding model repair problem as inputs. The algorithm implements a depth-first iterative-deepening (DFID) search [Kor85] for change sequence search trees. It searches the change sequence search tree induced by the problem instance and the representative function for a shortest solution for the instance. To this effect, the algorithm iteratively enumerates all change sequences encoded by the nodes of the search tree in increasing lengths. The variable *currentLengthBound* (l. 1) stores the bound of the length of sequences enumerated in the current iteration. In the body (ll. 3-15) of the outer loop (ll. 2-16), the algorithm performs a bounded depth-first search on the change sequence search tree. The algorithm aborts the search in a branch as soon as the length of the branch exceeds the current bound as specified by the value assigned to the variable *currentLengthBound*. At the end of the outer loop, the value stored in the variable *currentLengthBound* is increased by one (l. 16). For the depth-first search, the variable S (l. 3) is a stack that stores the explored nodes, from which further paths have to be explored. The search starts in the root node ε (l. 4). The algorithm uses

Algorithm 2 Computing a shortest solution for a model repair problem instance.

Input: A model repair problem instance $I = (\mathcal{P}, m)$ where $\mathcal{P} = (\mathcal{L}, O, P)$ and a \mathcal{P} -repair-representative function \mathcal{R} .

Output: Shortest solution $t \in O^*$ for I .

```

1: define currentLengthBound  $\leftarrow 0$  as positive integer
2: loop
3:   define  $S$  as empty stack of  $O^*$ 
4:    $S.push(\varepsilon)$ 
5:   while  $S$  not empty do
6:      $t \leftarrow S.pop()$ 
7:     if  $m \triangleright t \in M \wedge P(m \triangleright t)$  then
8:       return  $t$ 
9:     end if
10:    if  $m \triangleright t \in M \wedge |t| < currentLengthBound$  then
11:      for all  $o \in \mathcal{R}(m \triangleright t)$  do
12:         $S.push(t \& o)$ 
13:      end for
14:    end if
15:  end while
16:   $currentLengthBound \leftarrow currentLengthBound + 1$ 
17: end loop

```

the inner loop (ll. 5-15) to explore all nodes encoding change sequences having a length that is smaller than the current bound. To this effect, the algorithm first fetches the most recently explored node (l. 6). If this node represents a solution for the model repair problem instance (ll. 7-9), then the algorithm returns the change sequence encoded by the node (l. 8). Otherwise, if the length of the node is smaller than the current bound (ll. 10-14), the algorithm proceeds as follows: It concatenates all change operations relevant from the current node with respect to the repair-representative function (ll. 11-13) to the sequence encoded by the current node and pushes the resulting change sequences on the stack S (l. 12). The algorithm terminates as soon as it finds a shortest solution for the model repair problem instance. The algorithm always terminates because the change sequence search tree is guaranteed to contain a rooted path to a node that represents a shortest solution for the model repair problem instance (cf. Proposition 7.10).

Algorithm 2 returns the first shortest solutions that it finds. If the computation of all shortest solutions is required, the algorithm can be easily adapted as follows: The algorithm initializes a variable as a set of change sequences containing all shortest solutions. As soon as the algorithm finds the first shortest solution, it adds the solution to the set stored in the variable and continues the search with respect to the current length bound. The algorithm adds every further solution found in the current length

bound to the set. After visiting all nodes reachable with respect to the current bound, the algorithm returns the set stored in the variable instead of increasing the current length bound. The computation of all shortest solutions is interesting if there are further metrics available that measure which of these solutions is best to be presented to an engineer or for the direct application to repair the input model. Such metrics may be highly specific to the model repair problem, the model to repair, the modeling language, and the use case for which the repairing sequences are computed. We leave the development of such metrics for future work.

7.5.1 Algorithm Performance

Search tree algorithms are usually analyzed along three dimensions (*e.g.*, [Kor85]): Space complexity, time complexity, and optimality. The space complexity measures the amount of space used by the algorithm. The time complexity measures the number of operations the algorithm requires during its executions. Optimality asks whether the algorithm always finds an optimal solution on termination. Algorithm 2 is a DFID [Kor85] tree search algorithm. The algorithm is guaranteed to always find shortest solutions, if at least one exists (cf. [Kor85]). Solutions for model repair problem instances are guaranteed to exist (cf. Proposition 7.10). The space complexity of the algorithm is $\mathcal{O}(d)$ where d is the length of a shortest solution for the model repair problem instance given as input. Thus, the space complexity is linear in the length of a solution [Kor85]. The running time of the DFID algorithm is $\mathcal{O}(b^d)$ where b is the branching factor of the search tree and d is the depth of the nodes encoding the shortest solutions [Kor85]. For the algorithm, we assume the availability of an implementation of the repair representative-function \mathcal{R} and neglect the implementation's running time. The DFID algorithm is asymptotically optimal among the class of brute-force search algorithms [Kor85] in terms of space complexity, time complexity, and optimality. Therefore, there do not exist brute-force algorithms for searching the change sequence search tree for a shortest solution for a model repair problem instance with better asymptotic properties than the DFID algorithm. However, the algorithm can be enhanced with several performance improvements, as discussed in the following.

From a theoretical viewpoint, it is also possible to provide an algorithm for computing a shortest solution by implementing a breadth-first search [Kor85]. Such an algorithm would also always terminate (as by Proposition 7.10 solutions are guaranteed to exist) and its running time would also be $\mathcal{O}(b^d)$ where b is the branching factor of the search tree and d is the depth of the nodes encoding the shortest solution. However, the space complexity of the breadth-first search algorithm would also be $\mathcal{O}(b^d)$. This is a critical drawback compared to the DFID search algorithm, which only has a space complexity of $\mathcal{O}(d)$. Thus, this section solely focuses on DFID search algorithms.

7.5.2 Checking Change Operation Properties

In subsequent iterations of the outer loop of Algorithm 2 (cf. ll.2-17), the algorithm checks multiple times whether the application of the same change sequence results in a model that satisfies the property of the model repair problem (Algorithm 2, l. 7). As checking the property is usually computationally expensive (such as refinement checking), redundant property checks should be avoided as much as possible. This is achievable by not checking whether a change sequence yields a model that satisfies the property if the change sequence's length is less than or equal to the bound for the length (value stored in the variable *currentLengthBound*) of a previous iteration.

A change operation may, by construction, always change every model satisfying a property to a model that also satisfies the property. For example, if the property P contains all models that refine a model m , then the application of a refining operation to a refinement of m always yields a refinement of m . The change operation preserves the property:

Definition 7.9. Let $\mathcal{L} = (M, S, \text{sem})$ be a modeling language, O be a change operation suite for \mathcal{L} , and $P \subseteq M$ be a model property. A change operation $o \in O$ is called *P-preserving* iff $\forall m \in \text{dom}(o) : P(m) \Rightarrow P(o(m))$.

Vice versa, a change operation might never change a model, which does not satisfy a property, to a model that satisfies the property. For example, assume the property P contains all models that are a refinement of a model m . Then, the application of a generalizing operation to a model that is not a refinement of m always yields a model that is not a refinement of m . Thus, the change operation is \bar{P} -preserving, *i.e.*, it preserves the complement property of the property refinement. If a change sequence does not repair a model towards satisfying a property, then every change sequence obtained from prolonging the sequence with a change operation that preserves the complement property is no change sequence that repairs the model towards satisfying the property, either.

Proposition 7.14. Let $\mathcal{L} = (M, S, \text{sem})$ be a modeling language, $m \in M$ be a model, $P \subseteq M$ be a model property, and $t \in O^*$ be a change sequence with $m \triangleright t \in M$ and $m \triangleright t \notin P$. If $o \in O$ is \bar{P} -preserving and $m \triangleright t \in \text{dom}(o)$, then $m \triangleright (t \& o) \notin P$.

Proof. Let \mathcal{L} , m , P , and t be given as above. Assume $o \in O$ is \bar{P} -preserving and $m \triangleright t \in \text{dom}(o)$. As $m \triangleright t \notin P$, it holds that $m \triangleright t \in \bar{P}$. Therefore, as o is \bar{P} -preserving, it holds that $m \triangleright (t \& o) = o(m \triangleright t) \in \bar{P}$. \square

Unnecessary property checks are avoidable by omitting the property satisfaction checks for every change sequence that originates from appending a change operation that preserves the complement property to a sequence, which does not repair the original model. Incorporating this into the algorithm requires a procedure for checking whether a change operation preserves the complemented model property. To obtain a performance improvement, the check should be computationally inexpensive, as it is frequently executed.

Algorithm 3 Computing a shortest solution for a model repair problem instance with considering change operation properties.

Input: A model repair problem instance $I = (\mathcal{P}, m)$ where $\mathcal{P} = (\mathcal{L}, O, P)$ and a \mathcal{P} -repair-representative function \mathcal{R} .

Output: Shortest solution $t \in O^*$ for I .

```

1: define currentLengthBound  $\leftarrow 0$  as positive integer
2: loop
3:   define  $S$  as empty stack of  $O^*$ 
4:    $S.push(\varepsilon)$ 
5:   while  $S$  not empty do
6:      $t \leftarrow S.pop()$ 
7:     if  $|t| \geq \text{currentLengthBound}$  then
8:       if  $|t| = 0 \vee (|t| > 0 \wedge \text{last}(t) \text{ is not } \overline{P}\text{-preserving})$  then
9:         if  $m \triangleright t \in M \wedge P(m \triangleright t)$  then
10:          return  $t$ 
11:        end if
12:      end if
13:      else if  $m \triangleright t \in M$  then
14:        for all  $o \in \mathcal{R}(m \triangleright t)$  do
15:           $S.push(t \& o)$ 
16:        end for
17:      end if
18:    end while
19:     $\text{currentLengthBound} \leftarrow \text{currentLengthBound} + 1$ 
20: end loop

```

Algorithm 3 depicts a revised version of Algorithm 2 that incorporates the improvements described above. The differences to the original algorithm (ll. 7,8,12,13,17) are highlighted in blue. In contrast to Algorithm 2, the Algorithm 3 solely checks whether a change sequence repairs the input model in case the sequence's length is equal to the current bound for the lengths of explored change sequences (l. 7). If the length of a change sequence is smaller than the bound (l. 13), then the algorithm behaves as Algorithm 2. Before checking whether the currently explored change sequence repairs the input model, the algorithm checks whether the change operation, which has been most recently appended to the current change sequence, preserves the complement of the model repair problem's property (l. 8). If this is the case, then Proposition 7.14 guarantees that the change sequence is not a solution for the model repair problem instance because the prefix of the change sequence without the last operation is also not a solution for the instance (otherwise it had been returned in the previous iteration).

7.5.3 Propagating Properties Implying the Complement Model Property

In case a model does not satisfy a property, procedures for checking the property often yield witnesses that describe reasons why the model does not satisfy the property. For example, the semantic differencing operators described in Part II yield diff witnesses, which are concrete elements in the semantics of one model and no elements of the another model's semantics. More generally, the witnesses are interpretable to be properties that include all models that have a semantics containing the witness. If a model satisfies the witness property, then the model is also not a refinement of the model for which the witness was calculated. Therefore, the witness property implies the complement of the refinement property.

Definition 7.10. *A model property P implies a model property P' iff $P \subseteq P'$.*

A property P implies a property P' iff it is guaranteed that every model that satisfies P also satisfies P' . It may be computationally less expensive to check whether a model satisfies a property that implies another property than checking whether the model satisfies the latter property. During the exploration of the change sequence search tree, if the property checking procedure yields a property that implies the complement of the property that is to be satisfied, then the implying property can be propagated to subsequent iterations for checking whether the newly explored models satisfy the implying property. If a newly explored model satisfies the implying property, then the model is also guaranteed to not satisfy the property of the model repair problem. In case that checking the implying property is computationally less expensive than checking the property of the model repair problem, the propagation yields performance improvements.

Algorithm 4 is an algorithm obtained from incorporating the witness propagation into Algorithm 2. The relevant changes (ll. 2,9-17) are highlighted in blue. In Algorithm 4, the variable *complImplProp* stores the current property that implies the complement of the model repair problem's property (l. 2). The property variable *complImplProp* stores the disjunction of all computed properties that imply the complement of the model repair problem's property. The variable *complImplProp* is initialized as the property *false*, represented by the empty set of models. No model satisfies this property, which implies that it implies all other properties.

Before checking whether the model obtained from applying the currently processed change sequence to the input model satisfies the model repair problem property, the algorithm checks whether the model does not satisfies the property that implies the complement of the model repair problem's property (l. 9). If the model does not satisfy this property, then it may satisfy the property of the model repair problem. Otherwise, the model is guaranteed to not satisfy the model repair problem's property. Algorithm 4 stores the result from applying the property satisfaction check in the variable *checkingResult* (l. 10). The value of the variable is either a model property that implies the complement of the model repair problem property or the special value \checkmark . The

Algorithm 4 Computing a shortest solution for a model repair problem instance with considering properties implying the complement model property as witnesses.

Input: A model repair problem instance $I = (\mathcal{P}, m)$ where $\mathcal{P} = (\mathcal{L}, O, P)$ and a \mathcal{P} -repair-representative function \mathcal{R} .

Output: Shortest solution $t \in O^*$ for I .

```

1: define currentLengthBound  $\leftarrow 0$  as positive integer
2: define complImplProp  $\leftarrow \emptyset$  as property of the models of  $\mathcal{L}$ 
3: loop
4:   define  $S$  as empty stack of  $O^*$ 
5:    $S.push(\varepsilon)$ 
6:   while  $S$  not empty do
7:      $t \leftarrow S.pop()$ 
8:     if  $|t| = 0 \vee (|t| > 0 \wedge last(t) \text{ is not } \overline{P}\text{-preserving})$  then
9:       if  $m \triangleright t \in M \wedge m \triangleright t \notin complImplProp$  then
10:        define checkingResult as model property or  $\checkmark$ 
11:        checkingResult  $\leftarrow CHECKPROPERTY(\mathcal{P}, m \triangleright t)$ 
12:        if checkingResult  $= \checkmark$  then
13:          return  $t$ 
14:        else
15:          complImplProp  $\leftarrow complImplProp \cup checkingResult$ 
16:        end if
17:      end if
18:    end if
19:    if  $m \triangleright t \in M \wedge |t| < currentLengthBound$  then
20:      for all  $o \in \mathcal{R}(m \triangleright t)$  do
21:         $S.push(t \& o)$ 
22:      end for
23:    end if
24:  end while
25:  currentLengthBound  $\leftarrow currentLengthBound + 1$ 
26: end loop

```

special value \checkmark represents a successful property check, *i.e.*, that the checked model satisfies the property. The algorithm performs the property check by using the function `CheckProperty` (l. 11), which takes the model repair problem property and the model obtained from applying the currently processed sequence to the input model as inputs. The function `CheckProperty` is problem-specific and is required to produce \checkmark iff the model satisfies the property. Otherwise, if the model does not satisfy the property, the function `CheckProperty` is required to output a property that implies the complement of the model repair problem property. If the property check is successful (l. 12), then the

algorithm returns the currently processed change sequence (l. 13). Otherwise, it updates the value stored in the variable *complImplProp* to the disjunction of its current value and the property computed by the function *CheckProperty* (ll. 14-16).

7.5.4 Detecting Previously Explored Models

Two different change sequences may transform a model into the same model. If the change sequences of multiple nodes change the original model to the same model, then it suffices to continue the search in the node encoding the change sequence with the shortest length among all these sequences:

Proposition 7.15. *Let $I = (\mathcal{P}, m)$ where $\mathcal{P} = (\mathcal{L}, O, P)$ be a model repair problem instance and let $t, u \in O^*$ be two change sequences. If $m \triangleright t = m \triangleright u$ and $|t| < |u|$, then for all shortest solutions $v \in O^*$ for I , it holds that u is not a prefix of v .*

Proof. Let $m \in M$ be a model, $P \subseteq M$ be a property on models, and let $t, u \in O^*$ be two change sequences. Assume $m \triangleright t = m \triangleright u$ and $|t| < |u|$. Suppose towards a contradiction there exists a shortest change sequence v repairing m towards satisfying the property P such that u is a prefix of v . Let $s \in O^*$ such that $v = u \& s$. The sequence s exists because u is a prefix of v . As by assumption $m \triangleright t = m \triangleright u$, we have that $m \triangleright t \& s = m \triangleright u \& s$. Thus, $t \& s$ is a change sequence that repairs m towards satisfying the property P because $u \& s$ is a change sequence that repairs m towards satisfying P and $m \triangleright t \& s = m \triangleright u \& s$. This contradicts the assumption that v is a shortest change sequence that repairs m towards satisfying the property P because $|v| = |u \& s| = |u| + |s| > |t| + |s| = |t \& s|$ and $t \& s$ is a change sequence that repairs m towards satisfying P . \square

There are different possibilities for extending the algorithm to avoid the expansion of change sequences that are not prefixes of shortest solutions.

One possibility is to store the models obtained from applying already discovered change sequences together with the lengths of the already explored sequences in a set. Whenever a change sequence is newly discovered, it is possible to check whether the change sequence changes the original model to a model that is contained in the set and to check whether the length of the sequence that previously discovered the model is smaller than or equal to the length of the currently processed sequence. If this is the case, then the discovered change sequence is guaranteed to be not a prefix of a shortest solution (cf. Proposition 7.15). Thus, the expansion of the branch of the sequence can be aborted. While this variant preserves optimality and guarantees that change sequences corresponding to models that have already been discovered by shorter sequences are never expanded, the change drastically increases the algorithm's space complexity. If b is the branching factor of the subtree of depth $d \in \mathbb{N}$ of the complete change sequence search tree, then the algorithm has to store $O(b^d)$ models after exploring depth d in the worst case. Therefore, this variant is infeasible for model repair problem instances that induce large change sequence search trees.

Algorithm 5 Computing a shortest solution for a model repair problem instance, while storing previously explored models.

Input: A model repair problem instance $I = (\mathcal{P}, m)$ where $\mathcal{P} = (\mathcal{L}, O, P)$ and a \mathcal{P} -repair-representative function \mathcal{R} .

Output: Shortest solution $t \in O^*$ for I .

```

1: define currentLengthBound  $\leftarrow 0$  as positive integer
2: define complImplProp  $\leftarrow \emptyset$  as property of the models of  $\mathcal{L}$ 
3: loop
4:   define exploredWithLength  $\leftarrow \emptyset$  as set of  $M \times \mathbb{N}$ 
5:   define  $S$  as empty stack of  $O^*$ 
6:    $S.push(\varepsilon)$ 
7:   while  $S$  not empty do
8:      $t \leftarrow S.pop()$ 
9:     if  $m \triangleright t \in M \wedge \forall (e, l) \in \text{exploredWithLength} : (e \neq m \triangleright t \vee |l| > |t|)$  then
10:      exploredWithLength  $\leftarrow (\text{exploredWithLength} \setminus \{m \triangleright t\} \times \mathbb{N}) \cup \{(m \triangleright t, |t|)\}$ 
11:      if  $|t| = 0 \vee (|t| > 0 \wedge \text{last}(t) \text{ is not } \bar{P}\text{-preserving})$  then
12:        if  $m \triangleright t \in M \wedge m \triangleright t \notin \text{complImplProp}$  then
13:          define checkingResult as model property or  $\checkmark$ 
14:          checkingResult  $\leftarrow \text{CHECKPROPERTY}(\mathcal{P}, m \triangleright t)$ 
15:          if checkingResult  $= \checkmark$  then
16:            return  $t$ 
17:          else
18:            complImplProp  $\leftarrow \text{complImplProp} \cup \text{checkingResult}$ 
19:          end if
20:        end if
21:      end if
22:      if  $m \triangleright t \in M \wedge |t| < \text{currentLengthBound}$  then
23:        for all  $o \in \mathcal{R}(m \triangleright t)$  do
24:           $S.push(t \& o)$ 
25:        end for
26:      end if
27:    end if
28:  end while
29:  currentLengthBound  $\leftarrow \text{currentLengthBound} + 1$ 
30: end loop

```

Algorithm 5 is a revised version of Algorithm 4 that stores previously explored models and the lengths of the shortest explored change sequences that change the input model to the explored models. The most relevant changes (ll. 4,9,10,27) are highlighted in blue. The algorithm additionally initializes the variable *exploredWithLength* storing

a set of tuples of models and natural numbers (l. 4). In each iteration, each tuple $(e, l) \in \text{exploredWithLength}$ represents the information that the model e has been previously explored by a change sequence and that l is the length of the shortest explored change sequence that changes the input model to e . Whenever the algorithm processes a change sequence, it checks whether the model obtained from applying the change sequence to the input model has already been explored by a shorter change sequence in a previous iteration (l. 9). If this is the case, the algorithm aborts the search in the branch encoded by the change sequence and processes the next recently explored change sequence. Otherwise, it is possible that the branch encoded by the currently processed change sequence contains a shortest solution. Then, the algorithm updates the content of the variable *exploredWithLength* by adding the information that the currently processed change sequence is the shortest explored change sequence that changes the input model to the model obtained from applying the change sequence (l. 10).

Another possibility to abort searches in branches that do not contain a shortest solution includes checking whether the syntactic difference to a model obtained from applying a newly discovered change sequence to the original model is shorter than the newly discovered change sequence. If this is the case, then the newly discovered change sequence is not a prefix of a shortest solution (cf. Proposition 7.15). Therefore, the algorithm can abort expanding the branch corresponding to the change sequence. This change to the algorithm preserves optimality and does not increase its space complexity. However, it might be the case that the algorithm for computing the syntactic difference does not produce a shortest change sequence that transforms one model to another model for any two input models. Then, the algorithm may expand branches of change sequences that transform the original model to a model that has been already obtained from applying a shorter discovered change sequence. Therefore, this change does not guarantee that the search in branches of change sequences transforming the original model to an already discovered model are always aborted. The change also requires the availability of a syntactic differencing operator.

It is also possible to check whether a true prefix of the currently processed change sequence changes the original model to the same model as the currently processed sequence. In this case, the newly discovered sequence is not a prefix of a shortest solution (cf. Proposition 7.15). Incorporating this change preserves the optimality and the space complexity of the algorithm. After the change, the search space often decreases by an exponential factor. For instance, the algorithm obtained from incorporating the change detects whenever an inverse operation of the previously appended change operation is appended to the currently explored sequence. In this case, it aborts the search in the branch corresponding to the sequence. While this change does not change the space complexity of the algorithm, it does not guarantee that the resulting algorithm always aborts the search in branches that correspond to models that have been previously obtained from applying shorter explored change sequences.

Incorporating the check between the currently processed change sequence and the

Algorithm 6 Computing a shortest solution for a model repair problem instance, while checking whether the syntactic difference to a model is shorter than the currently processed sequences and checking whether a model has already been explored by a prefix of the currently processed change sequence.

Input: A model repair problem instance $I = (\mathcal{P}, m)$ where $\mathcal{P} = (\mathcal{L}, O, P)$ and a \mathcal{P} -repair-representative function \mathcal{R} .

Output: Shortest solution $t \in O^*$ for I .

```

1: define currentLengthBound  $\leftarrow 0$  as positive integer
2: define complImplProp  $\leftarrow \emptyset$  as property of the models of  $\mathcal{L}$ 
3: loop
4:   define  $S$  as empty stack of  $O^*$ 
5:    $S.push(\varepsilon)$ 
6:   while  $S$  not empty do
7:      $t \leftarrow S.pop()$ 
8:     if  $m \triangleright t \in M \wedge |\Delta(m, m \triangleright t)| \geq |t| \wedge \forall p \sqsubset t : m \triangleright p \neq m \triangleright t$  then
9:       if  $|t| = 0 \vee (|t| > 0 \wedge last(t) \text{ is not } \overline{P}\text{-preserving})$  then
10:        if  $m \triangleright t \in M \wedge m \triangleright t \notin complImplProp$  then
11:          define checkingResult as model property or  $\checkmark$ 
12:          checkingResult  $\leftarrow \text{CHECKPROPERTY}(\mathcal{P}, m \triangleright t)$ 
13:          if checkingResult =  $\checkmark$  then
14:            return  $t$ 
15:          else
16:            complImplProp  $\leftarrow complImplProp \cup checkingResult$ 
17:          end if
18:        end if
19:      end if
20:      if  $m \triangleright t \in M \wedge |t| < currentLengthBound$  then
21:        for all  $o \in \mathcal{R}(m \triangleright t)$  do
22:           $S.push(t \& o)$ 
23:        end for
24:      end if
25:    end if
26:  end while
27:  currentLengthBound  $\leftarrow currentLengthBound + 1$ 
28: end loop

```

syntactic difference as well as the check between the currently processed change sequence and its prefixes into Algorithm 4 yields Algorithm 6. The most important changes (ll. 8,25) are highlighted in blue. Whenever the algorithm starts processing a recently explored change sequence, it performs two checks (l. 8): At first, the algorithm checks

whether the length of the syntactic difference is smaller than the length of the currently processed sequence. Then, the algorithm checks whether a true prefix of the currently processed change sequence changes the input model to the same model as the currently processed sequence. If at least one of the checks is successful, then the currently processed sequence is not a prefix of a shortest solution for the model repair problem instance (cf. Proposition 7.15). Therefore, if at least one of the checks is successful, the algorithm aborts the search in the branch of the change sequence search tree encoded by the currently processed sequence. Otherwise, Algorithm 6 behaves as Algorithm 4 (ll. 9-24).

7.6 Applicability and Development Methodology

This section discusses the applicability of the framework for automatic model repairs.

Proposition 7.16. *There exists a \mathcal{P} -repair-representative function for every model repair problem \mathcal{P} .*

Proof. Let $\mathcal{P} = (\mathcal{L}, O, P)$ where $\mathcal{L} = (M, S, sem)$ be a model repair problem. For every instance $I = (\mathcal{P}, m)$ of \mathcal{P} , let t_m be an arbitrary shortest solution for I . Proposition 7.2 guarantees the existence of these solutions. We define the function \mathcal{R} as follows:

$$\mathcal{R} : M \rightarrow \wp_{fin}(O), m : \begin{cases} \emptyset, & \text{if } t_m = \varepsilon \\ \{t_m.0\}, & \text{if } t_m \neq \varepsilon. \end{cases}$$

The function \mathcal{R} is an oracle with perfect information. If the model of an instance of the model repair problem satisfies the property, then the function maps the model to the empty set. Otherwise, the function maps the model to exactly one change operation that does not delay the solution for the instance. Let $I = (\mathcal{P}, m)$ be an instance of \mathcal{P} . Assume $Del_I \neq O$. Then, it holds that the shortest solution t_m is not equal to the empty sequence, i.e., $t_m \neq \varepsilon$. Therefore, $\mathcal{R}(m) = \{t_m.0\}$. As t_m is a shortest solution for I , $t_m.0$ does not delay the solution for I and, thus, $\mathcal{R}(m) \cap (O \setminus Del_I) \neq \emptyset$.

From the above, we can conclude that \mathcal{R} satisfies $Del_I \neq O \Rightarrow \mathcal{R}(m) \cap (O \setminus Del_I) \neq \emptyset$ for every instance $I = (\mathcal{P}, m)$ of \mathcal{P} . By Proposition 7.8, this implies that \mathcal{R} is a \mathcal{P} -repair-representative function. \square

Thus, repair-representative functions always exist for all model repair problems. If an algorithm computing the repair-representative function for a model repair problem is available, then the algorithms presented in Section 7.5 can be used to fully automatically compute shortest solutions for all instances of the model repair problem. Therefore, the primary methodological challenge for developers is the identification and implementation of the repair-representative function.

The default process for the development of a repair-representative function for a concrete model repair problem is as follows: The developer restricts her view to an arbitrary

but fixed instance of the model repair problem. Then, she tries to derive properties of the (arbitrary but fixed) instance for obtaining results for all instances of the model repair problem. The developer first identifies a subset of the change operations applicable to the model of the instance that are guaranteed to delay the solution for the instance. These change operations do not need to be analyzed further. If the set of remaining change operations is infinite, then the developer tries to identify infinite sets of change operations in the set that pairwise induce an equally long shortest solution for the instance. If the identified sets divide the remaining change operations into finitely many partitions, then the developer can designate an arbitrary element from each of the sets and define the repair representative function: the model of the problem instance is mapped to the finite set containing the designated elements. In this case, as the model repair problem instance was chosen arbitrarily, the developer identified a repair-representative function. Otherwise, if the identified sets divide the remaining change operations into infinitely many partitions, then the developer reapplies the complete process with the goal to refine the identified sets of change operations.

The possibility to partition the non-delaying change operations into finitely many equivalence classes of change operations inducing an equally long shortest solution depends on whether sufficiently many change operations have been previously characterized to delay the solution. Thus, if the identified set of change operations that delay the solution is too small, then it might be impossible to partition the set of remaining change operations into finitely many equivalence classes of change operations inducing an equally long shortest solution. The set of equivalence classes may be infinite:

Proposition 7.17. *There exists a model repair problem instance $I = (\mathcal{P}, m)$ where $\mathcal{P} = (\mathcal{L}, O, P)$ such that O/\sim_I is infinite.*

Proof. We define the simple modeling language $Z = (\mathbb{Z}, \mathbb{Z}, sem)$ where each model is an integer $m \in \mathbb{Z}$, the semantic domain is the set of integers \mathbb{Z} , and the semantic mapping maps each model to the singleton set containing the model, *i.e.*, $sem_Z(m) \stackrel{\text{def}}{=} \{m\}$ for all $m \in \mathbb{Z}$. The property $P = \{0\} \subseteq \mathbb{Z}$ is defined as equality to the model $0 \in \mathbb{Z}$ of Z .

We define the change operation suite $O \stackrel{\text{def}}{=} \{add_k \mid k \in \mathbb{N}\} \cup \{dec\}$ for Z where $dec(m) = m - 1$ for all $m \in \mathbb{Z}$ and $add_k(m) = m + k$ for all $m \in \mathbb{Z}$, $k \in \mathbb{N}$. The change operation suite O contains infinitely many change operations. The operation dec decreases a model by one. For every natural number $k \in \mathbb{N}$, the operation add_k adds the value k to a model. It is easy to see that O is complete.

We define the model repair problem $\mathcal{P} = (Z, O, P)$ and an instance $I = (\mathcal{P}, 1)$. The set of equivalence classes of change operations inducing equally long shortest solutions for I is infinite: Let $k \in \mathbb{N}$. Then, it holds that $add_k(1) = 1 + k$. It is easy to see that a shortest solution for the model repair problem instance $I = (\mathcal{P}, add_k(1))$ is given by the change sequence $t = dec^{k+1}$, which contains the change operation dec exactly $k + 1$ times. We can conclude that for all $k \in \mathbb{N}$, it holds that $d_{\mathcal{P}}(add_k(1)) = k + 1$. Therefore, $add_k \not\sim_I add_l$ for all $l, k \in \mathbb{N}$ with $l \neq k$. As further $c_{\mathcal{P}}^*(dec(1)) = 0 \neq c_{\mathcal{P}}^*(add_k(1))$ for

all $k \in \mathbb{N}$, it holds that $dec \not\sim_I add_k$ for all $k \in \mathbb{N}$. We can conclude that all pairs of change operations from the infinite set O do not induce equally long shortest solutions for I , which implies that O/\sim_I is infinite. \square

Developers might be unsure about the possibility to partition the change operations that have not been characterized to delay the solution. Nevertheless, as the following shows, specifically designed change operation suites guarantee that the partitioning is always possible. If every change operation has an inverse, then for every problem instance, every subset of the change operation suite can be partitioned into finitely many sets, each containing change operations that pairwise induce an equally long shortest solution for the instance.

Proposition 7.18. *Let $I = (\mathcal{P}, m)$ where $\mathcal{P} = (\mathcal{L}, O, P)$ be a model repair problem instance. If for all change operations $o \in O$, there exists a change operation $i \in O$ that is an inverse of o , then O/\sim_I is finite.*

Proof. Let $I = (\mathcal{P}, m)$ where $\mathcal{P} = (\mathcal{L}, O, P)$ be a model repair problem instance. Assume for all change operations $o \in O$, there exists a change operation $i \in O$ that is an inverse of o . By Proposition 7.2, there exists a shortest solution $t \in O^*$ for I . Let $o \in O$ be a change operation and let $i \in O$ be the inverse of o . If $o(m) \in M$, then it holds that $c_{\mathcal{P}}^*(o(m)) \leq |i : t| = |t| + 1$ because $o(m) \triangleright (i : t) = m \triangleright t$ and $P(m \triangleright t)$. Therefore, for every change operation $o \in O$, it holds that $o(m) \notin M$ or $o(m) \in M \wedge c_{\mathcal{P}}^*(o(m)) \leq 1 + |t|$. Therefore, \sim_I has at most $2 + |t|$ many equivalence classes, i.e., $|O/\sim_I| \leq 2 + |t|$. \square

Therefore, if every change operation has an inverse, it is always possible to divide the change operations that have not been characterized as delaying into finitely many sets of change operations inducing an equally long shortest solution.

7.7 Composing Model Repair Problems

Complex model properties can be composed of multiple simpler model properties by using the basic set operations for unifying, intersecting, and complementing the sets representing the properties.

The composition of properties by using the basic set operations can be lifted to model repair problems for the same modeling language and the same change operations suite. For example, if (\mathcal{L}, O, P_1) and (\mathcal{L}, O, P_2) are model repair problems, $\overline{P_1} \neq \emptyset$, and $P_1 \cap P_2 \neq \emptyset$, then $(\mathcal{L}, O, P_1 \cup P_2)$, $(\mathcal{L}, O, \overline{P_1})$, and $(\mathcal{L}, O, P_1 \cap P_2)$ are also model repair problems.

The composition can be further lifted to model repair problem instances that use the same model. The *complement* of a model repair problem instance (\mathcal{P}, m) where $\mathcal{P} = (\mathcal{L}, O, P)$ is defined as (\mathcal{P}', m) where $\mathcal{P}' = (\mathcal{L}, O, \overline{P})$. The *union* of two model repair problem instances (\mathcal{P}_1, m) and (\mathcal{P}_2, m) is defined as (\mathcal{P}, m) where $\mathcal{P} = (\mathcal{L}, O, P_1 \cup P_2)$. The *intersection* of two model repair problem instances is defined analogously.

In general, the identification of equivalent change operations under “induce equally long shortest solution relations” and the identification of change operations that delay solutions is not trivial. Therefore, in case a model repair problem is a composition of two simpler model repair problems, it would be beneficial to be able to derive properties of the change operations in the context of the composed model repair problem from the properties of the change operations in the contexts of the simpler model repair problems. This would facilitate developers in constructing repair-representative functions for complex model repair problems that are composed of simpler model repair problems.

However, this section shows that it is, in most cases, not possible to derive properties of change operations in the context of a model repair problem that is composed of simpler model repair problems from the properties of the change operations in the contexts of the simpler model repair problems. Further, in most cases, it is not possible to derive a repair-representative function for a model repair problem that is composed of simpler model repair problems from the repair-representative functions for the simpler problems. The cases where derivations are possible are methodologically of little interest.

In the following, Section 7.7.1 is concerned with deriving whether a change operation delays the solution for an instance of a composed model repair problem. Section 7.7.2 discusses the derivation of change operations that induce equally long shortest solutions for instances of composed model repair problems. Subsequently, Section 7.7.3 is concerned with the derivation of repair-representative functions.

7.7.1 Derivation of Operations that Delay Solutions

If a change operation does not delay the solution for an instance of a model repair problem, then it is guaranteed to delay the solution for the complement of the instance.

Proposition 7.19. *Let $\mathcal{P} = (\mathcal{L}, O, P)$ where $\mathcal{L} = (M, \text{Sem}, \text{sem})$ be a model repair problem. Assume that $\overline{\mathcal{P}} = (\mathcal{L}, O, \overline{P})$ is a model repair problem. Then, for all change operations $o \in O$ and all models $m \in M$, it holds that $o \notin \text{Del}_{(\mathcal{P}, m)} \Rightarrow o \in \text{Del}_{(\overline{\mathcal{P}}, m)}$.*

Proof. Let $\mathcal{L} = (M, \text{Sem}, \text{sem})$, \mathcal{P} and $\overline{\mathcal{P}}$ be given as above. Let $o \in O$ be a change operation and let $m \in M$ be a model. Assume $o \notin \text{Del}_{(\mathcal{P}, m)}$. As $o \notin \text{Del}_{(\mathcal{P}, m)}$, it holds that $m \notin P$. Therefore, $m \in \overline{P}$. Thus, ε is a shortest solution for the model repair problem instance $(\overline{\mathcal{P}}, m)$. Therefore, o delays the solution for the instance $(\overline{\mathcal{P}}, m)$. \square

However, change operations that do not delay the solution for a model repair problem instance are usually not directly identified during the development of repair-representative functions (cf. Section 7.6). Therefore, Proposition 7.19 can be rarely applied and is methodologically of little interest for the development of repair-representative functions. In general, the methodologically more interesting other direction of the statement in Proposition 7.19 does not hold. It is not possible to determine whether a change opera-

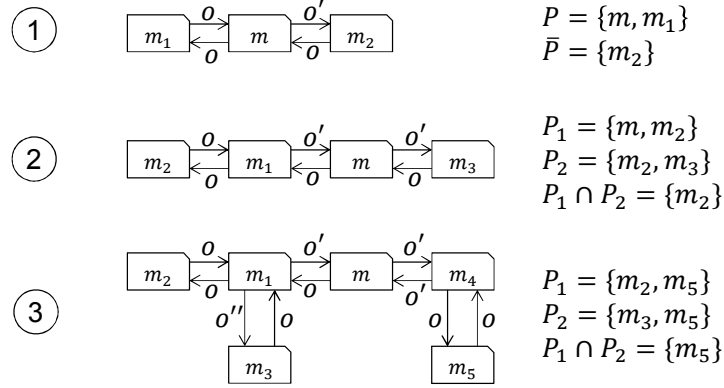


Figure 7.12: Illustration of the models, the properties, and the change operations used in the proofs of Proposition 7.20, Proposition 7.23, and Proposition 7.24.

tion delays the solution for a model repair problem instance from the information that the change operation delays the solution for the instance's complement.

Proposition 7.20. *There exists a model repair problem $\mathcal{P} = (\mathcal{L}, O, P)$ where $\mathcal{L} = (M, \text{Sem}, \text{sem})$, a model $m \in M$, and a change operation $o \in O$ such that $\bar{\mathcal{P}} = (\mathcal{L}, O, \bar{P})$ is a model repair problem and $o \in \text{Del}_{(\mathcal{P}, m)} \wedge o \in \text{Del}_{(\bar{\mathcal{P}}, m)}$.*

Proof. We define $\mathcal{L} \stackrel{\text{def}}{=} (M, M, \text{sem})$ where $M = \{m, m_1, m_2\}$, $\text{sem}(x) = \emptyset$ for all $x \in M$, $O \stackrel{\text{def}}{=} \{o, o'\}$ where $o \stackrel{\text{def}}{=} \{m : m_1, m_1 : m, m_2 : m\}$, $o' \stackrel{\text{def}}{=} \{m : m_2\}$, and $P = \{m, m_1\}$. The first part of Figure 7.12 illustrates the models, the change operations, and the model properties. It is easy to verify that $\mathcal{P} = (\mathcal{L}, O, P)$ and $\bar{\mathcal{P}} = (\mathcal{L}, O, \bar{P})$ are model repair problems and that $o \in \text{Del}_{(\mathcal{P}, m)} \wedge o \in \text{Del}_{(\bar{\mathcal{P}}, m)}$. \square

If a change operation delays the solution for two model repair problem instances using the same model, then the operation also delays the solution for the union of the instances.

Proposition 7.21. *Let $\mathcal{P}_1 = (\mathcal{L}, O, P_1)$ and $\mathcal{P}_2 = (\mathcal{L}, O, P_2)$ be model repair problems where $\mathcal{L} = (M, \text{Sem}, \text{sem})$. Let $\mathcal{P} = (\mathcal{L}, O, P_1 \cup P_2)$. Then, for all change operations $o \in O$ and all models $m \in M$, it holds that $o \in \text{Del}_{(\mathcal{P}_1, m)} \cap \text{Del}_{(\mathcal{P}_2, m)} \Rightarrow o \in \text{Del}_{(\mathcal{P}, m)}$.*

Proof. Let $\mathcal{L} = (M, \text{Sem}, \text{sem})$, \mathcal{P}_1 , \mathcal{P}_2 , and \mathcal{P} be given as above. Let $o \in O$ be a change operation and let $m \in M$ be a model. Assume $o \in \text{Del}_{(\mathcal{P}_1, m)} \cap \text{Del}_{(\mathcal{P}_2, m)}$. Suppose towards a contradiction that $o \notin \text{Del}_{(\mathcal{P}, m)}$. Then, there exists a shortest solution $t \in O^*$ for (\mathcal{P}, m) such that $|t| > 0$ and $t.0 = o$. By definition of solution, it must hold that $m \triangleright t \in P_1$ or $m \triangleright t \in P_2$. Without loss of generality, assume $m \triangleright t \in P_1$. Then, t is also a solution for (\mathcal{P}_1, m) . However, as $t.0 \in \text{Del}_{(\mathcal{P}_1, m)}$, it holds that t is not a shortest solution for (\mathcal{P}_1, m) . Thus, there exists a solution v for (\mathcal{P}_1, m) such that $|v| < |t|$. As v

is a solution for (\mathcal{P}_1, m) , it holds that $m \triangleright v \in P_1$ and, thus, v is also a solution for (\mathcal{P}, m) . However, this contradicts that t is a shortest solution for (\mathcal{P}, m) because $|v| < |t|$. \square

Analogously, if a change operation does not delay the solution for two model repair problem instances using the same model, then the operation does not delay the solution for the union of the instances.

Proposition 7.22. *Let $\mathcal{P}_1 = (\mathcal{L}, O, P_1)$ and $\mathcal{P}_2 = (\mathcal{L}, O, P_2)$ be model repair problems where $\mathcal{L} = (M, Sem, sem)$. Let $\mathcal{P} = (\mathcal{L}, O, P_1 \cup P_2)$. Then, for all change operations $o \in O$ and all models $m \in M$, it holds that $o \notin Del_{(\mathcal{P}_1, m)} \wedge o \notin Del_{(\mathcal{P}_2, m)} \Rightarrow o \notin Del_{(\mathcal{P}, m)}$.*

Proof. Let $\mathcal{L} = (M, Sem, sem)$, \mathcal{P}_1 , \mathcal{P}_2 , and \mathcal{P} be given as above. Let $o \in O$ be a change operation and let $m \in M$ be a model. Assume $o \notin Del_{(\mathcal{P}_1, m)} \wedge o \notin Del_{(\mathcal{P}_2, m)}$. Then, there exist change sequences $t, u \in O^*$ such that $t.0 = o$ and $u.0 = o$ and t is a shortest solution for (\mathcal{P}_1, m) and u is a shortest solution for (\mathcal{P}_2, m) . Suppose towards a contradiction that $o \in Del_{(\mathcal{P}, m)}$. This implies that t and u are not shortest solutions for (\mathcal{P}, m) because $t.0 = u.0 = o$. Thus, there exists a solution $v \in O^*$ for (\mathcal{P}, m) such that $|v| < |t|$ and $|v| < |u|$. As v is a solution for (\mathcal{P}, m) , it holds that $m \triangleright v \in P_1$ or $m \triangleright v \in P_2$. Without loss of generality, assume $m \triangleright v \in P_1$. Then, v is a solution for (\mathcal{P}_1, m) . However, this contradicts that t is a shortest solution for (\mathcal{P}_1, m) because $|v| < |t|$. \square

Proposition 7.21 and Proposition 7.22 are not transferable to the context of intersecting model repair problem instances. If a change operation delays the solution for two model repair problem instances using the same model, then the change operation does not necessarily delay the solution for the intersection of the instances.

Proposition 7.23. *There exist model repair problems $\mathcal{P}_1 = (\mathcal{L}, O, P_1)$, $\mathcal{P}_2 = (\mathcal{L}, O, P_2)$ where $\mathcal{L} = (M, Sem, sem)$, a model $m \in M$, and a change operation $o \in O$ such that $\mathcal{P} = (\mathcal{L}, O, P_1 \cap P_2)$ is a model repair problem and $o \in Del_{(\mathcal{P}_1, m)} \wedge o \in Del_{(\mathcal{P}_2, m)} \wedge o \notin Del_{(\mathcal{P}, m)}$.*

Proof. We define $\mathcal{L} \stackrel{\text{def}}{=} (M, M, sem)$ where $M = \{m, m_1, m_2, m_3\}$, $sem(x) = \emptyset$ for all $x \in M$, $O \stackrel{\text{def}}{=} \{o, o'\}$ where $o \stackrel{\text{def}}{=} \{m : m_1, m_1 : m_2, m_2 : m_1, m_3 : m\}$, $o' \stackrel{\text{def}}{=} \{m : m_3, m_1 : m\}$, $\mathcal{P}_1 \stackrel{\text{def}}{=} (\mathcal{L}, O, \{m, m_2\})$, and $\mathcal{P}_2 \stackrel{\text{def}}{=} (\mathcal{L}, O, \{m_2, m_3\})$. The second part of Figure 7.12 illustrates the outlined circumstance. It is easy to verify that $\mathcal{P} = (\mathcal{L}, O, \{m_2\})$ is a model repair problem and $o \in Del_{(\mathcal{P}_1, m)} \wedge o \in Del_{(\mathcal{P}_2, m)} \wedge o \notin Del_{(\mathcal{P}, m)}$. \square

Analogously, if a change operation does not delay the solution for two model repair problem instances using the same model, then the change operation does not necessarily not delay the solution for the intersection of the instances.

Proposition 7.24. *There exist model repair problems $\mathcal{P}_1 = (\mathcal{L}, O, P_1)$, $\mathcal{P}_2 = (\mathcal{L}, O, P_2)$ where $\mathcal{L} = (M, Sem, sem)$, a model $m \in M$, and a change operation $o \in O$ such that $\mathcal{P} = (\mathcal{L}, O, P_1 \cap P_2)$ is a model repair problem and $o \notin Del_{(\mathcal{P}_1, m)} \wedge o \notin Del_{(\mathcal{P}_2, m)} \wedge o \in Del_{(\mathcal{P}, m)}$.*

Proof. We define $\mathcal{L} \stackrel{\text{def}}{=} (M, M, sem)$ where $M = \{m, m_1, m_2, m_3, m_4, m_5\}$, $sem(x) = \emptyset$ for all $x \in M$, $O \stackrel{\text{def}}{=} \{o, o', o''\}$ where $o \stackrel{\text{def}}{=} \{m : m_1, m_1 : m_2, m_2 : m_1, m_3 : m_1, m_4 : m_5, m_5 : m_4\}$, $o' \stackrel{\text{def}}{=} \{m : m_4, m_1 : m, m_4 : m\}$, $o'' \stackrel{\text{def}}{=} \{m_1 : m_3\}$, $\mathcal{P}_1 \stackrel{\text{def}}{=} (\mathcal{L}, O, \{m_2, m_5\})$, and $\mathcal{P}_2 \stackrel{\text{def}}{=} (\mathcal{L}, O, \{m_3, m_5\})$. The third part of Figure 7.12 illustrates the outlined circumstance. It is easy to verify that $\mathcal{P} = (\mathcal{L}, O, \{m_5\})$ is a model repair problem and that $o \notin Del_{(\mathcal{P}_1, m)} \wedge o \notin Del_{(\mathcal{P}_2, m)} \wedge o \in Del_{(\mathcal{P}, m)}$ holds. \square

7.7.2 Derivation of Operations that Induce Equally Long Shortest Solutions

If two change operations induce equally long shortest solutions for a model repair problem instance, then the change operations do not necessarily induce an equally long shortest solution for the complement of the instance. Similarly, they do not necessarily not induce an equally long shortest solution for the complement of the instance.

Proposition 7.25. *For each of the following statements, there exists a modeling language $\mathcal{L} = (M, Sem, sem)$, a change operation suite O , a model repair problem $\mathcal{P} = (\mathcal{L}, O, P)$, a model $m \in M$, and two change operations $o, o' \in O$ such that $\overline{\mathcal{P}} = (\mathcal{L}, O, \overline{P})$ is a model repair problem and the statement is satisfied:*

1. $o \sim_{(\mathcal{P}, m)} o' \wedge o \not\sim_{(\overline{\mathcal{P}}, m)} o'$.
2. $o \sim_{(\mathcal{P}, m)} o' \wedge o \sim_{(\overline{\mathcal{P}}, m)} o'$.

Proof. Proof of 1: We define $\mathcal{L} \stackrel{\text{def}}{=} (M, M, sem)$ where $M = \{m, m_1, m_2\}$, $sem(m) = \emptyset$ for all $m \in M$, $O \stackrel{\text{def}}{=} \{o, o'\}$ where $o \stackrel{\text{def}}{=} \{m : m_1, m_1 : m_2, m_2 : m\}$, $o' \stackrel{\text{def}}{=} \{m : m_2\}$, and $P \stackrel{\text{def}}{=} \{m_1, m_2\}$. The first part of Figure 7.13 illustrates the models, the change operations, and the model properties. It is easy to verify that $\mathcal{P} = (\mathcal{L}, O, P)$ and $\overline{\mathcal{P}} = (\mathcal{L}, O, \overline{P})$ are model repair problems and that $o \sim_{(\mathcal{P}, m)} o' \wedge o \not\sim_{(\overline{\mathcal{P}}, m)} o'$.

Proof of 2: We define $\mathcal{L} \stackrel{\text{def}}{=} (M, M, sem)$ where $M = \{m, m_1\}$, $sem(m) = \emptyset$ for all $m \in M$, $O \stackrel{\text{def}}{=} \{o, o'\}$ where $o \stackrel{\text{def}}{=} \{m : m_1, m_1 : m\}$, $o' \stackrel{\text{def}}{=} \{m : m_1\}$, and $P = \{m_1\}$. The second part of Figure 7.13 illustrates the models, the change operations, and the model properties. It is easy to verify that $\mathcal{P} = (\mathcal{L}, O, P)$ and $\overline{\mathcal{P}} = (\mathcal{L}, O, \overline{P})$ are model repair problems and that $o \sim_{(\mathcal{P}, m)} o' \wedge o \sim_{(\overline{\mathcal{P}}, m)} o'$. \square

If two change operations induce an equally long shortest solution for two model repair problem instances using the same model, then they also induce an equally long shortest solution for the union of the instances.

Proposition 7.26. *Let $\mathcal{P}_1 = (\mathcal{L}, O, P_1)$ and $\mathcal{P}_2 = (\mathcal{L}, O, P_2)$ be model repair problems where $\mathcal{L} = (M, \text{Sem}, \text{sem})$. Let $\mathcal{P} = (\mathcal{L}, O, P_1 \cup P_2)$. Then, for all change operations $o \in O$ and all models $m \in M$, it holds that $(o \sim_{(\mathcal{P}_1, m)} o' \wedge o \sim_{(\mathcal{P}_2, m)} o') \Rightarrow o \sim_{(\mathcal{P}, m)} o'$.*

Proof. Let $\mathcal{L} = (M, \text{Sem}, \text{sem})$, O , \mathcal{P}_1 , \mathcal{P}_2 , and \mathcal{P} be given as above. Let $o, o' \in O$ be two change operations and let $m \in M$ be a model. Assume $o \sim_{(\mathcal{P}_1, m)} o' \wedge o \sim_{(\mathcal{P}_2, m)} o'$ holds. Then, it holds that $o(m), o'(m) \notin M$ or $o(m), o'(m) \in M \wedge d_{\mathcal{P}_1}(o(m)) = d_{\mathcal{P}_1}(o'(m)) \wedge d_{\mathcal{P}_2}(o(m)) = d_{\mathcal{P}_2}(o'(m))$. If $o(m), o'(m) \notin M$, then it holds by definition that $o \sim_{(\mathcal{P}, m)} o'$. Assume $o(m), o'(m) \in M \wedge d_{\mathcal{P}_1}(o(m)) = d_{\mathcal{P}_1}(o'(m)) \wedge d_{\mathcal{P}_2}(o(m)) = d_{\mathcal{P}_2}(o'(m))$. Without loss of generality, assume that $d_{\mathcal{P}_1}(o(m)) = d_{\mathcal{P}_1}(o'(m)) \leq d_{\mathcal{P}_2}(o(m)) = d_{\mathcal{P}_2}(o'(m))$.

In the following, we show that $d_{\mathcal{P}_1}(o(m)) = d_{\mathcal{P}}(o(m)) = d_{\mathcal{P}}(o'(m)) = d_{\mathcal{P}_1}(o'(m))$, which directly implies that $o \sim_{(\mathcal{P}, m)} o'$. As $d_{\mathcal{P}_1}(o(m))$ is the length of all shortest solutions for $(\mathcal{P}_1, o(m))$ and as $P_1 \subseteq P$, it holds that the lengths of the shortest solutions for $(\mathcal{P}, o(m))$ are bounded by $d_{\mathcal{P}_1}(o(m))$, i.e., $d_{\mathcal{P}}(o(m)) \leq d_{\mathcal{P}_1}(o(m))$. Suppose towards a contradiction that $d_{\mathcal{P}}(o(m)) \neq d_{\mathcal{P}_1}(o(m))$. Then, $d_{\mathcal{P}}(o(m)) < d_{\mathcal{P}_1}(o(m))$. Thus, there exists a change sequence $t \in O^*$ with $o(m) \triangleright t \in P_1 \cup P_2$ and $|t| < d_{\mathcal{P}_1}(o(m))$. If $o(m) \triangleright t \in P_1$, then t is also a solution for $(\mathcal{P}_1, o(m))$, which contradicts that $d_{\mathcal{P}_1}(o(m))$ is the length of all shortest solutions for $(\mathcal{P}_1, o(m))$ because $|t| < d_{\mathcal{P}_1}(o(m))$. If $o(m) \triangleright t \in P_2$, then t is also a solution for $(\mathcal{P}_2, o(m))$, which contradicts with $|t| < d_{\mathcal{P}_1}(o(m))$ and $d_{\mathcal{P}_1}(o(m)) \leq d_{\mathcal{P}_2}(o(m))$ that $d_{\mathcal{P}_2}(o(m))$ is the length of all shortest solutions for $(\mathcal{P}_2, o(m))$. Analogously, one can show that $d_{\mathcal{P}}(o'(m)) = d_{\mathcal{P}_1}(o'(m))$. \square

The other direction does not hold. It is possible that two change operations do not induce equally long shortest solutions for two model repair problem instances, although they induce an equally long shortest solution for the union of the instances.

Proposition 7.27. *There exist a modeling language $\mathcal{L} = (M, \text{Sem}, \text{sem})$, a change operation suite O , model repair problems $\mathcal{P}_1 = (\mathcal{L}, O, P_1)$, $\mathcal{P}_2 = (\mathcal{L}, O, P_2)$ a model $m \in M$, and two change operations $o, o' \in O$ such that $o \not\sim_{(\mathcal{P}_1, m)} o' \wedge o \not\sim_{(\mathcal{P}_2, m)} o' \wedge o \sim_{(\mathcal{P}, m)} o'$ where $\mathcal{P} = (\mathcal{L}, O, P_1 \cup P_2)$.*

Proof. We define $\mathcal{L} \stackrel{\text{def}}{=} (M, M, \text{sem})$ where $M = \{m, m_1, m_2\}$, $\text{sem}(x) = \emptyset$ for all $x \in M$, $O \stackrel{\text{def}}{=} \{o, o'\}$ where $o \stackrel{\text{def}}{=} \{m : m_1, m_1 : m\}$, $o' \stackrel{\text{def}}{=} \{m : m_2, m_2 : m\}$, $\mathcal{P}_1 = (\mathcal{L}, O, \{m, m_1\})$, and $\mathcal{P}_2 = (\mathcal{L}, O, \{m, m_2\})$. The third part of Figure 7.13 illustrates the models, the change operations, and the model properties. It is easy to verify that $\mathcal{P} = (\mathcal{L}, O, \{m, m_1, m_2\})$ is a model repair problem and that $o \not\sim_{(\mathcal{P}_1, m)} o' \wedge o \not\sim_{(\mathcal{P}_2, m)} o' \wedge o \sim_{(\mathcal{P}, m)} o'$. \square

It is not possible to conclude whether two change operations induce an equally long shortest solution for the intersection of two model repair problem instances from the information whether the two change operations induce an equally long shortest solution for the instances. If two change operations induce equally long shortest solutions for

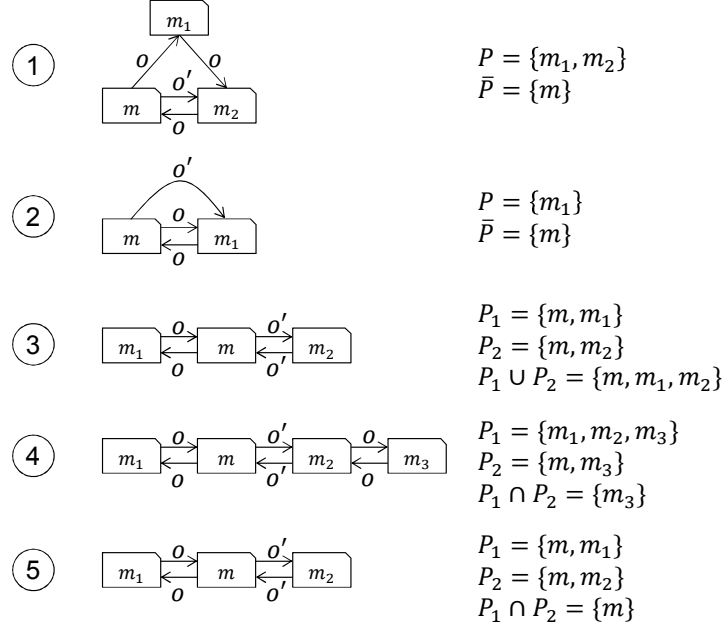


Figure 7.13: Illustration of the models, the properties, and the change operations used in the proofs of Proposition 7.25, Proposition 7.27, and Proposition 7.28.

two model repair problem instances, then they do not necessarily induce an equally long shortest solution for the intersection of the instances. Vice versa, if two change operations do not induce equally long shortest solution for two model repair problem instances, then they do not necessarily not induce an equally long shortest solution for the intersection of the instances.

Proposition 7.28. *For each of the following statements, there exist a modeling language $\mathcal{L} = (M, Sem, sem)$, a change operation suite O , model repair problems $\mathcal{P}_1 = (\mathcal{L}, O, P_1)$, $\mathcal{P}_2 = (\mathcal{L}, O, P_2)$, a model $m \in M$, and two change operations $o, o' \in O$ such that $\mathcal{P} = (\mathcal{L}, O, P_1 \cap P_2)$ is a model repair problem and the statement is satisfied:*

1. $o \sim_{(\mathcal{P}_1, m)} o' \wedge o \sim_{(\mathcal{P}_2, m)} o' \wedge o \not\sim_{(\mathcal{P}, m)} o'$.
2. $o \not\sim_{(\mathcal{P}_1, m)} o' \wedge o \not\sim_{(\mathcal{P}_2, m)} o' \wedge o \sim_{(\mathcal{P}, m)} o'$.

Proof. Proof of 1: We define $\mathcal{L} \stackrel{\text{def}}{=} (M, M, sem)$ where $M = \{m, m_1, m_2, m_3\}$, $sem(x) = \emptyset$ for all $x \in M$, $O \stackrel{\text{def}}{=} \{o, o'\}$ where $o \stackrel{\text{def}}{=} \{m : m_1, m_1 : m, m_2 : m_3, m_3 : m_2\}$, $o' \stackrel{\text{def}}{=} \{m : m_2, m_2 : m\}$, $\mathcal{P}_1 \stackrel{\text{def}}{=} (\mathcal{L}, O, \{m_1, m_2, m_3\})$, and $\mathcal{P}_2 \stackrel{\text{def}}{=} (\mathcal{L}, O, \{m, m_3\})$. The fourth part of Figure 7.13 illustrates the models, the change operations, and the model properties. It is easy to verify that $\mathcal{P} = (\mathcal{L}, O, \{m_3\})$ is a model repair problem and that $o \sim_{(\mathcal{P}_1, m)} o' \wedge o \sim_{(\mathcal{P}_2, m)} o' \wedge o \not\sim_{(\mathcal{P}, m)} o'$.

Proof of 2: We define $\mathcal{L} \stackrel{\text{def}}{=} (M, M, \text{sem})$ where $M = \{m, m_1, m_2\}$, $\text{sem}(x) = \emptyset$ for all $x \in M$, $O \stackrel{\text{def}}{=} \{o, o'\}$ where $o \stackrel{\text{def}}{=} \{m : m_1, m_1 : m\}$, $o' \stackrel{\text{def}}{=} \{m : m_2, m_2 : m\}$

$\mathcal{P}_1 = (\mathcal{L}, O, \{m, m_1\})$, and $\mathcal{P}_2 = (\mathcal{L}, O, \{m, m_2\})$. The fifth part of Figure 7.13 illustrates the models, the change operations, and the model properties. It is easy to verify that $\mathcal{P} = (\mathcal{L}, O, \{m\})$ is a model repair problem and that $o \not\sim_{(\mathcal{P}_1, m)} o' \wedge o \not\sim_{(\mathcal{P}_2, m)} o' \wedge o \sim_{(\mathcal{P}, m)} o'$ holds. \square

7.7.3 Repair-Representative Function Derivation

It is possible to construct a repair-representative function for the union of two model repair problems from repair-representative functions for the two problems. The constructed function maps each model to the union of the sets of change operations that are related to the model by the other two repair-representative functions.

Proposition 7.29. *Let $\mathcal{L} = (M, \text{Sem}, \text{sem})$ be a modeling language and let $\mathcal{P}_1 = (\mathcal{L}, O, P_1)$, $\mathcal{P}_2 = (\mathcal{L}, O, P_2)$ be model repair problems. If \mathcal{R}_1 is a \mathcal{P}_1 -repair-representative function and \mathcal{R}_2 is a \mathcal{P}_2 -repair-representative function, then $\mathcal{R} : M \rightarrow \wp_{fin}(O)$ with $\forall m \in M : \mathcal{R}(m) \stackrel{\text{def}}{=} \mathcal{R}_1(m) \cup \mathcal{R}_2(m)$ is a $(\mathcal{L}, O, P_1 \cup P_2)$ -repair-representative function.*

Proof. Let \mathcal{L} , $\mathcal{P}_1 = (\mathcal{L}, O, P_1)$, and $\mathcal{P}_2 = (\mathcal{L}, O, P_2)$ be given as above. Let \mathcal{R}_1 be a \mathcal{P}_1 -repair-representative function and let \mathcal{R}_2 be a \mathcal{P}_2 -repair-representative function. We define the function $\mathcal{R} : M \rightarrow \wp_{fin}(O)$ with $\forall m \in M : \mathcal{R}(m) \stackrel{\text{def}}{=} \mathcal{R}_1(m) \cup \mathcal{R}_2(m)$. Let $\mathcal{P} \stackrel{\text{def}}{=} (\mathcal{L}, O, P_1 \cup P_2)$ and let $m \in M$ be an arbitrary model.

By Proposition 7.9, there exist change sequences $t, u \in O^*$ such that t is a shortest solution for (\mathcal{P}_1, m) with $t.i \in \mathcal{R}_1(m \triangleright t \downarrow i)$ for all $i \in \mathbb{N}$ with $0 \leq i < |t|$ and u is a shortest solution for (\mathcal{P}_2, m) with $u.i \in \mathcal{R}_2(m \triangleright u \downarrow i)$ for all $i \in \mathbb{N}$ with $0 \leq i < |u|$. Without loss of generality, assume $|t| \leq |u|$. As t is a solution for (\mathcal{P}_1, m) , and $P_1 \subseteq P_1 \cup P_2$, it holds that t is also a solution for (\mathcal{P}, m) . Thus, it holds that $d_{\mathcal{P}}(m) \leq |t|$.

Suppose towards a contradiction that t is not a shortest solution for (\mathcal{P}, m) . Then, there exists a solution v for (\mathcal{P}, m) such that $|v| < |t|$. As v is a solution for (\mathcal{P}, m) , it holds that $m \triangleright v \in M \wedge m \triangleright v \in P_1 \cup P_2$. If $m \triangleright v \in P_1$, then t would not be a shortest solution for (\mathcal{P}, m) because $|v| < |t|$. If $m \triangleright v \in P_2$, then it would hold that v is a solution for (\mathcal{P}_2, m) , which would imply that $|u| < |v|$ and contradict the assumption that $|t| < |u|$ as $|v| < |t|$. We can conclude that t is a shortest solution for (\mathcal{P}, m) .

If $t = \varepsilon$, then $\text{Del}_I = O$ because t is a shortest solution for I . If $t \neq \varepsilon$, then $t.0 \notin \text{Del}_I$ because t is a shortest solution for I . Thus, $\text{Del}_I \neq O$ and $t.0 \in O \setminus \text{Del}_I$. As further $t.0 \in \mathcal{R}_1(m)$ and $\mathcal{R}(m) = \mathcal{R}_1(m) \cup \mathcal{R}_2(m)$, it holds that $t.0 \in \mathcal{R}(m)$. Thus, $t.0 \in \mathcal{R}(m) \cap (O \setminus \text{Del}_I)$. We can conclude that $\text{Del}_I \neq O \Rightarrow \mathcal{R}(m) \cap (O \setminus \text{Del}_I) \neq \emptyset$ holds. As the model m was chosen arbitrarily, we can conclude, it holds that $\text{Del}_I \neq O \Rightarrow \mathcal{R}(m) \cap (O \setminus \text{Del}_I) \neq \emptyset$ for every instance (\mathcal{P}, m) of \mathcal{P} . Using Proposition 7.8, we obtain that \mathcal{R} is a \mathcal{P} -repair-representative function. \square



Figure 7.14: Illustration of the models, the properties, and the change operations used in the proof of Proposition 7.30.

However, using the constructed repair-representative function to compute shortest solutions for instances of the union of two model repair problems is methodologically of little interest. The proof of Proposition 7.29 shows that the shortest solution for an instance of the union is equal to the shortest solution for one of the corresponding instances of the original model repair problems. Therefore, it is possible to compute shortest solutions for instances of the union by individually computing shortest solutions for the corresponding instances of the original problems.

Methodologically more interesting is the construction of a repair-representative function for the intersection of two model repair problems. In general, the construction used in Proposition 7.29 cannot be used to construct repair-representative functions for the intersection of model repair problems.

Proposition 7.30. *There exist a modeling language $\mathcal{L} = (M, Sem, sem)$, model repair problems $\mathcal{P}_1 = (\mathcal{L}, O, P_1)$ and $\mathcal{P}_2 = (\mathcal{L}, O, P_2)$, a \mathcal{P}_1 -repair-representative function \mathcal{R}_1 , and a \mathcal{P}_2 -repair-representative function \mathcal{R}_2 such that $(\mathcal{L}, O, P_1 \cap P_2)$ is a model repair problem and $\mathcal{R} : M \rightarrow \wp_{fin}(O)$ with $\forall m \in M : \mathcal{R}(m) \stackrel{\text{def}}{=} \mathcal{R}_1(m) \cup \mathcal{R}_2(m)$ is not a $(\mathcal{L}, O, P_1 \cap P_2)$ -repair representative function.*

Proof. We define $\mathcal{L} \stackrel{\text{def}}{=} (M, M, sem)$ where $M = \{m, m_1, m_2, m_3\}$, $sem(x) = \emptyset$ for all $x \in M$, $O \stackrel{\text{def}}{=} \{o, o', o''\}$ where $o \stackrel{\text{def}}{=} \{m : m_1, m_1 : m, m_2 : m, m_3 : m\}$, $o' \stackrel{\text{def}}{=} \{m : m_2\}$, $o'' \stackrel{\text{def}}{=} \{m : m_3\}$, $\mathcal{P}_1 \stackrel{\text{def}}{=} (\mathcal{L}, O, \{m_1, m_3\})$, and $\mathcal{P}_2 \stackrel{\text{def}}{=} (\mathcal{L}, O, \{m_2, m_3\})$. Figure 7.14 illustrates the models, the change operations, and the model properties.

It is easy to verify that $\mathcal{P} = (\mathcal{L}, O, \{m_3\})$ is a model repair problem, $\mathcal{R}_1 \stackrel{\text{def}}{=} \{m : \{o\}, m_1 : \emptyset, m_2 : \{o\}, m_3 : \emptyset\}$ is a \mathcal{P}_1 -repair-representative function, and $\mathcal{R}_2 \stackrel{\text{def}}{=} \{m : \{o'\}, m_1 : \{o\}, m_2 : \emptyset, m_3 : \emptyset\}$ is a \mathcal{P}_2 -repair-representative function. However, $\mathcal{R} : M \rightarrow \wp_{fin}(O)$ with $\forall x \in M : \mathcal{R}(x) \stackrel{\text{def}}{=} \mathcal{R}_1(x) \cup \mathcal{R}_2(x)$, i.e., $\mathcal{R} = \{m : \{o, o'\}, m_1 : \{o\}, m_2 : \{o\}, m_3 : \emptyset\}$ is not a \mathcal{P} -repair-representative function because $o'' \notin \mathcal{R}(m)$. \square

If \mathcal{R} is a repair-representative function and F is a function that maps each model to a finite superset of the set of change operations related to the model by the function \mathcal{R} , then F is also a repair-representative function.

Proposition 7.31. *Let $\mathcal{L} = (M, Sem, sem)$ be a modeling language and $\mathcal{P} = (\mathcal{L}, O, P)$ be a model repair problem. If \mathcal{R} is a \mathcal{P} -repair-representative function, then each function $F : M \rightarrow \wp_{fin}(O)$ with $\forall m \in M : \mathcal{R}(m) \subseteq F(m)$ is a \mathcal{P} -repair-representative function.*

Proof. Let $\mathcal{L} = (M, Sem, sem)$ and $\mathcal{P} = (\mathcal{L}, O, P)$ be given as above. Let \mathcal{R} be a \mathcal{P} -repair-representative function and let $F : M \rightarrow \wp_{fin}(O)$ with $\forall m \in M : \mathcal{R}(m) \subseteq F(m)$.

As \mathcal{R} is a \mathcal{P} -repair-representative function, by Proposition 7.8, it holds that $Del_I \neq O \Rightarrow \mathcal{R}(m) \cap (O \setminus Del_I) \neq \emptyset$ for every instance $I = (\mathcal{P}, m)$ of \mathcal{P} . As $\mathcal{R}(m) \subseteq F(m)$ for all $m \in M$, it holds that $Del_I \neq O \Rightarrow F(m) \cap (O \setminus Del_I) \neq \emptyset$ for every instance $I = (\mathcal{P}, m)$ of \mathcal{P} . Using Proposition 7.8, we obtain that F is a \mathcal{P} -repair-representative function. \square

Thus, there is no general construction for a repair-representative function for the intersection of two model repair problems from the repair-representative functions for the two problems where the constructed function maps each model to a subset of the union of the sets of change operations related to the model by the repair-representative functions for the two problems. If there were such a general construction, then the function constructed in the proof of Proposition 7.30 would be a repair-representative function of the model repair problem constructed in the proof of Proposition 7.30.

7.8 Related Work and Discussion

Syntactic differencing approaches (e.g. [AP03, KKT11, KKT13, KGE09, KGFE08, TELW14, TBK09]) are not concerned with the semantic changes of models caused by the application of syntactic changes. However, syntactic differencing approaches provide a fundamental basis for frameworks facilitating to detect the syntactic changes causing the non-satisfaction of properties in the form of change operations.

Semantic differencing (e.g. [MRR11a]) reveals the semantic differences of models but is not concerned with syntactic differences. If the semantic difference from one model to another model is not empty, semantic differencing approaches usually provide diff witnesses. Each diff witness is an element in the semantics of one of the models that is not an element in the semantics of the other model. However, semantic differencing neither reveals the syntactic differences that cause the witnesses nor reveals the syntactic differences causing that one of the models is not a refinement of the other model. In general, it is not a trivial task to manually detect all syntactic differences causing the existence of a concrete witness or causing that the semantic difference from one model to another model is not empty.

Diffuse [MR15, MR18] is a language-independent framework that combines semantic with syntactic differencing. This thesis and the Diffuse framework share the same fundamental definition of modeling language. However, the representation of syntactic differences is different in Diffuse and the framework of this thesis. In this thesis, syntactic differences are represented by sequences of change operations. In the Diffuse

framework, syntactic differences are represented by partially ordered sets of change operations [MR18]. Nevertheless, the framework of this thesis and Diffuse can be combined with little effort. Every sequentialization of a set of change operations is a change sequence. Vice versa, every change sequence is interpretable as a totally ordered set of change operations.

Diffuse introduces the notions of necessary, exhibiting, and sufficient sets of change operations [MR18]. Each of the three notions relates two models, a diff witness, and a concrete syntactic difference from one of the models to the other model to a subset of the syntactic difference. Combining our work with Diffuse may be interesting. On the one hand, it might be possible to lift the three notions to abstract from a concrete syntactic difference. For instance, it might be interesting to compute whether there exists a generally necessary change operation that must be applied to cause the existence of a concrete witness. On the other hand, Diffuse is tailored towards determining syntactic changes causing a concrete property. The concrete property is the containment of a diff witness in the semantics of a model. Generalizing Diffuse to abstract from the property of interest is an interesting future work direction.

Diffuse enables determining which change operations of a concrete syntactic difference should not be applied to prevent the existence of a specific witness. However, this analysis does not reveal how to obtain a refinement. The analyses enabled by Diffuse are backward-oriented. In contrast, our approach is forward-oriented and computes what needs to be done to obtain a model that satisfies a concrete property (such as, for example, refinement).

The model repair framework presented in this thesis is a generalization of our previous work [KR18a]. In [KR18a], we presented equivalences for change operations in the context of repairing failed model refinements. The framework [KR18a] is tailored towards model refinement properties. In this thesis, we generalized the framework for repairing models towards satisfying arbitrary properties. This thesis further presents algorithms with performance improvements, whereas the previous work suggests [KR18a] to use a simple breadth-first search. This thesis further examines the compositionality of model repair problems, which is not tackled in [KR18a].

Non-enumerative semantic differencing approaches [FLW11] compute aggregated descriptions that summarize semantic differences (not necessarily all) from one model to another model. Non-enumerative approaches have been applied to feature models and automata [FLW11] as well as to class diagrams [FALW14]. As an aggregated description may encode more information than a single witness, it is more suited to manually detect the syntactic elements causing the models' semantic differences than a single witness. The aggregated description describing the semantic difference facilitates the manual detection of syntactic changes required to repair refinement. However, also with existing non-enumerative approaches, the computation of syntactic changes that repair refinement is not automated. Our previous work [KR18a] and the framework presented in this thesis aims at automation. Further, non-enumerative semantic differencing ap-

proaches are tailored towards semantic differencing. Hence, in general, they cannot be used to detect syntactic elements causing the non-satisfaction of arbitrary properties. The framework presented in this thesis focuses on generality by abstracting from a concrete property. It might be interesting future work to generalize non-enumerative semantic differencing towards abstracting from semantic differences to consider arbitrary properties. It could be interesting to develop a framework that facilitates computing aggregated descriptions causing the non-satisfaction of arbitrary but fixed properties.

Automatic software repair aims at automatically repairing software bugs [Rin08, Har10, Mon18, WCW⁺18, GMM19]. It focuses on fixing bugs in GPL source code. The bugs are encoded by a specification encoding expected behaviors, for instance, given by a formal specification (*e.g.*, [Wei06]) or test cases (*e.g.*, [WNLGF09, LGDVFW12, WCW⁺18]). The goal of automatic software repair is the automatic computation of a patch (syntactic changes) for a program to eliminate bugs. Automatic software repair frameworks can be obviously interpreted as model repair problems. The models are all possible GPL programs. The properties are encoded by the formal specifications or the test cases. Computing patches for automatically repairing programs and computing solutions for model repair problems are both search activities [Har10] that can be categorized into the field of search-based software engineering [Har07, Har10]. Existing approaches for automatic program repair are usually not tailored towards completeness: The approaches may not find a change to the program that repairs the bugs, although there exists a program that does not contain the bugs [WNLGF09, WCW⁺18]. For instance, some approaches reduce the infinite search space by only considering changes that are based on already existing structures in the code (*e.g.*, [WNLGF09]) or by bounding the depth of the search space (*e.g.*, [WCW⁺18]). In contrast, the approach of this thesis targets at completeness and is based on partitioning the set of change operations applicable to a model into equivalence classes to reduce the search space. If the set of equivalence classes is finite, then automatic computation of repairing changes is possible by only considering finitely many representatives per model. Applying the change operation partitioning to methods for automatic software repair is interesting future work for decreasing the search space to achieve further performance improvements.

The model repair framework presented in this thesis can be interpreted to generalize classical planning problems [GNT16]. A classical planning problem consists of a planning domain, an initial state, and a goal. A planning domain consists of a set of states, a finite set of actions, and a state transition function. The state transition function is a partial function that maps a state and an action to a state. It describes the effect of applying actions in a state in terms of the state reached when applying a given action in a given state. The goal encodes a set of states. A solution for a planning problem is a sequence of actions that transforms the initial state to a state that satisfies the goal. When transferring the notions to the framework of this thesis, the set of all models is the analog to the set of states. The set of change operations is the analog to the set of actions. Properties are the analogs to goals. While the set of actions is required to be finite in

classical planning problems, the framework of this thesis does not assume that the set of actions is finite. Computability of shortest solutions is achieved via partitioning the set of change operations applicable to models into equivalence classes. The equivalence classes may be different for different models. The search space may consist of infinitely many change operations, whereas only finitely many change operations are considered for each model.

Chapter 8

Concrete Instantiations of the Model Repair Framework

This chapter presents instantiations of the model repair framework presented in Chapter 7 with the modeling languages presented in Part II and refinement, generalization, and refactoring properties.

Developers can use shortest change sequences for repairing refinements to detect the syntactic modeling elements that are responsible for the existence of semantic differences from one model to another model. If the former model is intended to be a refinement of the latter model, then the automatically computed changes can be applied to the former model to definitely obtain a refinement of the other model.

Solving a generalization model repair problem is especially useful when comparing models that represent underspecified specifications. Then, the computed change sequences facilitate developers in understanding why one specification does not imply the other specification and vice versa.

Computing solutions for refactoring model repair problems facilitates the repair of failed model refactorings. Developers often change models to increase the models' internal representation with respect to the understandability of the model in the context of its development project. Refactoring steps can fail. Shortest solutions for refactoring model repair problems can support developers in detecting the syntactic model elements causing that the successor model version is no refactoring of its predecessor version. The changes can also be applied for changing the model resulting from a failed refactoring step to a refactoring of the original model.

Section 8.1 introduces the refines, generalizes, and refactors model repair problems. Afterwards, Section 8.2, Section 8.3, Section 8.4, and Section 8.5 instantiate the automatic model repair framework with the TSPA, FD, SD, and AD modeling languages.

8.1 Refines, Generalizes, and Refactors Model Repair Problems

This section defines abbreviations for model properties and model repair problems using the properties refinement, generalization, and refactoring. This reduces notational overhead in the remainder of this chapter.

Definition 8.1. Let $\mathcal{L} = (M, Sem, sem)$ be a modeling language, O be a complete change operation suite for \mathcal{L} , and $m \in M$ be a consistent model.

- The *refines* property for the modeling language \mathcal{L} and the model m is defined as $P_{\subseteq}(\mathcal{L}, m) \stackrel{\text{def}}{=} \{n \in M \mid \emptyset \neq sem(n) \subseteq sem(m)\}$.
- The *generalizes* property for the modeling language \mathcal{L} and the model m is defined as $P_{\supseteq}(\mathcal{L}, m) \stackrel{\text{def}}{=} \{n \in M \mid sem(m) \subseteq sem(n)\}$.
- The *refactors* property for the modeling language \mathcal{L} and the model m is defined as $P_{=}(\mathcal{L}, m) \stackrel{\text{def}}{=} \{n \in M \mid sem(n) = sem(m)\}$.
- The *refines problem* for the modeling language \mathcal{L} , the change operation suite O , and the model m is defined as $\mathcal{P}_{\subseteq}(\mathcal{L}, O, m) \stackrel{\text{def}}{=} (\mathcal{L}, O, P_{\subseteq}(\mathcal{L}, m))$.
- The *generalizes problem* for the modeling language \mathcal{L} , the change operation suite O , and the model m is defined as $\mathcal{P}_{\supseteq}(\mathcal{L}, O, m) \stackrel{\text{def}}{=} (\mathcal{L}, O, P_{\supseteq}(\mathcal{L}, m))$.
- The *refactors problem* for the modeling language \mathcal{L} , the change operation suite O , and the model m is defined as $\mathcal{P}_{=}(\mathcal{L}, O, m) \stackrel{\text{def}}{=} (\mathcal{L}, O, P_{=}(\mathcal{L}, m))$.

The models satisfying a *refines* property are required to be consistent. This reflects the intention that inconsistent models without realizations are not useful. The consistency requirement is not necessary for *generalizes* and *refactors* model repair problems. If a model is consistent, then all models that are generalizations or refactorings of the model are also consistent.

Proposition 8.1. Let $\mathcal{L} = (M, Sem, sem)$ be a modeling language, O be a complete change operation suite for \mathcal{L} , and $m \in M$ be a consistent model. Then, $\mathcal{P}_{\subseteq}(\mathcal{L}, O, m)$, $\mathcal{P}_{\supseteq}(\mathcal{L}, O, m)$, and $\mathcal{P}_{=}(\mathcal{L}, O, m)$ are model repair problems.

Proof. Let \mathcal{L} , O , and m be given as above. As the model m is required to be consistent, it directly follows by definition that $m \in P_{\subseteq}(\mathcal{L}, m)$, $m \in P_{\supseteq}(\mathcal{L}, m)$, and $m \in P_{=}(\mathcal{L}, m)$. \square

If a model is not a refinement of another model and a change operation is *refactoring*, then the application of the change operation to the former model yields a model that is not a refinement of the other model, either. Generalizing change operations do not preserve the complement properties of *refines* properties because they may change inconsistent models to consistent refinements of other models. Refining change operations preserve the complement properties of *generalizes* properties. If a model is not a refactoring of another model, then any refactoring of the former model is also not a refactoring of the latter model.

Proposition 8.2. Let $\mathcal{L} = (M, Sem, sem)$ be a modeling language, O be a change operation suite for \mathcal{L} , $m \in M$ be a consistent model, and $o \in O$ be a change operation. The following statements hold:

1. If o is refactoring, then o is $\overline{P_{\subseteq}(\mathcal{L}, m)}$ -preserving.
2. If o is generalizing and $\forall m \in M : sem(m) \neq \emptyset$, then o is $\overline{P_{\subseteq}(\mathcal{L}, m)}$ -preserving.
3. If o is refining, then o is $\overline{P_{\supseteq}(\mathcal{L}, m)}$ -preserving.
4. If o is refactoring, then o is $\overline{P_{=}(\mathcal{L}, m)}$ -preserving.

Proof. Let \mathcal{L} , O , m , and o be given as above. Let $n \in M$ be a model.

Proof of 1) Assume o is refactoring, $n \in \overline{P_{\subseteq}(\mathcal{L}, m)}$, and $n \in dom(o)$. Then, $sem(n) = \emptyset$ or there exists $w \in sem(n)$ such that $w \notin sem(m)$. If $sem(n) = \emptyset$, then it holds that $\emptyset = sem(n) = sem(o(n))$ as o is refactoring. Therefore, $o(n) \in \overline{P_{\subseteq}(\mathcal{L}, m)}$. If there exists $w \in sem(n)$ such that $w \notin sem(m)$, then it holds that $sem(n) = sem(o(n))$ as o is refactoring. Thus, $w \in sem(o(n))$. Therefore, $o(n) \in \overline{P_{\subseteq}(\mathcal{L}, m)}$.

Proof of 2) Assume o is generalizing, $\forall m \in M : sem(m) \neq \emptyset$ holds, $n \in \overline{P_{\subseteq}(\mathcal{L}, m)}$, and $n \in dom(o)$. Then, $sem(n) = \emptyset$ or there exists $w \in sem(n)$ such that $w \notin sem(m)$. As $\forall m \in M : sem(m) \neq \emptyset$ holds, it especially holds that $sem(n) \neq \emptyset$. Thus, there must exist $w \in sem(n)$ such that $w \notin sem(m)$. As o is generalizing, it holds that $sem(n) \subseteq sem(o(n))$ and, thus, $w \in sem(o(n))$. Therefore, $o(n) \in \overline{P_{\subseteq}(\mathcal{L}, m)}$.

Proof of 3) Assume o is refining, $n \in \overline{P_{\supseteq}(\mathcal{L}, m)}$, and $n \in dom(o)$. Then, there exists $w \in sem(m)$ such that $w \notin sem(n)$. As o is refining, it holds that $sem(o(n)) \subseteq sem(n)$ and, thus, $w \notin sem(o(n))$. Therefore, $o(n) \in \overline{P_{\supseteq}(\mathcal{L}, m)}$.

Proof of 4) Assume o is refactoring, $n \in \overline{P_{=}(\mathcal{L}, m)}$, and $n \in dom(o)$. Then, $sem(o(n)) = sem(n) \neq sem(m)$, which implies that $o(n) \in \overline{P_{=}(\mathcal{L}, m)}$. \square

As discussed in Section 7.5, checking whether a change operation implies the complement of a model repair problem's property can be incorporated into algorithms for computing shortest solutions for instances of the model repair problem.

To avoid notational overhead in the remainder of this chapter, we assume that the models used in model repair problems are consistent without explicitly stating it.

8.2 Instantiations with the Time-Synchronous Port Automaton Language

This section instantiates the model repair framework with the refines, generalizes, and refactors properties and the TSPA modeling language $\mathcal{L}_{PA} = (M_{PA}, Sem_{PA}, [\cdot]^{PA})$ presented in Chapter 3.

Infinitely many state-addition, input-channel-addition, and output-channel-addition operations are applicable to each TSPA. In contrast, every TSPA only contains finitely many states and finitely many channels. The types of all channels are finite. Therefore, only finitely many state-deletion, transition-addition, transition-deletion, channel-deletion, and initial-state-change operations are applicable to each TSPA.

Therefore, it suffices to partition the state-addition and channel-addition operations that are applicable to a TSPA with respect to the property of a concrete model repair problem. The partitioning is similar for all instances of the three model repair problems for the TSPA language.

Section 8.2.1 presents a change operation partitioning for TSPA refinement repair problem instances. Section 8.2.2 presents a change operation partitioning for TSPA generalization repair problem instances. Afterwards, Section 8.2.3 presents a change operation partitioning for TSPA refactoring repair problem instances. Finally, Section 8.2.4 presents a repair-representative function for the TSPA model repair problems.

8.2.1 Time-Synchronous Port Automaton Refinement Repair

This section presents a change operation partitioning for the TSPA modeling language and the TSPA refinement repair problem.

Let A and A' be two TSPAs. Then, for all states x, y that are not used in A , the operations $addS_x$ and $addS_y$ induce an equally long shortest solution for the model repair problem instance $I = (\mathcal{P}_{\subseteq}(\mathcal{L}_{PA}, O_{PA}, A'), A)$. State names do not influence the communication histories of a TSPA. For every shortest change sequence t that repairs $addS_x(A)$, it is possible to construct a change sequence u of the same length $|u| = |t|$ such that u repairs $addS_y(A)$. The TSPAs $addS_x(A) \triangleright t$ and $addS_y(A) \triangleright u$ solely differ in the naming of their states. The TSPA $addS_x(A) \triangleright t$ can be obtained from the TSPA $addS_y(A) \triangleright u$ by renaming the state x (if it exists) to y and renaming the state y (if it exists) to x in $addS_y(A) \triangleright u$. Vice versa, the TSPA $addS_y(A) \triangleright u$ can be obtained from the TSPA $addS_x(A) \triangleright t$ by renaming the state x (if it exists) to y and renaming the state y (if it exists) to x in $addS_x(A) \triangleright t$. As the naming of states does not influence the communication histories in the semantics of TSPAs, the TSPAs $addS_x(A) \triangleright t$ and $addS_y(A) \triangleright u$ have equal semantics. Thus, if the TSPA $addS_x(A) \triangleright t$ is a consistent refinement of the TSPA A' , the TSPA $addS_y(A) \triangleright u$ is also a consistent refinement of A' .

Proposition 8.3. *Let $A = (I, O, S, \iota, \delta)$ and $A' = (I', O', S', \iota', \delta')$ be two TSPAs. Further, let $I = (\mathcal{P}, A)$ be an instance of the model repair problem $\mathcal{P} = \mathcal{P}_{\subseteq}(\mathcal{L}_{PA}, O_{PA}, A')$. For all state names $x, y \in U_N \setminus S$, it holds that $addS_x \sim_I addS_y$.*

Proof. (Sketch.) Let A, A', \mathcal{P} , and I be given as above. Let $x, y \in U_N \setminus S$ be two state names. Let $t \in O_{PA}^*$ be a shortest change sequence that repairs the TSPA $addS_x(A)$ towards satisfying $\mathcal{P}_{\subseteq}(\mathcal{L}_{PA}, A')$. We define the change sequence $u \in O_{PA}^*$ of length $|u| = |t|$ as the change sequence obtained from the sequence t via replacing each change operation in t affecting the state x with a change operation of the same type affecting y , instead, and vice versa. A change operation adding a transition from the state x to the state y with label a , for instance, is replaced by a change operation adding a transition from the state y to the state x with label a .

By construction of the change sequence u , the TSPA $addS_x(A) \triangleright t$ can be obtained from the TSPA $addS_y(A) \triangleright u$ via renaming the state x (if it exists) to y and renaming the state y (if it exists) to x in $addS_y(A) \triangleright u$. Vice versa, the TSPA $addS_y(A) \triangleright u$ can be obtained from the TSPA $addS_x(A) \triangleright t$ via renaming the state x (if it exists) to y and renaming the state y (if it exists) to x in $addS_x(A) \triangleright t$.

As the TSPAs $addS_x(A) \triangleright t$ and $addS_y(A) \triangleright u$ solely differ in the naming of their states and state names do not affect the communication histories in the semantics of a TSPA, the two TSPAs are semantically equivalent. Therefore, as $addS_x(A) \triangleright t$ is by assumption a consistent refinement of A' , the TSPA $addS_y(A) \triangleright u$ is also a consistent refinement of A' . From the above, we can conclude $d_{\mathcal{P}}(addS_x(A)) \geq d_{\mathcal{P}}(addS_y(A))$.

Analogously, we can show that for every shortest change sequence $t \in O_{PA}^*$ that repairs $addS_y(A)$ towards satisfying $P_{\subseteq}(\mathcal{L}_{PA}, A')$, there exists a change sequence $u \in O_{PA}^*$ with $|u| = |t|$ that repairs $addS_x(A)$ towards satisfying $P_{\subseteq}(\mathcal{L}_{PA}, A')$. From this, we can conclude $d_{\mathcal{P}}(addS_x(A)) \leq d_{\mathcal{P}}(addS_y(A))$.

From $d_{\mathcal{P}}(addS_x(A)) \geq d_{\mathcal{P}}(addS_y(A))$ and $d_{\mathcal{P}}(addS_x(A)) \leq d_{\mathcal{P}}(addS_y(A))$, we can conclude that $d_{\mathcal{P}}(addS_x(A)) = d_{\mathcal{P}}(addS_y(A))$. Thus, $addS_x$ and $addS_y$ induce an equally long shortest solution for I . \square

Let A and A' be two TSPAs. Then, for each channel c that is not used in A' , the operation $addIC_c$ delays the solution for $I = (P_{\subseteq}(\mathcal{L}_{PA}, O_{PA}, A'), A)$. A necessary condition for a TSPA to refine another TSPA is that the former does not use channels that are not used by the latter. If the former TSPA used a channel not used by the latter TSPA, then no communication history of the former TSPA would be a communication history of the latter TSPA. If t is a change sequence that repairs A and starts with a change operation that adds the input channel c , then the channel c added by $t.0$ needs to be removed by a channel-deletion operation contained in t because otherwise $A \triangleright t$ would be no refinement of A' . A shorter change sequence that repairs A can be obtained from the change sequence t by deleting the channel-addition and channel-deletion operations, before modifying the change operations in between the two operations such that they do not affect the channel c anymore.

Proposition 8.4. *Let $A = (I, O, S, \iota, \delta)$ and $A' = (I', O', S', \iota', \delta')$ be two TSPAs. Further, let $I = (P, A)$ be an instance of the model repair problem $\mathcal{P} = P_{\subseteq}(\mathcal{L}_{PA}, O_{PA}, A')$. For every channel $c \in \mathcal{C} \setminus (I' \cup O')$, it holds that $addIC_c$ delays the solution for I .*

Proof. (Sketch.) Let A , A' , \mathcal{P} , and I be given as above. Let $c \in \mathcal{C} \setminus (I' \cup O')$ be a channel name. Let $t \in O_{PA}^*$ be a change sequence that repairs $addIC_c(A)$ towards satisfying $P_{\subseteq}(\mathcal{L}_{PA}, A')$.

Then, the change sequence t must contain the channel deletion operation $delC_c$: Suppose towards a contradiction that t does not contain the change operation $delC_c$. Then, the TSPA $addIC_c(A) \triangleright t$ contains the channel c . Therefore, every communication history

of $\text{addIC}_c(A) \triangleright t$ is no communication history of A' because A' does not contain the channel c . This contradicts the assumption that $\text{addIC}_c(A) \triangleright t$ is a refinement of A' .

Let $i \in \mathbb{N}$ with $0 \leq i < |t|$ be the smallest index such that $t.i = \text{delC}_c$. Let $u \in O_{PA}^*$ with $|u| = |t| - 1$ be the change sequence obtained from the change sequence t by deleting the channel deletion operation at position i and removing the mappings from the channel c in the channel valuations that are used by the first i change operations in t . The change sequence $\text{addT}_{s,t,\{c:\varepsilon,d:\varepsilon\}}, \text{addS}_x, \text{delC}_c, \text{addS}_y, \text{addT}_{x,y,\{d:\varepsilon\}}, \text{remT}_{s,t,\{d:\varepsilon\}}$, for instance, is changed to $\text{addT}_{s,t,\{d:\varepsilon\}}, \text{addS}_x, \text{addS}_y, \text{addT}_{x,y,\{d:\varepsilon\}}, \text{remT}_{s,t,\{d:\varepsilon\}}$.

By construction of u , it holds that $\text{addIC}_c(A) \triangleright t = A \triangleright u$ and $|u| < |t|$. As t repairs addIC_c towards satisfying $P_{\subseteq}(\mathcal{L}_{PA}, A')$, this implies that u repairs A towards satisfying $P_{\subseteq}(\mathcal{L}_{PA}, A')$. Therefore, for every change sequence $t \in O_{PA}^*$ that repairs $\text{addIC}_c(A)$ towards satisfying $P_{\subseteq}(\mathcal{L}_{PA}, A')$, there exists a change $u \in O_{PA}^*$ with $|u| < |t|$ such that u repairs A towards satisfying $P_{\subseteq}(\mathcal{L}_{PA}, A')$. Thus, addIC_c delays the solution for I . \square

Let A and A' be two TSPAs. Then, for each channel c that is not used in A' , the operation addOC_c delays the solution for $I = (P_{\subseteq}(\mathcal{L}_{PA}, O_{PA}, A'), A)$. The reason is similar to the reason for input-channel-addition operations adding a channel that is not used in A' .

Proposition 8.5. *Let $A = (I, O, S, \iota, \delta)$ and $A' = (I', O', S', \iota', \delta')$ be two TSPAs. Further, let $I = (P, A)$ be an instance of the model repair problem $\mathcal{P} = P_{\subseteq}(\mathcal{L}_{PA}, O_{PA}, A')$. For every channel $c \in \mathcal{C} \setminus (I' \cup O')$, it holds that addOC_c delays the solution for I .*

Proof. The proof is analogous to the proof for Proposition 8.4. \square

8.2.2 Time-Synchronous Port Automaton Generalization Repair

This section presents a change operation partitioning for the TSPA modeling language and the TSPA generalization repair problem.

Let A and A' be two TSPAs. Then, for all states x, y that are not used in A , the operations addS_x and addS_y induce an equally long shortest solution for the model repair problem instance $I = (P_{\supseteq}(\mathcal{L}_{PA}, O_{PA}, A'), A)$. The reason for this is similar to the reason for the equivalence in the context of TSPA refinement model repair problems (cf. Section 8.2.1). The names of states do not directly influence the communication histories of TSPAs.

Proposition 8.6. *Let $A = (I, O, S, \iota, \delta)$ and $A' = (I', O', S', \iota', \delta')$ be two TSPAs. Further, let $I = (P, A)$ be an instance of the model repair problem $\mathcal{P} = P_{\supseteq}(\mathcal{L}_{PA}, O_{PA}, A')$. Then, for all state names $x, y \in U_N \setminus S$, it holds that $\text{addS}_x \sim_I \text{addS}_y$.*

Proof. The proof is analogous to the proof of Proposition 8.3. \square

Let A and A' be two TSPAs. Then, for each channel c that is not used in A' , the operation addIC_c delays the solution for $I = (\mathcal{P}_{\supseteq}(\mathcal{L}_{PA}, O_{PA}, A'), A)$. The reason is similar to the reason in the context of TSPA refinement model repair problems (cf. Section 8.2.1). A necessary condition for a TSPA to generalize another TSPA is that the former does not use channels that are not used by the latter. If the former TSPA used a channel not used by the latter, then no communication history of the latter TSPA would be a communication history of the former TSPA. From every change sequence t that repairs A and starts with a change operation that adds the input channel c , it is possible to construct a shorter change sequence that repairs A .

Proposition 8.7. *Let $A = (I, O, S, \iota, \delta)$ and $A' = (I', O', S', \iota', \delta')$ be two TSPAs. Further, let $I = (\mathcal{P}, A)$ be an instance of the model repair problem $\mathcal{P} = \mathcal{P}_{\supseteq}(\mathcal{L}_{PA}, O_{PA}, A')$. For every channel $c \in \mathcal{C} \setminus (I' \cup O')$, it holds that addIC_c delays the solution for I .*

Proof. The proof is analogous to the proof for Proposition 8.4. \square

Let A and A' be two TSPAs. Then, for each channel c that is not used in A' , the operation addOC_c delays the solution for $I = (\mathcal{P}_{\supseteq}(\mathcal{L}_{PA}, O_{PA}, A'), A)$. The reason is similar to the reason for input-channel-addition operations adding a channel that is not used in A' .

Proposition 8.8. *Let $A = (I, O, S, \iota, \delta)$ and $A' = (I', O', S', \iota', \delta')$ be two TSPAs. Further, let $I = (\mathcal{P}, A)$ be an instance of the model repair problem $\mathcal{P} = \mathcal{P}_{\supseteq}(\mathcal{L}_{PA}, O_{PA}, A')$. For every channel $c \in \mathcal{C} \setminus (I' \cup O')$, it holds that addOC_c delays the solution for I .*

Proof. The proof is analogous to the proof for Proposition 8.4. \square

8.2.3 Time-Synchronous Port Automaton Refactoring Repair

This section presents a change operation partitioning for the TSPA modeling language and the TSPA refactoring repair problem.

Let A and A' be two TSPAs. Then, for all states x, y that are not used in A , the operations addS_x and addS_y induce an equally long shortest solution for the model repair problem instance $I = (\mathcal{P}_{=}(\mathcal{L}_{PA}, O_{PA}, A'), A)$. The reason is similar to the reasons in the contexts of TSPA refinement repair problems (cf. Section 8.2.1) and TSPA generalization repair problems (cf. Section 8.2.2).

Proposition 8.9. *Let $A = (I, O, S, \iota, \delta)$ and $A' = (I', O', S', \iota', \delta')$ be two TSPAs. Further, let $I = (\mathcal{P}, A)$ be an instance of the model repair problem $\mathcal{P} = \mathcal{P}_{=}(\mathcal{L}_{PA}, O_{PA}, A')$. For all state names $x, y \in U_N \setminus S$, it holds that $\text{addS}_x \sim_I \text{addS}_y$.*

Proof. The proof is analogous to the proof for Proposition 8.3. \square

Let A and A' be two TSPAs. Then, for each channel c that is not used in A' , the operation addIC_c delays the solution for $I = (\mathcal{P}_=(\mathcal{L}_{PA}, O_{PA}, A'), A)$. The reason is similar to the reasons in the contexts of TSPA refinement model repair problems (cf. Section 8.2.1) and TSPA generalization model repair problems (cf. Section 8.2.2).

Proposition 8.10. *Let $A = (I, O, S, \iota, \delta)$ and $A' = (I', O', S', \iota', \delta')$ be two TSPAs. Further, let $I = (\mathcal{P}, A)$ be an instance of the model repair problem $\mathcal{P} = \mathcal{P}_=(\mathcal{L}_{PA}, O_{PA}, A')$. For every channel $c \in \mathcal{C} \setminus (I' \cup O')$, it holds that addIC_c delays the solution for I .*

Proof. The proof is analogous to the proof for Proposition 8.4. \square

Let A and A' be two TSPAs. Then, for each channel c that is not used in A' , the operation addOC_c delays the solution for $I = (\mathcal{P}_=(\mathcal{L}_{PA}, O_{PA}, A'), A)$. The reason is similar to the reasons in the contexts of TSPA refinement repair problems (cf. Section 8.2.1) and TSPA generalization repair problems (cf. Section 8.2.2).

Proposition 8.11. *Let $A = (I, O, S, \iota, \delta)$ and $A' = (I', O', S', \iota', \delta')$ be two TSPAs. Further, let $I = (\mathcal{P}, A)$ be an instance of the model repair problem $\mathcal{P} = \mathcal{P}_=(\mathcal{L}_{PA}, O_{PA}, A')$. For every channel $c \in \mathcal{C} \setminus (I' \cup O')$, it holds that addOC_c delays the solution for I .*

Proof. The proof is analogous to the proof for Proposition 8.4. \square

8.2.4 Repair-Representative Function and Example Applications

From the argumentations in Section 8.2.1, Section 8.2.2, and Section 8.2.3, we can conclude that it is possible to use the same repair-representative function for the TSPA refinement, generalization, and refactoring repair problems using the same models. For each TSPA $A \in M_{PA}$, we can construct the repair-representative function $\mathcal{R}_A : M_{PA} \rightarrow \wp_{fin}(O_{PA})$ for each of the TSPA model repair problems as follows: The function \mathcal{R}_A maps each TSPA to the set containing

- all state-deletion, transition-addition, transition-deletion, channel-deletion, and initial-state-change operations that are applicable to the TSPA,
- all state-addition, input-channel-addition, and output-channel-addition operations that have not been partitioned in Section 8.2.1, Section 8.2.2, and Section 8.2.3 and are applicable to the TSPA,
- exactly one arbitrary but fixed operation of each equivalence class of the state-addition, input-channel-addition, and output-channel-addition operations identified in Section 8.2.1, Section 8.2.2, and Section 8.2.3.

For example, Figure 8.1 depicts an excerpt of the change sequence search tree for computing a shortest solution for a TSPA generalization repair problem instance. The TSPA `aut1` is the model of the instance. The property of the generalization repair problem contains all TSPAs that are generalizations of the TSPA `aut2`.

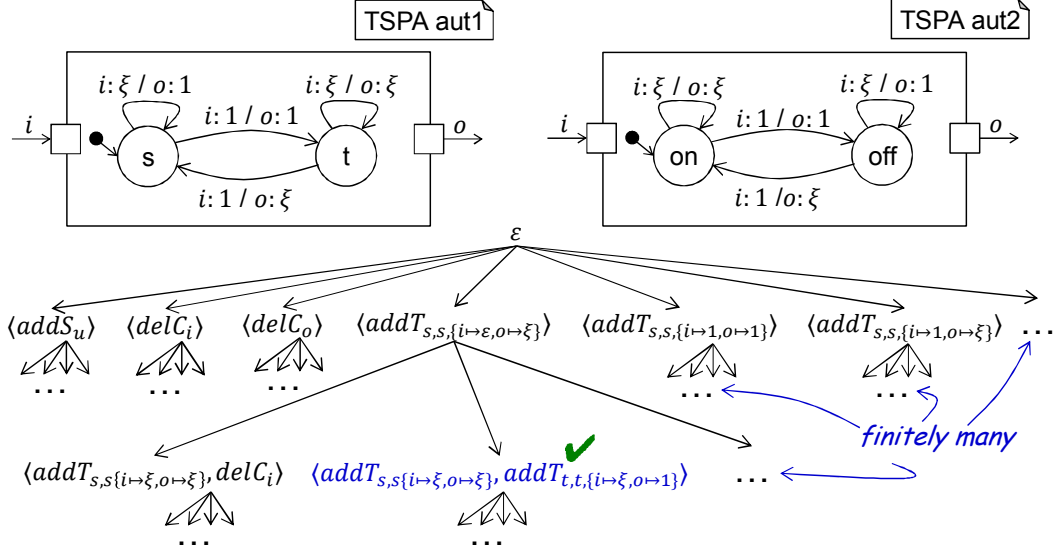


Figure 8.1: Two simple TSPAs and an excerpt of the change sequence search tree for computing a shortest solution for a TSPA generalization repair problem.

8.2.5 Implementation and Experiments

We implemented the model repair algorithms for the TSPA modeling language to perform experimental evaluations. The implementation is written in Java and uses the semantic differencing operator implementation presented in Section 3.3.

We performed experimental evaluations by executing the algorithms for computing shortest solutions for TSPA refinement repair problem instances using the seven example TSPAs presented in Appendix A. The purpose of the experiments is twofold. The first purpose is testing whether computing shortest repairing sequences is feasible for the example TSPA refinement repair problem instances. The second purpose is to compare the performances of the different algorithms in the context of the example TSPA refinement repair problems.

Heuristic for Witness Computation

The computational complexity of semantic differencing of TSPAs is high and the algorithms often execute the semantic differencing operator. To achieve performance improvements, we implemented a heuristic for fast witness computation.

The heuristic checks whether a TSPA A does not refine another TSPA B by searching a short diff witness. To this effect, the heuristic checks whether there exists a prefix of a behavior of A that is not a prefix of any behavior of B . The lengths of the checked

prefixes is bounded by the number of states in A . If the heuristic finds such a prefix, it proceeds by finding the prefix of an execution that produces the behavior with the prefix. After determining the prefix of the execution, the algorithm proceeds by searching a path that starts in the last state of the computed prefix of the execution and contains a cycle in the TSPA A . Composing the computed prefix of the behavior with the labels of the transitions on the path containing a cycle yields an ultimately periodic word that is a behavior in the semantic difference from A to B . If the heuristic successfully finds a witness, then A is not a refinement of B . However, if the heuristic does not find a diff witness, it is not guaranteed that A is not a refinement of B . In this case, we use the sound and complete semantic differencing operator for TSPAs as introduced in Section 3.3.

The function `SemDiffHeuristically` defined in Algorithm 7 is the heuristic for computing short diff witnesses. The function takes two TSPAs A and B as inputs. It either outputs a diff witness $w \in \delta(A, B)$ or the special symbol *nil* if no prefix of a diff witness of length smaller than or equal to the number of states in A exists. The function enumerates all paths of transitions starting in the initial state of A with a length smaller than or equal to the number of states in A . To this effect, the function executes an DFID search. For each path, it checks whether there exists a behavior in B having the prefix of the behavior encoded by the path as prefix. In the following algorithms, $src(t)$ denotes the source state of a transition t , $trg(t)$ denotes the target state of a transition t , and $lt(w)$ denotes the last element of a finite sequence w .

The function iterates over all integers i that are smaller than or equal to the number of states in A (ll. 2-24). In each iteration, it checks the execution paths of length equal to i . In the loop, the function initializes the variable *can* as an empty stack of sequences of transitions of A (l. 3). The stack *can* contains the execution path candidates that are to be processed. Then, the function pushes the sequences of length one containing the transitions originating from the initial state on the stack *can* (ll. 4-6). While the candidate stack is not empty (ll. 7-23), the function iteratively pops a candidate *cur* from the stack *can* (l. 8), checks the prefix of the behavior encoded by the candidate (ll. 9-17), and explores new candidates (ll. 19-21).

If the length of the current candidate *cur* is equal to the length i of the currently checked candidates (l. 9), the function checks the currently processed transition sequence (ll. 10-17). Therefore, it constructs the behavior prefix encoded by the currently processed transition sequence and stores the result in the variable *behPrefix* (ll. 10-13). Afterwards, the function checks whether the sequence stored in variable *behPrefix* is a prefix of a behavior of B by checking whether the set of states reachable by B via processing the sequence is empty (l. 14). For computing the set of reachable states, it calls the function `PossibleStatesAfterInput` (l. 14). If the set is empty, then there does not exist a behavior of B that has the sequence as a prefix. In this case, the function calls the function `BehaviorStartingIn` (l. 15) to compute a behavior starting in the target state of the last transition contained in the currently processed transition

Algorithm 7 Heuristic for computing a diff witness contained in the semantic difference from a TSPA A to a TSPA B .

Input: Two TSPAs $A = (\Sigma, S, \iota, \delta)$ and B .

Output: A behavior $w \in \delta(A, B)$ or the special symbol nil .

```

1: function SEMDIFFHEURISTICALLY( $A, B$ )
2:   for all  $i \in \mathbb{N}$  with  $i < |S|$  in increasing value do
3:     define  $can$  as empty stack of  $\delta^*$ 
4:     for all  $t \in \{(s, a, t) \in \delta \mid s = \iota\}$  do
5:        $can.push(t)$   $\triangleright$  push sequence of length one containing transition  $t$ 
6:     end for
7:     while  $can$  not empty do
8:        $cur \leftarrow can.pop()$ 
9:       if  $|cur| = i$  then
10:        define  $behPrefix \leftarrow \varepsilon$  as empty sequence
11:        for all  $(s, a, t) \in cur$  in ascending order do
12:           $behPrefix \leftarrow behPrefix \& a$ 
13:        end for
14:        if  $POSSIBLESTATESAFTERINPUT(B, behPrefix) = \emptyset$  then
15:           $(np, p) = BEHAVIORSTARTINGIN(A, trg(lt(cur)))$ 
16:          return  $behPrefix \& np \& p^\infty$ 
17:        end if
18:      else
19:        for all  $t \in \{(s, a, t) \in \delta \mid s = trg(lt(cur))\}$  do
20:           $can.push(cur \& t)$ 
21:        end for
22:      end if
23:    end while
24:  end for
25:  return  $nil$ 
26: end function

```

sequence. This function returns a tuple (np, p) of channel assignment sequences where np models a non-period part of a behavior and p models the periodic part of the behavior. The behavior $np \& p^\infty$ is a behavior of the TSPA obtained from the TSPA A by changing the initial state to the target state of the last transition contained in the currently processed sequence. Therefore, the sequence $behPrefix \& np \& p^\infty$ is a behavior of A that is not a behavior of B . The function returns this sequence (l. 16).

In case the length of the currently processed sequence is not equal to the currently checked length i , the algorithm computes new candidates that are added to the candidate stack (ll. 19-21). Therefore, the function iterates over all transitions starting in the target

Algorithm 8 Computing the set of states reachable by a TSPA after processing a finite prefix of a behavior.

Input: A TSPA $A = (\Sigma, S, \iota, \delta)$ and a finite sequence of channel assignments b .

Output: The set of states reachable by A after processing b .

```

1: function POSSIBLESTATESAFTERINPUT( $A, b$ )
2:   define  $curStates \leftarrow \{\iota\}$  as set of  $S$ 
3:   for all  $v \in b$  in ascending order do
4:     define  $newCurStates \leftarrow \emptyset$  as set of  $S$ 
5:     for all  $(s, a, t) \in \delta$  do
6:       if  $s \in curStates \wedge a = v$  then
7:          $newCurStates \leftarrow newCurStates \cup \{t\}$ 
8:       end if
9:     end for
10:     $curStates \leftarrow newCurStates$ 
11:  end for
12:  return  $curStates$ 
13: end function

```

state of the last transition contained the currently processed transition sequence (l. 19). For each of these transition, the algorithm pushes the result from concatenating the transition to the currently processed transition sequence on the stack (l. 20). If no diff witness is found, the function returns the special symbol *nil* (l. 25).

The function `PossibleStatesAfterInput` is depicted in Algorithm 8. The function takes a TSPA A and a finite sequence of channel assignments b as inputs. It outputs the set of states reachable by A after processing b . To this effect, the function simulates all runs of the TSPA A that are possible when processing the sequence of channel assignments b . The function initializes the variable $curStates$ as the singleton set containing the initial state of the TSPA (l. 2). Afterwards, it iterates over the channel assignments in b in ascending order (ll. 3-11) and iteratively computes the sets of possible states reachable by processing the channel assignments in b up to the currently processed channel assignment v (ll. 4-10). To this effect, it initializes the variable $newCurStates$ as the empty set (l. 4). Then, the function iterates over all possible transitions of the TSPA (ll. 5-9). The function adds the target state of each transition that starts in one of the states contained in the set $curStates$ and is labeled with the currently processed channel assignment v (l. 6) to the set $newCurStates$ (l. 7). After iterating through all transitions, the set $newCurStates$ contains all states reachable by the TSPA after processing the prefix of the sequence b up to and including the currently processed channel assignment v . Therefore, after the loop, the algorithm sets the variable $curStates$ to the value of the variable $newCurStates$ (l. 10) and proceeds with the following channel assignment contained in b . After processing the complete sequence, the function returns

Algorithm 9 Computing the non-periodic and the periodic part of a behavior of a TSPA starting in a given state.

Input: A TSPA $A = (\Sigma, S, \iota, \delta)$ and a state $i \in S$.

Output: A tuple (np, p) of finite sequences of channel assignments representing the behavior $np \& p^\infty$ of the TSPA A when starting in state i .

```

1: function BEHAVIORSTARTINGIN( $A, i$ )
2:   define  $visited \leftarrow \{i\}$  as set of  $S$ 
3:   define  $toProcess$  as FIFO queue of  $\delta^*$ 
4:   for all  $t \in \{(s, a, t) \in \delta \mid s = i\}$  do
5:      $toProcess.add(t)$ 
6:   end for
7:   while  $toProcess$  not empty do
8:     define  $cur \leftarrow toProcess.poll()$  as element of  $\delta^*$ 
9:     define  $(s, a, t) \leftarrow lt(cur)$  as element of  $\delta$ 
10:    for all  $i \in \mathbb{N}$  with  $i < |cur|$  in increasing value do
11:      define  $(u, b, v) \leftarrow cur.i$ 
12:      if  $u = t$  then
13:        define  $np \leftarrow \varepsilon$  as sequence of channel assignments
14:        define  $p \leftarrow \varepsilon$  as sequence of channel assignments
15:        for all  $j \in \mathbb{N}$  with  $j < |cur|$  in increasing value do
16:          define  $(w, c, x) \leftarrow cur.j$  as element of  $\delta$ 
17:          if  $j < i$  then
18:             $np \leftarrow np \& c$ 
19:          else
20:             $p \leftarrow p \& c$ 
21:          end if
22:        end for
23:        return  $(np, p)$ 
24:      end if
25:    end for
26:    for all  $t \in \{(u, b, c) \in \delta \mid u = t\}$  do
27:       $toProcess.add(cur \& t)$ 
28:    end for
29:  end while
30: end function

```

the set $curStates$ (l. 12).

The function `BehaviorStartingIn` is depicted in Algorithm 9. It takes a TSPA A and a state i of A as inputs. The function outputs a tuple (np, p) of finite sequences of channel assignments such that $np \& p^\infty$ is a behavior of A when starting in the state

i. The function executes a breadth-first search in the TSPA A starting in the state i to compute a path in A that contains a cycle. First, it initializes the set *visited* as the singleton set containing the state i (l. 2). This set stores the states visited during the breadth-first search. Then, the function initializes the variable *toProcess* as an empty first-in-first-out queue of sequences of transitions of the TSPA A (l. 3). The queue stores the sequences of transitions encoding explored paths that need to be expanded during the breadth-first search. All sequences of length one containing a transition starting in the state i are initially added to the queue (ll. 4-6). While the queue *toProcess* is not empty (l. 7), the function executes the breadth-first search (ll. 8-29). In the loop, the function polls the first element of the queue *toProcess* and stores the result in the variable *cur* (l. 8). The transition sequence stored in the variable *cur* is expanded in the current iteration of the loop. To this effect, the function retrieves the last transition (s, a, t) of the sequence (l. 9). Then, it iterates over all integers that are smaller than the length of the sequence *cur* (ll. 10-25). For each of these integers i , the function retrieves the i -th transition (u, b, v) of the sequence *cur* (l. 11) and checks whether $u = t$ (l. 12), *i.e.*, it checks whether the target state of the last transition is equal to the starting state of the i -th transition. If this is the case, the function found a cycle and partitions the discovered transitions in the non-periodic part defining the sequence np and the periodic part defining the sequence p (l. 13-23). Otherwise, the function iterates over all transitions starting in the target state t of the last transition (ll. 26-28) and adds the result from appending the transition to the currently processed sequence *cur* to the queue (l. 27). For determining the non-periodic and the periodic parts (ll. 13-23), the function first initializes the variables np and p as empty sequences of channel assignments (ll. 13-14). Then, the function iterates over all integers j that are smaller than the length of the currently processed sequence *cur* (ll. 15-22) and retrieves the j -th transition of the sequence *cur* (l. 16). If the integer j is smaller than the integer i (l. 17), the j -th transition is not part of the cycle and the algorithm appends the label of the transition to the non-periodic part (l. 18). Otherwise (l. 19), the transition is part of the cycle and the function appends the label of the transition to the periodic part (l. 20). After iterating over all transitions, the function returns the tuple (np, p) (l. 23).

As every TSPA is consistent, the implementation of a consistency checking procedure for the TSPA modeling language is not necessary.

Proposition 8.12. *Every TSPA is consistent.*

Proof. Let $A = (\Sigma, S, \iota, \delta)$ where $\Sigma = (I, O)$ be a TSPA. For each state $s \in S$ let $(u^s, a^s, v^s) \in \delta$ be an arbitrary but fixed transition with $u^s = s$. The transitions exist because A is by definition reactive. We recursively define the execution $e = s_0, \theta_0, s_1, \theta_1, \dots$ of A by $s_0 = \iota$, $s_i = v^{s_{i-1}}$ for all $i \in \mathbb{N}$ with $i > 0$, and $\theta_i = a^{s_i}$ for all $i \in \mathbb{N}$. Then, by construction, e is an execution of A . Therefore, $\llbracket A \rrbracket^{PA} \neq \emptyset$ holds. \square

Including the heuristic for diff witness computation yields Algorithm 10 for checking

Algorithm 10 Checking whether the TSPA A is a consistent refinement of the TSPA B including a heuristic for diff witness computation.

Input: Two TSPAs A and B .

Output: \checkmark , if A is a refinement of B . Otherwise, $w \in \delta(A, B)$.

```

1:  $w \leftarrow \text{SEMDIFFHEURISTICALLY}(A, B)$ 
2: if  $w \neq \text{nil}$  then
3:   return  $w$ 
4: end if
5: return  $\text{SEMTPADIFF}(A, B)$ 

```

whether a TSPA A is a consistent refinement of a TSPA B . First, the algorithm tries to heuristically compute a diff witness in the semantic difference from A to B by calling the function `SemDiffHeuristically` (l. 1). If the function found a witness (l. 2), then the algorithm returns the witness (l. 3). Otherwise, the algorithm returns the result from calling the function `SemTSPADiff` (l. 5). The function is required to return \checkmark , if A refines B . Otherwise, the function is required to return a diff witness contained in the semantic difference from A to B . For the implementation of the function `SemTSPADiff`, we use the semantic differencing operator for TSPAs presented in Section 3.3. Each diff witness w returned by Algorithm 10 is easily translatable to the property $P = \{m \in M_{PA} \mid w \in \llbracket m \rrbracket^{PA}\}$ containing all models containing this witness in their semantics. This property implies the complement of the consistently refines property.

Search Space Restrictions

For pragmatic reasons, to increase the performance of the algorithms, we restrict the search spaces of the TSPA refinement repair problems as described in the following.

We do not consider transition-addition operations adding transitions with a label that is not used on any transition of the repair problem's model. This restriction is reasonable because any behavior caused by any execution visiting a transition with a label not used in the repair problem's model cannot be a behavior of the repair problem's model.

For each currently processed model, we focus on operations for modifying the parts of the model producing the computed witness for the semantic difference from the model to the repair problem's model. As the witness is contained in the semantics of the processed model, there exists an execution of the model that causes the witness. Thus, at least one transition visited during the execution must be deleted from the model to eliminate the existence of the witness in the semantics of the model. However, it is not necessarily possible to directly delete one of the transitions as the deletion could cause that the resulting TSPA is not reactive. Therefore, adding a transition before deleting the transition is sometimes necessary to preserve the reactivity of intermediately computed TSPAs. If the deletion is not directly possible, we consider transition-addition

Model	Problem	Time				
		Algo. 2	Algo. 3	Algo. 4	Algo 5	Algo. 6
mod4Ctr	threeCtr	154ms	141ms	141ms	119ms	123ms
threeCtr	mod4Ctr	682ms	626ms	459ms	432ms	481ms
reset	threeCtr	TO	TO	TO	OOM	TO
reset	mod4Ctr	TO	TO	TO	OOM	TO
impl	spec	1458ms	289ms	288ms	1152ms	410ms
spec	impl	TO	TO	TO	OOM	TO
aut1	aut2	566ms	528ms	124ms	161ms	145ms
aut2	aut1	445ms	365ms	250ms	230ms	252ms

Figure 8.2: The execution times of Algorithm 2 - Algorithm 6 when given the TSPA refinement repair problem instances as inputs.

operations adding a transition from the source state of the execution's transition such that the transition's input channel assignment equals the input channel assignment of the transition of the execution. Otherwise, if the transition-deletion operation is already applicable, we do not consider any transition-addition operations adding transitions originating from the transition's source state. For each processed model, among all the applicable transition-deletion and transition-addition operations, we do only consider the operations described above.

Experiments

We performed experimental evaluations with the seven example TSPAs presented in Appendix A. For each pair of thematically related TSPAs where one of the TSPAs is not a refinement of the other TSPA, we executed the algorithms presented in Section 7.5 to compute solutions for the corresponding refinement repair problems. For testing whether intermediately computed TSPAs satisfy the refines property, we used Algorithm 10 for refinement checking.

All experiments were executed on a laptop computer with an Intel Core i7-8650U CPU @ 1.90GHz processor, 16GB RAM, and a Samsung PM981 512GB SSD hard drive using Windows 10 and Java 1.8.0_192.

Figure 3.4 summarizes the computation times of the algorithms. We have set a time-out of 15 minutes for each computation. Timeouts are indicated by TO. During three experiments, Java reported an `OutOfMemoryError`, which is indicated by OOM. For instance, Algorithm 2 took 154ms to compute a solution for the model repair problem instance $(\mathcal{P}_{\subseteq}(\mathcal{L}_{PA}, O_{PA}, \text{threeCtr}), \text{mod4Ctr})$, *i.e.*, to compute a sequence $t \in O_{PA}$ such that $\emptyset \neq \llbracket \text{mod4Ctr} \triangleright t \rrbracket^{PA} \subseteq \llbracket \text{threeCtr} \rrbracket^{PA}$. In the cases where the algorithms terminated, the computation times range from 119ms to 1458ms. Figure 8.3 depicts the

Model	Problem	Solution
mod4Ctr	threeCtr	$addT_{3,inc3,3}, delT_{3,inc3,0}$
threeCtr	mod4Ctr	$addT_{3,inc3,0}, delT_{3,inc3,3}$
impl	spec	$addT_{stopped,stp,idle}, delT_{stopped,emgOff2,driving}$
aut1	aut2	$addT_{s,\{i:\xi,o:\xi\},s}, delT_{s,\{i:\xi,o:1\},s},$ $addT_{t,\{i:\xi,o:1\},t}, delT_{t,\{i:\xi,o:\xi\},t},$
aut2	aut1	$addT_{on,\{i:\xi,o:1\},on}, delT_{on,\{i:\xi,o:\xi\},on},$ $addT_{off,\{i:\xi,o:\xi\},off}, delT_{off,\{i:\xi,o:1\},off}$

Figure 8.3: The shortest solutions computed by Algorithm 4 for the TSPA refinement repair problem instances.

solutions computed by Algorithm 4 for the inputs where the computations terminated.

Algorithm 3 was faster than Algorithm 2 and Algorithm 4 was at least as fast as Algorithm 3 for all computations that did not time out. In three cases, Algorithm 5 was faster than Algorithm 4. In the other two cases, Algorithm 4 was faster than Algorithm 5. This might be because the computed solutions are short and storing already visited models as well as checking whether a currently processed model was already visited causes additional overhead. Algorithm 4 was faster than Algorithm 6 in four cases. This might be because the computed solutions are short and checking the prefixes of currently processed change sequences causes additional overhead in Algorithm 6.

We conclude that the algorithms handle the small examples, where only short repairing sequences are required, sufficiently quick. However, the algorithms do not scale well for inputs where long repairing sequences are required. This is not surprising as the running times of the algorithms are exponential in the lengths of the shortest solutions (cf. Section 7.5.1). For all computations where the algorithms timed out, more than 20 changes are required to repair the models.

8.3 Instantiations with the Feature Diagram Language

This section instantiates the model repair framework with the FD modeling language $\mathcal{L}_{FD} = (M_{FD}, Sem_{FD}, \llbracket \cdot \rrbracket^{FD})$ presented in Chapter 4 and the refines, generalizes, and refactors model repair problems.

Infinitely many feature-addition and root-renaming operations are applicable to each FD. In contrast, every FD only contains finitely many features. For this reason, only finitely many feature-deletion, implies-constraint-addition, implies-constraint-deletion, excludes-constraint-addition, excludes-constraint-deletion, or-group-creation, xor-to-or-conversion, or-to-xor-conversion, mandatory-to-optional, optional-to-mandatory, feature-group-insertion, and feature-group-exclusion operations are applicable to each FD. The computation of the change operations described above that are applicable to an FD is a

straight-forward task.

Therefore, it suffices to partition the feature-addition and root-renaming operations that are applicable to an FD with respect their properties in the context of a model repair problem. The partitioning is similar for all instances of the three model repair problems for the FD language.

Section 8.3.1 presents a change operation partitioning for FD refinement repair problem instances. Then, Section 8.3.2 presents a change operation partitioning for FD generalization repair problem instances. Afterwards, Section 8.3.3 presents a change operation partitioning for FD refactoring repair problem instances. Finally, Section 8.3.4 presents a repair-representative function for the FD model repair problems.

8.3.1 Feature Diagram Refinement Repair

This section presents a change operation partitioning for the FD modeling language and the FD refinement repair problem.

Let fd_1 and fd_2 be two FDs. Then, for all features f, g that are neither used in fd_1 nor in fd_2 , the operations $rnmRoot_f$ and $rnmRoot_g$ induce an equally long shortest solution for the model repair problem instance $I = (\mathcal{P}_{\subseteq}(\mathcal{L}_{FD}, O_{FD}, fd_2), fd_1)$. Intuitively, for every shortest change sequence t that repairs $rnmRoot_f(fd_1)$, it is possible to construct a change sequence u of the same length $|u| = |t|$ that repairs $rnmRoot_g(fd_1)$. The FD $rnmRoot_f(fd_1) \triangleright t$ constrains the feature f in the same way as the FD $rnmRoot_g(fd_1) \triangleright u$ constrains the feature g . Vice versa, the FD $rnmRoot_f(fd_1) \triangleright t$ constrains the feature g in the same way as the FD $rnmRoot_g(fd_1) \triangleright u$ constrains the feature f . The valid configurations of $rnmRoot_g(fd_1) \triangleright u$ can be obtained from the valid configurations of $rnmRoot_f(fd_1) \triangleright t$ by exchanging the features f and g in the valid configurations of $rnmRoot_f(fd_1) \triangleright t$. As the features f and g are not used in the FD fd_2 , the FD fd_2 does not constrain the features f and g . Therefore, as $rnmRoot_f(fd_1) \triangleright t$ is by assumption a consistent refinement of fd_2 , the FD $rnmRoot_g(fd_1) \triangleright u$ is also a consistent refinement of fd_2 . A similar argumentation shows that for every shortest change sequence t that repairs $rnmRoot_g(fd_1)$, it is possible to construct a change sequence u of the same length $|u| = |t|$ that repairs $rnmRoot_f(fd_1)$.

Proposition 8.13. *Let $fd_i = (F_i, E_i, r_i, M_i, Or_i, Xor_i, I_i, X_i)$ for $i \in \{1, 2\}$ be two FDs. Let $I = (\mathcal{P}, fd_1)$ be an instance of the model repair problem $\mathcal{P} = \mathcal{P}_{\subseteq}(\mathcal{L}_{FD}, O_{FD}, fd_2)$. Then, for all features $f, g \in U_N \setminus (F_1 \cup F_2)$, it holds that $rnmRoot_f \sim_I rnmRoot_g$.*

Proof. (Sketch.) Let fd_i for $i \in \{1, 2\}$, \mathcal{P} , and I be given as above. Let $f, g \in U_N \setminus (F_1 \cup F_2)$ be two features that are neither used in fd_1 nor in fd_2 . As $f, g \notin F_1$, the operations $rnmRoot_f$ and $rnmRoot_g$ are applicable to fd_1 . Assume $t \in O_{FD}^*$ is a shortest change sequence that repairs $rnmRoot_f(fd_1)$ towards satisfying $P_{\subseteq}(\mathcal{L}_{FD}, fd_2)$. We define $u \in O_{FD}^*$ as the change sequence of length $|u| = |t|$ obtained from t by replacing each change operation in t affecting f by a change operation of the same type affecting

g and vice versa. An implies-constraint-addition operation adding an implies constraint from f to g , for instance, is replaced by an implies-constraint-addition operation adding an implies constraint from g to f , i.e., “ f implies g ” is replaced by “ g implies f ”.

By construction of the change sequence u , the FD $rnmRoot_f(fd_1) \triangleright t$ can be obtained from the FD $rnmRoot_g(fd_1) \triangleright u$ by renaming the feature f (if it exists) to g and renaming the feature g (if it exists) to f in $rnmRoot_g(fd_1) \triangleright u$. Vice versa, the FD $rnmRoot_g(fd_1) \triangleright u$ can be obtained from the FD $rnmRoot_f(fd_1) \triangleright t$ by renaming the feature f (if it exists) to g and renaming the feature g (if it exists) to f in $rnmRoot_f(fd_1) \triangleright t$.

The feature f is constrained in $rnmRoot_f(fd_1) \triangleright t$ in the same way the feature g is constrained in $rnmRoot_g(fd_1) \triangleright u$. Vice versa, the feature g is constrained in $rnmRoot_f(fd_1) \triangleright t$ in the same way the feature f is constrained in $rnmRoot_g(fd_1) \triangleright u$. The valid configurations of $rnmRoot_g(fd_1) \triangleright u$ can be obtained from the valid configurations of $rnmRoot_f(fd_1) \triangleright t$ via replacing the feature f by the feature g and replacing the feature g by the feature f in the valid configurations of $rnmRoot_f(fd_1) \triangleright t$. Analogously, it is possible to obtain the valid configurations of $rnmRoot_f(fd_1) \triangleright t$ from the valid configurations of $rnmRoot_g(fd_1) \triangleright u$. This especially implies that $rnmRoot_g(fd_1) \triangleright u$ is consistent because $rnmRoot_f(fd_1) \triangleright t$ is consistent.

Let c be a valid configuration of $rnmRoot_g(fd_1) \triangleright u$. Then, the above implies that the configuration c' obtained from c by replacing the feature f in c (if it exists in c) by the feature g and replacing the feature g in c (if it exists in c) by the feature f is a valid configuration of $rnmRoot_f(fd_1) \triangleright t$. As the FD $rnmRoot_f(fd_1) \triangleright t$ is by assumption a consistent refinement of fd_2 , the configuration c' is also a valid configuration of fd_2 . As the features f, g are not used by fd_2 , using Proposition 4.1, we obtain that the validity of c' in fd_2 implies that $c' \cup \{f, g\}$ is valid in fd_2 . As $c \subseteq c' \cup \{f, g\}$ and f, g are not used in fd_2 , this again implies with Proposition 4.1 that c is valid in fd_2 . Therefore, $rnmRoot_g(fd_1) \triangleright u$ is a consistent refinement of fd_2 . We can conclude that $d_{\mathcal{P}}(rnmRoot_f(fd_1)) \geq d_{\mathcal{P}}(rnmRoot_g(fd_1))$.

Analogously, we can show that for every shortest change sequence t that repairs $rnmRoot_g(fd_1)$ towards satisfying $P_{\subseteq}(\mathcal{L}_{FD}, fd_2)$, there exists a change sequence u with $|u| = |t|$ such that u repairs $rnmRoot_f(fd_1)$ towards satisfying $P_{\subseteq}(\mathcal{L}_{FD}, fd_2)$. Therefore, $d_{\mathcal{P}}(rnmRoot_f(fd_1)) \leq d_{\mathcal{P}}(rnmRoot_g(fd_1))$.

As it holds that $d_{\mathcal{P}}(rnmRoot_f(fd_1)) \geq d_{\mathcal{P}}(rnmRoot_g(fd_1))$ and it further holds that $d_{\mathcal{P}}(rnmRoot_f(fd_1)) \leq d_{\mathcal{P}}(rnmRoot_g(fd_1))$, we can conclude that $d_{\mathcal{P}}(rnmRoot_f(fd_1)) = d_{\mathcal{P}}(rnmRoot_g(fd_1))$. Thus, $rnmRoot_f$ and $rnmRoot_g$ induce an equally long shortest solution for I . \square

Similarly, feature-addition operations that add features not used in the FDs induce equally long shortest solutions for FD refinement model repair problem instances. Let fd_1 and fd_2 be two FDs and let p be an arbitrary feature used in fd_1 . Then, for all features f, g that are neither used in fd_1 nor in fd_2 , the operations $addF_{p,f}$ and $addF_{p,g}$ induce an equally long shortest solution for the model repair problem instance

$I = (\mathcal{P}_{\subseteq}(\mathcal{L}_{FD}, O_{FD}, fd_2), fd_1)$. The reason for this is similar to the reason for the equivalence of root renaming operations: For every shortest change sequence t that repairs $addF_{p,f}(fd_1)$, it is possible to construct a change sequence u of the same length $|u| = |t|$ that repairs $addF_{p,g}(fd_1)$. The FD $addF_{p,f}(fd_1) \triangleright t$ constrains the feature f in the same way as the FD $addF_{p,g}(fd_1) \triangleright u$ constrains the feature g . Vice versa, the FD $addF_{p,f}(fd_1) \triangleright t$ constrains the feature g in the same way as the FD $addF_{p,g}(fd_1)(fd_1) \triangleright u$ constrains the feature f . The valid configurations of $addF_{p,g}(fd_1) \triangleright u$ can be obtained from the valid configurations of $addF_{p,f}(fd_1) \triangleright t$ by exchanging the features f and g in the valid configurations of $addF_{p,f}(fd_1) \triangleright t$. As the features f and g are not used in the FD fd_2 , the FD fd_2 does not constrain the features f and g . Therefore, as $addF_{p,f}(fd_1) \triangleright t$ is by assumption a consistent refinement of fd_2 , the FD $addF_{p,g}(fd_1) \triangleright u$ is also a consistent refinement of fd_2 . A similar argumentation shows that for every shortest change sequence t that repairs $addF_{p,g}(fd_1)$, it is possible to construct a change sequence u of the same length $|u| = |t|$ that repairs $addF_{p,f}(fd_1)$.

Proposition 8.14. *Let $fd_i = (F_i, E_i, r_i, M_i, Or_i, Xor_i, I_i, X_i)$ for $i \in \{1, 2\}$ be two FDs and let $p \in F_1$ be a feature of fd_1 . Further, let $I = (\mathcal{P}, fd_1)$ be an instance of the model repair problem $\mathcal{P} = \mathcal{P}_{\subseteq}(\mathcal{L}_{FD}, O_{FD}, fd_2)$. Then, for all features $f, g \in U_N \setminus (F_1 \cup F_2)$, it holds that $addF_{p,f} \sim_I addF_{p,g}$.*

Proof. The proof is analogous to the proof for Proposition 8.13. □

8.3.2 Feature Diagram Generalization Repair

This section presents a change operation partitioning for the FD modeling language and the FD generalization repair problem.

Let fd_1 and fd_2 be two FDs. Then, for all features f, g that are neither used in fd_1 nor in fd_2 , the operations $rnmRoot_f$ and $rnmRoot_g$ induce an equally long solution for the model repair problem instance $I = (\mathcal{P}_{\subseteq}(\mathcal{L}_{FD}, O_{FD}, fd_2), fd_1)$. From each shortest change sequence t that repairs the FD $rnmRoot_f(fd_1)$, it is possible to construct a change sequence u of same length $|u| = |t|$ that repairs the FD $rnmRoot_g(fd_1)$. The change sequence u is constructed in the same way as in the context of the FD refinement repair problem (cf. Section 8.3.1). If there existed a configuration c that is valid in fd_2 and not valid in $rnmRoot_g(fd_1) \triangleright u$, then the configuration c' obtained from the configuration c by exchanging the feature f (if it exists in c) with the feature g and exchanging the feature g (if it exists in c) with the feature f would not be valid configuration of $rnmRoot_f(fd_1) \triangleright t$. However, c' would be a valid configuration of fd_2 because the configuration c is valid in fd_2 and fd_2 neither restricts the feature f nor the feature g . This would contradict the assumption that $rnmRoot_f(fd_1) \triangleright t$ is a generalization of fd_2 .

Proposition 8.15. *Let $fd_i = (F_i, E_i, r_i, M_i, Or_i, Xor_i, I_i, X_i)$ for $i \in \{1, 2\}$ be two FDs. Let $I = (\mathcal{P}, fd_1)$ be an instance of the model repair problem $\mathcal{P} = \mathcal{P}_{\subseteq}(\mathcal{L}_{FD}, O_{FD}, fd_2)$. Then, for all features $f, g \in U_N \setminus (F_1 \cup F_2)$, it holds that $rnmRoot_f \sim_I rnmRoot_g$.*

Proof. (Sketch.) Let fd_i for $i \in \{1, 2\}$, \mathcal{P} , and I be given as above. Let $f, g \in U_N \setminus (F_1 \cup F_2)$ be two features neither used in fd_1 nor in fd_2 . As $f, g \notin F_1$, the operations $rnmRoot_f$ and $rnmRoot_g$ are applicable to fd_1 . Assume $t \in O_{FD}^*$ is a shortest change sequence that repairs $rnmRoot_f(fd_1)$ towards satisfying $P_{\supseteq}(\mathcal{L}_{FD}, fd_2)$. We define $u \in O_{FD}^*$ as the change sequence of length $|u| = |t|$ obtained from t by replacing each change operation in t affecting f by a change operation of the same type affecting g and vice versa.

By construction of the change sequence u , the FD $rnmRoot_f(fd_1) \triangleright t$ can be obtained from the FD $rnmRoot_g(fd_1) \triangleright u$ by renaming the feature f (if it exists) to g and renaming the feature g (if it exists) to f in $rnmRoot_g(fd_1) \triangleright u$. Vice versa, the FD $rnmRoot_g(fd_1) \triangleright u$ can be obtained from the FD $rnmRoot_f(fd_1) \triangleright t$ by renaming the feature f (if it exists) to g and renaming the feature g (if it exists) to f in $rnmRoot_f(fd_1) \triangleright t$.

The feature f is constrained in $rnmRoot_f(fd_1) \triangleright t$ in the same way the feature g is constrained in $rnmRoot_g(fd_1) \triangleright u$. Vice versa, the feature g is constrained in $rnmRoot_f(fd_1) \triangleright t$ in the same way the feature f is constrained in $rnmRoot_g(fd_1) \triangleright u$. Therefore, the valid configurations of $rnmRoot_g(fd_1) \triangleright u$ can be obtained from the valid configurations of $rnmRoot_f(fd_1) \triangleright t$ via replacing the feature f by the feature g and replacing the feature g by the feature f in the valid configurations of $rnmRoot_f(fd_1) \triangleright t$. Analogously, it is possible to obtain the valid configurations of $rnmRoot_f(fd_1) \triangleright t$ from the valid configurations of $rnmRoot_g(fd_1) \triangleright u$.

Let $c \in \llbracket fd_2 \rrbracket^{FD}$ be a valid configuration of fd_2 . As by assumption $rnmRoot_f(fd_1) \triangleright t$ is a generalization of fd_2 , the configurations c is also valid in $rnmRoot_f(fd_1) \triangleright t$.

Suppose towards a contradiction that c is not valid in $rnmRoot_g(fd_1) \triangleright u$. Then, it follows from the above that the configuration c' obtained from c by replacing the feature f in c (if it exists in c) by the feature g and replacing the feature g in c (if it exists in c) by the feature f is not a valid configuration of $rnmRoot_f(fd_1) \triangleright t$. But as $f, g \notin F_2$, the features f and g are unconstrained in fd_2 and, therefore, Proposition 4.1 guarantees with $c \in \llbracket fd_2 \rrbracket^{FD}$ that $c' \in \llbracket fd_2 \rrbracket^{FD}$, which stands with $c' \notin \llbracket rnmRoot_f(fd_1) \triangleright t \rrbracket^{FD}$ in contradiction to the assumption that $rnmRoot_f(fd_1) \triangleright t$ is a generalization of fd_2 . We can conclude $d_{\mathcal{P}}(rnmRoot_f(fd_1)) \geq d_{\mathcal{P}}(rnmRoot_g(fd_1))$.

Analogously, we can show that for every shortest change sequence t that repairs $rnmRoot_g(fd_1)$ towards satisfying $P_{\supseteq}(\mathcal{L}_{FD}, fd_2)$, there exists a change sequence u with $|u| = |t|$ such that u repairs $rnmRoot_f(fd_1)$ towards satisfying $P_{\supseteq}(\mathcal{L}_{FD}, fd_2)$. From this, we can conclude $d_{\mathcal{P}}(rnmRoot_f(fd_1)) \leq d_{\mathcal{P}}(rnmRoot_g(fd_1))$.

As it holds that $d_{\mathcal{P}}(rnmRoot_f(fd_1)) \geq d_{\mathcal{P}}(rnmRoot_g(fd_1))$ and it further holds that $d_{\mathcal{P}}(rnmRoot_f(fd_1)) \leq d_{\mathcal{P}}(rnmRoot_g(fd_1))$, we can conclude that $d_{\mathcal{P}}(rnmRoot_f(fd_1)) = d_{\mathcal{P}}(rnmRoot_g(fd_1))$. Thus, $rnmRoot_f$ and $rnmRoot_g$ induce an equally long shortest solution for I . \square

Similarly, feature-addition operations that add features not used in the FDs induce equally long shortest solutions for FD generalization model repair problem instances.

Proposition 8.16. *Let $fd_i = (F_i, E_i, r_i, M_i, Or_i, Xor_i, I_i, X_i)$ for $i \in \{1, 2\}$ be two FDs and let $p \in F_1$ be a feature of fd_1 . Further, let $I = (\mathcal{P}, fd_1)$ be an instance of the model repair problem $\mathcal{P} = \mathcal{P}_{\supseteq}(\mathcal{L}_{FD}, O_{FD}, fd_2)$. Then, for all features $f, g \in U_N \setminus (F_1 \cup F_2)$, it holds that $addF_{p,f} \sim_I addF_{p,g}$.*

Proof. The proof is analogous to the proof for Proposition 8.15. \square

8.3.3 Feature Diagram Refactoring Repair

This section presents a change operation partitioning for the FD modeling language and the FD refactoring repair problem.

Root-renaming operations changing the root to a feature not used in the FDs induce equally long shortest solutions for FD refactoring model repair problem instances.

Proposition 8.17. *Let $fd_i = (F_i, E_i, r_i, M_i, Or_i, Xor_i, I_i, X_i)$ for $i \in \{1, 2\}$ be two FDs. Let $I = (\mathcal{P}, fd_1)$ be an instance of the model repair problem $\mathcal{P} = \mathcal{P}_{=}(\mathcal{L}_{FD}, O_{FD}, fd_2)$. Then, for all features $f, g \in U_N \setminus (F_1 \cup F_2)$, it holds that $rnmRoot_f \sim_I rnmRoot_g$.*

Proof. The proof is a combination of the proofs for Proposition 8.13 and Proposition 8.15. \square

Similarly, feature-addition operations adding features not used in the FDs induce equally long shortest solutions for FD refactoring model repair problem instances.

Proposition 8.18. *Let $fd_i = (F_i, E_i, r_i, M_i, Or_i, Xor_i, I_i, X_i)$ for $i \in \{1, 2\}$ be two FDs and let $p \in F_1$ be a feature of fd_1 . Further, let $I = (\mathcal{P}, fd_1)$ be an instance of the model repair problem $\mathcal{P} = \mathcal{P}_{=}(\mathcal{L}_{FD}, O_{FD}, fd_2)$. Then, for all features $f, g \in U_N \setminus (F_1 \cup F_2)$, it holds that $addF_{p,f} \sim_I addF_{p,g}$.*

Proof. The proof is analogous to the proof for Proposition 8.17. \square

8.3.4 Example Repair-Representative Function and Application

From the argumentations in Section 8.3.1, Section 8.3.2, and Section 8.3.3, we can conclude that it is possible to use the same repair-representative function for the FD refinement, generalization, and refactoring repair problems using the same models. For each $fd \in M_{FD}$, we can construct the repair-representative function $\mathcal{R}_{fd} : M_{FD} \rightarrow \wp_{fin}(O_{FD})$ for each of the FD model repair problems using the FD fd as follows: The function \mathcal{R}_{fd} maps each FD to the set containing

- all feature-deletion, implies-constraint-addition, implies-constraint-deletion, xor-to-or-conversion, or-to-xor conversion, excludes-constraint-addition, optional-to-mandatory, excludes-constraint-deletion, or-group-creation, mandatory-to-optional, feature-group-insertion, feature-group-exclusion operations that are applicable to the FD,

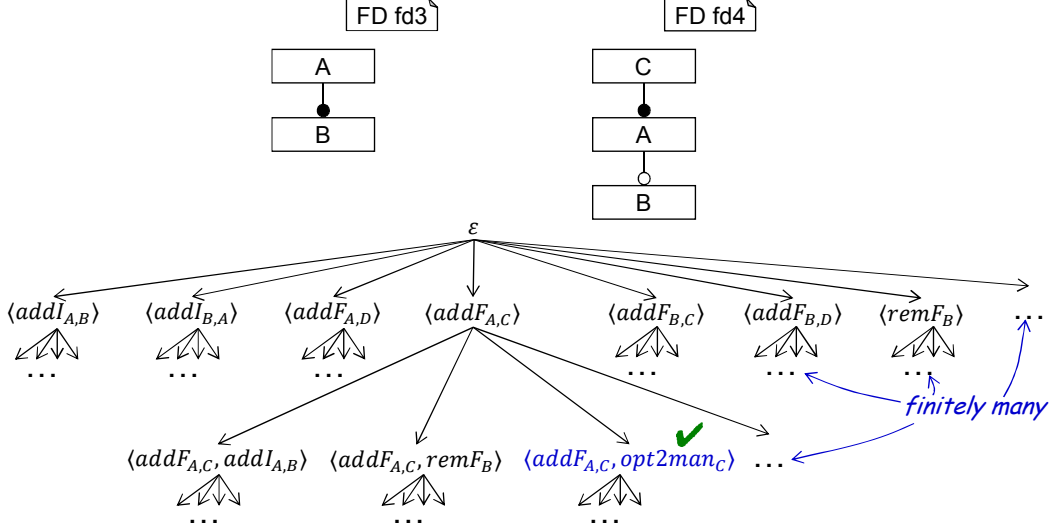


Figure 8.4: Two simple FDs and an excerpt of the change sequence search tree for computing a shortest solution for an FD refinement repair problem.

- all root-renaming and feature-addition, operations that are not partitioned in Section 8.3.1, Section 8.3.2, and Section 8.3.3 and are applicable to the FD, and
- exactly one arbitrary but fixed change operation of each equivalence class of the root-renaming and feature-addition operations identified in Section 8.3.1, Section 8.3.2, and Section 8.3.3.

For example, Figure 8.4 depicts an excerpt of the change sequence search tree for computing a shortest solution for an FD refinement repair problem instance. The FD `fd3` is the model of the instance. The property of the refinement repair problem contains all consistent FDs that refine the FD `fd4`.

8.3.5 Implementation and Experiments

We implemented the model repair algorithms for the FD modeling language to perform experimental evaluations. The implementation is written in Java and uses the semantic differencing operator implementation presented in Section 4.3.

We performed experimental evaluations by executing the algorithms for computing shortest solutions for FD refinement repair problem instances using the ten example FDs presented in Appendix B. The purpose of the experiments is twofold. The first purpose is testing whether computing shortest repairing sequences is feasible for the example FD refinement repair problem instances. The second purpose is comparing the

Algorithm 11 Heuristic for computing configurations in the semantics of an FD.

Input: An FD $fd = (F, E, r, M, Or, Xor, I, X)$.

Output: A set $C \subseteq \llbracket fd \rrbracket^{FD}$ of configurations in the semantics of fd .

```

1: function EXPLICITCONFIGS( $fd$ )
2:   define  $configs \leftarrow$  SUBTREECONFIGS( $r, fd$ ) as set of configurations
3:    $configs \leftarrow \{C \in configs \mid \forall (a, b) \in I : a \in C \Rightarrow b \in C\}$ 
4:    $configs \leftarrow \{C \in configs \mid \forall (a, b) \in X : a \in C \Rightarrow b \notin C\}$ 
5:   return  $configs$ 
6: end function

```

performances of the different algorithms in the context of the example FD refinement repair problems.

Heuristics for Consistency Checking and Witness Computation

The computational complexity of semantic differencing and consistency checking is high. Further, the algorithms often execute the semantic differencing operator and consistency checks. To achieve performance improvements, we implemented heuristics for fast witness computation and consistency checking.

For checking whether an FD fd is a refinement of an FD fd' , the heuristic first computes all configurations that are valid in fd and solely contain features that are used in fd . Then, the heuristic checks whether one of these configurations is not valid in fd' . If this is the case, then the configuration is a diff witness contained in the semantic difference from fd to fd' . However, even if all configurations in the set are also valid in fd' , the semantic difference from fd to fd' might be not empty. Thus, if no diff witness is found as described above, we use the semantic differencing operator as presented in Section 4.3 for executing a sound and complete refinement check. The computed set of configurations is also used for checking whether fd is consistent. By Proposition 4.2, an FD is consistent iff there exists a configuration that is valid in the FD and solely contains features that are used in the FD. Thus, the computed set is not empty iff the FD fd is consistent.

The function `ExplicitConfigs` defined in Algorithm 11 is the function for computing all configurations that are valid in an FD and solely contain features that are used in the FD. The function takes an FD as input. It outputs a finite subset of the semantics of the FD. Initially, it computes the set of all configurations that satisfy the child feature and group constraints of the FD and solely contain features that are used in the FD (l. 2). To this effect, the function calls the function `SubtreeConfigs`. The computed set of configurations is stored in the variable `configs`. The computed configurations do not necessarily satisfy the implies and excludes constraints of the FD. Therefore, the function proceeds with filtering all configurations not satisfying the implies and excludes

Algorithm 12 Computing the explicitly modeled configurations of a subtree of an FD while ignoring cross-tree constraints.

Input: An FD $fd = (F, E, r, M, Or, Xor, I, X)$ and a feature $p \in F$.

Output: Configurations of the subtree of fd starting at the feature p when ignoring cross-tree constraints.

```

1: function SUBTREECONFIGS( $p, fd$ )
2:   define  $configs \leftarrow \{\{p\}\}$  as set of configurations
3:   for all  $(a, b) \in \{(a, b) \in E \mid a = p \wedge \forall (c, G) \in Or \cup Xor : b \notin G\}$  do
4:     define  $bConfigs \leftarrow \text{SUBTREECONFIGS}(b, fd)$  as set of configurations
5:     if  $(a, b) \in M$  then
6:        $configs \leftarrow \{A \cup B \mid A \in configs \wedge B \in bConfigs\}$ 
7:     else
8:        $configs \leftarrow configs \cup \{A \cup B \mid A \in configs \wedge B \in bConfigs\}$ 
9:     end if
10:  end for
11:  for all  $(a, G) \in \{(a, G) \in Or \mid a = p\}$  do
12:    define  $oldConfigs \leftarrow configs$  as set of configurations
13:     $configs \leftarrow \emptyset$ 
14:    for all  $S \subseteq G$  with  $S \neq \emptyset$  do
15:      define  $subsetConfigs \leftarrow oldConfigs$  as set of configurations
16:      for all  $c \in S$  do
17:        define  $cConfigs \leftarrow \text{SUBTREECONFIGS}(c, fd)$ 
18:         $subsetConfigs \leftarrow \{A \cup B \mid A \in subsetConfigs \wedge B \in cConfigs\}$ 
19:      end for
20:       $configs \leftarrow configs \cup subsetConfigs$ 
21:    end for
22:  end for
23:  for all  $(a, G) \in \{(a, G) \in Xor \mid a = p\}$  do
24:    define  $oldConfigs \leftarrow configs$  as set of configurations
25:     $configs \leftarrow \emptyset$ 
26:    for all  $c \in G$  do
27:      define  $cConfigs \leftarrow \text{SUBTREECONFIGS}(c, fd)$ 
28:       $configs \leftarrow configs \cup \{A \cup B \mid A \in oldConfigs \wedge B \in cConfigs\}$ 
29:    end for
30:  end for
31:  return  $configs$ 
32: end function

```

constraints from the set stored in the variable $configs$ (ll. 3,4). Finally, the function returns the set of computed configurations (l. 5).

Algorithm 13 Checking whether an FD is inconsistent.

Input: An FD $fd = (F, E, r, M, Or, Xor, I, X)$.

Output: `true`, if fd contains an excludes constraint between features that obviously are core features of fd . Otherwise, `false`.

```

1: function COREINCONSISTENT( $fd$ )
2:   define  $core \leftarrow \text{COMPUTECORE}(fd)$  as set of features
3:   for all  $(a, b) \in X$  do
4:     if  $a \in core \wedge b \in core$  then
5:       return true
6:     end if
7:   end for
8:   return false
9: end function

```

The function `SubTreeConfigs` defined in Algorithm 12 takes a feature p and a feature diagram fd as inputs. It recursively computes the set of all configurations that satisfy the child feature and group constraints between the features in the subtree of the feature diagram fd starting at the feature p . The computed configurations solely contain features that are used in the subtree. Initially, the function initializes the variable $configs$ as a singleton set containing the configuration solely containing the feature p (l. 2). This represents that the feature p must be selected if the subtree starting at feature p is selected. Afterwards, the function iterates over all edges of the tree starting at p (ll. 3-10) that do not end at a feature participating in a group. Thus, the loop iterates over all edges targeting an optional or mandatory child feature of p . In the loop, the function computes the configurations obtained when recursively calling the function with the child feature and the FD as parameters and stores the result in the variable $bConfigs$ (l. 4). If the child feature is a mandatory child (l. 5), then the set of computed configurations stored in $configs$ is updated to the pairwise union of the sets of configurations contained in the sets $configs$ and $bConfigs$ (l. 6). Applying this operation represents that the child feature must be selected when its parent p is selected. Otherwise (ll. 7-9), the child feature is an optional child of its parent. In this case, the variable $configs$ is set to the union of the set stored in the variable and the pairwise union of the sets of configurations contained in the sets $configs$ and $bConfigs$ (l. 8). Applying this operation represents that the child feature can (but may not) be selected when its parent p is selected. Afterwards, the function iterates over all or-groups (ll. 11-22) and over all xor-groups (ll. 23-30) of the FD fd . In the bodies of the loops, the function updates the set of configurations stored in the variable $configs$ with respect to choosing different subsets of the features participating in the groups. The update processes are similar to the update process for the mandatory and optional child features. Finally, the function returns the value stored in the variable $configs$ (l. 31).

Algorithm 14 Computing features that are part of the core of an FD.

Input: An FD $fd = (F, E, r, M, Or, Xor, I, X)$.

Output: A set of features C that is a subset of the set of all core features of fd .

```

1: function COMPUTECORE( $fd$ )
2:   define  $core \leftarrow \{r\}$  as set of features
3:   define  $S$  as empty stack of features
4:    $S.push(r)$ 
5:   while  $S$  not empty do
6:     define  $cur \leftarrow S.pop()$  as feature
7:     for all  $(a, b) \in M$  do
8:       if  $a = cur \wedge b \notin core$  then
9:          $core \leftarrow core \cup \{b\}$ 
10:         $S.push(b)$ 
11:      end if
12:    end for
13:    for all  $(a, G) \in Or \cup Xor$  do
14:      if  $a = cur \wedge |G| = 1 \wedge G \not\subseteq core$  then
15:         $core \leftarrow core \cup G$ 
16:         $S.pushAll(G)$ 
17:      end if
18:    end for
19:    for all  $(a, b) \in I$  do
20:      if  $a = cur \wedge b \notin core$  then
21:        define  $p \leftarrow b$  as feature
22:        while  $p \neq r$  do
23:          if  $p \notin core$  then
24:             $core \leftarrow core \cup \{p\}$ 
25:             $S.push(p)$ 
26:          end if
27:           $p \leftarrow parent(p)$ 
28:        end while
29:      end if
30:    end for
31:  end while
32:  return  $core$ 
33: end function

```

The heuristic for checking whether an FD is consistent relies on computing features that are obviously core features [BSRC10] of the FD, before checking whether there exists an excludes constraint between two of the core features. A feature is a core feature of

Algorithm 15 Checking whether the FD fd is a consistent refinement of the FD fd' including heuristics for diff witness computation and checking consistency.

Input: Two FDs $fd = (F, E, r, M, Or, Xor, I, X)$ and fd' .

Output: inconsistent, if fd is inconsistent. Otherwise, \checkmark , if fd refines fd' and fd is consistent. Otherwise, $w \in \delta(fd, fd')$, if fd is consistent and $\delta(fd, fd') \neq \emptyset$.

```

1: if COREINCONSISTENT( $fd$ ) then
2:   return inconsistent
3: end if
4: define  $configs \leftarrow$  EXPLICITCONFIGS( $fd$ ) as set of configurations
5: if  $configs = \emptyset$  then
6:   return inconsistent
7: end if
8: for all  $C \in configs$  do
9:   if  $C \notin \llbracket fd' \rrbracket^{FD}$  then
10:    return  $C$ 
11:   end if
12: end for
13: return SEMFDDIFF( $fd, fd'$ )

```

an FD iff every valid configuration of FD contains the feature [BSRC10]. For example, the root feature of an FD is a core feature of the FD.

The function `CoreInconsistent` defined in Algorithm 13 computes features that are obviously core features of an FD, before checking whether an excludes constraint between the computed core features is not satisfied. To this effect, it initializes the variable *core* and sets its value to the result from calling the function `ComputeCore` (l. 2). The set *core* contains core features of the input FD. Afterwards, it iterates over all excludes constraints of the FD (ll. 3-7) and checks whether two of the core features exclude each other (l. 4). If this is the case, the function returns `true` (l. 5). Otherwise, the function returns `false` (l. 8).

The function `ComputeCore` is defined in Algorithm 14. It recursively computes the set of all features that must be chosen because of mandatory child feature constraints, group constraints induced by groups containing a single feature, and requires constraints. First, it initializes the variable *core* as the singleton set containing the root feature (l. 2). Afterwards, it initializes the variable *S* as an empty stack of features (l. 3), before it pushes the root feature on the stack (l. 4). In the following loop (ll. 5-31), the function computes core features of the FD and adds them to the set stored in the variable *core*. The stack *S* contains the features that are still to be processed for computing the core features. It contains core features that may have dependencies to other core features. While the stack is not empty (l. 5), the function pops the first element of the stack (l. 6) and stores the value in the variable *cur*. Afterwards, it adds all features that

8.3 INSTANTIATIONS WITH THE FEATURE DIAGRAM LANGUAGE

Model	Problem	Time				
		Algo. 2	Algo. 3	Algo. 4	Algo 5	Algo. 6
car	car1	218ms	231ms	157ms	207ms	346ms
car1	car	48s	48s	1808ms	6017ms	3667ms
car1	car2	46s	46s	1764ms	6292ms	3154ms
car2	car1	186ms	182ms	178ms	233ms	233ms
tablet1	tablet2	TO	TO	32s	1m 52s	50s
tablet1	tablet3	TO	TO	1m 45s	TO	2m 43s
tablet2	tablet1	212ms	235ms	201ms	440ms	298ms
tablet2	tablet3	200ms	195ms	192ms	348ms	243ms
tablet3	tablet1	243ms	274ms	186ms	368ms	275ms
tablet3	tablet2	177ms	194ms	172ms	317ms	220ms
fd1	fd2	8s	10s	362ms	1131ms	961ms
fd1	fd3	105ms	98ms	136ms	196ms	126ms
fd1	fd4	112ms	105ms	106ms	198ms	198ms
fd2	fd1	849ms	768ms	145ms	140ms	253ms
fd2	fd4	115ms	102ms	101ms	170ms	189ms
fd3	fd1	1542ms	1433ms	171ms	215ms	326ms
fd3	fd2	1m 7s	1m 1s	2169ms	2202ms	4002ms
fd3	fd4	102ms	94ms	110ms	138ms	175ms
fd4	fd2	15s	12s	1186ms	1326ms	2792ms
fd4	fd3	112ms	91ms	94ms	104ms	194ms

Figure 8.5: The execution times of Algorithm 2 - Algorithm 6 when given the FD refinement repair problem instances as inputs.

belong to the core because of mandatory child dependencies with parent *cur* (ll. 7-12), group dependencies with parent *cur* (ll. 13-18), and requires constraint dependencies of requires constraints starting at *cur* (ll.19-30) to the set *core*. After the execution of the outer loop (ll. 5-31), the variable *core* stores a set of features that are core features of *fd*. Finally, the function returns this set (l. 32).

Including the heuristics for diff witness computation and consistency checking yields Algorithm 15 for checking whether an FD *fd* is a consistent refinement of an FD *fd'*. Initially, the algorithm heuristically checks whether the features of the core exclude each other by calling the function *CoreInconsistent* (l. 1). If this is the case, the algorithm returns the special value *inconsistent* (l. 3). Otherwise, the algorithm computes the explicitly modeled configurations of the input FD by calling the function *ExplicitConfigs* and stores the result in the variable *configs* (l. 4). The set stored in the variable *configs* is empty iff the FD *fd* is inconsistent. In this case (l. 5), the algorithm returns the special value *inconsistent*. Otherwise, the algorithm iterates over the configurations of the set stored in the variable *configs* (ll. 8-12). For each configuration, the algorithm checks whether it is valid in *fd'*. If the configuration is not valid in *fd'*, then it is a diff witness contained in the semantic difference from *fd* to *fd'*.

Model	Problem	Solution
car	car1	$exclGrp_{\text{hybrid}}, addX_{\text{fingerprint}, \text{fingerprint}}$
car1	car	$addX_{\text{fingerprint}, \text{fingerprint}}, addX_{\text{gas}, \text{gas}}, addF_{\text{fingerprint}, \text{hybrid}}$
car1	car2	$addX_{\text{fingerprint}, \text{fingerprint}}, addX_{\text{gas}, \text{gas}}, addF_{\text{fingerprint}, \text{hybrid}}$
car2	car1	$exclGrp_{\text{hybrid}}, addX_{\text{fingerprint}, \text{fingerprint}}$
tablet1	tablet2	$addX_{\text{cellular}, \text{cellular}}, addF_{\text{cellular}, \text{dis12}}, addF_{\text{cellular}, 256\text{GB}}$
tablet1	tablet3	$addX_{\text{P200}, \text{P200}}, addF_{\text{P200}, \text{dis12}}, addF_{\text{P200}, 256\text{GB}}$
tablet2	tablet1	$addX_{\text{dis12}, \text{dis12}}$
tablet2	tablet3	$addX_{64\text{GB}, 64\text{GB}}$
tablet3	tablet1	$addI_{\text{processor}, 64\text{GB}}$
tablet3	tablet2	$addX_{256\text{GB}, 256\text{GB}}$
fd1	fd2	$addX_{\text{C}, \text{C}}, addF_{\text{A}, \text{E}}, opt2man_{\text{E}}$
fd1	fd3	$exclGrp_{\text{C}}$
fd1	fd4	$exclGrp_{\text{B}}$
fd2	fd1	$addF_{\text{C}, \text{D}}$
fd2	fd4	$man2opt_{\text{E}}, exclGrp_{\text{B}}$
fd3	fd1	$addF_{\text{B}, \text{D}}, addX_{\text{B}, \text{D}}$
fd3	fd2	$addF_{\text{B}, \text{E}}, opt2man_{\text{E}}, addF_{\text{B}, \text{C}}, addX_{\text{B}, \text{C}}$
fd3	fd4	$addF_{\text{B}, \text{C}}, opt2man_{\text{C}}$
fd4	fd2	$opt2man_{\text{B}}, rnmRoot_{\text{E}}, addF_{\text{E}, \text{C}}, addX_{\text{E}, \text{C}}$
fd4	fd3	$opt2man_{\text{B}}$

Figure 8.6: The shortest solutions computed by Algorithm 4 for the FD refinement repair problem instances.

In this case, the algorithm returns the configuration (l. 10). Otherwise, the algorithm returns the result from calling the function `SemFDDiff` (l. 13). The function is required to return \checkmark , if fd is a refinement of fd' . Otherwise, the function is required to return a diff witness contained in the semantic difference from fd to fd' . For the implementation of the function `SemFDDiff`, it is possible to use the sound and complete semantic differencing operator for FDs presented in Section 4.3. If Algorithm 15 returns a diff witness, this witness is easily translatable to the model property containing all models containing this witness in their semantics. This property implies the complement of the consistently refines property.

Experiments

We performed experimental evaluations with ten example FDs. Appendix B presents the example FDs in detail. For each pair of thematically related FDs where one of the FDs is not a refinement of the other FD, we executed the algorithms presented in Section 7.5 to compute solutions for the corresponding refinement repair problems. For testing whether intermediately computed FDs satisfy the refines property, we used Algorithm 15 for consistency and refinement checking.

All experiments were executed on a laptop computer with an Intel Core i7-8650U CPU @ 1.90GHz processor, 16GB RAM, and a Samsung PM981 512GB SSD hard drive using Windows 10 and Java 1.8.0.192.

Figure 8.5 summarizes the computation times of the algorithms. We have set a timeout of 15 minutes for each computation. In Figure 3.4, timeouts are indicated by TO. For instance, Algorithm 2 took 218ms to compute a solution for the model repair problem instance $(\mathcal{P}_{\subseteq}(\mathcal{L}_{FD}, O_{FD}, \text{car1}), \text{car})$, *i.e.*, to compute a sequence $t \in O_{PA}$ such that $\emptyset \neq \llbracket \text{car} \triangleright t \rrbracket^{PA} \subseteq \llbracket \text{car1} \rrbracket^{PA}$. In the cases where the algorithms terminated, the computation times range from 91ms to 2m 43s. Algorithm 4 successfully computed solutions for all instances in at most 1m 45s. Figure 8.3 depicts the solutions computed by Algorithm 4 for the input FDs where the computations terminated.

Algorithm 2 and Algorithm 3 have performed about equally well. In all cases where the length of the computed solutions is longer than two, Algorithm 4 performed better than the other algorithms. Thus, in the most cases, Algorithm 4 was faster than Algorithm 5 and Algorithm 6. This might be because the computed solutions are short and the additional computations in Algorithm 5 and Algorithm 6 cause additional overhead.

We conclude that the algorithms handle the small examples, where only short repairing sequences are required, sufficiently quick. However, the algorithms do not scale well for inputs where long repairing sequences are required. In one case, three algorithms timed out and the fastest algorithm took 1m 45s for computing a solution. This is not surprising as the running times of the algorithms are exponential in the lengths of the shortest solutions (cf. Section 7.5.1).

8.4 Instantiations with the Sequence Diagram Language

This section instantiates the model repair framework with the refines, generalizes, and refactors properties and the SD modeling language $\mathcal{L}_{SD} = (M_{SD}, Sem_{SD}, \llbracket \cdot \rrbracket^{SD})$ presented in Chapter 5.

Infinitely many object-addition and action-addition operations are applicable to each SD. In contrast, every SD only contains finitely many objects and finitely many actions. Thus, only finitely many object-deletion, tag-object-as-complete, untag-object-as-complete, tag-object-as-visible, untag-object-as-visible, tag-object-as-initial, untag-object-as-initial, interaction-addition, and interaction-deletion operations are applicable to each SD.

Therefore, it suffices to partition the object-addition and action-addition operations that are applicable to an SD with respect to the property of a concrete model repair problem. The partitioning is similar for all instances of the three model repair problems for the SD language.

Section 8.4.1 presents a change operation partitioning for SD refinement repair problem instances. Then, Section 8.4.2 presents a change operation partitioning for SD

generalization repair problem instances. Afterwards, Section 8.4.3 presents a change operation partitioning for SD refactoring repair problem instances. Finally, Section 8.4.4 presents a repair-representative function for the SD model repair problems.

8.4.1 Sequence Diagram Refinement Repair

This section presents a change operation partitioning for the SD modeling language and the SD refinement repair problem.

Let sd and sd' be two SDs. Then, for all objects o, p that are neither used in sd nor in sd' , the operations $addO_o$ and $addO_p$ induce an equally long shortest solution for the model repair problem instance $I = (\mathcal{P}_{\subseteq}(\mathcal{L}_{SD}, O_{SD}, sd'), sd)$. For every shortest change sequence t that repairs $addO_o(sd)$, it is possible to construct a change sequence u of the same length $|u| = |t|$ such that u repairs $addO_p(sd)$. The SD $addO_o(sd) \triangleright t$ constrains the object o in the same way the SD $addO_p(sd) \triangleright u$ constrains the object p and vice versa. Dually, the SD $addO_p(sd) \triangleright u$ constrains the object o in the same way the SD $addO_o(sd) \triangleright t$ constrains the object p and vice versa. The system runs that are valid in $addO_p(sd) \triangleright u$ can be obtained from the system runs that are valid in $addO_o(sd) \triangleright t$ by exchanging all occurrences of the object o with p and exchanging all occurrences of the object p with o in the system runs that are valid in $addO_o(sd) \triangleright t$. As the objects o and p are not used in sd' , the SD equally constrains the interactions of both objects (due to other tagged objects). Therefore, as $addO_o(sd) \triangleright t$ is by assumption a consistent refinement of sd' and sd' equally constrains the interactions of the objects o and p , $addO_p(sd)$ is also a consistent refinement of sd' .

Proposition 8.19. *Let $sd = (O, O_c, O_v, O_i, A, d)$ and $sd' = (O', O'_c, O'_v, O'_i, A', d')$ be two SDs. Let $I = (\mathcal{P}, sd)$ be an instance of the model repair problem $\mathcal{P} = \mathcal{P}_{\subseteq}(\mathcal{L}_{SD}, O_{SD}, sd')$. Then, for all objects $o, p \in U_N \setminus (O \cup O')$, it holds that $addO_o \sim_I addO_p$.*

Proof. (Sketch.) Let sd , sd' , \mathcal{P} , and I be given as above. Let $o, p \in U_N \setminus (O \cup O')$ be two object names. Let $t \in O_{SD}^*$ be a shortest change sequence that repairs the SD $addO_o(sd)$ towards satisfying $\mathcal{P}_{\subseteq}(\mathcal{L}_{SD}, sd')$. We define the change sequence $u \in O_{SD}^*$ of length $|u| = |t|$ as the change sequence obtained from the sequence t via replacing each change operation in t affecting the object o with the change operation of the same type affecting the object p , instead, and vice versa. A change operation adding an interaction from the object o to the object p with action a at position i , for instance, is replaced by a change operation adding an interaction from the object p to the object o with action a at position i .

By construction of the change sequence u , the SD $addO_o(sd) \triangleright t$ can be obtained from the SD $addO_p(sd) \triangleright u$ via renaming the object o (if it exists) to p and renaming the object p (if it exists) to o in $addO_p(sd) \triangleright u$. Vice versa, the SD $addO_p(sd) \triangleright u$ can be obtained from the SD $addO_o(sd) \triangleright t$ via renaming the object o (if it exists) to p and renaming the object p (if it exists) to o in $addO_o(sd) \triangleright t$. Thus, the object o (respectively

p) is constrained in the SD $\text{add}O_o(sd) \triangleright t$ in the same way the object p (respectively o) is constrained in $\text{add}O_p(sd) \triangleright u$ and vice versa. This especially implies that $\text{add}O_p(sd) \triangleright u$ is consistent because $\text{add}O_o(sd) \triangleright t$ is consistent.

Let (Obj, Act, τ) be a system run that is valid in $\text{add}O_p(sd) \triangleright u$. The above implies that the system run (Obj', Act, τ') obtained from (Obj, Act, τ) by replacing the object o in each interaction of τ and in the set of object O by the object p and replacing the object p in each interaction of τ and in the set of objects O by the object o is valid in $\text{add}O_o(sd) \triangleright t$. As $\text{add}O_o(sd) \triangleright t$ is a consistent refinement of sd' , the system run (Obj', Act, τ') is also a system run of sd' . As the interactions of the objects o and p are equally constrained in sd' (by objects tagged with stereotypes) because the objects are not used in sd' , the system run (Obj, Act, τ) must also be a valid in sd' . From the above, we can conclude $d_{\mathcal{P}}(\text{add}O_o(sd)) \geq d_{\mathcal{P}}(\text{add}O_p(sd))$.

Analogously, we can show that for every shortest change sequence $t \in O_{SD}^*$ that repairs $\text{add}O_p(sd)$ towards satisfying $P_{\subseteq}(\mathcal{L}_{SD}, sd')$, there exists a change sequence $u \in O_{SD}^*$ with $|u| = |t|$ that repairs $\text{add}O_o(sd)$ towards satisfying $P_{\subseteq}(\mathcal{L}_{SD}, sd')$. From this, we can conclude $d_{\mathcal{P}}(\text{add}O_o(sd)) \leq d_{\mathcal{P}}(\text{add}O_p(A))$.

From $d_{\mathcal{P}}(\text{add}O_o(sd)) \geq d_{\mathcal{P}}(\text{add}O_p(A))$ and $d_{\mathcal{P}}(\text{add}O_o(sd)) \leq d_{\mathcal{P}}(\text{add}O_p(A))$, we can conclude that $d_{\mathcal{P}}(\text{add}O_o(sd)) = d_{\mathcal{P}}(\text{add}O_p(A))$. Thus, $\text{add}O_o$ and $\text{add}O_p$ induce an equally long shortest solution for I . \square

Let sd and sd' be two SDs. Then, for all actions a, b that are neither used in sd nor in sd' , the operations $\text{add}Act_a$ and $\text{add}Act_b$ induce an equally long shortest solution for the model repair problem instance $I = (\mathcal{P}_{\subseteq}(\mathcal{L}_{SD}, O_{SD}, sd'), sd)$. For every shortest change sequence t that repairs $\text{add}Act_a(sd)$, it is possible to construct a change sequence u of the same length $|u| = |t|$ such that u repairs $\text{add}Act_b(sd)$. The SD $\text{add}Act_a(sd) \triangleright t$ constrains the interactions with the action a in the same way the SD $\text{add}Act_b(sd) \triangleright u$ constrains the interactions with the action b and vice versa. Dually, the SD $\text{add}Act_b(sd) \triangleright u$ constrains the interactions with the action a in the same way the SD $\text{add}Act_a(sd) \triangleright t$ constrains the interactions with the action b and vice versa. The system runs that are valid in $\text{add}Act_b(sd) \triangleright u$ can be obtained from the system runs that are valid in $\text{add}Act_a(sd) \triangleright t$ by exchanging all occurrences of the action a with the action b and exchanging all occurrences of the action b with the action a in the system runs that are valid in $\text{add}Act_a(sd) \triangleright t$. As the actions a and b are not used in sd' , the SD equally constrains the interactions with the actions a and b . Therefore, as the SD $\text{add}Act_a(sd) \triangleright t$ is by assumption a consistent refinement of the SD sd' and sd' equally constrains the interactions with the actions a and b , it holds that the SD $\text{add}Act_b(sd) \triangleright u$ is also a consistent refinement of the SD sd' .

Proposition 8.20. *Let $sd = (O, O_c, O_v, O_i, A, d)$ and $sd' = (O', O'_c, O'_v, O'_i, A', d')$ be two SDs. Let $I = (\mathcal{P}, sd)$ be an instance of the model repair problem $\mathcal{P} = \mathcal{P}_{\subseteq}(\mathcal{L}_{SD}, O_{SD}, sd')$. Then, for all actions $a, b \in U_N \setminus (A \cup A')$, it holds that $\text{add}Act_a \sim_I \text{add}Act_b$.*

Proof. (Sketch.) Let sd, sd', \mathcal{P} , and I be given as above. Let $a, b \in U_N \setminus (A \cup A')$ be two action labels that are neither used in sd nor in sd' . Assume $t \in O_{SD}^*$ is a shortest change sequence that repairs $addAct_a(sd)$ towards satisfying $P_{\subseteq}(\mathcal{L}_{SD}, sd')$. We define the change sequence u of length $|t|$ as the change sequence obtained from the sequence t via replacing each change operation in t affecting the action a with a change operation of the same type that affects the action b , instead, and vice versa. A change operation adding an interaction from the object o to the object p with action a at position i , for instance, is replaced by a change operation adding an interaction from the object o to the object p with action b at position i .

As $addAct_a$ and $addAct_b$ are applicable to sd , the SD sd does not contain any interactions with actions a or b . Therefore, by construction of the change sequence u , the SD $addAct_a(sd) \triangleright t$ can be obtained from the AD $addAct_b(sd) \triangleright u$ via exchanging all occurrences of the action a in the SD $addAct_b(sd) \triangleright u$ with the action b and exchanging all occurrences of the action b with the action a . Vice versa, the SD $addAct_b(sd) \triangleright u$ can be obtained from the SD $addAct_a(sd) \triangleright t$ via exchanging the actions a and b .

Therefore, if (Obj, Act, τ) is a system run in the semantics of $addAct_a(sd) \triangleright t$, then the system run (Obj, Act', τ') obtained from the system run (Obj, Act, τ) by replacing all occurrences of a by b and all occurrences of b by a in τ and Act is a system run in the semantics of $addAct_b(sd) \triangleright u$ and vice versa. This especially implies that $addAct_b(sd) \triangleright u$ is consistent because $addAct_a(sd) \triangleright t$ is consistent.

Let (Obj, Act, τ) be a system run that is valid in $addAct_b(sd) \triangleright u$. The above implies that the system run (Obj, Act', τ') obtained from (Obj, Act, τ) by replacing the action a in each interaction of τ and in the set of actions Act by the action b and replacing the action b in each interaction of τ and in the set of actions Act by the action a is valid in $addAct_a(sd) \triangleright t$. As $addAct_a(sd) \triangleright t$ is a refinement of sd' , the system run (Obj, Act', τ') is also valid in sd' . As the interactions with the action a and b are equally constrained in sd' (by objects tagged with stereotypes) because the actions are not used in sd' , the system run (Obj, Act, τ) must also be valid in sd' . From the above, we can conclude $d_{\mathcal{P}}(addAct_a(sd)) \geq d_{\mathcal{P}}(addAct_b(sd))$.

Analogously, we can show that for every shortest change sequence $t \in O_{SD}^*$ that repairs $addAct_b(sd)$ towards satisfying $P_{\subseteq}(\mathcal{L}_{SD}, sd')$, there exists a change sequence $u \in O_{SD}^*$ with $|u| = |t|$ that repairs $addAct_a(sd)$ towards satisfying $P_{\subseteq}(\mathcal{L}_{SD}, sd')$. From this, we can conclude $d_{\mathcal{P}}(addAct_a(sd)) \leq d_{\mathcal{P}}(addAct_b(sd))$.

From $d_{\mathcal{P}}(addAct_a(sd)) \geq d_{\mathcal{P}}(addAct_b(sd))$ and $d_{\mathcal{P}}(addAct_a(sd)) \leq d_{\mathcal{P}}(addAct_b(sd))$, we can conclude that $d_{\mathcal{P}}(addAct_a(sd)) = d_{\mathcal{P}}(addAct_b(sd))$. Thus, $addAct_a$ and $addAct_b$ induce an equally long shortest solution for I . \square

Proposition 8.21. *Let $sd = (O, O_c, O_v, O_i, A, d)$ and $sd' = (O', O'_c, O'_v, O'_i, A', d')$ be two SDs. Let $I = (\mathcal{P}, sd)$ be an instance of the model repair problem $\mathcal{P} = \mathcal{P}_{\subseteq}(\mathcal{L}_{SD}, O_{SD}, sd')$. Let objects $o, p \in O$ be objects, $a, b \in A$ be actions, and $i \in \mathcal{N}$ with $0 \leq i < |d|$ be an index. If $\forall j \in \mathbb{N}$ with $0 \leq j < |d'|$, it holds that $d'.j \neq (o,) \text{ addIA}_a \sim_I \text{ addAct}_b$.*

8.4.2 Sequence Diagram Generalization Repair

This section presents a change operation partitioning for the SD modeling language and the SD generalization repair problem.

Let sd and sd' be two SDs. Then, for all objects o, p that are neither used in sd nor in sd' , the operations $addO_o$ and $addO_p$ induce an equally long shortest solution for the model repair problem instance $I = (\mathcal{P}_{\supseteq}(\mathcal{L}_{SD}, O_{SD}, sd'), sd)$. For every shortest change sequence t that repairs $addO_o(sd)$, it is possible to construct a change sequence u of the same length $|u| = |t|$ such that u repairs $addO_p(sd)$. The change sequence u is constructed in the same way as for SD refinement repair problems (cf. Section 8.4.1). The system runs that are valid in $addO_p(sd) \triangleright u$ can be obtained from the system runs that are valid in $addO_o(sd) \triangleright t$ by exchanging all occurrences of the object o with p and exchanging all occurrences of the object p with o in the system runs that are valid in $addO_o(sd) \triangleright t$. As the objects o and p are not used in sd' , the SD equally constrains the interactions of both objects. As $addO_o(sd) \triangleright t$ is by assumption a generalization of sd' and sd' equally constrains the interactions of the objects o and p , $addO_p(sd)$ is also a generalization of sd' .

Proposition 8.22. *Let $sd = (O, O_c, O_v, O_i, A, d)$ and $sd' = (O', O'_c, O'_v, O'_i, A', d')$ be two SDs. Let $I = (\mathcal{P}, sd)$ be an instance of the model repair problem $\mathcal{P} = \mathcal{P}_{\supseteq}(\mathcal{L}_{SD}, O_{SD}, sd')$. Then, for all objects $o, p \in U_N \setminus (O \cup O')$, it holds that $addO_o \sim_I addO_p$.*

Proof. (Sketch.) Let sd , sd' , \mathcal{P} , and I be given as above. Let $o, p \in U_N \setminus (O \cup O')$ be two object names. Let $t \in O_{SD}^*$ be a shortest change sequence that repairs the SD $addO_o(sd)$ towards satisfying $\mathcal{P}_{\supseteq}(\mathcal{L}_{SD}, sd')$. We define the change sequence $u \in O_{SD}^*$ of length $|u| = |t|$ as the sequence obtained from t via replacing each change operation in t affecting the object o with the change operation of the same type affecting p , instead, and vice versa. A change operation adding an interaction from the object o to the object p with action a at position i , for instance, is replaced by a change operation adding an interaction from the object p to the object o with action a at position i .

By construction of the change sequence u , the SD $addO_o(sd) \triangleright t$ can be obtained from the SD $addO_p(sd) \triangleright u$ via renaming the object o (if it exists) to p and renaming the object p (if it exists) to o in $addO_p(sd) \triangleright u$. Vice versa, the SD $addO_p(sd) \triangleright u$ can be obtained from the SD $addO_o(sd) \triangleright t$ via renaming the object o (if it exists) to p and renaming the object p (if it exists) to o in $addO_o(sd) \triangleright t$. Thus, the object o (respectively p) is constrained in the SD $addO_o(sd) \triangleright t$ in the same way the object p (respectively o) is constrained in $addO_p(sd) \triangleright u$. Therefore, if (Obj, Act, τ) is valid in $addO_o(sd) \triangleright t$, then the system run (Obj', Act, τ') obtained from the system run (Obj, Act, τ) via exchanging the object o in the interactions of τ and in the set Obj with the object p and exchanging the object p with the object o is valid in $addO_p(sd) \triangleright u$.

Let (Obj, Act, τ) be a system run that is valid in sd' . As the objects o and p are not used in the SD sd' , the SD sd' equally constrains the interactions of the objects o and

p (due to objects that are possibly tagged in sd'). Let (Obj', Act, τ') be the system run obtained from the system run (Obj, Act, τ) by replacing the object o in the interactions of τ and in the set of object Obj by the object p and replacing the object p in the interactions and in the set Obj by the object o . As sd' equally constrains the objects p and o and as (Obj, Act, τ) is valid in sd' , the system run (Obj', Act, τ') is valid in sd' .

As (Obj', Act, τ') is valid in sd' and $addO_o(sd) \triangleright t$ is a generalization of sd' , the system run (Obj', Act, τ') is also valid in $addO_o(sd) \triangleright t$. From the above, it follows that the system run (Obj, Act, τ) (obtained from (Obj', Act, τ') via exchanging the object o in the interactions of τ' and in the set Obj' with the object p and exchanging the object p with the object o) is valid in $addO_p(sd) \triangleright u$. We can conclude $d_{\mathcal{P}}(addO_o(sd)) \geq d_{\mathcal{P}}(addO_p(sd))$.

Analogously, we can show that for every shortest change sequence $t \in O_{SD}^*$ that repairs $addO_p(sd)$ towards satisfying $P_{\supseteq}(\mathcal{L}_{SD}, sd')$, there exists a change sequence $u \in O_{SD}^*$ with $|u| = |t|$ that repairs $addO_o(sd)$ towards satisfying $P_{\supseteq}(\mathcal{L}_{SD}, sd')$. From this, we can conclude $d_{\mathcal{P}}(addO_o(sd)) \leq d_{\mathcal{P}}(addO_p(sd))$.

From $d_{\mathcal{P}}(addO_o(sd)) \geq d_{\mathcal{P}}(addO_p(sd))$ and $d_{\mathcal{P}}(addO_o(sd)) \leq d_{\mathcal{P}}(addO_p(sd))$, we can conclude that $d_{\mathcal{P}}(addO_o(sd)) = d_{\mathcal{P}}(addO_p(sd))$. Thus, $addO_o$ and $addO_p$ induce an equally long shortest solution for I . \square

Let sd and sd' be two SDs. Then, for all actions a, b that are neither used in sd nor in sd' , the operations $addAct_a$ and $addAct_b$ induce an equally long shortest solution for the model repair problem instance $I = (\mathcal{P}_{\supseteq}(\mathcal{L}_{SD}, O_{SD}, sd'), sd)$. For every shortest change sequence t that repairs $addAct_a(sd)$, it is possible to construct a change sequence u of the same length $|u| = |t|$ such that u repairs $addAct_b(sd)$. The change sequence u is constructed in the same way as for SD refinement repair problems (cf. Section 8.4.1). The system runs that are valid in $addA_b(sd) \triangleright u$ can be obtained from the system runs that are valid in $addA_a(sd) \triangleright t$ by exchanging all occurrences of the action a with the action b and exchanging all occurrences of the action b with the action a in the system runs that are valid in $addAct_a(sd) \triangleright t$. As the actions a and b are not used in sd' , the SD equally constrains the interactions with the actions a and b . Therefore, as $addA_a(sd) \triangleright t$ is by assumption a generalization of sd' and sd' equally constrains the interactions with the actions a and b , $addA_b(sd)$ is also a generalization of sd' .

Proposition 8.23. *Let $sd = (O, O_c, O_v, O_i, A, d)$ and $sd' = (O', O'_c, O'_v, O'_i, A', d')$ be two SDs. Let $I = (\mathcal{P}, sd)$ be an instance of the model repair problem $\mathcal{P} = \mathcal{P}_{\supseteq}(\mathcal{L}_{SD}, O_{SD}, sd')$. Then, for all actions $a, b \in U_N \setminus (A \cup A')$, it holds that $addAct_a \sim_I addAct_b$.*

Proof. (Sketch.) Let sd, sd', \mathcal{P} , and I be given as above. Let $a, b \in U_N \setminus (A \cup A')$ be two actions that are neither used in sd nor in sd' . Assume $t \in O_{SD}^*$ is a shortest change sequence that repairs $addAct_a(sd)$ towards satisfying $P_{\supseteq}(\mathcal{L}_{SD}, sd')$. We define the change sequence u of length $|u| = |t|$ as the change sequence obtained from the sequence t via replacing each change operation in t affecting the action a with a change operation of the same type that affects the action b , instead, and vice versa. A change operation

adding an interaction from the object o to the object p with action a at position i , for instance, is replaced by a change operation adding an interaction from the object o to the object p with action b at position i .

As $addAct_a$ and $addAct_b$ are applicable to sd , the SD sd does not contain any interactions with the action a or b . Therefore, by construction of the change sequence u , the SD $addAct_a(sd) \triangleright t$ can be obtained from the AD $addAct_b(sd) \triangleright u$ via exchanging all occurrences of the action a in the SD $addAct_b(sd) \triangleright u$ with the action b and exchanging all occurrences of the action b with the action a . Vice versa, the SD $addAct_b(sd) \triangleright u$ can be obtained from the SD $addAct_a(sd) \triangleright t$ via exchanging the actions a and b .

Therefore, if (Obj, Act, τ) is a system run in the semantics of $addAct_a(sd) \triangleright t$, then the system run (Obj, Act', τ') obtained from (Obj, Act, τ) by replacing all occurrences of a with b in τ and Act' and replacing all occurrences of b with a in τ and Act' is a trace in the semantics of $addAct_b(sd) \triangleright u$ and vice versa.

Let (Obj, Act, τ) be a system run that is valid in sd' . As the actions a and b are not used in the SD sd' , the SD sd' equally constrains all interactions with the actions a or b (due to objects that are possibly tagged in sd'). Let (Obj, Act', τ') be the system run obtained from the system run (Obj, Act, τ) by replacing the action a in the interactions of τ and in the set Act by the action b and replacing the action b in the interactions of τ and in the set Act by the action a . As sd' equally constrains the interactions with the actions a and b and as (Obj, Act, τ) is valid in sd' , the system run (Obj, Act', τ') is valid in sd' .

As (Obj, Act', τ') is valid in sd' and $addAct_a(sd) \triangleright t$ is a generalization of sd' , the system run (Obj, Act', τ') is also valid in $addAct_a(sd) \triangleright t$. From the above, it follows that the system run (Obj, Act, τ) (obtained from (Obj, Act', τ') via exchanging the action a in the interactions of τ' and in the set Act with the action b and exchanging the action b with the action a) is valid in $addAct_b(sd) \triangleright u$. From this, we can conclude $d_P(addAct_a(sd)) \geq d_P(addAct_b(sd))$.

Analogously, we can show that for every shortest change sequence $t \in O_{SD}^*$ that repairs $addAct_b(sd)$ towards satisfying $P_{\supseteq}(\mathcal{L}_{SD}, sd')$, there exists a change sequence $u \in O_{SD}^*$ with $|u| = |t|$ that repairs $addAct_a(sd)$ towards satisfying $P_{\supseteq}(\mathcal{L}_{SD}, sd')$. From this, we can conclude $d_P(addAct_a(sd)) \leq d_P(addAct_b(sd))$.

From $d_P(addAct_a(sd)) \geq d_P(addAct_b(sd))$ and $d_P(addAct_a(sd)) \leq d_P(addAct_b(sd))$, we can conclude that $d_P(addAct_a(sd)) = d_P(addAct_b(sd))$. Thus, $addAct_a$ and $addAct_b$ induce an equally long shortest solution for I . \square

8.4.3 Sequence Diagram Refactoring Repair

This section presents a change operation partitioning for the SD modeling language and the SD refactoring repair problem.

Let sd and sd' be two SDs. Then, for all objects o, p that are neither used in sd nor in sd' , the operations $addO_o$ and $addO_p$ induce an equally long shortest solution

for the model repair problem instance $I = (\mathcal{P}_=(\mathcal{L}_{SD}, O_{SD}, sd'), sd)$. The reason is a combination of the reasons for the equivalences in the contexts of SD refinement repair problems (cf. Section 8.4.1) and SD generalization repair problems (cf. Section 8.4.2).

Proposition 8.24. *Let $sd = (O, O_c, O_v, O_i, A, d)$ and $sd' = (O', O'_c, O'_v, O'_i, A', d')$ be two SDs. Let $I = (\mathcal{P}, sd)$ be an instance of the model repair problem $\mathcal{P} = \mathcal{P}_=(\mathcal{L}_{SD}, O_{SD}, sd')$. Then, for all objects $o, p \in U_N \setminus (O \cup O')$, it holds that $addO_o \sim_I addO_p$.*

Proof. The proof is a combination of the proofs for Proposition 8.19 and Proposition 8.22. \square

Let sd and sd' be two SDs. Then, for all actions a, b that are neither used in sd nor in sd' , the operations $addAct_a$ and $addAct_b$ induce an equally long shortest solution for the model repair problem instance $I = (\mathcal{P}_=(\mathcal{L}_{SD}, O_{SD}, sd'), sd)$. The reason is a combination of the reasons for the equivalences in the contexts of SD refinement repair problems (cf. Section 8.4.1) and SD generalization repair problems (cf. Section 8.4.2).

Proposition 8.25. *Let $sd = (O, O_c, O_v, O_i, A, d)$ and $sd' = (O', O'_c, O'_v, O'_i, A', d')$ be two SDs. Let $I = (\mathcal{P}, sd)$ be an instance of the model repair problem $\mathcal{P} = \mathcal{P}_=(\mathcal{L}_{SD}, O_{SD}, sd')$. Then, for all actions $a, b \in U_N \setminus (A \cup A')$, it holds that $addAct_a \sim_I addAct_b$.*

Proof. The proof is a combination of the proofs for Proposition 8.20 and Proposition 8.23. \square

8.4.4 Repair-Representative Function and Example Applications

From the argumentations in Section 8.4.1, Section 8.4.2, and Section 8.4.3, we can conclude that it is possible to use the same repair-representative function for the SD refinement, generalization, and refactoring repair problems using the same models. For each SD $sd \in M_{SD}$, we can construct the repair-representative function $\mathcal{R}_{sd} : M_{SD} \rightarrow \wp_{fin}(O_{SD})$ for each of the SD model repair problems as follows: The function \mathcal{R}_{sd} maps each SD to the set containing

- all tag-object-as-complete, untag-object-as-complete, untag-object-as-visible, untag-object-as-initial, tag-object-as-visible, object-deletion, interaction-addition, tag-object-as-initial, interaction-deletion, operations that are applicable to the SD,
- all object-addition and action-addition operations that have not been partitioned in Section 8.4.1, Section 8.4.2, and Section 8.4.3 and are applicable to the SD,
- exactly one arbitrary but fixed change operation of each equivalence class of the object-addition and action-addition operations identified in Section 8.4.1, Section 8.4.2, and Section 8.4.3.

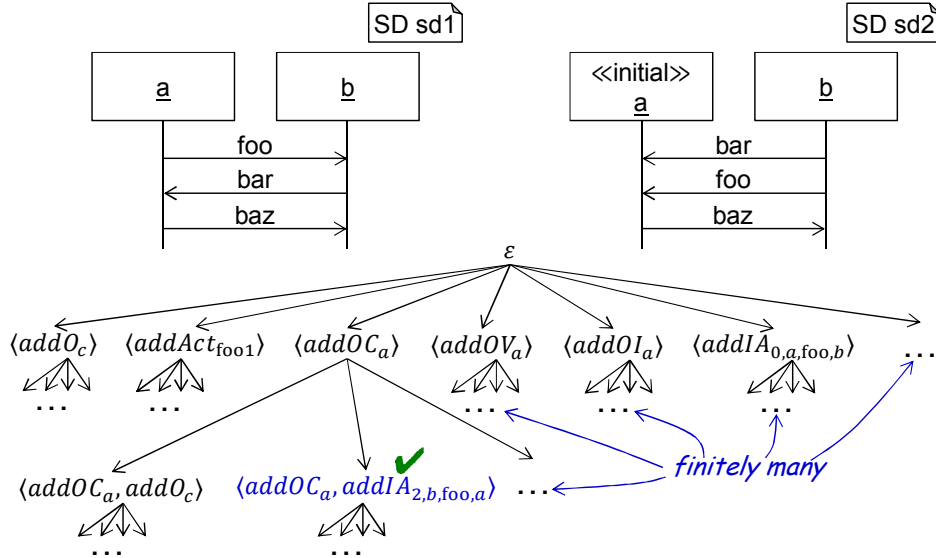


Figure 8.7: Two simple SDs and an excerpt of the change sequence search tree for computing a shortest solution for an SD refinement repair problem.

For example, Figure 8.7 depicts an excerpt of the change sequence search tree for computing a shortest solution for an SD refinement repair problem instance. The SD sd1 is the model of the instance. The property of the refinement repair problem contains all consistent SDs that are refinements of the SD sd2.

8.4.5 Implementation and Experiments

We implemented the model repair algorithms for the SD modeling language to perform experimental evaluations. The implementation is written in Java and uses the semantic differencing operator implementation presented in Section 5.3.

We performed experimental evaluations by executing the algorithms for computing shortest solutions for SD refinement repair problem instances using the seven example SDs presented in Appendix C. The purpose of the experiments is twofold. The first purpose is testing whether computing shortest repairing sequences is feasible for the example SD refinement repair problem instances. The second purpose is comparing the performances of the different algorithms in the context of the example SD refinement repair problems.

Algorithm 16 Checking whether the SD sd is a refinement of the SD sd' including a heuristic for diff witness computation.

Input: Two SDs $sd = (O, O_c, O_v, O_i, A, d)$ and $sd' = (O', O'_c, O'_v, O'_i, A', d')$.

Output: A system run $r \in \delta(sd, sd')$ if $\delta(sd, sd') \neq \emptyset$. Otherwise, the special symbol \checkmark if sd is a refinement of sd' .

```

1: if  $|d| < |d'|$  then
2:   return  $(O, A, d)$ 
3: else if  $|d| = |d'| \wedge d \neq d'$  then
4:   return  $(O, A, d)$ 
5: else if  $(O, A, d) \notin \llbracket sd' \rrbracket^{SD}$  then
6:   return  $(O, A, d)$ 
7: end if
8: return SEMSDDIFF( $sd, sd'$ )
    
```

Heuristic for Witness Computation

The computational complexity of semantic differencing of SDs is high and the algorithms often execute the semantic differencing operator. To achieve performance improvements, we implemented a heuristic for fast witness computation. The heuristic compares the lengths of sequences of diagram interactions and relies on the following observations.

Proposition 8.26. *Let $sd = (O, O_c, O_v, O_i, A, d)$ and $sd' = (O', O'_c, O'_v, O'_i, A', d')$ be two SDs. Then, the following statements hold:*

1. *If $|d| < |d'|$, then $r \in \delta(sd, sd')$ where $r = (O, A, d)$.*
2. *If $|d| = |d'|$ and $d \neq d'$, then $r \in \delta(sd, sd')$ where $r = (O, A, d)$.*

Proof. Let sd and sd' be given as above.

Proof of 1.: Assume $|d| < |d'|$. Then, by definition of SD semantics, it holds that $r = (O, A, d) \in \llbracket sd \rrbracket^{SD}$. As $|d| < |d'|$, it must hold that $r \notin \llbracket sd' \rrbracket^{SD}$ because, by definition of SD semantics, $r \in \llbracket sd' \rrbracket^{SD}$ would imply that $|d| \geq |d'|$.

Proof of 2.: Assume $|d| = |d'|$ and $d \neq d'$. Then, by definition of SD semantics, it holds that $r = (O, A, d) \in \llbracket sd \rrbracket^{SD}$. As $|d| = |d'|$ and $d \neq d'$ it holds that $r \notin \llbracket sd' \rrbracket^{SD}$ because, by definition of SD semantics, $r \in \llbracket sd' \rrbracket^{SD}$ would imply that $d = d'$ as d' is the only trace of all systems runs in the semantics of sd' with traces of length $|d'|$. \square

Algorithm 16 is the semantic differencing algorithm including the heuristic. The algorithm takes two SDs sd and sd' as inputs. For checking whether sd is a refinement of sd' , the heuristic compares the sequences of diagram interactions of the SDs according to Proposition 8.26 (ll. 1,3). If one of the conditions stated in Proposition 8.26 is satisfied, the algorithm returns the corresponding diff witness (ll. 2,4). Then, for fast diff witness

computation, the algorithm checks whether the system run containing exactly the objects and diagram interactions as the SD sd is valid in sd' (l. 5). As shown in the proof of Proposition 8.26, this system run is guaranteed to be an element of the semantics of sd . If the system run is not an element of the semantics of sd' , then the algorithm returns it as a diff witness (l. 6). Finally, if the algorithm does not heuristically find a diff witness, it returns the result from calling the function SemSDDiff (l. 8). The function is required to return \checkmark if sd is a refinement of sd' . Otherwise, the function is required to return a diff witness contained in the semantic difference from sd to sd' . For the evaluation, we use the semantic differencing operator implementation presented in Section 5.3. Each diff witness w returned by Algorithm 16 can easily be transformed to the property $\{m \in M_{SD} \mid w \in \llbracket m \rrbracket^{SD}\}$ containing all models containing the witness w in their semantics. As every SD is consistent, it is not necessary to include a SD consistency check in Algorithm 16.

Proposition 8.27. *Every SD is consistent.*

Proof. Let $sd = (O, O_c, O_v, O_i, A, d)$ be an SD. By definition of SD semantics, it directly follows that $(O, A, d) \in \llbracket sd \rrbracket^{SD}$. \square

Search Space Restrictions

For pragmatic reasons, to increase the performance of the algorithms, we further restrict the search spaces as follows.

We consider only interaction-addition operations adding interactions that either occur in one of the models or involve an action that does not occur in any interaction of the models. We do not consider interaction-addition operations that would cause the model obtained from applying a currently processed change sequence to contain the interaction more often than the repair problem's model. These restrictions are appropriate because the application of other interaction-addition operations would result in models that impose different or stronger constraints than the model of the repair problem.

For any model obtained from applying a currently processed change sequence, if there exist interactions occurring in the repair problem's model that do not occur in the current model, then we do not consider interaction-addition operations different from the operations for adding these interactions. This restriction is reasonable because the repaired model must contain the interactions occurring in the repair problem's model as these interactions are required to occur in the traces of the system runs in the semantics.

For the same reason, if the set of actions of a model obtained from applying a currently processed change sequence does not contain an action that is used in the sequence of diagram interactions of the repair problem's model, then we solely consider an action-addition operator for adding such an action.

For each model resulting from applying a currently processed change sequence, we only consider operations for tagging objects in the model that are also tagged accordingly in

Model	Problem	Time				
		Algo. 2	Algo. 3	Algo. 4	Algo 5	Algo. 6
sd1	sd2	3576ms	1245ms	1011ms	634ms	1021ms
sd2	sd1	250ms	224ms	228ms	232ms	374ms
rob1	rob3	6886ms	3468ms	1898ms	1972ms	3505ms
rob1	rob4	1041ms	526ms	525ms	539ms	872ms
rob1	rob5	627ms	625ms	669ms	680ms	1140ms
rob2	rob1	559ms	549ms	26s	18s	33s
rob2	rob3	3867ms	2374ms	1500ms	1343ms	2301ms
rob2	rob4	748ms	706ms	TO	12m 22s	TO
rob2	rob5	3269ms	3096ms	TO	TO	TO
rob3	rob1	790ms	510ms	28s	19s	30s
rob3	rob4	1185ms	604ms	TO	9m 16s	TO
rob3	rob5	2754ms	3065ms	TO	TO	TO
rob4	rob1	500ms	535ms	551ms	526ms	552ms
rob4	rob2	757ms	480ms	505ms	464ms	473ms
rob4	rob3	11s	6879ms	2081ms	2239ms	2790ms
rob4	rob5	642ms	653ms	1156ms	1065ms	1419ms
rob5	rob1	538ms	1074ms	922ms	950ms	1025ms
rob5	rob2	522ms	731ms	708ms	897ms	1022ms
rob5	rob3	11s	8902ms	3684ms	4127ms	4374ms
rob5	rob4	591ms	584ms	599ms	1060ms	974ms

Figure 8.8: The execution times of Algorithm 2 - Algorithm 6 when given the SD refinement repair problem instances as inputs.

the model of the repair problem. This restriction is reasonable because the application of other tagging operations would result in models that impose different or stronger constraints than the model of the repair problem.

Experiments

We performed experimental evaluations with seven example SDs. Appendix C presents the example SDs in detail. For each pair of thematically related SDs where one of the SDs is not a refinement of the other SD, we executed the algorithms presented in Section 7.5 to compute solutions for the corresponding refinement repair problems. For testing whether intermediately computed SDs satisfy the refines property, we used Algorithm 16 for refinement checking.

All experiments were executed on a laptop computer with an Intel Core i7-8650U CPU @ 1.90GHz processor, 16GB RAM, and a Samsung PM981 512GB SSD hard drive using Windows 10 and Java 1.8.0_192.

Figure 8.8 summarizes the computation times of the algorithms. We have set a timeout of 15 minutes for each computation. In Figure 3.4, timeouts are indicated by TO.

For instance, Algorithm 2 took 3567ms to compute a solution for the model repair problem instance $(\mathcal{P}_{\subseteq}(\mathcal{L}_{SD}, O_{SD}, sd2), sd1)$, *i.e.*, to compute a sequence $t \in O_{PA}$ such that $\emptyset \neq \llbracket sd1 \triangleright t \rrbracket^{PA} \subseteq \llbracket sd2 \rrbracket^{PA}$. In the cases where the algorithms terminated, the computation times range from 224ms to 12m 22s. Algorithm 3 successfully computed solutions for all instances in at most 8902ms. Figure 8.9 depicts the solutions computed by Algorithm 3 for the input SDs. Figure 8.9 uses the abbreviations for action and object names that are introduced in Figure 5.10 (cf. p. 106) to save space.

In the most cases, Algorithm 3 and Algorithm 2 have performed about equally well. In two cases, Algorithm 2 took 11s to calculate solutions, whereas Algorithm 3 took less than one second in both cases. In all but two cases where the lengths of the computed solutions was greater than three, Algorithm 3 performed better than the other algorithms. Each of the algorithms Algorithm 4 - Algorithm 6 timed out for at least one of the inputs. The algorithms Algorithm 2 and Algorithm 3 did not time out for any input and took at most 8902ms for computing a solution. We conclude that Algorithm 2 and Algorithm 3 are best to compute solutions for the SD refinement repair problem instances.

The algorithms handle the small examples, where only short repairing sequences are required, sufficiently quick. However, the running times of the algorithms are exponential in the lengths of the shortest solutions (cf. Section 7.5.1). Thus, the algorithms might not scale well for larger SDs.

8.5 Instantiations with the Activity Diagram Language

This section instantiates the model repair framework with the refines, generalizes, and refactors properties and the AD modeling language $\mathcal{L}_{AD} = (M_{AD}, Sem_{AD}, \llbracket \cdot \rrbracket^{AD})$ presented in Chapter 6.

Infinitely many label-addition, action-insertion, xor-fragment-insertion, and-fragment-insertion, and cyclic-fragment-insertion operations are applicable to each AD. However, only finitely many label-deletion, action-deletion, xor-fragment-deletion, and-fragment-deletion, cyclic-fragment-deletion, fragment-branch-insertion as well as fragment-branch-deletion operations are applicable to an AD. Computing the applicable operations for an AD is straight-forward.

Thus, it suffices to partition the label-addition, action-insertion, xor-fragment-insertion, and-fragment-insertion, and cyclic-fragment-insertion operations applicable to each AD into finitely many equivalence classes with respect to an AD model repair problem.

Section 8.5.1 presents a change operation partitioning for AD refinement repair problem instances. Then, Section 8.5.2 presents a change operation partitioning for AD generalization repair problem instances. Afterwards, Section 8.5.3 presents a change operation partitioning for AD refactoring repair problem instances. Finally, Section 8.5.4 presents a repair-representative function for the AD model repair problems.

Model	Problem	Solution
sd1	sd2	$addI A_{2,b,foo,a}, addOV_a$
sd2	sd1	$addI A_{0,a,foo,b}$
rob1	rob3	$addOC_c$
rob1	rob4	$addAct_{af}, addI A_{7,ae,af,c}$
rob1	rob5	$addAct_{ab}, addAct_{aa}, addI A_{7,ae,ab,c}, addI A_{5,ui,aa,ae}$
rob2	rob1	$addAct_{gs}, addAct_s, addO_{sp}, addI A_{2,sp,s,pl}, addI A_{2,pl,gs,sp}$
rob2	rob3	$addOC_c$
rob2	rob4	$addAct_{af}, addAct_{gs}, addAct_s, addI A_{5,ae,af,c}, addO_{sp}, addI A_{2,sp,s,pl}, addI A_{2,pl,gs,sp}$
rob2	rob5	$addAct_{ab}, addAct_{gs}, addAct_s, addAct_{aa}, addI A_{5,ae,ab,c}, addI A_{3,ui,aa,ae}, addO_{sp}, addI A_{2,sp,s,pl}, addI A_{2,pl,gs,sp}$
rob3	rob1	$addAct_{gs}, addAct_s, addO_{sp}, addI A_{2,sp,s,pl}, addI A_{2,pl,gs,sp}$
rob3	rob4	$addAct_{af}, addAct_{gs}, addAct_s, addI A_{5,ae,af,c}, addO_{sp}, addI A_{2,sp,s,pl}, addI A_{2,pl,gs,sp}$
rob3	rob5	$addAct_{ab}, addAct_{gs}, addAct_s, addAct_{aa}, addI A_{5,ae,ab,c}, addI A_{3,ui,aa,ae}, addO_{sp}, addI A_{2,sp,s,pl}, addI A_{2,pl,gs,sp}$
rob4	rob1	$addAct_{as}, addI A_{7,ae,as,c}$
rob4	rob2	$addAct_{as}, addI A_{7,ae,as,c}$
rob4	rob3	$addAct_{as}, addI A_{6,ae,as,c}, addOC_c$
rob4	rob5	$addAct_{ab}, addAct_{aa}, addI A_{7,ae,ab,c}, addI A_{5,ui,aa,ae}$
rob5	rob1	$addAct_{as}, addI A_{8,ae,as,c}$
rob5	rob2	$addAct_{as}, addI A_{8,ae,as,c}$
rob5	rob3	$addAct_{as}, addI A_{7,ae,as,c}, addOC_c$
rob5	rob4	$addAct_{af}, addI A_{8,ae,af,c}$

Figure 8.9: The shortest solutions computed by Algorithm 3 for the SD refinement repair problem instances.

8.5.1 Activity Diagram Refinement Repair

This section presents a change operation partitioning for the AD modeling language and the AD refinement repair problem.

Let ad and ad' be two ADs, (a, b) be a transition of ad and c be an action label of ad . Then, for all node names d, e that are not used in ad , the operations $addA_{a,b,d,c}$ and $addA_{a,b,e,c}$ induce an equally long shortest solution for the model repair problem instance $I = (\mathcal{P}_{\subseteq}(\mathcal{L}_{AD}, O_{AD}, ad'), ad)$. The reason for this is that node names do not influence the traces in the semantics of ADs. For every shortest change sequence t that repairs $addA_{a,b,d,c}(ad)$, it is possible to construct a change sequence u of the same length $|u| = |t|$ that repairs $addA_{a,b,e,c}(ad)$. The ADs $addA_{a,b,d,c}(ad) \triangleright t$ and $addA_{a,b,e,c}(ad) \triangleright u$ solely differ in the naming of their nodes. More precisely, the AD $addA_{a,b,d,c}(ad) \triangleright t$ can be obtained from the AD $addA_{a,b,e,c}(ad) \triangleright u$ by renaming the node d (if it exists) to e and renaming the node e (if it exists) to d in $addA_{a,b,e,c}(ad) \triangleright u$. Analogously, the AD $addA_{a,b,e,c}(ad) \triangleright u$ can be obtained from the AD $addA_{a,b,d,c}(ad) \triangleright t$. As node names do not directly influence the traces in the semantics of ADs, the ADs have equal semantics.

Proposition 8.28. *Let $ad = (L, N, t, AND, XOR, C, T, l)$ and $ad' = (L', N', t', AND', XOR', C', T', l')$ be two ADs, $(a, b) \in T$ be a transition of ad , and $c \in L$ be an action label of ad . Further, let $I = (\mathcal{P}, ad)$ be an instance of the model repair problem $\mathcal{P} = \mathcal{P}_{\subseteq}(\mathcal{L}_{AD}, O_{AD}, ad')$. Then, for all names $d, e \in U_N \setminus N$, it holds $addA_{a,b,d,c} \sim_I addA_{a,b,e,c}$.*

Proof. (Sketch.) Let $ad, ad', (a, b), c, \mathcal{P}$, and I be given as above. Let $d, e \in U_N \setminus N$ be two node names that are not used in ad . Assume $t \in O_{AD}^*$ is a shortest change sequence that repairs $addA_{a,b,d,c}(ad)$ towards satisfying $\mathcal{P}_{\subseteq}(\mathcal{L}_{AD}, ad')$. We define u as the change sequence of length $|u| = |t|$ obtained from t by replacing each change operation in t affecting the node d by the change operation of the same type affecting the node e , instead, and vice versa. A change operation for adding an action node x labeled y between d and another node n , for instance, is replaced by a change operation adding the action node x labeled y between the node e and the node n . Change operations that do not affect the nodes e and d , such as label addition operations, are left unchanged.

By construction of the change sequence u , the AD $addA_{a,b,d,c}(ad) \triangleright t$ can be obtained from the AD $addA_{a,b,e,c}(ad) \triangleright u$ by renaming the node d in $addA_{a,b,e,c}(ad) \triangleright u$ (if it exists) to e and renaming the node e in $addA_{a,b,e,c}(ad) \triangleright u$ (if it exists) to d . Vice versa, the AD $addA_{a,b,e,c}(ad) \triangleright u$ can be obtained from the AD $addA_{a,b,d,c}(ad) \triangleright t$ by applying the same renaming in $addA_{a,b,d,c}(ad) \triangleright t$.

As the ADs $addA_{a,b,e,c}(ad) \triangleright u$ and $addA_{a,b,d,c}(ad) \triangleright t$ solely differ in the naming of their nodes and node names do not affect the traces in the semantics of ADs, the two ADs are semantically equivalent. Therefore, as the AD $addA_{a,b,d,c}(ad) \triangleright t$ is by assumption a consistent refinement of the AD ad' , the AD $addA_{a,b,e,c}(ad) \triangleright u$ is also a consistent refinement of the AD ad' . From the above, we can conclude that $d_{\mathcal{P}}(addA_{a,b,d,c}(ad)) \geq d_{\mathcal{P}}(addA_{a,b,e,c}(ad))$ holds.

Analogously, we can show that for every shortest change sequence $t \in O_{AD}^*$ that repairs $addA_{a,b,e,c}(ad)$ towards satisfying $P_{\subseteq}(\mathcal{L}_{AD}, ad')$, there exists a change sequence $u \in O_{AD}^*$ with $|u| = |t|$ such that u repairs $addA_{a,b,d,c}(ad)$ towards satisfying $P_{\subseteq}(\mathcal{L}_{AD}, ad')$. From this, we can conclude that $d_{\mathcal{P}}(addA_{a,b,d,c}(ad)) \leq d_{\mathcal{P}}(addA_{a,b,e,c}(ad))$.

Therefore, from $d_{\mathcal{P}}(addA_{a,b,d,c}(ad)) \geq d_{\mathcal{P}}(addA_{a,b,e,c}(ad))$ and $d_{\mathcal{P}}(addA_{a,b,d,c}(ad)) \leq d_{\mathcal{P}}(addA_{a,b,e,c}(ad))$, we can conclude $d_{\mathcal{P}}(addA_{a,b,d,c}(ad)) = d_{\mathcal{P}}(addA_{a,b,e,c}(ad))$. Thus, $addA_{a,b,d,c}$ and $addA_{a,b,e,c}$ induce an equally long shortest solution for I . \square

Let ad and ad' be two ADs. Then, for all action labels a, b that are neither used in ad nor in ad' , the operations $addL_a$ and $addL_b$ induce an equally long shortest solution for the model repair problem instance $I = (\mathcal{P}_{\subseteq}(\mathcal{L}_{AD}, O_{AD}, ad'), ad)$. For every shortest change sequence t that repairs $addL_a(ad)$, it is possible to construct a change sequence u of the same length $|u| = |t|$ such that u repairs $addL_b(ad)$. The ADs $addL_a \triangleright t$ and $addL_b \triangleright u$ solely differ in the labeling of their action nodes. Every node in $addL_a(ad) \triangleright t$ that is labeled with a is labeled with b in $addL_b(ad) \triangleright u$ and vice versa. Similarly, every node in $addL_b(ad) \triangleright u$ that is labeled with a is labeled with b in $addL_a(ad) \triangleright t$ and vice versa. Thus, for every trace in the semantics of $addL_a(ad) \triangleright t$, the trace obtained from replacing each occurrence of a by b and replacing each occurrence of b by a in the trace is a trace in the semantics of $addL_b(ad) \triangleright u$ and vice versa. As the AD ad' does neither use the action label a nor the action label b , its semantics does not contain any trace that uses one of the labels. As further $addL_a(ad) \triangleright t$ is a consistent refinement of ad' , the semantics of the AD $addL_a(ad) \triangleright t$ does not contain any trace using the labels, either. Therefore, the ADs $addL_a(ad) \triangleright t$ and $addL_b(ad) \triangleright u$ must be semantically equivalent, which implies that $addL_b(ad) \triangleright u$ is a consistent refinement of ad' .

Proposition 8.29. *Let $ad = (L, N, t, AND, XOR, C, T, l)$ and $ad' = (L', N', t', AND', XOR', C', T', l')$ be two ADs. Further, let $I = (\mathcal{P}, ad)$ be an instance of the model repair problem $\mathcal{P} = \mathcal{P}_{\subseteq}(\mathcal{L}_{AD}, O_{AD}, ad')$. Then, for all labels $a, b \in U_N \setminus (L \cup L')$, it holds that $addL_a \sim_I addL_b$.*

Proof. (Sketch.) Let ad , ad' , \mathcal{P} , and I be given as above. Let $a, b \in U_N \setminus (L \cup L')$ be two action labels that are neither used in ad nor in ad' . Assume $t \in O_{AD}^*$ is a shortest change sequence that repairs $addL_a(ad)$ towards satisfying $P_{\subseteq}(\mathcal{L}_{AD}, ad')$. We define the change sequence u of length $|t|$ as the change sequence obtained from the sequence t via replacing each change operation in t affecting the action label a with a change operation of the same type that affects the label b , instead, and vice versa. A change operation inserting an action with label a between two succeeding nodes, for instance, is replaced by a change operation inserting the same action with label b between the same nodes.

As $addL_a$ and $addL_b$ are applicable to ad , the AD ad cannot contain any actions labeled a or b . Therefore, by construction of the change sequence u , the AD $addL_a(ad) \triangleright t$ can be obtained from the AD $addL_b(ad) \triangleright u$ via changing the label of each action in $addL_b(ad) \triangleright u$ that is labeled with a to b and changing the label of each action that is labeled with b

to a . Vice versa, the AD $addL_b(ad) \triangleright u$ can be obtained from the AD $addL_a(ad) \triangleright t$ via applying the same relabeling.

Therefore, if τ is a trace in the semantics of $addL_a(ad) \triangleright t$, then the trace τ' obtained from replacing all occurrences of a by b and all occurrences of b by a in τ is a trace in the semantics of $addL_b(ad) \triangleright u$ and vice versa. This especially implies that $addL_b(ad) \triangleright u$ is consistent because $addL_a(ad) \triangleright t$ is consistent.

Suppose towards a contradiction that $addL_b(ad) \triangleright u$ is not a refinement of ad' . Then, there exists a trace τ in the semantics of $addL_b(ad) \triangleright u$ that is not an element of the semantics of ad' . From the above, it follows that the trace τ' obtained from replacing all occurrences of a by b and all occurrences of b by a in τ is a trace in the semantics of $addL_a(ad) \triangleright t$. As $addL_a(ad) \triangleright t$ is a refinement of ad' and the action labels a and b do not exist in ad' , the trace τ' does neither contain a nor b . Therefore, $\tau = \tau'$. However, this contradicts that τ' is not an element of the semantics of ad' because τ' is an element of the semantics of $addL_a(ad) \triangleright t$ and $addL_a(ad) \triangleright t$ is a refinement of ad' . From the above, we can conclude that $d_{\mathcal{P}}(addL_a(ad)) \geq d_{\mathcal{P}}(addL_b(ad))$.

Analogously, we can show that for every shortest change sequence $t \in O_{AD}^*$ that repairs $addL_b(ad)$ towards satisfying $P_{\subseteq}(\mathcal{L}_{AD}, ad')$, there exists a change sequence $u \in O_{AD}^*$ with $|u| = |t|$ such that u repairs $addL_a(ad)$ towards satisfying $P_{\subseteq}(\mathcal{L}_{AD}, ad')$. From this, we can conclude that $d_{\mathcal{P}}(addL_a(ad)) \leq d_{\mathcal{P}}(addL_b(ad))$.

Therefore, from $d_{\mathcal{P}}(addL_a(ad)) \geq d_{\mathcal{P}}(addL_b(ad))$ and $d_{\mathcal{P}}(addL_a(ad)) \leq d_{\mathcal{P}}(addL_b(ad))$, we can conclude $d_{\mathcal{P}}(addL_a(ad)) = d_{\mathcal{P}}(addL_b(ad))$. Thus, $addL_a$ and $addL_b$ induce an equally long shortest solution for I . \square

Let ad and ad' be two ADs, (x, y) be a transition of ad and k be an action label of ad . Then, for all node names d, m, a, e, n, b that are neither used in ad nor in ad' such that d, m, a as well as e, n, b are pairwise different, the operations $addXor_{x,y,d,m,a,k}$ and $addXor_{x,y,e,n,b,k}$ induce an equally long shortest solution for the model repair problem instance $I = (P_{\subseteq}(\mathcal{L}_{AD}, O_{AD}, ad'), ad)$. The reason for this is similar to the reason for the equivalences in the context of action-addition operations. Node names do not influence the traces in the semantics of ADs. For every shortest change sequence t that repairs $addXor_{x,y,d,m,a,k}(ad)$, it is possible to construct a change sequence u of the same length $|u| = |t|$ that repairs $addXor_{x,y,e,n,b,k}(ad)$. The ADs $addXor_{x,y,d,m,a,k}(ad) \triangleright t$ and $addXor_{x,y,e,n,b,k}(ad) \triangleright u$ solely differ in the naming of their nodes. The AD $addXor_{x,y,d,m,a,k}(ad) \triangleright t$ can be obtained from the AD $addXor_{x,y,e,n,b,k}(ad) \triangleright u$ by renaming the node d (if it exists) to e , renaming the node m (if it exists) to n , and renaming the node a (if it exists) to b . Analogously, the ad $addXor_{x,y,e,n,b,k} \triangleright u$ can be obtained from the AD $addXor_{x,y,d,m,a,k} \triangleright t$ by renaming the nodes. As the names of nodes do not influence the traces in the semantics of ADs, the ADs have equal semantics.

For analogous reasons, and-fragment-addition operations (respectively cyclic-fragment-addition operations) adding and-fragments (respectively cyclic fragments) containing actions with the same label between the same nodes are equivalent.

Proposition 8.30. *Let $ad = (L, N, t, AND, XOR, C, T, l)$ and $ad' = (L', N', t', AND', XOR', C', T', l')$ be two ADs, let $(x, y) \in T$ be a transition of ad , and let $k \in L$ be an action label of ad . Further, let $I = (\mathcal{P}, ad)$ be an instance of the model repair problem $\mathcal{P} = \mathcal{P}_{\subseteq}(\mathcal{L}_{AD}, O_{AD}, ad')$. Then, for all node names $d, m, a, e, n, b \in U_N \setminus N$ with $d \neq m$, $d \neq a$, $m \neq a$, $e \neq n$, $e \neq b$, $n \neq b$, the following statements hold:*

- $addXor_{x,y,d,m,a,k} \sim_I addXor_{x,y,e,n,b,k}$.
- $addAnd_{x,y,d,m,a,k} \sim_I addAnd_{x,y,e,n,b,k}$.
- $addC_{x,y,d,m,a,k} \sim_I addC_{x,y,e,n,b,k}$.

Proof. The proofs are analogous to the proof for Proposition 8.28. \square

8.5.2 Activity Diagram Generalization Repair

This section presents a change operation partitioning for the AD modeling language and the AD generalization repair problem.

Let ad and ad' be two ADs, (a, b) be a transition of ad and c be an action label of ad . Then, for all node names d, e that are not used in ad , the operations $addA_{a,b,d,c}$ and $addA_{a,b,e,c}$ induce an equally long shortest solution for the model repair problem instance $I = (\mathcal{P}_{\supseteq}(\mathcal{L}_{AD}, O_{AD}, ad'), ad)$. The reason for this is similar to the reason for the equivalence in the context of AD refinement repair problems. Action node names do not influence the traces in the semantics of ADs. For every shortest change sequence t that repairs $addA_{a,b,d,c}(ad)$, it is possible to construct a change sequence u of the same length $|u| = |t|$ that repairs $addA_{a,b,e,c}(ad)$. The change sequence u is constructed in the same way as for AD refinement repair problems (cf. Section 8.5.1). Thus, the ADs $addA_{a,b,d,c}(ad) \triangleright t$ and $addA_{a,b,e,c}(ad) \triangleright u$ have equal semantics. As $addA_{a,b,d,c}(ad) \triangleright t$ is by assumption a generalization of ad' , the AD $addA_{a,b,e,c}(ad) \triangleright u$ is also a generalization of the AD ad' .

Proposition 8.31. *Let $ad = (L, N, t, AND, XOR, C, T, l)$ and $ad' = (L', N', t', AND', XOR', C', T', l')$ be two ADs, $(a, b) \in T$ be a transition of ad , and $c \in L$ be an action label of ad . Further, let $I = (\mathcal{P}, ad)$ be an instance of the model repair problem $\mathcal{P} = \mathcal{P}_{\supseteq}(\mathcal{L}_{AD}, O_{AD}, ad')$. Then, for all names $d, e \in U_N \setminus N$, it holds $addA_{a,b,d,c} \sim_I addA_{a,b,e,c}$.*

Proof. The proof is analogous to the proof for Proposition 8.28. \square

Let ad and ad' be two ADs. Then, for all action labels a, b that are neither used in ad nor in ad' , the operations $addL_a$ and $addL_b$ induce an equally long shortest solution for the model repair problem instance $I = (\mathcal{P}_{\supseteq}(\mathcal{L}_{AD}, O_{AD}, ad'), ad)$. For every shortest change sequence t that repairs $addL_a(ad)$, it is possible to construct a change sequence u of the same length $|u| = |t|$ such that u repairs $addL_b(ad)$. The change sequence u is constructed in the same way as for AD refinement repair problems (cf. Section 8.5.1).

For every trace in the semantics of $\text{add}L_a(ad) \triangleright t$, the trace obtained from replacing each occurrence of a by b and replacing each occurrence of b by a in the trace is a trace in the semantics of $\text{add}L_b(ad) \triangleright u$ and vice versa. As the AD ad' does neither use the action label a nor the action label b , its semantics does not contain any trace that uses one of the labels. If there did exist a trace in the semantics of ad' that is not a trace in the semantics of $\text{add}L_b(ad) \triangleright u$, then this trace would not be an element in the semantics of $\text{add}L_a(ad) \triangleright t$. This would contradict that $\text{add}L_a(ad) \triangleright t$ is a generalization of ad' .

Proposition 8.32. *Let $ad = (L, N, t, AND, XOR, C, T, l)$ and $ad' = (L', N', t', AND', XOR', C', T', l')$ be two ADs. Further, let $I = (\mathcal{P}, ad)$ be an instance of the model repair problem $\mathcal{P} = \mathcal{P}_{\supseteq}(\mathcal{L}_{AD}, O_{AD}, ad')$. Then, for all labels $a, b \in U_N \setminus (L \cup L')$, it holds that $\text{add}L_a \sim_I \text{add}L_b$.*

Proof. (Sketch.) Let ad , ad' , \mathcal{P} , and I be given as above. Let $a, b \in U_N \setminus (L \cup L')$ be two action labels that are neither used in ad nor in ad' . Assume $t \in O_{AD}^*$ is a shortest change sequence that repairs $\text{add}L_a(ad)$ towards satisfying $P_{\supseteq}(\mathcal{L}_{AD}, ad')$. We define the change sequence u of length $|t|$ as the change sequence obtained from the sequence t via replacing each change operation in t affecting the action label a with a change operation of the same type that affects the label b , instead, and vice versa. A change operation inserting an action with label a between two succeeding nodes, for instance, is replaced by a change operation inserting the same action with label b between the same nodes.

As $\text{add}L_a$ and $\text{add}L_b$ are applicable to ad , the AD ad cannot contain any actions labeled a or b . Therefore, by construction of the change sequence u , the AD $\text{add}L_a(ad) \triangleright t$ can be obtained from the AD $\text{add}L_b(ad) \triangleright u$ via changing the label of each action in $\text{add}L_b(ad) \triangleright u$ that is labeled with a to b and changing the label of each action that is labeled with b to a . Vice versa, the AD $\text{add}L_b(ad) \triangleright u$ can be obtained from the AD $\text{add}L_a(ad) \triangleright t$ via applying the same relabeling.

Therefore, if τ is a trace in the semantics of $\text{add}L_a(ad) \triangleright t$, then the trace τ' obtained from replacing all occurrences of a by b and all occurrences of b by a is a trace in the semantics of $\text{add}L_b(ad) \triangleright u$ and vice versa. Suppose towards a contradiction that $\text{add}L_b(ad) \triangleright u$ is not a generalization of ad' . Then, there exists a trace τ in the semantics of ad' that is no element of the semantics of $\text{add}L_b(ad) \triangleright u$. As $\text{add}L_a(ad \triangleright t)$ is by assumption a generalization of ad' , the trace τ is also an element of the semantics of $\text{add}L_a(ad \triangleright t)$. From the above, it follows that the trace τ' obtained from replacing all occurrences of a by b and all occurrences of b by a in τ is a trace in the semantics of $\text{add}L_b(ad) \triangleright u$. As τ is an element of the semantics of ad' and the action labels a and b do not exist in ad' , the trace τ' does neither contain a nor b . Therefore, $\tau = \tau'$. However, this contradicts that τ is not an element of the semantics of $\text{add}L_b(ad) \triangleright u$ because τ' is an element of the semantics of $\text{add}L_b(ad) \triangleright u$ and $\tau = \tau'$. From the above, we can conclude that $d_{\mathcal{P}}(\text{add}L_a(ad)) \geq d_{\mathcal{P}}(\text{add}L_b(ad))$.

Analogously, we can show that for every shortest change sequence $t \in O_{AD}^*$ that repairs $\text{add}L_b(ad)$ towards satisfying $P_{\supseteq}(\mathcal{L}_{AD}, ad')$, there exists a change sequence $u \in O_{AD}^*$ with

$|u| = |t|$ such that u repairs $\text{add}L_a(ad)$ towards satisfying $P_{\supseteq}(\mathcal{L}_{AD}, ad')$. From this, we can conclude that $d_{\mathcal{P}}(\text{add}L_a(ad)) \leq d_{\mathcal{P}}(\text{add}L_b(ad))$.

Therefore, from $d_{\mathcal{P}}(\text{add}L_a(ad)) \geq d_{\mathcal{P}}(\text{add}L_b(ad))$ and $d_{\mathcal{P}}(\text{add}L_a(ad)) \leq d_{\mathcal{P}}(\text{add}L_b(ad))$, we can conclude $d_{\mathcal{P}}(\text{add}L_a(ad)) = d_{\mathcal{P}}(\text{add}L_b(ad))$. Thus, $\text{add}L_a$ and $\text{add}L_b$ induce an equally long shortest solution for I . \square

For similar reasons as for AD refinement repair problems, xor-fragment-addition operations (respectively and-fragment-addition operations and cyclic-fragment addition operations) adding xor-fragments (respectively and-fragments and cyclic-fragments) containing actions with the same label between the same nodes are equivalent.

Proposition 8.33. *Let $ad = (L, N, t, AND, XOR, C, T, l)$ and $ad' = (L', N', t', AND', XOR', C', T', l')$ be two ADs, let $(x, y) \in T$ be a transition of ad , and let $k \in L$ be an action label of ad . Further, let $I = (\mathcal{P}, ad)$ be an instance of the model repair problem $\mathcal{P} = \mathcal{P}_{\supseteq}(\mathcal{L}_{AD}, O_{AD}, ad')$. Then, for all node names $d, m, a, e, n, b \in U_N \setminus N$ with $d \neq m$, $d \neq a$, $m \neq a$, $e \neq n$, $e \neq b$, $n \neq b$, the following statements hold:*

- $\text{addXor}_{x,y,d,m,a,k} \sim_I \text{addXor}_{x,y,e,n,b,k}$.
- $\text{addAnd}_{x,y,d,m,a,k} \sim_I \text{addAnd}_{x,y,e,n,b,k}$.
- $\text{addC}_{x,y,d,m,a,k} \sim_I \text{addC}_{x,y,e,n,b,k}$.

Proof. The proofs are analogous to the proof for Proposition 8.28. \square

8.5.3 Activity Diagram Refactoring Repair

This section presents a change operation partitioning for the AD modeling language and the AD refactoring repair problem.

Let ad and ad' be two ADs, $(a, b) \in T$ be a transition of ad and c be an action label of ad . Then, for all node names d, e that are not used in ad , the operations $\text{add}A_{a,b,d,c}$ and $\text{add}A_{a,b,e,c}$ induce an equally long shortest solution for the model repair problem instance $I = (\mathcal{P}_{=}, (\mathcal{L}_{AD}, O_{AD}, ad'), ad)$. The reason for this is similar to the reason for the equivalence in the contexts of AD refinement model repair problems (cf. Section 8.5.1) and AD generalization model repair problems (cf. Section 8.5.2). Action node names do not influence the traces in the semantics of ADs.

Proposition 8.34. *Let $ad = (L, N, t, AND, XOR, C, T, l)$ and $ad' = (L', N', t', AND', XOR', C', T', l')$ be two ADs, $(a, b) \in T$ be a transition of ad , and $c \in L$ be an action label of ad . Further, let $I = (\mathcal{P}, ad)$ be an instance of the model repair problem $\mathcal{P} = \mathcal{P}_{=}, (\mathcal{L}_{AD}, O_{AD}, ad')$. Then, for all names $d, e \in U_N \setminus N$, it holds $\text{add}A_{a,b,d,c} \sim_I \text{add}A_{a,b,e,c}$.*

Proof. The proof is analogous to the proof for Proposition 8.28. \square

Let ad and ad' be two ADs. Then, for all action labels a, b that are neither used in ad nor in ad' , the operations $addL_a$ and $addL_b$ induce an equally long shortest solution for the model repair problem instance $I = (\mathcal{P}_=(\mathcal{L}_{AD}, O_{AD}, ad'), ad)$. The reason is a combination of the reasons for the equivalences of the operations in the contexts of AD refinement repair problems (cf. Section 8.5.1) and AD generalization repair problems (cf. Section 8.5.2).

Proposition 8.35. *Let $ad = (L, N, t, AND, XOR, C, T, l)$ and $ad' = (L', N', t', AND', XOR', C', T', l')$ be two ADs. Further, let $I = (\mathcal{P}, ad)$ be an instance of the model repair problem $\mathcal{P} = \mathcal{P}_=(\mathcal{L}_{AD}, O_{AD}, ad')$. Then, for all labels $a, b \in U_N \setminus (L \cup L')$, it holds that $addL_a \sim_I addL_b$.*

Proof. The proof is a combination of the proofs for Proposition 8.29 and Proposition 8.32. \square

For the same reasons as for AD refinement and generalization repair problems, xor-fragment-addition operations (respectively and-fragment-addition operations and cyclic-fragment addition operations) adding xor-fragments (respectively and-fragments and cyclic-fragments) containing actions with the same label between the same nodes induce equally long shortest solutions.

Proposition 8.36. *Let $ad = (L, N, t, AND, XOR, C, T, l)$ and $ad' = (L', N', t', AND', XOR', C', T', l')$ be two ADs, let $(x, y) \in T$ be a transition of ad , and let $k \in L$ be an action label of ad . Further, let $I = (\mathcal{P}, ad)$ be an instance of the model repair problem $\mathcal{P} = \mathcal{P}_=(\mathcal{L}_{AD}, O_{AD}, ad')$. Then, for all node names $d, m, a, e, n, b \in U_N \setminus N$ with $d \neq m$, $d \neq a$, $m \neq a$, $e \neq n$, $e \neq b$, $n \neq b$, the following statements hold:*

- $addXor_{x,y,d,m,a,k} \sim_I addXor_{x,y,e,n,b,k}$.
- $addAnd_{x,y,d,m,a,k} \sim_I addAnd_{x,y,e,n,b,k}$.
- $addC_{x,y,d,m,a,k} \sim_I addC_{x,y,e,n,b,k}$.

Proof. The proofs are analogous to the proof for Proposition 8.28. \square

8.5.4 Repair-Representative Function and Example Applications

From the argumentations in Section 8.5.1, Section 8.5.2, and Section 8.5.3, we can conclude that it is possible to use the same repair-representative function for the AD refinement, generalization, and refactoring repair problems using the same models. For each $ad \in M_{AD}$, we can construct the repair-representative function $\mathcal{R}_{ad} : M_{AD} \rightarrow \wp_{fin}(O_{AD})$ for each of the AD model repair problems as follows: The function \mathcal{R}_{ad} maps each AD to the set containing

- all label-deletion, action-deletion, xor-fragment-deletion, and-fragment-deletion, cyclic-fragment-deletion, fragment-branch-insertion, and fragment-branch-deletion operations that are applicable to the AD,
- all label-addition, action-insertion, xor-fragment-insertion, and-fragment-insertion, and cyclic-fragment-insertion operations that have not been partitioned in Section 8.5.1, Section 8.5.2, and Section 8.5.3 and are applicable to the AD,
- exactly one arbitrary but fixed change operation of each equivalence class of the label-addition, action-insertion, cyclic-fragment-insertion, xor-fragment-insertion, and and-fragment-insertion operations identified in Section 8.5.1, Section 8.5.2, and Section 8.5.3.

For example, Figure 8.10 depicts an excerpt of the change sequence search tree for computing a shortest solution for an AD refactoring repair problem instance. The AD *ad1* is the model of the instance. The property of the refactoring repair problem contains all ADs that are refactorings of the AD *ad2*.

8.5.5 Implementation and Experiments

We implemented the model repair algorithms for the AD modeling language to perform experimental evaluations. The implementation is written in Java and uses the semantic differencing operator implementation presented in Section 6.3.

We performed experimental evaluations by executing the algorithms for computing shortest solutions for AD refinement repair problem instances using the eight example ADs presented in Appendix D. The purpose of the experiments is twofold. The first purpose is testing whether computing shortest repairing sequences is feasible for the example AD refinement repair problem instances. The second purpose is comparing the performances of the different algorithms in the context of the example AD refinement repair problems.

Heuristic for Witness Computation

The computational complexity of semantic differencing of ADs is high and the algorithms often execute the semantic differencing operator. To achieve performance improvements, we implemented a heuristic for fast witness computation.

The heuristic relies on quickly computing short diff witnesses contained in the semantic difference from an AD *ad* to an AD *ad'*. It translates the input ADs to their associated NFAs and searches for a word accepted by $nfa(ad)$ that is not accepted by $nfa(ad')$. Such a word is a trace contained in the semantic difference from *ad* to *ad'*. The heuristic solely considers words with a length smaller than or equal to an arbitrary but fixed upper bound. For the evaluation, we solely consider words of length at most eight.

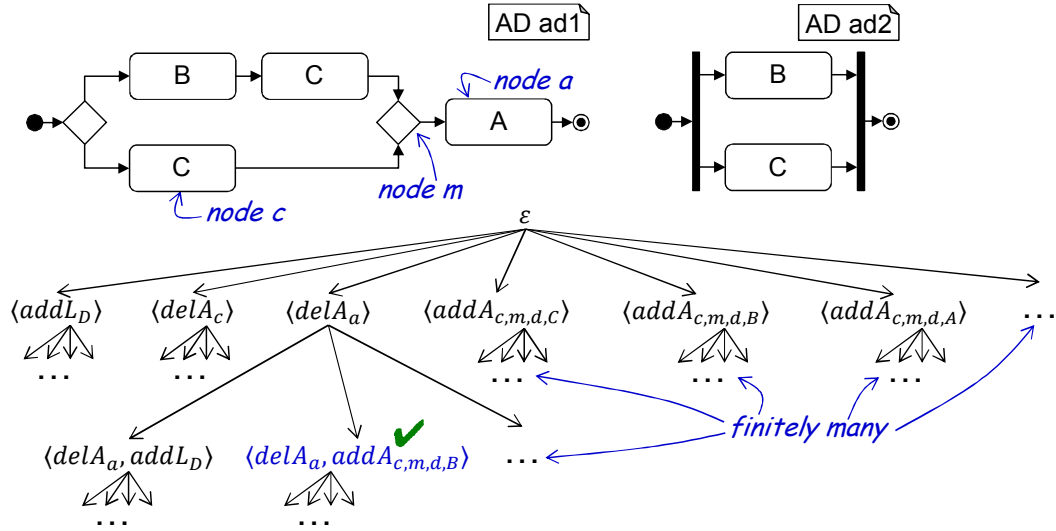


Figure 8.10: Two simple ADs and an excerpt of the change sequence search tree for computing a shortest solution for an AD refactoring repair problem.

Algorithm 17 is the heuristic for the witness computation. It takes two NFAs A and B and a natural number $n \in \mathbb{N}$ as inputs. It outputs a word $w \in \mathcal{L}_*(A) \setminus \mathcal{L}_*(B)$ if there exists a sequence s of consecutive transitions in A with $|s| \leq n$ starting in the initial state such that the word encoded by s is not contained in the language recognized by B . Otherwise, the algorithm outputs the special symbol nil . In Algorithm 17, $lt(w)$ denotes the last element of a finite sequence w . The algorithm executes a DFID search in the NFA A for finding a word accepted by A of length at most n that is not accepted by the NFA B . To this effect, it iterates over all natural numbers i that are smaller than or equal to n in increasing value (ll. 1-26). In each iteration (ll. 2-25), the algorithm checks whether there exists a word of length at most i that is accepted by A and not accepted by B . In the loop, the algorithm initializes the variable can as an empty stack of transitions of A (l. 2). Afterwards, the algorithm pushes the sequences of length one containing exactly the transitions starting in the initial state on the stack can (ll. 3-5). While the stack can is not empty (l. 6), the algorithm checks the explored sequences of transitions that have been pushed on the stack cur (ll. 7-24). Therefore, the algorithm pops the most recently explored transition sequence from the stack on stores the result in the variable cur (l. 7). If the length of the transition sequence is equal to the current length bound i (l. 8), the algorithm checks whether the transition sequence is a run on a word that is accepted by A and not accepted by B (ll. 9-18). First, it fetches the last transition contained in the transition sequence (l. 9). If the transition sequence ends in a final state of the NFA A (l. 10), then the NFA accepts the word corresponding to the

Algorithm 17 Checking whether the NFA A accepts a word of length at most n that is not accepted by the NFA B .

Input: Two NFAs $A = (S, \Sigma, \delta, i, F)$ and $B = (S', \Sigma', \delta', i', F')$ and an integer $n \in \mathbb{N}$.

Output: A word $w \in \mathcal{L}_*(A) \setminus \mathcal{L}_*(B)$ if there exists a sequence s of consecutive transitions in A with $|s| \leq n$ starting in the initial state such that the word encoded by s is not contained in the language recognized by B . Otherwise, the special symbol *nil*.

```

1: for all  $i \in \mathbb{N}$  with  $i \leq n$  in increasing value do
2:   define  $can$  as empty stack of  $\delta^*$ 
3:   for all  $t \in \{(u, a, v) \in \delta \mid u = i\}$  do
4:      $can.push(t)$ 
5:   end for
6:   while  $can$  not empty do
7:     define  $cur \leftarrow can.pop()$  as element of  $\delta^*$ 
8:     if  $|cur| = i$  then
9:       define  $(u, a, v) \leftarrow lt(cur)$  as element of  $\delta$ 
10:      if  $v \in F$  then
11:        define  $w \leftarrow \varepsilon$  as empty word
12:        for all  $(u, a, v) \in cur$  in ascending order do
13:           $w \leftarrow w \& a$ 
14:        end for
15:        if  $w \notin \mathcal{L}_*(B)$  then
16:          return  $w$ 
17:        end if
18:      end if
19:    else
20:      define  $(u, a, v) \leftarrow lt(cur)$  as element of  $\delta$ 
21:      for all  $t \in \{(w, b, x) \in \delta \mid w = v\}$  do
22:         $can.push(cur \& t)$ 
23:      end for
24:    end if
25:  end while
26: end for
27: return nil

```

transition sequence. In this case, the algorithm checks whether the NFA B also accepts this word (ll. 11-17). To this effect, the algorithm constructs the word w corresponding to the transition sequence (ll. 11-14). Afterwards, it checks whether the NFA B accepts the word w (l. 15). If B does not accept the word, then the algorithm returns the word w as a diff witness (l. 16). If the currently processed sequence is smaller than the current length bound i , the algorithm proceeds by prolonging the currently processed sequence

cur with the transitions starting in the target state of the last transition contained in *cur* (ll. 20-23). For this task, it fetches the last transition contained in the sequence *cur* (l. 20). Then, it iterates over all transitions starting in the target state of the last transition (l. 21). The concatenation of each of these transitions to the currently processed sequence is pushed on the stack of candidates *can* (l. 22). If no witness of length at most n is found, the algorithm returns the special symbol *nil* (l. 27).

In the algorithms for computing repairing sequences, we first use the heuristic for checking whether one of the ADs is a refinement of the other AD by using the NFAs associated to the ADs as inputs. If the heuristic does not find a diff witness, we use the semantic differencing operator introduced in Section 6.3 for checking whether the AD is a refinement of the other AD. Whether an AG is consistent can be checked by checking whether the language accepted by the NFA associated to the AG is empty.

Search Space Restrictions

For pragmatic reasons, to increase the performance of the algorithms, we further restrict the search space as follows.

If an operation would add an action node such that the word constructed by concatenating the label of this node with the label of a next successive action node does not appear as the word of two successive actions or as the word obtained from concatenating the labels of two actions contained in different branches of the same and-fragment in the model of the repair problem, then we do not consider this operation. Vice versa, if the word constructed by concatenating the label of a predecessor action node with the label of the added action does not appear as the concatenation of the labels of two successive actions or as the word obtained from concatenating the labels of two actions contained in different branches of the same and-fragment in the repair problem's model, then we do not consider the operation adding the action. These restrictions are appropriate because otherwise there could be traces in the semantics of the AD that result from the existence of the added action and are no traces of the repair problem's model.

Similarly, we do not consider operations adding actions into branches of and-fragments where the concatenation of the action label with the label of an action of another branch of the same fragment does not appear as the concatenation of the labels of two successive actions or as the word obtained from concatenating the labels of two actions contained in different branches of the same and-fragment in the repair problem's model.

We do only consider fragment-addition operations adding a fragment containing an action that is also contained in a fragment of the same type of the repair problem's model. This restriction avoids unnecessary branching and decreases the resulting model's complexity regarding the depths of nested fragments.

We do only consider fragment-addition operations adding a fragment containing an action where the number of fragments including the added action in the resulting model is not greater than the greatest number of fragments in the repair problem's model

containing an action with the same label. This restriction avoids unnecessary complexity regarding the depths of nested fragments.

We do not consider any label-addition operations adding labels that are not used in the repair problem’s model because no trace of the repair problem’s model contains this action label.

If the repair problem’s model contains actions having a label that is not used in the currently processed model, then we solely consider a label-addition operation adding one of the labels. This restriction is reasonable because the repaired model must usually contain an action labeled with the unused label because the existence of an action with the label in the repair problem’s model causes the existence of a trace in the semantics of the model that uses the label. If the addition of the label is not necessary because the repaired model does not contain an action using the label, then the operation can easily be removed from the computed repairing sequence. Similarly, if the currently processed model contains actions labeled with a label not used in the repair problem’s model, then we solely consider an operation for deleting one of the actions. This restriction is reasonable because the repaired model must not contain actions labeled with a label not used in the repair problem’s model because the existence of the action causes a trace in the semantics of the model that is not a trace in the semantics of the repair problem’s model. We do not consider any label-deletion operations as these delay the repair.

Experiments

We performed experimental evaluations with seven example ADs. Appendix D presents the example ADs in detail. For each pair of thematically related ADs where one of the ADs is not a refinement of the other AD, we executed the algorithms presented in Section 7.5 to compute solutions for the corresponding refinement repair problems. For testing whether intermediately computed ADs satisfy the refines property, we used the semantic differencing operator including the heuristic as presented in Section 8.5.5.

All experiments were executed on a laptop computer with an Intel Core i7-8650U CPU @ 1.90GHz processor, 16GB RAM, and a Samsung PM981 512GB SSD hard drive using Windows 10 and Java 1.8.0_192.

Figure 8.11 summarizes the computation times of the algorithms. During two experiments, Java reported an `OutOfMemoryError`, which is indicated by OOM in Figure 3.4. For instance, Algorithm 2 took 268ms to compute a solution for the model repair problem instance $(\mathcal{P}_{\subseteq}(\mathcal{L}_{SD}, O_{SD}, \text{hire2}), \text{hire1})$, *i.e.*, to compute a sequence $t \in O_{PA}$ such that $\emptyset \neq \llbracket \text{hire1} \triangleright t \rrbracket^{PA} \subseteq \llbracket \text{hire2} \rrbracket^{PA}$. In the cases where the algorithms terminated, the computation times range from 211ms to 8m 12s. Algorithm 2 successfully computed solutions for all instances in at most 6m 8s. Figure 8.12 depicts the solutions computed by Algorithm 2 for the input ADs.

Algorithm 5 produced an `OutOfMemoryError` for two of the inputs. The other algorithms computed solutions for all inputs. Algorithm 2 performed better than Algo-

Model	Problem	Time				
		Algo. 2	Algo. 3	Algo. 4	Algo 5	Algo. 6
hire1	hire2	268ms	417ms	360ms	246ms	454ms
hire2	hire1	211ms	335ms	408ms	270ms	373ms
claim1	claim2	3m 44s	4m 44s	4m 52s	OOM	4m 13s
claim1	claim3	1507ms	3282ms	2485ms	3081ms	1451ms
claim1	claim4	6m 8s	8m 12s	7m 12s	OOM	6m 54s
claim2	claim1	280ms	450ms	262ms	266ms	271ms
claim2	claim4	1079ms	2093ms	847ms	1143ms	854ms
claim3	claim1	631ms	1095ms	695ms	1149ms	656ms
claim3	claim2	1796ms	2615ms	1564ms	2854ms	3480ms
claim3	claim4	47s	52s	8454ms	13s	12s
claim4	claim1	1511ms	2205ms	1372ms	2392ms	2269ms
claim4	claim2	4m 30s	4m 25s	3m 13s	4m 36s	3m 19s
claim4	claim3	7890ms	6272ms	1825ms	3341ms	2342ms
ad1	ad2	309ms	340ms	324ms	497ms	319ms
ad2	ad1	514ms	717ms	317ms	557ms	334ms

Figure 8.11: The execution times of Algorithm 2 - Algorithm 6 when given the AD refinement repair problem instances as inputs.

rithm 3 in all but two cases. Algorithm 2 performed better than Algorithm 4 in eight out of 15 cases. In no case Algorithm 5 or Algorithm 6 performed significantly better than all of the other algorithms.

The algorithms handle the small examples, where only short repairing sequences are required, sufficiently quick. However, the algorithms do not scale well for inputs where many changes are required. For one instance, the fastest algorithm needed 6m 8s to compute a solution. In general, the running times of the algorithms are exponential in the lengths of the shortest solutions (cf. Section 7.5.1).

Model	Problem	Solution
hire1	hire2	$addL_{Assign\ Keys}, addA_{J1,MI,n1,Assign\ Keys}$
hire2	hire1	$delA_{AK}$
claim1	claim2	$addL_{Close\ Claim}, addL_{Update\ Cust. \ Record}, delA_{Acc}, delA_{RAD}, addA_{D3,M3,n1,Send\ Declinature},$ $addA_{M3,M2,n2,Update\ Cust. \ Record}, addA_{n2,M2,n3,Close\ Claim}, delFB_{D1,M1}$
claim1	claim3	$addL_{Close\ Claim}, addL_{Update\ Cust. \ Record}, addA_{M3,M2,n1,Update\ Cust. \ Record}, addA_{n1,M2,n2,Close\ Claim}$
claim1	claim4	$addL_{Close\ Claim}, addL_{Update\ Cust. \ Record}, delA_{SC}, addA_{M3,M2,n1,Update\ Cust. \ Record},$ $addA_{n1,M2,n2,Close\ Claim}, addA_{J1,M2,n3,Send\ Confirmation}$
claim2	claim1	$addL_{Call\ Customer}, addL_{Retrieve\ Add. \ Data}, delA_{Acc}, delA_{UCR}$
claim2	claim4	$addL_{Call\ Customer}, addL_{Retrieve\ Add. \ Data}, delA_{SC}, addA_{J1,M2,n1,Send\ Confirmation}$
claim3	claim1	$delA_{CIC}, delA_{UCR}, delA_{CC1}$
claim3	claim2	$delA_{CC1}, delA_{Acc}, delA_{RAD}, addA_{D3,M3,n1,Send\ Declinature}, delFB_{D1,M1}$
claim3	claim4	$delA_{CC1}, delA_{SC}, addA_{J2,M2,n1,Send\ Confirmation}$
claim4	claim1	$delA_{C1C}, delA_{UCR}, delA_{A1}, addA_{J2,M2,n1,Payout}$
claim4	claim2	$delA_{CC}, delA_{RAD}, delA_{A1}, addA_{D3,M3,n1,Send\ Declinature}, addA_{J2,M2,n2,Payout}, delFB_{D1,M1}$
claim4	claim3	$delA_{A1}, addA_{J2,M2,n1,Payout}$
ad1	ad2	$delA_A, addA_{C,M1,n1,B}$
ad2	ad1	$addL_A, delA_B, addA_{J1,f,n1,A}$

Figure 8.12: The solutions computed by Algorithm 2 for the AD refinement repair problem instances.

Part IV

Epilogue

Chapter 9

Conclusion and Future Work

This chapter concludes the thesis. Section 9.1 summarizes the main results of this thesis. Afterwards, Section 9.2 presents recommendations for future work.

9.1 Summary and Main Results

Figure 9.1 overviews the formal frameworks and the most important concepts that have been presented in this thesis.

Chapter 2 presented a general framework for defining modeling languages by unifying previous works. Each modeling language consists of a set of models, a semantic domain, and a semantic mapping. The set of models is an abstract representation of the modeling language's syntax. The semantic domain is a set containing elements of a well-understood mathematical structure (*e.g.*, sequences, sets, functions). The elements of the semantic domain are mathematical abstractions of possible realizations of models. The semantic mapping defines the meaning of models. It relates each model to a set of elements from the semantic domain. Each element in the semantics of a model is interpretable to be a possible realization of the model. The semantic difference from a model to another model are all elements in the semantics of the former model that are not elements in the semantics of the latter model. Each such element is called a diff witness from the former model to the latter model. A semantic differencing operator for a modeling language is an automatic procedure that takes two models as input and outputs a finite set of diff witnesses from one of the models to other model. A model is a refinement of another model, if the semantic difference from the former model to the latter model is empty. Analogously, a model is a generalization of another model, if each element in the semantics of the latter model is also an element of the semantics of the former model. Two models are refactorings of each other, if their semantics are equal. Model evolution possibilities are described by change operations. Each change operation is a partial function mapping models to models. Sets of change operations are called change operation suites. Change operation suites can be used to describe all evolution possibilities of the models of a modeling language. A change operation suite is complete for a modeling language, if every model of the language can be transformed to any other model of the language via applying change operations from the change operation suite.

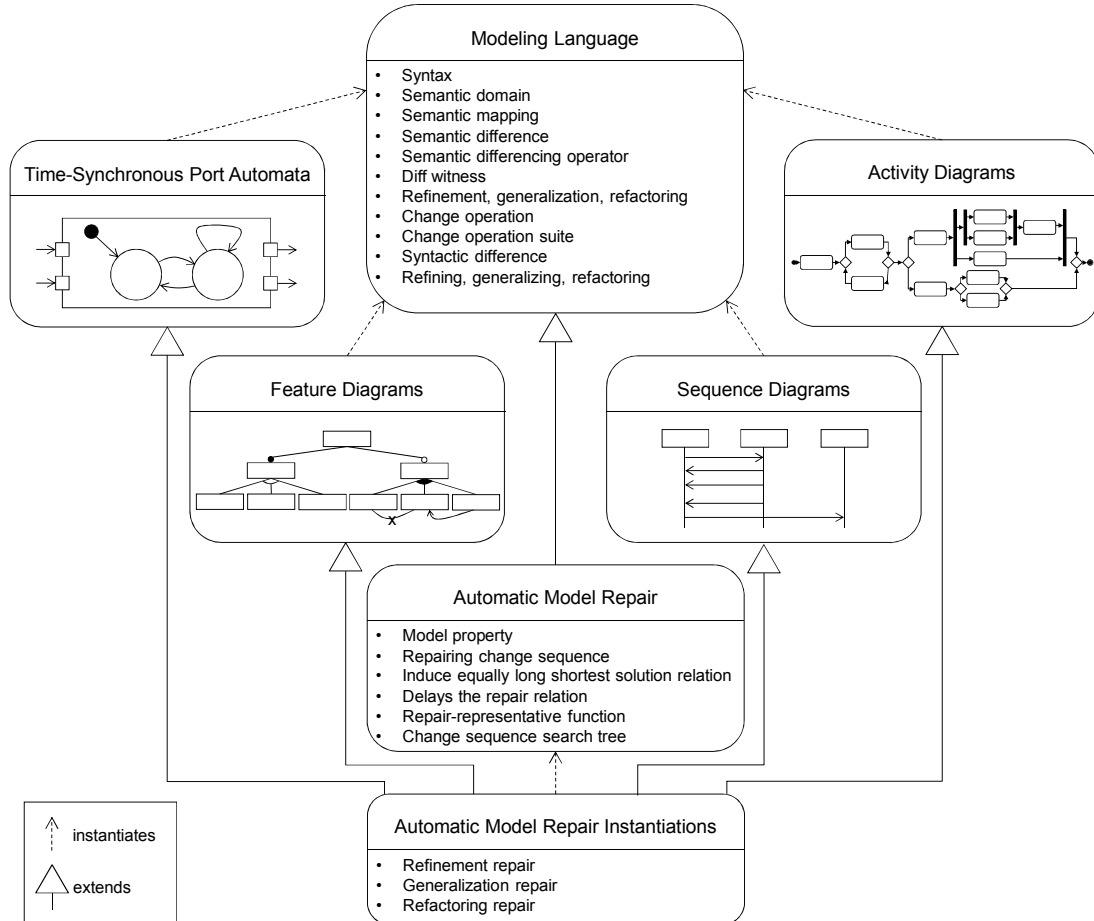


Figure 9.1: Overview of the framework developed in this thesis.

A syntactic difference from a model to another model is a sequence of change operations where the application of the sequence's change operations in order to the former model yields the latter model. A change operation is refining if its application to a model is guaranteed to yield a model that is a refinement of the former model. Generalizing and refactoring change operations are defined analogously. The following four chapters instantiated the general framework for with four concrete modeling languages.

Chapter 3 presented the TSPA modeling language. The models are automata describing the behaviors of reactive components participating in interactive systems. The transitions of a TSPA describe the reactions of the automaton in response to receiving messages via its input channels in terms of its internal state changes and the messages emitted by the automaton via its output channels. The semantic domain contains all

possible communication histories. Each communication history describes infinite streams of messages emitted via output channels in reaction to the receipt of infinite streams of messages received via input channels. The semantic mapping maps each automaton to the set of all communication histories that it describes. The semantic differencing operator is based on a translation to BAs. Each infinite word accepted by the BA resulting from translating a TSPA encodes a communication history of the TSPA. Thus, each word accepted by the BA resulting from translating a TSPA that is not a word accepted by the BA resulting from translating another TSPA encodes a communication history of the former TSPA that is not a communication history of the latter TSPA. This enables reducing semantic differencing of TSPAs to language inclusion checking for BAs. Chapter 3 further presented a complete change operation suite for the TSPA modeling language. For each change operation, it was analyzed whether it is refining, generalizing, or refactoring.

Chapter 4 presented a FD modeling language. The set of models is the set of all well-formed FDs. The semantic domain is the set of all possible feature configurations. Each feature configuration is a finite set of features. The semantic mapping is based on an open-world assumption and differs from the usual semantic mapping for FDs. Each FD is mapped to the set of all configurations that are valid in the FD. With the semantics presented in this thesis, the configurations of a FD may contain features that are not used in the FD. Features that are not used in a feature diagram are considered to be unconstrained by the FD. This implies that the semantics of each FD is an infinite set when assuming an infinite universe of features. The main enabler for the semantic differencing operator for the FD modeling language is the following observation: A FD is a refinement of another FD iff each configuration that is valid in the former FD and solely contains features that are used in at least one of the two FDs is also valid in the latter FD. As each FD only contains a finite set of features, it suffices to check whether one of finitely many configuration is valid in the one FD and not valid in the other FD. It is possible to provide an implementation of the semantic differencing procedure via a reduction to the satisfiability problem for propositional formulas. Chapter 4 also presented a complete change operation suite for the FD modeling language. For each change operation, it was analyzed whether it is refining, generalizing, or refactoring.

Chapter 5 presented a SD modeling language. The set of models is the set of all well-formed SDs. The semantic domain is the set of all possible system runs. Each system run describes a finite trace of interactions between objects. Each interaction consists of a source object, a target object, and an action. The semantic mapping maps each SD to the set of all systems runs that are valid in the SD. Systems runs that are valid in a SD may contain interactions using actions between objects that are not used in the SD. If the set of possible objects or the set of possible actions is infinite, then there exists an SD where the set of all traces of all valid system runs of the SD is not representable by a regular language. Nevertheless, the semantic differencing operator is based on a reduction to language inclusion between NFAs. The main enabler for the

semantic differencing operator is the observation that it suffices to search a regular set of traces of systems runs for a trace of a system run that is valid in one SD and not valid in another SD to conclude whether refinement holds. The semantic differencing operator takes two SDs as input, constructs two NFAs and checks whether the language recognized by one of the NFAs is a subset of the language recognized by the other NFA. In case language inclusion holds, the former sequence diagram is a refinement of the other SD. Otherwise, there exists a system run that is valid in the former SD and not valid in the other SD. Chapter 5 also presented a complete change operation suite for the SD modeling language. For each change operation, it was analyzed whether the change operation is refining, generalizing, or refactoring.

Chapter 6 presented an AD modeling language. The set of models is the set of all well-formed ADs. The semantic domain is the set of all possible finite execution traces. The semantic mapping maps each AD to all finite execution traces that it explicitly describes. The definition of the semantics of ADs is based on a translation from ADs to NFAs. The semantics of an AD is defined as the language recognized by the NFA resulting from translating the AD. This enables reducing semantic differencing of ADs to language inclusion checking between NFAs. Chapter 6 further presented a complete change operation suite for the AD modeling language. For each change operation, we analyzed whether it is refining, generalizing, or refactoring.

Chapter 7 presented a theoretical and language-independent framework for automatic model repairs. The framework extends the framework for defining modeling languages introduced in Chapter 2. Properties of models are represented by the sets of models that satisfy the property. A change sequence repairs a model towards satisfying a property, if the application of the change sequence's change operations in order yields a model that satisfies the property. The set of change operations applicable to a model is usually infinite. This hampers the automatic computation of repairing sequences. Therefore, Chapter 7 introduced equivalence relations for partitioning the change operations that are applicable to models. Each equivalence relation depends on a model and a property. A repair-representative function for a (non-empty) property is a function that maps each model to a finite set of change operation containing at least one element from a special equivalence class that depends on the model and the property. For each model, each repair representative function induces a change sequence search tree. The change sequence search tree is a possibly infinite, finitely branching, rooted tree, which contains a node that encodes a change sequence that repairs the model towards satisfying the property. Thus, it is possible to reduce the search for a repairing change sequence to a search in a finitely branching rooted tree that is guaranteed to contain a solution.

Chapter 8 illustrated the applicability of the framework for automatic model repairs. It instantiated the model repair framework with the four modeling languages presented in Chapter 3 - Chapter 6 and the properties refinement, generalization, and refactoring. The performance of different algorithms for computing solutions for refinement repair problem instances using models of the TSPA, FD, SD, and AD modeling languages were

experimentally evaluated. No algorithm performed best for all instances. Therefore, it seems to depend on the instance which of the algorithms performs best.

We conclude that it is possible to develop semantic differencing operators for various modeling languages with heterogeneous semantics. The instantiations of the model repair framework further demonstrated the possibility to develop non-trivial automatic model evolution analyses relating the syntax of models with their semantics. This opens interesting future work directions as discussed in the following section.

9.2 Possible Future Work Directions

A possible future work direction is the development of semantic differencing operators for more concrete modeling languages. Candidates for this task are the different languages contained in the UML [OMG15] and the SysML [OMG17]. The results of this thesis indicate that the development of semantic differencing operators for these modeling languages could be possible when choosing adequate semantics.

Existing syntactic and semantic differencing approaches and combinations of syntactic and semantic differencing are limited to comparing exactly two models of the same language. However, describing a complex system with a single model is impractical [HKR⁺07]. Therefore, in MDD, complex systems are usually modeled by a set of models, often by using heterogeneous languages [Rum13] as, for example, provided by the UML. Each model describes a different viewpoint on the system under development. To this effect, model composition is an important concept to achieve modularity and to enable independent parallel system development [Rum13]. The semantics of a set of models is then obtained by adequately composing the semantics of the models contained in the system. As the semantics of the complete system is composed of the semantics of the individual models, two different systems of models may be semantically equivalent, although the individual models of the two systems are pairwise semantically different. This phenomenon occurs, because each model in the system is part of the context of each other model. Currently, most semantic differencing operators do not consider the context of models. Based on the fundamental definition of modeling language, a possible future work direction is the development of a theoretically grounded framework for composing models and for lifting modeling languages with adequate composition operators to system modeling languages as described above. It could be possible to compose the semantic and the syntactic differencing operators for the individual languages to obtain semantic and syntactic differencing operators for composed modeling languages. This yields great benefits for developing analyses for systems of models as the syntax and semantics of system modeling languages are significantly more complex than the syntax and the semantics of the languages from which they originate.

Bibliography

- [ABH⁺17] Kai Adam, Arvid Butting, Robert Heim, Oliver Kautz, Jérôme Pfeiffer, Bernhard Rumpe, and Andreas Wortmann. *Modeling Robotics Tasks for Better Separation of Concerns, Platform-Independence, and Reuse*. Aachener Informatik-Berichte, Software Engineering, Band 28. Shaker Verlag, December 2017.
- [ACC⁺11] Parosh Aziz Abdulla, Yu-Fang Chen, Lorenzo Clemente, Lukáš Holík, Chih-Duo Hong, Richard Mayr, and Tomáš Vojnar. Advanced Ramsey-Based Büchi Automata Inclusion Testing. In Joost-Pieter Katoen and Barbara König, editors, *CONCUR 2011 – Concurrency Theory*, pages 187–202. Springer Berlin Heidelberg, 2011.
- [ACLF10] Mathieu Acher, Philippe Collet, Philippe Lahire, and Robert France. Composing Feature Models. In Mark van den Brand, Dragan Gašević, and Jeff Gray, editors, *Software Language Engineering*, pages 62–81. Springer Berlin Heidelberg, 2010.
- [AEH⁺99] Marc Andries, Gregor Engels, Annegret Habel, Berthold Hoffmann, Hans-Jörg Kreowski, Sabine Kuske, Detlef Plump, Andy Schürr, and Gabriele Taentzer. Graph Transformation for Specification and Programming. *Science of Computer Programming*, 34(1):1–54, April 1999.
- [AHC⁺12] Mathieu Acher, Patrick Heymans, Philippe Collet, Clément Quinton, Philippe Lahire, and Philippe Merle. Feature Model Differences. In Jolita Ralyté, Xavier Franch, Sjaak Brinkkemper, and Stanisław Wrycza, editors, *Advanced Information Systems Engineering*, pages 629–645. Springer Berlin Heidelberg, 2012.
- [Alu15] Rajeev Alur. *Principles of Cyber-Physical Systems*. The MIT Press, 2015.
- [AP03] Marcus Alanen and Ivan Porres. Difference and Union of Models. In Perdita Stevens, Jon Whittle, and Grady Booch, editors, *UML 2003 – The Unified Modeling Language. Modeling Languages and Applications*, pages 2–17. Springer Berlin Heidelberg, 2003.

- [ASW09] Kerstin Altmanninger, Martina Seidl, and Manuel Wimmer. A survey on model versioning approaches. *International Journal of Web Information Systems*, 5(3):271–304, 2009.
- [AVT⁺15] Vincent Aravantinos, Sebastian Voss, Sabine Teufl, Florian Hölzl, and Bernhard Schätz. AutoFOCUS 3: Tooling Concepts for Seamless, Model-based Development of Embedded Systems. In Iulia Dragomir, Susanne Graf, Gabor Karsai, Florian Noyrit, Iulian Ober, Damiano Torre, Yvan Labiche, Marcela Genero, and Maged Elaasar, editors, *Joint proceedings of ACES-MB 2015 – Model-based Architecting of Cyber-physical and Embedded Systems and WUCOR 2015 – UML Consistency Rules*, pages 19–26. CEUR, 2015.
- [Bat05] Don Batory. Feature Models, Grammars, and Propositional Formulas. In Henk Obbink and Klaus Pohl, editors, *Software Product Lines*, pages 7–20. Springer Berlin Heidelberg, 2005.
- [BC11] Manfred Broy and María Victoria Cengarle. UML formal semantics: lessons learned. *Software & Systems Modeling*, 10(4), October 2011.
- [BCGR09] Manfred Broy, María Victoria Cengarle, Hans Grönniger, and Bernhard Rumpe. Definition of the System Model. In Kevin Lano, editor, *UML 2 Semantics and Applications*. John Wiley & Sons, Inc., October 2009.
- [BCM⁺92] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic Model Checking: 10^{20} States and Beyond. *Information and Computation*, 98(2):142–170, June 1992.
- [BCW17] Marco Brambilla, Jordi Cabot, and Manuel Wimmer. *Model-Driven Software Engineering in Practice: Second Edition*. Morgan & Claypool Publishers, second edition, 2017.
- [BEK⁺18] Arvid Butting, Robert Eikermann, Oliver Kautz, Bernhard Rumpe, and Andreas Wortmann. Controlled and Extensible Variability of Concrete and Abstract Syntax with Independent Language Features. In Rafael Capilla, Malte Lochau, and Lidia Fuentes, editors, *Proceedings of the 12th International Workshop on Variability Modelling of Software-Intensive Systems*, pages 75–82. ACM, February 2018.
- [BEK⁺19] Arvid Butting, Robert Eikermann, Oliver Kautz, Bernhard Rumpe, and Andreas Wortmann. Systematic Composition of Independent Language Features. *Journal of Systems and Software*, 152:50–69, June 2019.
- [Ber86] Valdis Berzins. On Merging Software Extensions. *Acta Informatica*, 23:607–619, November 1986.

- [Bet16] Lorenzo Bettini. *Implementing Domain Specific Languages with Xtext and Xtend: Second Edition*. Packt Publishing, second edition, 2016.
- [BHP⁺98] Manfred Broy, Franz Huber, Barbara Paech, Bernhard Rumpe, and Katharina Spies. Software and System Modeling Based on a Unified Formal Semantics. In Manfred Broy and Bernhard Rumpe, editors, *Requirements Targeting Software and Systems Engineering (RTSE'97)*, pages 43–68. Springer Berlin Heidelberg, 1998.
- [BK08] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. MIT Press, 2008.
- [BKL⁺12] Petra Brosch, Gerti Kappel, Philip Langer, Martina Seidl, Konrad Wieland, and Manuel Wimmer. An Introduction to Model Versioning. In Marco Bernardo, Vittorio Cortellessa, and Alfonso Pierantonio, editors, *Formal Methods for Model-Driven Engineering: 12th International School on Formal Methods for the Design of Computer, Communication, and Software Systems*, pages 336–398. Springer Berlin Heidelberg, 2012.
- [BKL⁺16] Johannes Bürdek, Timo Kehrer, Malte Lochau, Dennis Reuling, Udo Kelter, and Andy Schürr. Reasoning about product-line evolution using complex feature model differences. *Automated Software Engineering*, 23(4):687–733, 2016.
- [BKRW17] Arvid Butting, Oliver Kautz, Bernhard Rumpe, and Andreas Wortmann. Semantic Differencing for Message-Driven Component & Connector Architectures. In *IEEE International Conference on Software Architecture (ICSA)*, pages 145–154. IEEE, 2017.
- [BKRW19] Arvid Butting, Oliver Kautz, Bernhard Rumpe, and Andreas Wortmann. Continuously analyzing finite, message-driven, time-synchronous component & connector systems during architecture evolution. *Journal of Systems and Software*, 149:437–461, 2019.
- [BMMR12] Luciano Baresi, Angelo Morzenti, Alfredo Motta, and Matteo Rossi. A Logic-based Semantics for the Verification of Multi-diagram UML Models. *ACM Sigsoft Software Engineering Notes*, 37(4), 2012.
- [BPvdL05] Günter Böckle, Klaus Pohl, and Frank van der Linden. A Framework for Software Product Line Engineering. In *Software Product Line Engineering: Foundations, Principles, and Techniques*, pages 19–38. Springer Berlin Heidelberg, 2005.
- [Bro10] Manfred Broy. A Logical Basis for Component-Oriented Software and Systems Engineering. *The Computer Journal*, 53(10):1758–1782, 2010.

- [BS01] Manfred Broy and Ketil Stølen. *Specification and Development of Interactive Systems: Focus on Streams, Interfaces, and Refinement*. Springer, 2001.
- [BSRC10] David Benavides, Sergio Segura, and Antonio Ruiz-Cortés. Automated Analysis of Feature Models 20 Years Later: A Literature Review. *Information Systems*, 35(6):615–636, 2010.
- [BTRC05] David Benavides, Pablo Trinidad, and Antonio Ruiz-Cortés. Automated Reasoning on Feature Models. In Oscar Pastor and João Falcão e Cunha, editors, *Advanced Information Systems Engineering*, pages 491–503. Springer Berlin Heidelberg, 2005.
- [Büc90] Julius Richard Büchi. On a decision method in restricted second order arithmetic. In Saunders Mac Lane and Dirk Siefkes, editors, *The Collected Works of J. Richard Büchi*, pages 425–435. Springer, 1990.
- [CFJ⁺16] Benoit Combemale, Robert France, Jean-Marc Jézéquel, Bernhard Rumpe, James Steel, and Didier Vojtisek. *Engineering Modeling Languages: Turning Domain Knowledge into Tools*. Chapman & Hall/CRC, November 2016.
- [CGR09] María Victoria Cengarle, Hans Grönniger, and Bernhard Rumpe. Variability within Modeling Language Definitions. In Andy Schürr and Bran Selic, editors, *Model Driven Engineering Languages and Systems*, pages 670–684. Springer Berlin Heidelberg, 2009.
- [CN01] Paul Clements and Linda Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2001.
- [CQT⁺16] Everton Cavalcante, Jean Quilbeuf, Louis-Marie Traonouez, Flávio Oquendo, Thaís Batista, and Axel Legay. Statistical Model Checking of Dynamic Software Architectures. In Bedir Tekinerdogan, Uwe Zdun, and Ali Babar, editors, *Software Architecture: 10th European Conference, ECSA 2016, Copenhagen, Denmark, November 28 – December 2, 2016, Proceedings*, pages 185–200. Springer International Publishing, 2016.
- [CRP07] Antonio Cicchetti, Davide Di Ruscio, and Alfonso Pierantonio. A Meta-model Independent Approach to Difference Representation. *Journal of Object Technology*, 6(9):165–185, October 2007.
- [CW07] Krzysztof Czarnecki and Andrzej Wasowski. Feature Diagrams and Logics: There and Back Again. In *11th International Software Product Line Conference (SPLC 2007)*, pages 23–34. IEEE, September 2007.

- [DEKR19] Imke Drave, Robert Eikermann, Oliver Kautz, and Bernhard Rumpe. Semantic Differencing of Statecharts for Object-Oriented Systems. In Slimane Hammoudi, Luis Ferreira Pires, and Bran Selić, editors, *Proceedings of the 7th International Conference on Model-Driven Engineering and Software Development*, pages 272–280. SciTePress, 2019.
- [DKMR19] Imke Drave, Oliver Kautz, Judith Michael, and Bernhard Rumpe. Semantic Evolution Analysis of Feature Models. In Thorsten Berger, Philippe Collet, Laurence Duchien, Thomas Fogdal, Patrick Heymans, Timo Kehrer, Jabier Martinez, Raúl Mazo, Leticia Montalvillo, Camille Salinesi, Xhevahire Tërnav, Thomas Thüm, and Tewfik Ziadi, editors, *Proceedings of the 23rd International Systems and Software Product Line Conference*, pages 245–255. ACM, 2019.
- [DKMR20] Imke Drave, Oliver Kautz, Judith Michael, and Bernhard Rumpe. Pre-Study on the Usefulness of Difference Operators for Modeling Languages in Software Development. Technical Report AIB-2020-05, RWTH Aachen University, May 2020.
- [DLL⁺10] Alexandre David, Kim G. Larsen, Axel Legay, Ulrik Nyman, and Andrzej Wasowski. Timed I/O Automata: A Complete Specification Theory for Real-time Systems. In *Proceedings of the 13th ACM International Conference on Hybrid Systems: Computation and Control*, pages 91–100. ACM, 2010.
- [EEPT06] Hartmut Ehrig, Karsten Ehrig, Ulrike Prange, and Gabriele Taentzer. *Fundamentals of Algebraic Graph Transformation*. Springer Berlin Heidelberg, 2006.
- [EKM98] Jacob Elgaard, Nils Klarlund, and Anders Møller. MONA 1.x: New techniques for WS1S and WS2S. In Alan J. Hu and Moshe Y. Vardi, editors, *Computer Aided Verification*, pages 516–520. Springer Berlin Heidelberg, 1998.
- [End72] Herbert B. Enderton. *A Mathematical Introduction to Logic*. Academic Press, 1972.
- [EPK06] Klaus-D. Engel, Richard F. Paige, and Dimitrios S. Kolovos. Using a Model Merging Language for Reconciling Model Versions. In Arend Rensink and Jos Warmer, editors, *Model Driven Architecture – Foundations and Applications*, pages 143–157. Springer Berlin Heidelberg, 2006.

- [Esh06] Rik Eshuis. Symbolic Model Checking of UML Activity Diagrams. *ACM Transactions on Software Engineering and Methodology*, 15(1):1–38, 2006.
- [FALW14] Uli Fahrenberg, Mathieu Acher, Axel Legay, and Andrzej Wąsowski. Sound Merging and Differencing for Class Diagrams. In Stefania Gnesi and Arend Rensink, editors, *Fundamental Approaches to Software Engineering*, pages 63–78. Springer Berlin Heidelberg, 2014.
- [Far02] Berndt Farwer. ω -automata. In Erich Grädel, Wolfgang Thomas, and Thomas Wilke, editors, *Automata Logics, and Infinite Games: A Guide to Current Research*, pages 3–21. Springer Berlin Heidelberg, 2002.
- [FLW11] Uli Fahrenberg, Axel Legay, and Andrzej Wąsowski. Vision Paper: Make a Difference! (Semantically). In Jon Whittle, Tony Clark, and Thomas Kühne, editors, *Model Driven Engineering Languages and Systems*, pages 490–500. Springer Berlin Heidelberg, 2011.
- [FR07] Robert France and Bernhard Rumpe. Model-Driven Development of Complex Software: A Research Roadmap. In Lionel C. Briand and Alexander L. Wolf, editors, *Future of Software Engineering (FOSE '07)*, pages 37–54. IEEE, 2007.
- [Fuc95] Max Fuchs. Formal Design of a Modulo-N Counter. Technical Report TUM-I9512, Technische Universität München, 1995.
- [GJK⁺13] Stefan Gulan, Sven Johr, Roberto Kretschmer, Stefan Rieger, and Michael Ditze. Graphical Modelling meets Formal Methods. In *IEEE International Conference on Industrial Informatics (INDIN)*, pages 716–721. IEEE, 2013.
- [GKLE10] Christian Gerth, Jochen M. Küster, Markus Luckey, and Gregor Engels. Precise Detection of Conflicting Change Operations Using Process Model Terms. In Dorina C. Petriu, Nicolas Rouquette, and Øystein Haugen, editors, *Model Driven Engineering Languages and Systems*, pages 93–107. Springer Berlin Heidelberg, 2010.
- [GKLE13] Christian Gerth, Jochen M. Küster, Markus Luckey, and Gregor Engels. Detection and resolution of conflicting change operations in version management of process models. *Software & Systems Modeling*, 12(3):517–535, 2013.
- [GLKE10] Christian Gerth, Markus Luckey, Jochen Malte Küster, and Gregor Engels. Detection of Semantically Equivalent Fragments for Business Pro-

- cess Model Change Management. In *IEEE International Conference on Services Computing*, pages 57–64. IEEE, 2010.
- [GMM19] Luca Gazzola, Daniela Micucci, and Leonardo Mariani. Automatic Software Repair: A Survey. *IEEE Transactions on Software Engineering*, 45(1):34–67, January 2019.
- [GNT16] Malik Ghallab, Dana Nau, and Paolo Traverso. *Automated Planning and Acting*. Cambridge University Press, 2016.
- [GR95] Radu Grosu and Bernhard Rumpe. Concurrent Timed Port Automata. Technical Report TUM-I9533, TU Munich, 1995.
- [GR11] Hans Grönniger and Bernhard Rumpe. Modeling Language Variability. In Radu Calinescu and Ethan Jackson, editors, *Foundations of Computer Software. Modeling, Development, and Verification of Adaptive Systems*, pages 17–32. Springer Berlin Heidelberg, 2011.
- [Grö10] Hans Grönniger. *Systemmodell-basierte Definition objektbasierter Modellierungssprachen mit semantischen Variationspunkten*. Aachener Informatik-Berichte, Software Engineering, Band 4. Shaker, August 2010.
- [GRR10] Hans Grönniger, Dirk Reiß, and Bernhard Rumpe. Towards a Semantics of Activity Diagrams with Semantic Variation Points. In Dorina C. Petriu, Nicolas Rouquette, and Øystein Haugen, editors, *Model Driven Engineering Languages and Systems*, pages 331–345. Springer Berlin Heidelberg, 2010.
- [GS05] Radu Grosu and Scott A. Smolka. Safety-Liveness Semantics for UML 2.0 Sequence Diagrams. In Jörg Desel and Yosinori Watanabe, editors, *Fifth International Conference on Application of Concurrency to System Design (ACSD’05)*, pages 6–14. IEEE, 2005.
- [Hab16] Arne Haber. *MontiArc - Architectural Modeling and Simulation of Interactive Distributed Systems*. Aachener Informatik-Berichte, Software Engineering, Band 24. Shaker Verlag, September 2016.
- [Hal60] Paul R. Halmos. *Naive Set Theory*. Van Nostrand, 1960.
- [Har07] Mark Harman. The Current State and Future of Search Based Software Engineering. In Lionel C. Briand and Alexander L. Wolf, editors, *Future of Software Engineering (FOSE ’07)*, pages 342–357. IEEE, 2007.
- [Har10] Mark Harman. Technical Perspective: Automated Patching Techniques: The Fix Is In. *Communications of the ACM*, 53(5):108, 2010.

- [Hec06] Reiko Heckel. Graph Transformation in a Nutshell. *Electronic Notes in Theoretical Computer Science*, 148(1):187–198, February 2006.
- [HF10] Florian Hölzl and Martin Feilkas. AutoFocus 3 - A Scientific Tool Prototype for Model-Based Development of Component-Based, Reactive, Distributed Systems. In Holger Giese, Gabor Karsai, Edward Lee, Bernhard Rumpe, and Bernhard Schätz, editors, *Model-Based Engineering of Embedded Real-Time Systems: International Dagstuhl Workshop, Dagstuhl Castle, Germany, November 4-9, 2007. Revised Selected Papers*, pages 317–322. Springer Berlin Heidelberg, 2010.
- [HHRS05] Øystein Haugen, Knut Eilif Husa, Ragnhild Kobro Runde, and Ketil Stølen. STAIRS towards formal design with sequence diagrams. *Software & Systems Modeling*, 4:355–367, 2005.
- [HKR⁺07] Christoph Herrmann, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. An Algebraic View on the Semantics of Model Composition. In David H. Akehurst, Régis Vogel, and Richard F. Paige, editors, *Model Driven Architecture- Foundations and Applications*, pages 99–113. Springer Berlin Heidelberg, 2007.
- [HM08] David Harel and Shahar Maoz. Assert and negate revisited: Modal semantics for UML sequence diagrams. *Software & Systems Modeling*, 7:237—252, 2008.
- [HMU06] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. Addison-Wesley, 2006.
- [HN96] David Harel and Amnon Naamad. The STATEMATE Semantics of Statecharts. *ACM Transactions on Software Engineering and Methodology*, 5(4):293–333, 1996.
- [HR00] David Harel and Bernhard Rumpe. Modeling Languages: Syntax, Semantics and All That Stuff (Part I: The Basic Stuff). Technical Report MCS00-16, Mathematics & Computer Science, Weizmann Institute of Science, 2000.
- [HR04] David Harel and Bernhard Rumpe. Meaningful Modeling: What’s the Semantics of ”Semantics”? *Computer*, 37(10):64–72, 2004.
- [HR17] Katrin Hölldobler and Bernhard Rumpe. *MontiCore 5 Language Workbench Edition 2017*. Aachener Informatik-Berichte, Software Engineering, Band 32. Shaker Verlag, December 2017.

- [HRR12] Arne Haber, Jan Oliver Ringert, and Bernard Rumpe. MontiArc - Architectural Modeling of Interactive Distributed and Cyber-Physical Systems. Technical Report AIB-2012-03, RWTH Aachen University, 2012.
- [HU69] John E. Hopcroft and Jeffrey D. Ullman. *Formal Languages and Their Relation to Automata*. Addison-Wesley, 1969.
- [Jac06] Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis*. MIT Press, 2006.
- [Kan00] Robert Kantrowitz. A Principle of Countability. *Mathematics Magazine*, 73:40–42, 2000.
- [KG10] Christoph Knieke and Ursula Goltz. An Executable Semantics for UML 2 Activity Diagrams. In *Proceedings of the International Workshop on Formalization of Modeling Languages*. ACM, 2010.
- [KGE09] Jochen M. Küster, Christian Gerth, and Gregor Engels. Dependent and Conflicting Change Operations of Process Models. In Richard F. Paige, Alan Hartman, and Arend Rensink, editors, *Model Driven Architecture - Foundations and Applications*, pages 158–173. Springer Berlin Heidelberg, 2009.
- [KGFE08] Jochen M. Küster, Christian Gerth, Alexander Förster, and Gregor Engels. Detecting and Resolving Process Model Differences in the Absence of a Change Log. In Marlon Dumas, Manfred Reichert, and Ming-Chien Shan, editors, *Business Process Management*, pages 244–260. Springer Berlin Heidelberg, 2008.
- [KKT11] Timo Kehrer, Udo Kelter, and Gabriele Taentzer. A Rule-Based Approach to the Semantic Lifting of Model Differences in the Context of Model Versioning. In *26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*, pages 163–172. IEEE, 2011.
- [KKT13] Timo Kehrer, Udo Kelter, and Gabriele Taentzer. Consistency-Preserving Edit Scripts in Model Versioning. In *28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 191–201. IEEE, 2013.
- [KLD02] K. C. Kang, Jaejoon Lee, and P. Donohoe. Feature-Oriented Product Line Engineering. *IEEE Software*, 19(4):58–65, 2002.
- [KLSV03] Dilsun K. Kaynar, Nancy A. Lynch, Roberto Segala, and Frits W. Vaandrager. Timed I/O Automata: A Mathematical Framework for Modeling

- and Analyzing Real-Time Systems. In *RTSS 2003. 24th IEEE Real-Time Systems Symposium*, pages 166–177. IEEE, 2003.
- [KMRR17] Oliver Kautz, Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. CD2Alloy: A Translation of Class Diagrams to Alloy. Technical Report AIB-2017-06, RWTH Aachen University, July 2017.
- [Kön27] Dénes König. Über eine Schlussweise aus dem Endlichen ins Unendliche (Punktmengen - Kantenfärben - Verwandtschaftsbeziehungen - Schachspiel). *Acta litterarum ac scientiarum Regiae Universitatis Hungaricae Francisco-Josephinae : Sectio scientiarum mathematicarum*, 3:121–130, 1927.
- [Kor85] Richard E. Korf. Depth-first Iterative-Deepening: An Optimal Admissible Tree Search. *Artificial Intelligence*, 27(1):97–109, September 1985.
- [KR18a] Oliver Kautz and Bernhard Rumpe. On Computing Instructions to Repair Failed Model Refinements. In *Proceedings of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*, pages 289–299. ACM, October 2018.
- [KR18b] Oliver Kautz and Bernhard Rumpe. Semantic Differencing of Activity Diagrams by a Translation into Finite Automata. In Regina Hebig and Thorsten Berger, editors, *Proceedings of MODELS 2018 Workshops: ModComp, MRT, OCL, FlexMDE, EXE, COMMitMDE, MDETools, GEMOC, MORSE, MDE4IoT, MDEbug, MoDeVVa, ME, MULTI, HuFaMo, AMMoRe, PAINS co-located with ACM/IEEE 21st International Conference on Model Driven Engineering Languages and Systems (MODELS 2018)*. CEUR, October 2018.
- [KRR18] Oliver Kautz, Alexander Roth, and Bernhard Rumpe. *Achievements, Failures, and the Future of Model-Based Software Engineering*, pages 221–236. Springer International Publishing, June 2018.
- [KRRvW17] Evgeny Kusmenko, Alexander Roth, Bernhard Rumpe, and Michael von Wenckstern. Modeling Architectures of Cyber-Physical Systems. In Anthony Anjorin and Huáscar Espinoza, editors, *Modelling Foundations and Applications*, pages 34–50. Springer International Publishing, 2017.
- [KRSvW18] Evgeny Kusmenko, Bernhard Rumpe, Sascha Schneiders, and Michael von Wenckstern. Highly-Optimizing and Multi-Target Compiler for Embedded System Models: C++ Compiler Toolchain for the Component and Connector Language EmbeddedMontiArc. In *Proceedings of the 21th*

- ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*, pages 447–457. ACM, October 2018.
- [KRV10] Holger Krahn, Bernhard Rumpe, and Steven Völkel. MontiCore: a framework for compositional development of domain specific languages. *International Journal on Software Tools for Technology Transfer*, 12:353–372, March 2010.
- [Lee08] Edward A. Lee. Cyber Physical Systems: Design Challenges. In *The 11th IEEE Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*, pages 363–369. IEEE, 2008.
- [LGDVFW12] Claire Le Goues, Michael Dewey-Vogt, Stephanie Forrest, and Westley Weimer. A Systematic Study of Automated Program Repair: Fixing 55 out of 105 Bugs for \$8 Each. In *34th International Conference on Software Engineering*, pages 3–13. IEEE, 2012.
- [LK11] Lunjin Lu and Dae-Kyoo Kim. Required Behavior of Sequence Diagrams: Semantics and Refinement. In Isabelle Perseil, Karin Breitman, and Roy Sterritt, editors, *Sixteenth IEEE International Conference on Engineering of Complex Computer Systems*, pages 127–136. IEEE, 2011.
- [LK14] Lunjin Lu and Dae-Kyoo Kim. Required Behavior of Sequence Diagrams: Semantics and Conformance. *ACM Transactions on Software Engineering and Methodology*, 23(2), April 2014.
- [LMK14a] Philip Langer, Tanja Mayerhofer, and Gerti Kappel. A Generic Framework for Realizing Semantic Model Differencing Operators. In Stefan Sauer and Manuel Wimmer, editors, *Joint Proceedings of MODELS 2014 Poster Session and the ACM Student Research Competition (SRC) co-located with the 17th International Conference on Model Driven Engineering Languages and Systems (MODELS 2014)*. CEUR, 2014.
- [LMK14b] Philip Langer, Tanja Mayerhofer, and Gerti Kappel. Semantic Model Differencing Utilizing Behavioral Semantics Specifications. In Juergen Dingel, Wolfram Schulte, Isidro Ramos, Silvia Abrahão, and Emilio Insfran, editors, *Model-Driven Engineering Languages and Systems*, pages 116–132. Springer International Publishing, 2014.
- [LvO92] Ernst Lippe and Norbert van Oosterom. Operation-Based Merging. *ACM SIGSOFT Software Engineering Notes*, 17(5):78–87, 1992.
- [McM93] Kenneth L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

BIBLIOGRAPHY

- [MD08] Tom Mens and Serge Demeyer, editors. *Software Evolution*. Springer Berlin Heidelberg, 2008.
- [Men02] Tom Mens. A State-of-the-Art Survey on Software Merging. *IEEE Transactions on Software Engineering*, 28(5):449–462, 2002.
- [MFBC12] Pierre-Alain Muller, Frédéric Fondement, Benoît Baudry, and Benoît Combemale. Modeling modeling modeling. *Software & Systems Modeling*, 11(3):347–359, 2012.
- [MGH05] Akhil Mehra, John Grundy, and John Hosking. A Generic Approach to Supporting Diagram Differencing and Merging for Collaborative Design. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, pages 204–213. ACM, 2005.
- [Mil99] Robin Milner. *Communicating and Mobile Systems: The π -calculus*. Cambridge University Press, 1999.
- [Mon18] Martin Monperrus. Automatic Software Repair: A Bibliography. *ACM Computing Surveys*, 51(1), April 2018.
- [MR15] Shahar Maoz and Jan Oliver Ringert. A Framework for Relating Syntactic and Semantic Model Differences. In *ACM/IEEE 18th International Conference on Model Driven Engineering Languages and Systems (MODELS)*, pages 24–33. IEEE, 2015.
- [MR18] Shahar Maoz and Jan Oliver Ringert. A framework for relating syntactic and semantic model differences. *Software & Systems Modeling*, 17(3):753–777, 2018.
- [MRR11a] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. A Manifesto for Semantic Model Differencing. In Juergen Dingel and Arnor Solberg, editors, *Models in Software Engineering*, pages 194–203. Springer Berlin Heidelberg, 2011.
- [MRR11b] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. ADDiff: Semantic Differencing for Activity Diagrams. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, pages 179–189. ACM, 2011.
- [MRR11c] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. An Operational Semantics for Activity Diagrams using SMV. Technical Report AIB-2011-07, RWTH Aachen University, Aachen, Germany, July 2011.

- [MRR11d] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. CD2Alloy: Class Diagrams Analysis Using Alloy Revisited. In Jon Whittle, Tony Clark, and Thomas Kühne, editors, *Model Driven Engineering Languages and Systems*, pages 592–607. Springer Berlin Heidelberg, 2011.
- [MRR11e] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. CDDiff: Semantic Differencing for Class Diagrams. In Mira Mezini, editor, *ECOOP 2011 – Object-Oriented Programming*, pages 230–254. Springer Berlin Heidelberg, 2011.
- [MRR11f] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. Semantically Configurable Consistency Analysis for Class and Object Diagrams. In Jon Whittle, Tony Clark, and Thomas Kühne, editors, *Model Driven Engineering Languages and Systems*, pages 153–167. Springer Berlin Heidelberg, 2011.
- [MRR11g] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. Summarizing Semantic Model Differences. In *International Workshop on Models and Evolution at ACM/IEEE 14th International Conference on Model Driven Engineering Languages and Systems*, 2011.
- [MRR12] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. An Interim Summary on Semantic Model Differencing. *Softwaretechnik-Trends*, 32(4):44–46, 2012.
- [MW11] Zoltán Micskei and Hélène Waeselynck. The many meanings of UML 2 Sequence Diagrams: a survey. *Software & Systems Modeling*, 10(4):489–514, October 2011.
- [Nag76] Manfred Nagl. Formal Languages of Labelled Graphs. *Computing*, 16:113–137, march 1976.
- [Nag79] Manfred Nagl. *Graph-Grammatiken: Theorie, Anwendungen, Implementierung*. Vieweg, 1979.
- [NR69] Peter Naur and Brian Randell, editors. *Software Engineering: Report on a conference sponsored by the NATO SCIENCE COMMITTEE, Garmisch, Germany, 7th to 11th October 1968*, 1969.
- [OMG15] Object Management Group. OMG Unified Modeling Language (OMG UML), Version 2.5., May 2015. <https://www.omg.org/spec/UML/2.5/PDF/> [accessed 2020-06-19].

- [OMG17] Object Management Group. OMG System Modeling Language Version 1.5 (OMG SysML), May 2017. <https://www.omg.org/spec/SysML/1.5/PDF> [accessed 2020-06-19].
- [Oqu04] Flavio Oquendo. π -ADL: an Architecture Description Language based on the higher-order typed π -calculus for specifying dynamic and mobile software architectures. *ACM SIGSOFT Software Engineering Notes*, 29(3):1–14, 2004.
- [OWK03] Dirk Ohst, Michael Welle, and Udo Kelter. Differences between versions of UML diagrams. In *Proceedings of the 9th European Software Engineering Conference Held Jointly with 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 227–236. ACM, 2003.
- [PBv05] Klaus Pohl, Günter Böckle, and Frank van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer Berlin Heidelberg, 2005.
- [PR97] Jan Philipps and Bernhard Rumpe. Refinement of Information Flow Architectures. In *First IEEE International Conference on Formal Engineering Methods*, pages 203–212. IEEE, 1997.
- [PR99] Jan Philipps and Bernhard Rumpe. Refinement of Pipe-and-Filter Architectures. In Jeannette M. Wing, Jim Woodcock, and Jim Davies, editors, *FM’99 — Formal Methods*, pages 96–115. Springer Berlin Heidelberg, 1999.
- [Rin08] Martin C. Rinard. Technical Perspective: Patching Program Errors. *Communications of the ACM*, 51(12), December 2008.
- [Rin14] Jan Oliver Ringert. *Analysis and Synthesis of Interactive Component and Connector Systems*. Aachener Informatik-Berichte, Software Engineering, Band 19. Shaker Verlag, 2014.
- [RR11] Jan Oliver Ringert and Bernhard Rumpe. A Little Synopsis on Streams, Stream Processing Functions, and State-Based Stream Processing. *International Journal of Software and Informatics*, 5, 2011.
- [RS59] Michael Oser Rabin and Dana Scott. Finite Automata and Their Decision Problems. *IBM Journal of Research and Development*, 3(2):114–125, 1959.
- [Rud76] Walter Rudin. *Principles of Mathematical Analysis*. McGraw-Hill Education Ltd, third edition edition, 1976.

- [Rum96] Bernhard Rumpe. *Formale Methodik des Entwurfs verteilter objektorientierter Systeme*. Herbert Utz Verlag Wissenschaft, 1996.
- [Rum13] Bernhard Rumpe. Towards Model and Language Composition. In *Proceedings of the First Workshop on the Globalization of Domain Specific Languages*, pages 4–7. ACM, 2013.
- [Rum16] Bernhard Rumpe. *Modeling with UML: Language, Concepts, Methods*. Springer International, July 2016.
- [Rum17] Bernhard Rumpe. *Agile Modeling with UML: Code Generation, Testing, Refactoring*. Springer International, May 2017.
- [RW18] Bernhard Rumpe and Andreas Wortmann. Abstraction and Refinement in Hierarchically Decomposable and Underspecified CPS-Architectures. In Lohstroh, Marten and Derler, Patricia Sirjani, Marjan, editor, *Principles of Modeling: Essays Dedicated to Edward A. Lee on the Occasion of His 60th Birthday*, pages 383–406. Springer International Publishing, 2018.
- [Saf88] Shmuel Safra. On The Complexity of ω -Automata. In *29th Annual Symposium on Foundations of Computer Science*, pages 319–327. IEEE, 1988.
- [SBPM09] David Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF: Eclipse Modeling Framework 2.0*. Addison-Wesley, second edition, 2009.
- [SC13] Matthew Stephan and James R. Cordy. A Survey of Model Comparison Approaches and Applications. In Slimane Hammoudi, Luís Ferreira Pires, Joaquim Filipe, and Rui César das Neves, editors, *Proceedings of the 1st International Conference on Model-Driven Engineering and Software Development*, pages 265–277. SciTePress, 2013.
- [Sch06] Douglas C. Schmidt. Guest editor’s introduction: Model-driven engineering. *Computer*, 39(2):25–31, February 2006.
- [Sch12] Martin Schindler. *Eine Werkzeuginfrastruktur zur agilen Entwicklung mit der UML/P*. Aachener Informatik-Berichte, Software Engineering, Band 11. Shaker Verlag, 2012.
- [Sel03] Bran Selic. The Pragmatics of Model-Driven Development. *IEEE Software*, 20(5):19–25, 2003.

- [SHTB06] Pierre Yves Schobbens, Patrick Heymans, Jean-Christophe Trigaux, and Yves Bontemps. Feature Diagrams: A Survey and a Formal Semantics. In Martin Glinz and Robyn Lutz, editors, *14th IEEE International Requirements Engineering Conference (RE'06)*, pages 139–148. IEEE, 2006.
- [SHTB07] Pierre-Yves Schobbens, Patrick Heymans, Jean-Christophe Trigaux, and Yves Bontemps. Generic Semantics of Feature Diagrams. *Computer Networks*, 51(2):456–479, February 2007.
- [SO00] Wasim Sadiq and Maria E. Orlowska. Analyzing process models using graph reduction techniques. *Information Systems*, 25(2):117–134, April 2000.
- [Sta73] Herbert Stachowiak. *Allgemeine Modelltheorie*, 1973.
- [Ste08] Perdita Stevens. Towards an Algebraic Theory of Bidirectional Transformations. In Hartmut Ehrig, Reiko Heckel, Grzegorz Rozenberg, and Gabriele Taentzer, editors, *Graph Transformations*, pages 1–17. Springer Berlin Heidelberg, 2008.
- [Stö03a] Harald Störrle. Assert, Negate and Refinement in UML-2 Interactions. In Jan Jürjens, Bernhard Rumpe, Robert France, and Eduardo B. Fernandez, editors, *Critical Systems Development with UML - Proceedings of the UML'03 workshop*, volume TUM-I0323, pages 79–93. TU Munich, September 2003.
- [Stö03b] Harald Störrle. Semantics of Interactions in UML 2.0. In John Hosking and Philip Cox, editors, *IEEE Symposium on Human Centric Computing Languages and Environments*, pages 129–136. IEEE, 2003.
- [Stö05] Harald Störrle. Semantics and Verification of Data Flow in UML 2.0 Activities. *Electronic Notes in Theoretical Computer Science*, 127(4):35–52, 2005.
- [Sug16] Prompong Sugunnasil. Detecting deadlock in activity diagram using process automata. In *International Computer Science and Engineering Conference (ICSEC)*. IEEE, 2016.
- [TBD⁺08] Pablo Trinidad, David Benavides, Amador Durán, Antonio Ruiz-Cortés, and Miguel Toro. Automated error analysis for the agilization of feature modeling. *Journal of Systems and Software*, 81(6), June 2008.
- [TBK09] Thomas Thüm, Don Batory, and Christian Kästner. Reasoning about Edits to Feature Models. In *2009 31st International Conference on Software Engineering*, pages 254–264. IEEE, 2009.

- [TELW14] Gabriele Taentzer, Claudia Ermel, Philip Langer, and Manuel Wimmer. A fundamental approach to model versioning based on graph modifications: from theory to implementation. *Software & Systems Modeling*, 13(1):239–272, 2014.
- [TKB⁺14] Thomas Thüm, Christian Kästner, Fabian Benduhn, Jens Meinicke, Gunter Saake, and Thomas Leich. FeatureIDE: An extensible framework for feature-oriented software development. *Science of Computer Programming*, 79:70–85, 2014.
- [TM17] Rachel Tzoref-Brill and Shahar Maoz. Syntactic and Semantic Differencing for Combinatorial Models of Test Designs. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 621–631. IEEE, 2017.
- [TSJ10] Nasi Tantitharanukul, Prompong Sugunnasil, and Watcharee Jumpamule. Detecting Deadlock and Multiple Termination in BPMN Model Using Process Automata. In *ECTI-CON2010: The 2010 ECTI International Conference on Electrical Engineering/Electronics, Computer, Telecommunications and Information Technology*, pages 478–482. IEEE, 2010.
- [VBD⁺13] Markus Voelter, Sebastian Benz, Christian Dietrich, Birgit Engelmann, Mats Helander, Lennart Kats, Eelco Visser, and Guido Wachsmuth. *DSL Engineering: Designing, Implementing and Using Domain-Specific Languages*. CreateSpace Independent Publishing Platform, 2013.
- [VC15] Edoardo Vacchi and Walter Cazzola. Neverlang: A framework for feature-oriented language development. *Computer Languages, Systems & Structures*, 43:1–40, 2015.
- [vdA99] Wil M. P. van der Aalst. Formalization and Verification of Event-driven Process Chains. *Information and Software Technology*, 41(10):639–650, July 1999.
- [vdAHV02] W. M. P. van der Aalst, A. Hirnschall, and H. M. W. Verbeek. An Alternative Way to Analyze Workflow Graphs. In Anne Banks Pidduck, M. Tamer Ozsu, John Mylopoulos, and Carson C. Woo, editors, *Advanced Information Systems Engineering*, pages 535–552. Springer Berlin Heidelberg, 2002.
- [vdB12] Pim van den Broek. Intersection of Feature Models. In *Proceedings of the 16th International Software Product Line Conference - Volume 2*, pages 61–65. ACM, 2012.

- [vdBGN10] Pim van den Broek, Ismênia Galvão, and Joost Noppen. Merging Feature Models. In Goetz Botterweck, Stan Jarzabek, Tomoji Kishi, Jaejoon Lee, and Steve Livengood, editors, *Proceedings of the 14th International Software Product Line Conference Volume 2 - Workshops, Industrial Track, Doctoral Symposium, Demonstrations and Tools*, pages 83–89. Lancaster University, 2010.
- [VVL07] Jussi Vanhatalo, Hagen Völzer, and Frank Leymann. Faster and More Focused Control-Flow Analysis for Business Process Models Through SESE Decomposition. In Bernd J. Krämer, Kwei-Jay Lin, and Priya Narasimhan, editors, *Service-Oriented Computing – ICSOC 2007*, pages 43–55. Springer Berlin Heidelberg, 2007.
- [WCW⁺18] Ming Wen, Junjie Chen, Rongxin Wu, Dan Hao, and Shing-Chi Cheung. Context-aware Patch Generation for Better Automated Program Repair. In *Proceedings of the 40th International Conference on Software Engineering*, pages 1–11. ACM, 2018.
- [Wei06] Westley Weimer. Patches as Better Bug Reports. In *Proceedings of the 5th International Conference on Generative Programming and Component Engineering*, page 181–190. ACM, 2006.
- [Wes10] Bernhard Westfechtel. A Formal Approach to Three-Way Merging of EMF Models. In Davide Di Ruscio and Dimitris S. Kolovos, editors, *Proceedings of the 1st International Workshop on Model Comparison in Practice*, pages 31–41. ACM, 2010.
- [WNLGF09] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. Automatically Finding Patches Using Genetic Programming. In *2009 31st International Conference on Software Engineering*, pages 364–374. IEEE, 2009.
- [Wor16] Andreas Wortmann. *An Extensible Component & Connector Architecture Description Infrastructure for Multi-Platform Modeling*. Aachen Informatik-Berichte, Software Engineering, Band 25. Shaker Verlag, November 2016.
- [www20] RABIT Tool Homepage, 2020. <http://www.languageinclusion.org/> [accessed 2020-06-19].
- [XS07] Zhenchang Xing and Eleni Stroulia. Differencing logical UML models. *Automated Software Engineering*, 14(2):215–259, 2007.

- [ZZM04] Wei Zhang, Haiyan Zhao, and Hong Mei. A Propositional Logic-Based Method for Verification of Feature Models. In Jim Davies, Wolfram Schulte, and Mike Barnett, editors, *Formal Methods and Software Engineering*, pages 115–130. Springer Berlin Heidelberg, 2004.

Appendix A

Time-Synchronous Port Automata for Experimental Evaluations

This appendix presents the TSPAs used in the experimental evaluations for the TSPA semantic differencing operator and the model repair instantiations for the TSPA language. The TSPAs are presented in the common graphical notation.

Figure A.1 depicts the three TSPAs `mod4Ctr`, `threeCtr`, and `reset` modeling the behaviors of a binary counters. The TSPAs depicted in Figure A.1 are also used as examples in Section 3.3.

Figure A.2 depicts two simple TSPAs. The TSPAs `aut1` and `aut2` depicted in Figure A.2 are also used as examples in Section 8.2.4.

Figure A.3 the two TSPAs `impl1` and `spec` modeling the behaviors of a mobile robot. The TSPAs are also used as examples in Section 7.1.2.

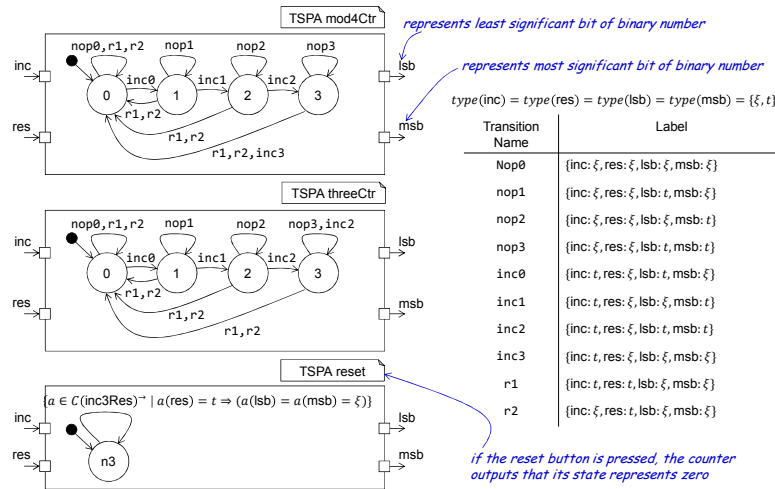


Figure A.1: Three TSPAs modeling the behaviors of binary counters.

APPENDIX A TIME-SYNCHRONOUS PORT AUTOMATA FOR EXPERIMENTAL EVALUATIONS

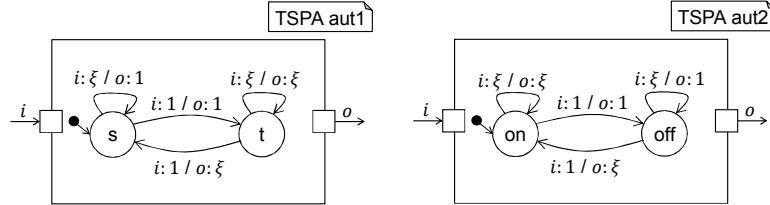


Figure A.2: Two simple TSPAs.

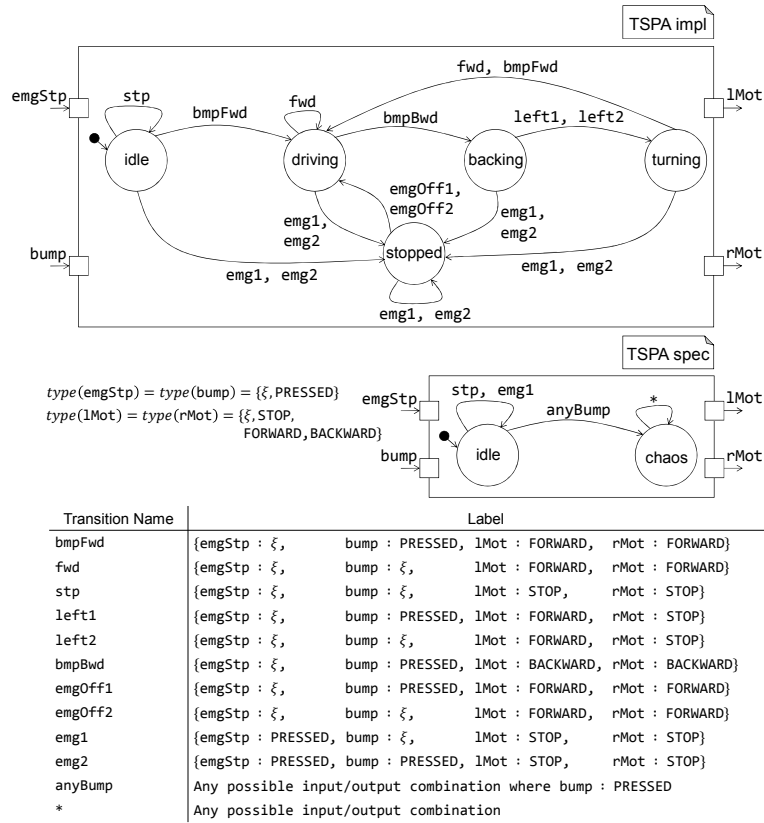


Figure A.3: Two TSPAs modeling the behaviors of a mobile robot.

Appendix B

Feature Diagrams for Experimental Evaluations

This appendix presents the FDs used in the experimental evaluations for the FD semantic differencing operator and the model repair instantiations for the FD language. The FDs are presented in the common graphical notation.

Figure B.1 depicts three FDs modeling the valid configurations of a car. The FD `car` is also used as an example in Section 4.1. The FDs `car1` and `car2` are also used as examples in Section 7.1.3.

Figure B.2 depicts three FDs modeling the valid configurations of a tablet computer. The three FDs are also used as examples in Section 4.3.

Figure B.3 depicts four FDs using all syntactic FD modeling elements. The FDs `fd1` and `fd2` are also used as examples in Section 4.3. The FDs `fd3` and `fd4` are also used as examples in Section 8.3.4.

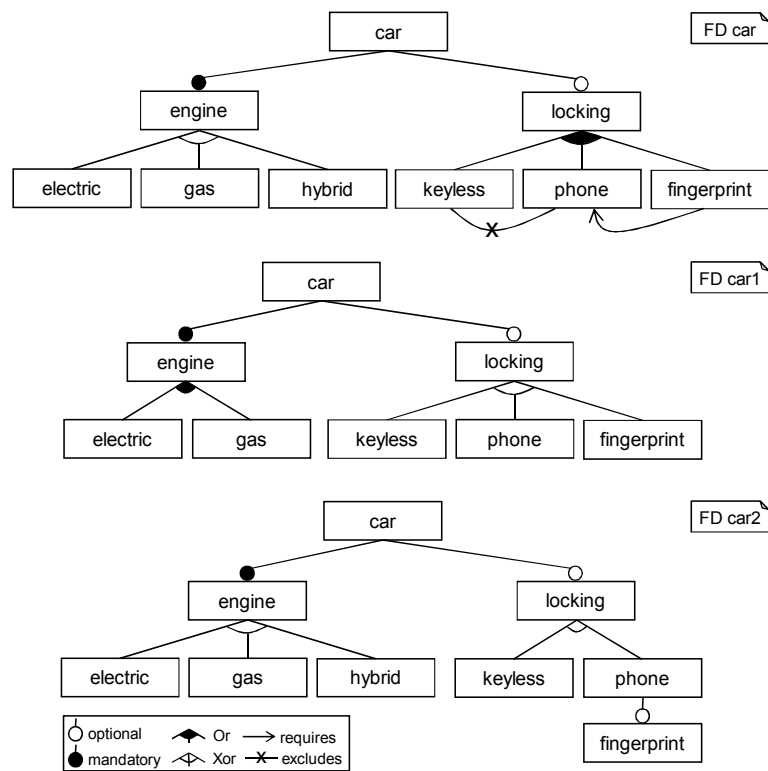


Figure B.1: Three FDs modeling the valid configurations of a car.

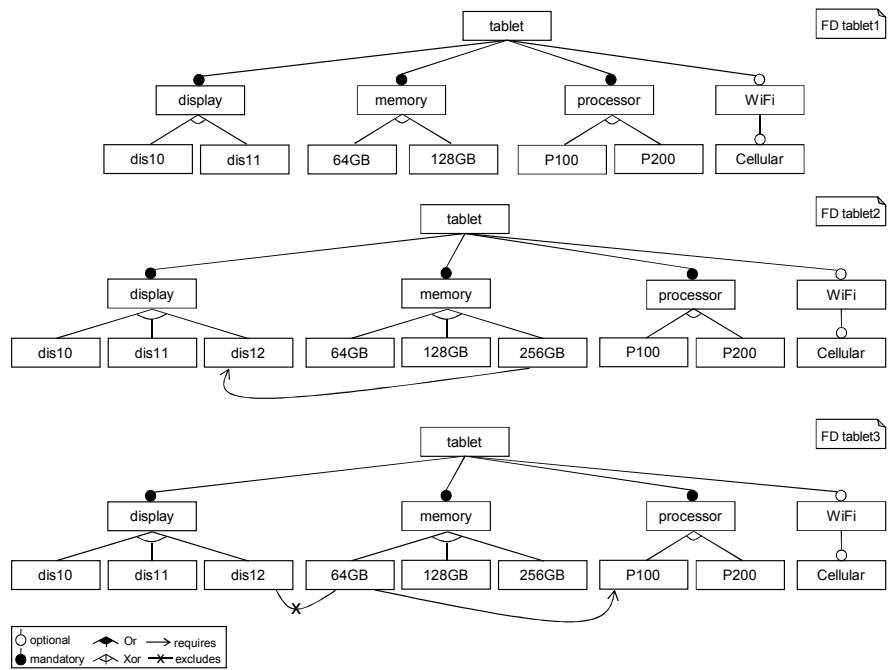


Figure B.2: Three FDs modeling the valid configurations of a car.

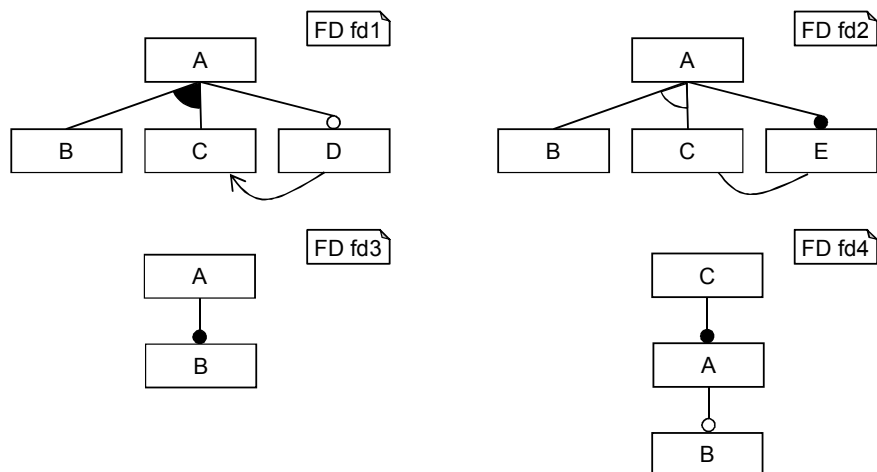


Figure B.3: Four FDs modeling using all syntactic FD modeling elements.

Appendix C

Sequence Diagrams for Experimental Evaluations

This appendix presents the SDs used in the experimental evaluations for the SD semantic differencing operator and the model repair instantiations for the SD language. The SDs are presented in the common graphical notation.

Figure C.1 depicts two simple FDs. The FDs `sd1` and `sd2` depicted in Figure C.1 are also used as an example in Section 8.4.4.

Figure C.2 depicts five SDs modeling the the interactions between objects in a software system implementing a mobile service robot. The three SDs `rob1`, `rob2`, and `rob3` are also used as examples in Section 5.3. The SDs `rob4` and `rob5` are also used as examples in Section 7.1.

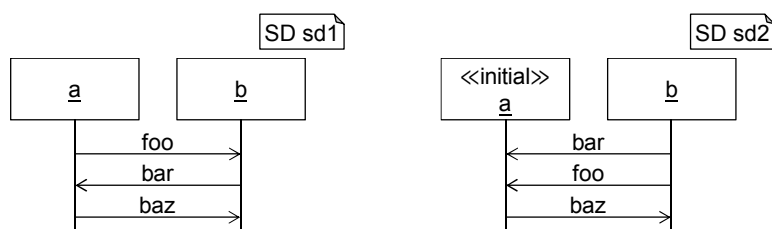


Figure C.1: Two simple SDs.

APPENDIX C SEQUENCE DIAGRAMS FOR EXPERIMENTAL EVALUATIONS

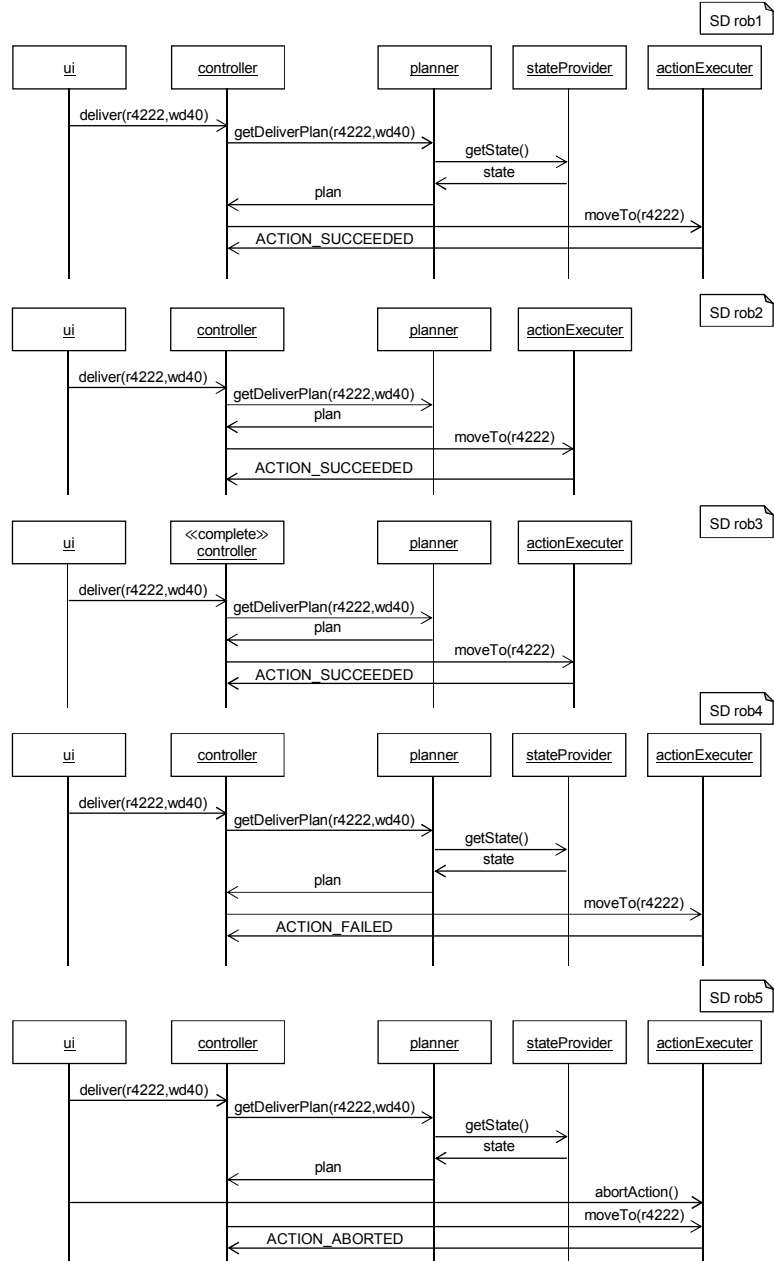


Figure C.2: Five SDs modeling the interactions between objects in a software system implementing a mobile service robot.

Appendix D

Activity Diagrams for Experimental Evaluations

This appendix presents the ADs used in the experimental evaluations for the AD semantic differencing operator and the model repair instantiations for the AD language. The ADs are presented in the common graphical notation.

Figure D.1 depicts two simple ADs. The ADs `ad1` and `ad2` depicted in Figure D.1 are also used as an example in Section 8.5.4.

Figure D.2 depicts the two ADs `hire1` and `hire2`. The ADs depicted in Figure D.2 are also used as an example in Section 6.1 and Section 6.2.

Figure D.3 and Figure D.4 depict four ADs modeling workflows for handling incoming claims in an insurance company. The ADs `claim1` and `claim2` are also used as examples in Section 6.1 and Section 6.3. The ADs `claim3` and `claim4` are also used as examples in Section 7.1.1.

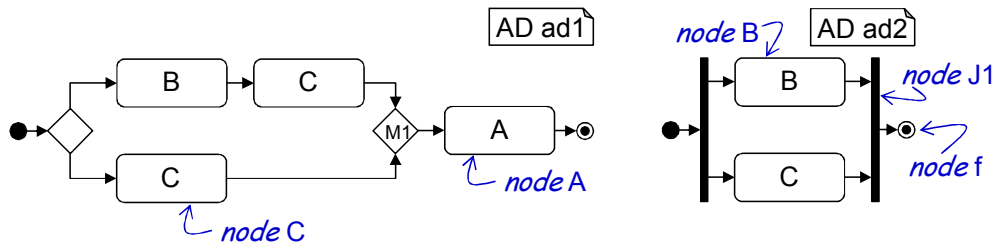


Figure D.1: Two simple ADs.

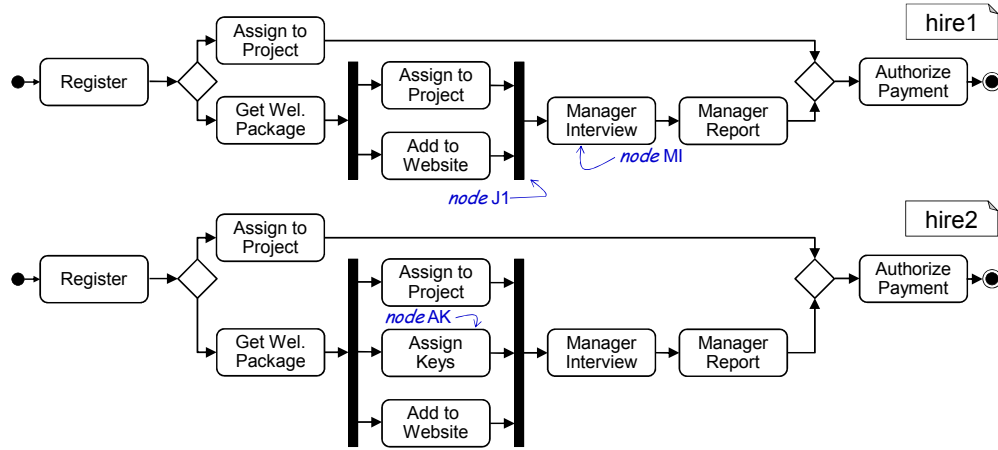


Figure D.2: Two ADs modeling workflows for registering new employees.

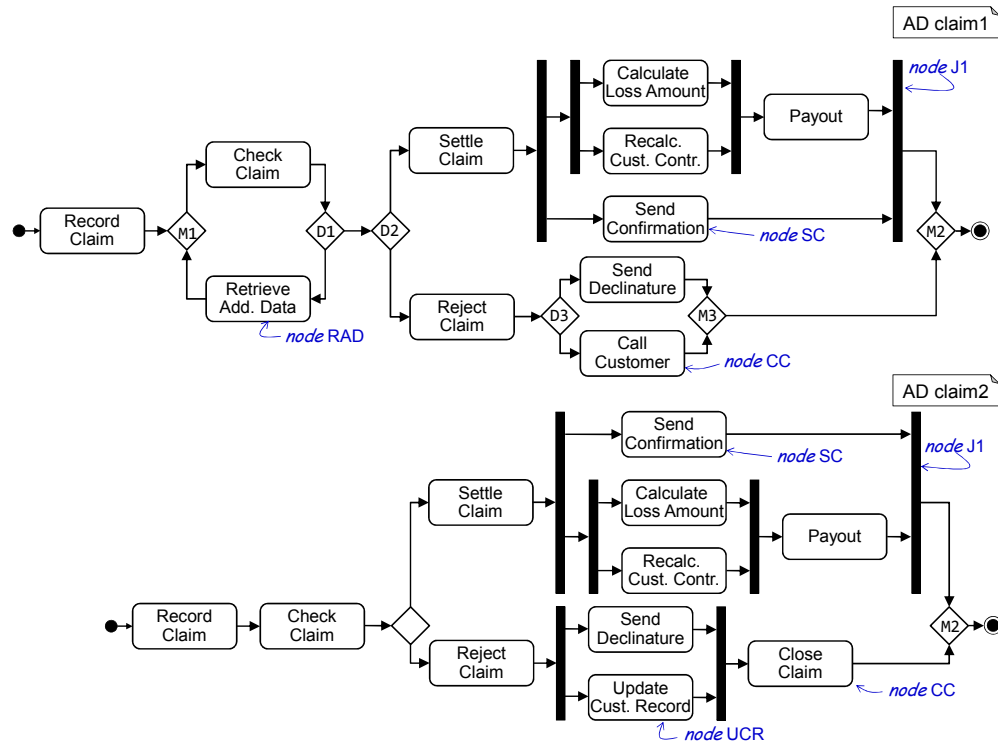


Figure D.3: The ADs `claim1` and `claim2` model workflows for handling incoming claims in an insurance company.

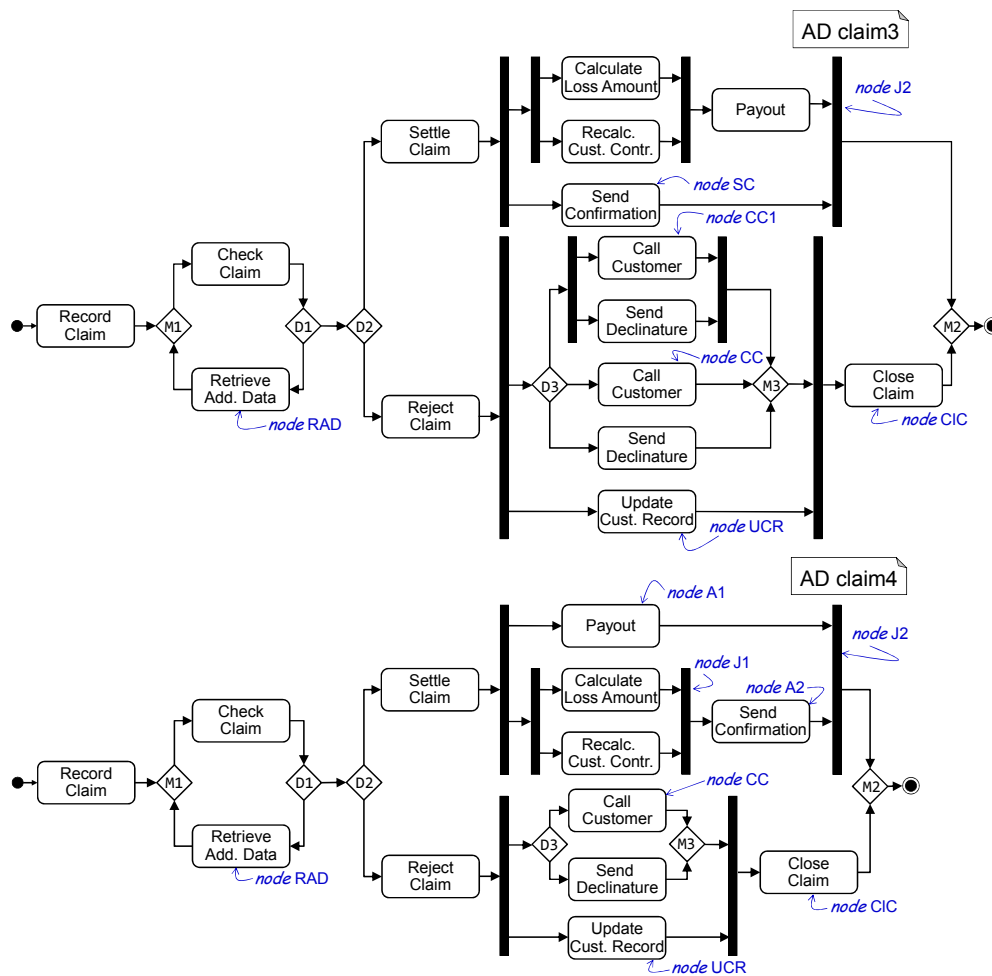


Figure D.4: The ADs claim3 and claim4 model workflows for handling incoming claims in an insurance company.

List of Figures

2.1	The conceptual parts of the complete definition of a modeling language inspired by [CGR09, Grö10].	14
2.2	The template for defining change operations.	23
2.3	An example template defining NFA change operations.	24
3.1	The TSPA <code>switch</code> models the behavior of a toggle switch.	34
3.2	The three TSPAs <code>mod4Ctr</code> , <code>threeCtr</code> , and <code>reset</code> . The TSPAs <code>mod4Ctr</code> and <code>threeCtr</code> represent implementations with incomparable semantics. Both implementations are refinements of the TSPA <code>reset</code> modeling a highly underspecified specification.	40
3.3	The number of states and transitions of the TSPAs used for the semantic differencing experiments.	42
3.4	The time needed by the semantic differencing operator for semantic differencing of the pairs of example TSPAs.	43
3.5	TSPA change operations and their properties.	44
3.6	State-addition operations $addS_s$ for all state names $s \in U_N$	44
3.7	State-deletion operations $delS_d$ for all state names $d \in U_N$	46
3.8	Transition-addition operations $addT_{s,t,a}$ for all state names $s, t \in U_N$ and channel assignments a	47
3.9	Transition-deletion operations $delT_{s,t,a}$ for all state names $s, t \in U_N$ and channel assignments a	48
3.10	Input-channel-addition operations $addIC_c$ for all channel names $c \in C$. . .	49
3.11	Output-channel-addition operations $addOC_c$ for all channel names $c \in C$. .	50
3.12	Channel deletion operations $delC_c$ for all channel names $c \in C$	51
3.13	Initial-state-change operations $chgI_s$ for all state names $s \in C$	52
4.1	An FD containing all FD modeling elements inspired by a FD from [MR15].	58
4.2	Three FDs modeling the valid configurations of a tablet computer.	65
4.3	Two FDs and the Alloy module resulting from translating the FDs.	68
4.4	The number of features in the example FDs and the number of constraints in the Alloy predicates generated from the FDs.	69
4.5	The time needed by the semantic differencing operator for semantic differencing of the pairs of example FDs.	70
4.6	Feature diagram change operation properties.	71

LIST OF FIGURES

4.7	Feature-addition operations $addF_{f,g}$ for all feature names $f, g \in U_N$	72
4.8	Feature-deletion operations $delF_f$ for all feature names $f \in U_N$	73
4.9	Implies-constraint-addition operations $addI_{f,g}$ for feature names $f, g \in U_N$	74
4.10	Implies-constraint-deletion operations $delI_{f,g}$ for feature names $f, g \in U_N$	75
4.11	Excludes-constraint-addition operations $addX_{f,g}$ for features $f, g \in U_N$	76
4.12	Excludes-constraint-deletion operations $delX_{f,g}$ for features $f, g \in U_N$	77
4.13	Or-group-creation operations $createOr_f$ for all feature names $f \in U_N$	78
4.14	Xor-to-or-conversion operations $xor2or_{p,G}$ for all feature names $p \in U_N$ and finite sets of feature names $G \subseteq U_N$	79
4.15	Or-to-xor-conversion operations $or2xor_{p,G}$ for all feature names $p \in U_N$ and finite sets of feature names $G \subseteq U_N$	81
4.16	Mandatory feature to optional feature conversion operations $man2opt_f$ for all feature names $f \in U_N$	82
4.17	Optional-to-mandatory-conversion operations $opt2man_f$ for all feature names $f \in U_N$	83
4.18	Feature-group-insertion operations $add2Grp_{f,G}$ for all feature names $f \in$ U_N , and finite sets of feature names $G \subseteq U_N$	84
4.19	Example FDs illustrating that group-insertion operations and group-exclusion operations are neither refining nor generalizing.	84
4.20	Feature-group-exclusion operations $exclGrp_f$ for all feature names $f \in U_N$	85
4.21	Root-rename operations $rnmRoot_f$ for all feature names $f \in U_N$	86
5.1	The SD <code>bid1</code> inspired by a similar SD from [Rum16].	92
5.2	The SD <code>bid2</code> only contains the semantically relevant modeling elements of the SD <code>bid1</code> depicted in Figure 5.1.	93
5.3	An SD that contains all SD modeling elements.	95
5.4	A simple SD containing two interactions and an object tagged as complete. . . .	96
5.5	Consecutive SD versions <code>rob1</code> , <code>rob2</code> , and <code>rob3</code> modeling systems runs of an autonomous service robot with pairwise different but not disjoint semantics.	98
5.6	Four SDs <code>sd1</code> , <code>sd2</code> , <code>sd3</code> , and <code>sd4</code> where <code>sd1</code> is not a refinement of <code>sd2</code> and <code>sd3</code> is not a refinement of <code>sd4</code>	102
5.7	The automata constructed for semantic differencing of the SDs <code>rob1</code> and <code>rob2</code> depicted in Figure 5.5.	104
5.8	The automata constructed for semantic differencing of the SDs <code>rob2</code> and <code>rob3</code> depicted in Figure 5.5.	105
5.9	The number of states and transitions of the NFAs constructed from the SDs for semantic differencing.	106
5.10	Abbreviations of action and object names used for describing the diff witnesses presented in Figure 5.11.	106

5.11	The time needed by the semantic differencing operator for semantic differencing of the pairs of example SDs and the traces of the computed diff witnesses.	108
5.12	Sequence diagram change operation properties.	109
5.13	Object-addition operations $addO_o$ for all object names $o \in \mathcal{O}$	109
5.14	Object-addition operations are not generalizing.	111
5.15	Object-deletion operations $delO_o$ for all object names $o \in \mathcal{O}$	111
5.16	Tag-object-as-complete operations $addOC_o$ for all object names $o \in \mathcal{O}$	112
5.17	Untag-object-as-complete operations $delOC_o$ for all object names $o \in \mathcal{O}$	113
5.18	Tag-object-as-visible operations $addOV_o$ for all object names $o \in \mathcal{O}$	114
5.19	Untag-object-as-visible operations $delOV_o$ for all object names $o \in \mathcal{O}$	116
5.20	Tag-object-as-initial operations $addOI_o$ for all object names $o \in \mathcal{O}$	117
5.21	Untag-object-as-initial operations $delOI_o$ for all object names $o \in \mathcal{O}$	118
5.22	Action-addition operations $addAct_a$ for all action names $a \in \mathcal{A}$	119
5.23	Action-deletion operations $delAct_a$ for all action names $a \in \mathcal{A}$	120
5.24	Interaction-addition operations $addIA_{i,o,a,p}$ for all indices $i \in \mathbb{N}$, object names $o, p \in \mathcal{O}$, and action names $a \in \mathcal{A}$	121
5.25	Interaction-addition and interaction-deletion operations are neither refining nor generalizing.	122
5.26	Interaction deletion operations $delIA_i$ for all indices $i \in \mathbb{N}$	122
6.1	An AD adapted from [MR18] that contains all AD modeling elements.	130
6.2	Two activity diagrams adapted from [MRR11b].	133
6.3	Reachable part of the NFA associated with <code>hire1</code> (cf. Figure 6.2).	136
6.4	An AD adapted from [MR18].	138
6.5	The number of nodes and transitions of the AGs and the number of states and transitions of the NFAs constructed from the AGs for semantic differencing.	138
6.6	The time needed by the semantic differencing operator for semantic differencing of the pairs of example AGs.	140
6.7	Activity graph change operation properties.	141
6.8	Label-addition operations $addL_k$ for all action labels $k \in U_N$	142
6.9	Label-deletion operations $delL_k$ for all action labels $k \in U_N$	143
6.10	Action-insertion operations $addA_{x,y,a,k}$ for all node names $x, y, a \in U_N$ and action labels $k \in U_N$	144
6.11	Action-deletion operations $delA_a$ for all action node names $a \in U_N$	145
6.12	Xor-fragment-insertion operations $addXor_{x,y,d,m,a,k}$ for node names $x, y, d, m, a \in U_N$ and action labels $k \in U_N$	146
6.13	Xor-fragment-deletion operations $delXor_{d,m,a}$ for node names $d, m, a \in U_N$	147
6.14	And-fragment-insertion operations $addAnd_{x,y,f,j,a,k}$ for node names $x, y, f, j, a \in U_N$ and action labels $k \in U_N$	148

6.15	And-fragment-deletion operations $delAnd_{f,j,a}$ for node names $f, j, a \in U_N$.	149
6.16	Cyclic-fragment-insertion operations $addC_{x,y,d,m,a,k}$ for node names $x, y, d, m, a \in U_N$ and action labels $k \in U_N$.	150
6.17	Cyclic-fragment-deletion operations $delC_{d,m,a}$ for node names $d, m, a \in U_N$.	151
6.18	Fragment-branch-insertion operations $addFB_{n,m}$ for nodes $n, m \in U_N$.	152
6.19	The AG $ag2$ can be obtained from the AG $ag1$ by applying a fragment-branch-insertion operation. The AG $ag1$ can be obtained from the AG $ag2$ by applying a fragment-branch-deletion operation.	153
6.20	Fragment-branch-deletion operations $delFB_{n,m}$ for node names $n, m \in U_N$.	153
7.1	Two ADs taken from [KR18a] and inspired by [KGE09, KGFE08] modeling workflows in the context of an insurance company.	165
7.2	Two TSPAs adapted from similar automata presented in [KR18b] that initially appeared in [Rin14].	166
7.3	Two FDs adapted from [MR18, KR18b] and inspired by a similar FD example from [MR15].	168
7.4	Two SDs adapted inspired by similar SDs from [ABH ⁺ 17].	170
7.5	The FD $fd2$ does not refine the FD $fd1$.	172
7.6	Schematic illustration of the relations between models, model properties, and change operations.	173
7.7	Schematic representation of the relation between two change operations o, o' that induce an equally long shortest solution for a model repair problem instance using the model m .	177
7.8	Schematic representation of the quotient of the set of all change operations under the equivalence relation \sim_I where I is a model repair problem instance.	180
7.9	Schematic representation of a change operation that delays the solution for a model repair problem instance using the model m .	181
7.10	Schematic illustration of a repair-representative function mapping the model m to a set of change operations.	185
7.11	Schematic representation of a change sequence search tree for a repair problem instance that uses the model m .	187
7.12	Illustration of the models, the properties, and the change operations used in the proofs of Proposition 7.20, Proposition 7.23, and Proposition 7.24.	204
7.13	Illustration of the models, the properties, and the change operations used in the proofs of Proposition 7.25, Proposition 7.27, and Proposition 7.28.	208
7.14	Illustration of the models, the properties, and the change operations used in the proof of Proposition 7.30.	210
8.1	Two simple TSPAs and an excerpt of the change sequence search tree for computing a shortest solution for a TSPA generalization repair problem.	223

8.2	The execution times of Algorithm 2 - Algorithm 6 when given the TSPA refinement repair problem instances as inputs.	230
8.3	The shortest solutions computed by Algorithm 4 for the TSPA refinement repair problem instances.	231
8.4	Two simple FDs and an excerpt of the change sequence search tree for computing a shortest solution for an FD refinement repair problem. . . .	237
8.5	The execution times of Algorithm 2 - Algorithm 6 when given the FD refinement repair problem instances as inputs.	243
8.6	The shortest solutions computed by Algorithm 4 for the FD refinement repair problem instances.	244
8.7	Two simple SDs and an excerpt of the change sequence search tree for computing a shortest solution for an SD refinement repair problem. . . .	253
8.8	The execution times of Algorithm 2 - Algorithm 6 when given the SD refinement repair problem instances as inputs.	256
8.9	The shortest solutions computed by Algorithm 3 for the SD refinement repair problem instances.	258
8.10	Two simple ADs and an excerpt of the change sequence search tree for computing a shortest solution for an AD refactoring repair problem. . . .	267
8.11	The execution times of Algorithm 2 - Algorithm 6 when given the AD refinement repair problem instances as inputs.	271
8.12	The solutions computed by Algorithm 2 for the AD refinement repair problem instances.	272
9.1	Overview of the framework developed in this thesis.	276
A.1	Three TSPAs modeling the behaviors of binary counters.	301
A.2	Two simple TSPAs.	302
A.3	Two TSPAs modeling the behaviors of a mobile robot.	302
B.1	Three FDs modeling the valid configurations of a car.	304
B.2	Three FDs modeling the valid configurations of a car.	305
B.3	Four FDs modeling using all syntactic FD modeling elements.	305
C.1	Two simple SDs.	307
C.2	Five SDs modeling the interactions between objects in a software system implementing a mobile service robot.	308
D.1	Two simple ADs.	309
D.2	Two ADs modeling workflows for registering new employees.	310
D.3	The ADs <code>claim1</code> and <code>claim2</code> model workflows for handling incoming claims in an insurance company.	310

LIST OF FIGURES

D.4 The ADs <code>claim3</code> and <code>claim4</code> model workflows for handling incoming claims in an insurance company.	311
---	-----

Acronyms

AD activity diagram. 129

AG activity graph. 130

BA Büchi automaton. 10

DFID depth-first iterative-deepening. 189

FD feature diagram. 57

GPL general-purpose programming language. 3

MDD Model-driven development. 3

NFA nondeterministic finite automaton with epsilon moves. 9

SD sequence diagram. 91

TSPA time-synchronous port automaton. 33

UML Unified Modeling Language. 4

Glossary of Notation for Foundations

$A \times B$	Cartesian product of A and B	7
$s \& t$	Concatenation of the sequences s and t	8
$E_A(q)$	Epsilon closure of the state q in the NFA A	9
$dom(f)$	Domain of the total or partial function f	7
$f : A \rightharpoonup B$	f is a partial function from A to B	7
$f _R$	Restriction of the function f to the set R	8
$f : A \rightarrow B$	f is a total function from A to B	7
$f(x) = \perp$	Partial function f is undefined on the argument x	8
$A \cap B$	Intersection of the sets A and B	7
$\mathcal{L}_\omega(B)$	Language recognized by the BA B	10
$\mathcal{L}_*(A)$	Language recognized by the NFA A	10
\mathbb{N}	The set of natural numbers	7
$\wp(A)$	Powerset of the set A	7
$\wp_{fin}(A)$	Set of all finite subsets of the set A	7
$s \downarrow i$	Prefix of length i of sequence s	8
\sqsubseteq	Prefix relation over sequences	8
\sqsubset	True prefix relation over sequences	8
$a : s$	Result from prepending the symbol a in front of the sequence s	8
$A \setminus B$	Relative complement of the set B in the set A	7
$rt(s)$	Result from removing the first element from the non-empty sequence s	8
Σ^*	Set of all finite sequences over the set Σ	8
Σ^∞	Set of all infinite sequences over the set Σ	8
$ s $	Length of a sequence s	8

GLOSSARY OF NOTATION FOR FOUNDATIONS

\underline{s}	Set of natural numbers smaller than or equal to length of the sequence s	8
$s.i$	$(i + 1)$ -th element of the sequence s	8
$A \cup B$	Union of the sets A and B	7

General Glossary of Notation

$act(i)$	Action of the interaction i	93
\mathcal{A}	Infinite set of actions used in SDs	93
AG	Set of all AGs	132
$addA_{x,y,a,k}$	Action-insertion operation adding an action node a with label k between the connected nodes x and y	144
$delA_a$	Action-deletion operation deleting action node a	144
$delAnd_{f,j,a}$	And-fragment-deletion operation deleting the and-fragment defined by fork node f and join node j containing action a	148
$addAnd_{x,y,f,j,a,k}$	And-fragment-insertion operation adding an and-fragment defined by fork node f and join node j containing action a labeled k between the connected nodes x and y	147
$delC_{d,m,a}$	Cyclic-fragment-deletion operation deleting the cyclic-fragment defined by decision node d and merge node m containing action a	150
$addC_{x,y,d,m,a,k}$	Cyclic-fragment-insertion operation adding a cyclic-fragment defined by decision node d and merge node m containing action a labeled k between the connected nodes x and y	149
$delFB_{n,m}$	Fragment-branch-deletion operation deleting the transition (n, m) if (n, m) is an and-, a xor-, or a cyclic-fragment and the transition exists.	152
$addFB_{n,m}$	Fragment-branch-insertion operation adding a transition from n to m if (n, m) is an and-, a xor-, or a cyclic-fragment	152
$addL_k$	Label-addition operation adding label k	142
$delL_k$	Label-deletion operation deleting label k	143
$delXor_{d,m,a}$	Xor-fragment-deletion operation deleting the xor-fragment defined by decision node d and merge node m containing action a	146

$addXor_{x,y,d,m,a,k}$	Xor-fragment-insertion operation adding a xor-fragment defined by decision node d and merge node m containing action a labeled k between the connected nodes x and y	145
$A(ag)$	Set of action nodes of the AG ag	132
$D(ag)$	Set of decision nodes of the AG ag	132
$F(ag)$	Set of fork nodes of the AG ag	132
$J(ag)$	Set of join nodes of the AG ag	132
$M(ag)$	Set of merge nodes of the AG ag	132
$Bool$	Type of booleans including ξ	35
$ba(A)$	BA associated to TSPA A	39
O_{AD}	Complete change operation suite for ADs	154
O_{FD}	Complete change operation suite for FDs	86
O_{SD}	Complete change operation suite for SDs	123
O_{PA}	Complete change operation suite for TSPAs	53
$m \triangleright t \in M$	Change sequence t is applicable to model m	20
$m \triangleright t$	Application of the change sequence t to the model m	20
B^{\rightarrow}	Set of all channel assignments over set of channels B	35
\mathcal{C}	Set of all channels	35
Σ^{Ω}	Set of all communication histories over the channel signature Σ	37
$his(\sigma)$	Communication history produced by the execution σ	38
\overline{P}	The complement property of the property P	171
C	Feature configuration	60
Del_I	Set of change operations that delay the solution for the model repair problem instance I	181
$\delta_{ag}^{-}(n)$	Set of transitions of AG ag ending in node n	132
$\delta_{ag}^{+}(n)$	Set of transitions of AG ag starting in node n	132
$d_{\mathcal{P}}(m)$	Lengths of shortest solutions repairing model m towards satisfying property P	175
$execs(A)$	Set of all executions of TSPA A	38
$addX_{f,g}$	Excludes-constraint-addition operation adding an excludes constraint from feature f to feature g	76
$delX_{f,g}$	Excludes-constraint-deletion operation deleting an excludes constraint from feature f to feature g	77

$addF_{f,g}$	Feature-addition operation adding feature g as optional child of feature f	72
$delF_f$	Feature-deletion operation deleting feature f	73
$exclGrp_f$	Feature-group-exclusion operation excluding the feature f from its group	85
$add2Grp_{f,G}$	Feature-group-insertion operation adding the feature f to group G	83
$addI_{f,g}$	Implies-constraint-addition operation adding an implies constraint from feature f to feature g	74
$delI_{f,g}$	Implies-constraint-deletion operation deleting an implies constraint from feature f to feature g	75
$man2opt_f$	Mandatory-to-optional-conversion operation making the mandatory feature f optional	81
$opt2man_f$	Optional-to-mandatory-conversion operation making the optional feature f mandatory	82
$or2xor_{p,G}$	Or-to-xor-conversion operation converting or-group G of feature p to an xor-group	80
$createOr_f$	Or-group creation operation creating an or-group containing feature f	78
$rnmRoot_f$	Root-rename operation renaming the root to f	85
$xor2or_{p,G}$	Xor-to-or-conversion operation converting xor-group G of feature p to an or-group	79
$\mathcal{P}_{\supseteq}(\mathcal{L}, O, m)$	Generalizes problem for the modeling language \mathcal{L} , the change operation suite O , and the model m	216
$P_{\supseteq}(\mathcal{L}, m)$	Generalizes property for the modeling language \mathcal{L} and the model m	216
$[o]_I$	Equivalence class of change operation o under \sim_I	179
Q / \sim_I	Quotient of set of change operations Q under \sim_I	179
$o \sim_I o'$	Change operations o and o' induce an equally long shortest solution for the model repair problem instance I	178
$o \not\sim_I o'$	Change operations o and o' do not induce an equally long shortest solution for the model repair problem instance I	178
\mathcal{I}	Set of all possible interactions between objects in SDs	93
$I(r)$	Set of all possible interactions between objects and actions of system run r	95
\mathcal{L}	Modeling language	15

\mathcal{L}_{AD}	AD modeling language	154
\mathcal{L}_{FD}	FD modeling language	86
\mathcal{L}_{SD}	SD modeling language	123
\mathcal{L}_{PA}	TSPA modeling language	54
\mathcal{M}	Set of messages	35
M_{AD}	Set of all AD models	154
M_{FD}	Set of all FD models	59
M_{PA}	Set of all TSPA models	36
M_{SD}	Set of all SD models	94
$nfa(ag)$	NFA associated to the AG ag	134
$obj(i)$	Objects of the interaction i	93
\mathcal{O}	Infinite set of objects used in SDs	93
$parent(f)$	Parent of feature f in a FD that is clear from the context	59
$parent_{fd}(f)$	Parent of feature f in FD fd	59
$\mathcal{P}_{=}(L, O, m)$	Refactors problem for the modeling language L , the change operation suite O , and the model m	216
$P_{=}(L, m)$	Refactors property for the modeling language L and the model m	216
$\mathcal{P}_{\subseteq}(L, O, m)$	Refines problem for the modeling language L , the change operation suite O , and the model m	216
$P_{\subseteq}(L, m)$	Refines property for the modeling language L and the model m	216
\mathcal{R}	Repair-representative function	184
$addAct_a$	Action-addition operation adding action a	119
$delAct_a$	Action-deletion operation deleting action a	120
$addIA_{i,o,a,p}$	Interaction-addition operation adding interaction (o, a, p) at position i	121
$delIA_i$	Interaction-deletion operation deleting the interaction at position i	122
$addO_o$	Object-addition operation adding object o	109
$delO_o$	Object-deletion operation deleting object o	110
$addOC_o$	Tag-as-complete operation tagging o as complete	112
$addOI_o$	Tag-as-initial operation tagging object o as initial	116
$addOV_o$	Tag-as-visible operation tagging object o as visible	114

$delOC_o$	Untag-as-complete operation untagging o as complete	113
$delOI_o$	Untag-as-initial operation untagging o as initial	118
$delOV_o$	Untag-as-visible operation untagging o as visible	115
$\delta(m, m')$	Semantic difference from model m to model m'	16
$\delta_{\mathcal{L}}(m, m')$	Semantic difference from model m to model m' in modeling language \mathcal{L}	16
Sem	Semantic domain	15
Sem_{FD}	Semantic domain of the FD modeling language	86
Sem_{AD}	Semantic domain of the AD modeling language	154
Sem_{SD}	Semantic domain of the SD modeling language	123
Sem_{PA}	Semantic domain of the TSPA modeling language	54
sem	Semantic mapping	15
$\llbracket ad \rrbracket^{AD}$	Semantics of the AD ad	154
$\llbracket fd \rrbracket^{FD}$	Semantics of the FD fd	60
$\llbracket sd \rrbracket^{SD}$	Semantics of the SD sd	96
$\llbracket A \rrbracket^{PA}$	Semantics of the TSPA A	38
M	Non-empty, countable set of models	15
$src(i)$	Source of the interaction i	93
Δ_{AD}	Syntactic differencing operator for ADs	156
Δ_{FD}	Syntactic differencing operator for FDs	87
Δ_{SD}	Syntactic differencing operator for SDs	124
Δ_{PA}	Syntactic differencing operator for TSPAs	53
$trg(i)$	Target of the interaction i	93
$traces(ag)$	Trace semantics of the AG ag	134
$T(\mathcal{R}, I)$	Change sequence search tree induced by repair-representative function \mathcal{R} and model repair problem instance I	186
$delC_c$	Channel-deletion operation deleting the channel c	51
$chngI_s$	Initial-state-change operation changing the initial state to s	52
$addIC_c$	Input-channel-addition operation adding the input channel c	49
$addOC_c$	Output-channel-addition operation adding the output channel c	50
$addS_s$	State-addition operation adding state s	44
$delS_d$	State-deletion operation deleting state d	45
$addT_{s,t,a}$	Transition-addition operation adding (s, a, t)	46
$delT_{s,t,a}$	Transition-deletion operation deleting (s, a, t)	47
$type$	Function mapping channels to their types	35

GENERAL GLOSSARY OF NOTATION

U_N	Countable and infinite set of names	22
ξ	Empty pseudo message	35

Index

- abstract syntaxes, 14
- accomplishable, 171
- action-addition operation, 119
- action-deletion operation, 120, 144
- action-insertion operation, 144
- actions, 93
- activity diagram, 129, 154
- activity graph, 130, 131
- and-fragment-deletion operation, 148
- and-fragment-insertion operation, 147

- Büchi automaton, 10
- bijective, 8

- cartesian product, 7
- change operation, 19
- change operation suite, 20
- change sequence, 19
- change sequence search tree, 186
- channel, 35
- channel assignment, 35
- channel signature, 35
- channel-deletion operation, 51
- child of a node, 11
- communication history, 37
- complete change operation suite, 20
- complete stereotype, 92, 94
- composition of properties, 202
- composition of repair problems, 202
- concatenation, 8
- concrete syntax, 14
- configuration, 57, 60
- consistent, 16
- countable, 9

- cyber-physical system, 33
- cycle, 11
- cyclic-fragment-deletion operation, 150
- cyclic-fragment-insertion operation, 149

- delays the solution, 180, 181
- depth-first iterative deepening, 189
- diff witness, 4, 16
- differencing operator, 3, 4, 17, 20
- domain, 7

- empty sequence, 8
- epsilon closure, 9
- equivalence class, 179
- equivalence relation, 7
- evolution step, 161
- excludes-constraint-addition, 76
- excludes-constraint-deletion, 77
- execution, 38

- feature diagram, 57, 59
- feature-addition operation, 72
- feature-deletion operation, 73
- feature-group-exclusion operation, 85
- feature-group-insertion operation, 83
- finite function, 8
- finite sequence, 8
- finitely branching, 11
- Focus, 33
- fragment-branch-insertion, 152
- function, 7
- function restriction, 8

- generalization, 17
- generalizes problem, 216

- generalizes property, 216
- generalizing change operation, 21
- grammar, 14
- graph, 11
- implies-constraint-addition, 74
- implies-constraint-deletion operation, 75
- inconsistent, 16
- induce equally long, 177
- infinite branch, 11
- infinite sequence, 8
- infinite tree, 11
- initial stereotype, 95
- initial-state-change operation, 52
- injective, 8
- input-channel-addition operation, 49
- interaction, 93
- interaction-addition operation, 121
- interaction-deletion operation, 122
- interactive system, 33
- intersection of sets, 7
- intial stereotype, 92
- inverse, 21
- König's Lemma, 11
- label-addition operation, 142
- label-deletion operation, 143
- language recognized by a BA, 10
- language recognized by an NFA, 10
- length of a sequence, 8
- mandatory-to-optional-conversion, 81
- metamodel, 14
- model, 3, 13
- model repair problem, 174, 202
- model repair problem instance, 174
- modeling language, 15
- natural numbers, 7
- Nondeterministic finite automaton, 9
- object-addition operation, 109
- object-deletion operation, 110
- objects, 93
- optional-to-mandatory-conversion, 82
- or-group-creation operation, 78
- or-to-xor-conversion operation, 80
- output-channel-addition operation, 50
- parent of a node, 11
- partial function, 7
- partition, 7
- path, 11
- powerset, 7
- prefix operator, 8
- prefix relation, 8
- property, 171
- property implication, 194
- property preserving, 192
- pseudo message, 35
- quotient, 179
- rafactors problem, 216
- reactive, 36
- refactoring, 17
- refactoring change operation, 21
- refactors property, 216
- refinement, 4, 17, 163
- refinement calculus, 163
- refinement step, 161, 163
- refines problem, 216
- refines property, 216
- refining change operation, 21
- reflexive, 7
- relative complement of sets, 7
- repair-representative function, 183
- repairing change sequence, 171
- repairs a model towards, 171
- root, 11
- root-rename operation, 85
- rooted tree, 11
- run of an NFA, 10

- semantic difference, 4, 16
- semantic differencing, 4, 161, 164
- semantic differencing operator, 4, 17
- semantic domain, 15
- semantic mapping, 15
- semantics of ADs, 154
- semantics of FDs, 60
- semantics of SDs, 96
- semantics of TSPAs, 38
- sequence diagram, 91, 94
- set of finite subsets, 7
- shortest repairing change sequence, 172
- shortest solution, 174
- solution, 174
- state-addition operation, 44
- state-deletion operation, 45
- stepwise refinement, 33
- stream, 37
- surjective, 8
- symmetric, 7
- syntactic difference, 20, 161
- syntactic differencing, 3, 163
- syntactic differencing operator, 3, 20
- syntax, 14
- system run, 95
- tag-object-as-complete operation, 112
- tag-object-as-initial operation, 116
- tag-object-as-visible operation, 114
- time-synchronous, 34
- time-synchronous port automaton, 35
- trace semantics, 134
- transition-addition operation, 46
- transition-deletion operation, 47
- transitive, 7
- tree, 11
- type, 35
- undefined, 8
- underspecification, 16
- union of sets, 7
- universe of names, 22
- untag-object-as-complete operation, 113
- untag-object-as-initial operation, 118
- untag-object-as-visible operation, 115
- valid in a feature diagram, 60
- valid in a sequence diagram, 96
- visible stereotype, 92, 95
- walk, 11
- xor-fragment-insertion operation, 145
- xor-to-or-conversion operation, 79

Related Interesting Work from the SE Group, RWTH Aachen

The following section gives an overview on related work done at the SE Group, RWTH Aachen. More details can be found on the website www.se-rwth.de/topics/ or in [HMR⁺19]. The work presented here mainly has been guided by our mission statement:

Our mission is to define, improve, and industrially apply *techniques*, *concepts*, and *methods* for *innovative and efficient development* of software and software-intensive systems, such that *high-quality* products can be developed in a *shorter period of time* and with *flexible integration* of *changing requirements*. Furthermore, we *demonstrate the applicability* of our results in various domains and potentially refine these results in a domain specific form.

Agile Model Based Software Engineering

Agility and modeling in the same project? This question was raised in [Rum04]: “Using an executable, yet abstract and multi-view modeling language for modeling, designing and programming still allows to use an agile development process.”, [JWCR18] addresses the question how digital and organizational techniques help to cope with physical distance of developers and [RRSW17] addresses how to teach agile modeling. Modeling will increasingly be used in development projects, if the benefits become evident early, e.g. with executable UML [Rum02] and tests [Rum03]. In [GKRS06], for example, we concentrate on the integration of models and ordinary programming code. In [Rum12] and [Rum16], the UML/P, a variant of the UML especially designed for programming, refactoring and evolution, is defined. The language workbench MontiCore [GKR⁺06, GKR⁺08, HR17] is used to realize the UML/P [Sch12]. Links to further research, e.g., include a general discussion of how to manage and evolve models [LRSS10], a precise definition for model composition as well as model languages [HKR⁺09] and refactoring in various modeling and programming languages [PR03]. In [FHR08] we describe a set of general requirements for model quality. Finally, [KRV06] discusses the additional roles and activities necessary in a DSL-based software development project. In [CEG⁺14] we discuss how to improve the reliability of adaptivity through models at runtime, which will allow developers to delay design decisions to runtime adaptation.

Artifacts in Complex Development Projects

Developing modern software solutions has become an increasingly complex and time consuming process. Managing the complexity, size, and number of the artifacts developed and used during a project together with their complex relationships is not trivial [BGRW17]. To keep track of relevant structures, artifacts, and their relations in order to be able e.g. to evolve or adapt models and their implementing code, the *artifact model* [GHR17] was introduced. [BGRW18] explains its applicability in systems engineering based on MDSE projects.

An artifact model basically is a meta-data structure that explains which kinds of artifacts, namely code files, models, requirements files, etc. exist and how these artifacts are related to each other. The artifact model therefore covers the wide range of human activities during the development down to fully automated, repeatable build scripts. The artifact model can be used to optimize parallelization during the development and building, but also to identify deviations of the real architecture and dependencies from the desired, idealistic architecture, for

cost estimations, for requirements and bug tracing, etc. Results can be measured using metrics or visualized as graphs.

Artificial Intelligence in Software Engineering

MontiAnna is a family of explicit domain specific languages for the concise description of the architecture of (1) a neural network, (2) its training, and (3) the training data [KNP⁺19]. We have developed a compositional technique to integrate neural networks into larger software architectures [KRRvW17] as standardized machine learning components [KPRS19]. This enables the compiler to support the systems engineer by automating the lifecycle of such components including multiple learning approaches such as supervised learning, reinforcement learning, or generative adversarial networks. According to [MRR11g] the semantic difference between two models are the elements contained in the semantics of the one model that are not elements in the semantics of the other model. A smart semantic differencing operator is an automatic procedure for computing diff witnesses for two given models. Smart semantic differencing operators have been defined for Activity Diagrams [MRR11a], Class Diagrams [MRR11d], Feature Models [DKMR19], Statecharts [DEKR19], and Message-Driven Component and Connector Architectures [BKRW17, BKRW19]. We also developed a modeling language-independent method for determining syntactic changes that are responsible for the existence of semantic differences [KR18].

We apply logic, knowledge representation and intelligent reasoning to software engineering to perform correctness proofs, execute symbolic tests or find counterexamples using a theorem prover. And we have applied it to challenges in intelligent flight control systems and assistance systems for air or road traffic management [KRRS19, HRR12] and based it on the core ideas of Broy's Focus theory [RR11, BR07]. Intelligent testing strategies have been applied to automotive software engineering [EJK⁺19, DGH⁺19, KMS⁺18], or more generally in systems engineering [DGH⁺18]. These methods are realized for a variant of SysML Activity Diagrams and Statecharts.

Machine Learning has been applied to the massive amount of observable data in energy management for buildings [FLP⁺11a, KLPR12] and city quarters [GLPR15] to optimize the operation efficiency and prevent unneeded CO2 emissions or reduce costs. This creates a structural and behavioral system theoretical view on cyber-physical systems understandable as essential parts of digital twins [RW18, BDH⁺20].

Generative Software Engineering

The UML/P language family [Rum12, Rum11, Rum16] is a simplified and semantically sound derivate of the UML designed for product and test code generation. [Sch12] describes a flexible generator for the UML/P based on the MontiCore language workbench [KRV10, GKR⁺06, GKR⁺08, HR17]. In [KRV06], we discuss additional roles necessary in a model-based software development project. [GKRS06, GHK⁺15a] discuss mechanisms to keep generated and handwritten code separated. In [Wei12], we demonstrate how to systematically derive a transformation language in concrete syntax. [HMSNRW16] presents how to generate extensible and statically type-safe visitors. In [MSNRR16], we propose the use of symbols for ensuring the validity of generated source code. [GMR⁺16] discusses product lines of template-based code generators. We also developed an approach for engineering reusable language components [HLMSN⁺15b, HLMSN⁺15a].

To understand the implications of executability for UML, we discuss needs and advantages of executable modeling with UML in agile projects in [Rum04], how to apply UML for testing in [Rum03], and the advantages and perils of using modeling languages for programming in [Rum02].

Unified Modeling Language (UML)

Starting with an early identification of challenges for the standardization of the UML in [KER99] many of our contributions build on the UML/P variant, which is described in the books [Rum16, Rum17] respectively [Rum12, Rum13] and is implemented in [Sch12]. Semantic variation points of the UML are discussed in [GR11]. We discuss formal semantics for UML [BHP⁺98] and describe UML semantics using the “System Model” [BCGR09a], [BCGR09b], [BCR07b] and [BCR07a]. Semantic variation points have, e.g., been applied to define class diagram semantics [CGR08]. A precisely defined semantics for variations is applied, when checking variants of class diagrams [MRR11c] and objects diagrams [MRR11e] or the consistency of both kinds of diagrams [MRR11f]. We also apply these concepts to activity diagrams [MRR11b] which allows us to check for semantic differences of activity diagrams [MRR11a]. The basic semantics for ADs and their semantic variation points is given in [GRR10]. We also discuss how to ensure and identify model quality [FHR08], how models, views and the system under development correlate to each other [BGH⁺98], and how to use modeling in agile development projects [Rum04], [Rum02]. The question how to adapt and extend the UML is discussed in [PFR02] describing product line annotations for UML and more general discussions and insights on how to use meta-modeling for defining and adapting the UML are included in [EFLR99], [FELR98] and [SRVK10].

Domain Specific Languages (DSLs)

Computer science is about languages. Domain Specific Languages (DSLs) are better to use, but need appropriate tooling. The MontiCore language workbench [GKR⁺06, KRV10, Kra10, GKR⁺08, HR17] allows the specification of an integrated abstract and concrete syntax format [KRV07b, HR17] for easy development. New languages and tools can be defined in modular forms [KRV08, GKR⁺07, Völ11, HLMSN⁺15b, HLMSN⁺15a, HRW18, BEK⁺18a, BEK⁺18b, BEK⁺19] and can, thus, easily be reused. We discuss the roles in software development using domain specific languages in [KRV14]. [Wei12] presents a tool that allows to create transformation rules tailored to an underlying DSL. Variability in DSL definitions has been examined in [GR11, GMR⁺16]. [BDL⁺18] presents a method to derive internal DSLs from grammars. In [BJRW18], we discuss the translation from grammars to accurate metamodels. Successful applications have been carried out in the Air Traffic Management [ZPK⁺11] and television [DHH⁺20] domains. Based on the concepts described above, meta modeling, model analyses and model evolution have been discussed in [LRSS10] and [SRVK10]. DSL quality [FHR08], instructions for defining views [GHK⁺07], guidelines to define DSLs [KKP⁺09] and Eclipse-based tooling for DSLs [KRV07a] complete the collection.

Software Language Engineering

For a systematic definition of languages using composition of reusable and adaptable language components, we adopt an engineering viewpoint on these techniques. General ideas on how to

engineer a language can be found in the GeMoC initiative [CBCR15, CCF⁺15] and the concern-oriented language development approach [CKM⁺18]. As said, the MontiCore language workbench provides techniques for an integrated definition of languages [KRV07b, Kra10, KRV10, HR17, HRW18, BEK⁺19]. In [SRVK10] we discuss the possibilities and the challenges using metamodels for language definition. Modular composition, however, is a core concept to reuse language components like in MontiCore for the frontend [Völ11, KRV08, HLMSN⁺15b, HLMSN⁺15a, HMSNRW16, HR17, BEK⁺18a, BEK⁺18b, BEK⁺19] and the backend [RRRW15, MSNRR16, GMR⁺16, HR17, BEK⁺18b]. In [GHK⁺15b, GHK⁺15a], we discuss the integration of handwritten and generated object-oriented code. [KRV14] describes the roles in software development using domain specific languages. Language derivation is to our believe a promising technique to develop new languages for a specific purpose that rely on existing basic languages [HRW18]. How to automatically derive such a transformation language using concrete syntax of the base language is described in [HRW15, Wei12] and successfully applied to various DSLs. We also applied the language derivation technique to tagging languages that decorate a base language [GLRR15] and delta languages [HHK⁺15a, HHK⁺13], where a delta language is derived from a base language to be able to constructively describe differences between model variants usable to build feature sets. The derivation of internal DSLs from grammars is discussed in [BDL⁺18] and a translation of grammars to accurate metamodels in [BJRW18].

Modeling Software Architecture & the MontiArc Tool

Distributed interactive systems communicate via messages on a bus, discrete event signals, streams of telephone or video data, method invocation, or data structures passed between software services. We use streams, statemachines and components [BR07] as well as expressive forms of composition and refinement [PR99, RW18] for semantics. Furthermore, we built a concrete tooling infrastructure called MontiArc [HRR12] for architecture design and extensions for states [RRW13b]. In [RRW13a], we introduce a code generation framework for MontiArc. MontiArc was extended to describe variability [HRR⁺11] using deltas [HRRS11, HKR⁺11] and evolution on deltas [HRRS12]. Other extensions are concerned with modeling cloud architectures [NPR13] and with the robotics domain [AHRW17a, AHRW17b]. [GHK⁺07] and [GHK⁺08a] close the gap between the requirements and the logical architecture and [GKPR08] extends it to model variants. [MRR14b] provides a precise technique to verify consistency of architectural views [Rin14, MRR13] against a complete architecture in order to increase reusability. We discuss the synthesis problem for these views in [MRR14a]. Co-evolution of architecture is discussed in [MMR10] and modeling techniques to describe dynamic architectures are shown in [HRR98, BHK⁺17, KKR19].

Compositionality & Modularity of Models

[HKR⁺09] motivates the basic mechanisms for modularity and compositionality for modeling. The mechanisms for distributed systems are shown in [BR07, RW18] and algebraically underpinned in [HKR⁺07]. Semantic and methodical aspects of model composition [KRV08] led to the language workbench MontiCore [KRV10, HR17] that can even be used to develop modeling tools in a compositional form [HR17, HLMSN⁺15b, HLMSN⁺15a, HMSNRW16, MSNRR16, HRW18, BEK⁺18a, BEK⁺18b, BEK⁺19]. A set of DSL design guidelines incorporates reuse through this form of composition [KKP⁺09]. [Völ11] examines the composition of context conditions respec-

tively the underlying infrastructure of the symbol table. Modular editor generation is discussed in [KRV07a]. [RRRW15] applies compositionality to Robotics control. [CBCR15] (published in [CCF⁺15]) summarizes our approach to composition and remaining challenges in form of a conceptual model of the “globalized” use of DSLs. As a new form of decomposition of model information we have developed the concept of tagging languages in [GLRR15]. It allows to describe additional information for model elements in separated documents, facilitates reuse, and allows to type tags.

Semantics of Modeling Languages

The meaning of semantics and its principles like underspecification, language precision and detailedness is discussed in [HR04]. We defined a semantic domain called “System Model” by using mathematical theory in [RKB95, BHP⁺98] and [GKR96, KRB96]. An extended version especially suited for the UML is given in [BCGR09b] and in [BCGR09a] its rationale is discussed. [BCR07a, BCR07b] contain detailed versions that are applied to class diagrams in [CGR08]. To better understand the effect of an evolved design, detection of semantic differencing as opposed to pure syntactical differences is needed [MRR10]. [MRR11a, MRR11b] encode a part of the semantics to handle semantic differences of activity diagrams and [MRR11f, MRR11f] compare class and object diagrams with regard to their semantics. In [BR07], a simplified mathematical model for distributed systems based on black-box behaviors of components is defined. Meta-modeling semantics is discussed in [EFLR99]. [BGH⁺97] discusses potential modeling languages for the description of an exemplary object interaction, today called sequence diagram. [BGH⁺98] discusses the relationships between a system, a view and a complete model in the context of the UML. [GR11] and [CGR09] discuss general requirements for a framework to describe semantic and syntactic variations of a modeling language. We apply these on class and object diagrams in [MRR11f] as well as activity diagrams in [GRR10]. [Rum12] defines the semantics in a variety of code and test case generation, refactoring and evolution techniques. [LRSS10] discusses evolution and related issues in greater detail. [RW18] discusses an elaborated theory for the modeling of underspecification, hierarchical composition, and refinement that can be practically applied for the development of CPS.

Evolution and Transformation of Models

Models are the central artifacts in model driven development, but as code they are not initially correct and need to be changed, evolved and maintained over time. Model transformation is therefore essential to effectively deal with models. Many concrete model transformation problems are discussed: evolution [LRSS10, MMR10, Rum04], refinement [PR99, KPR97, PR94], decomposition [PR99, KRW20], synthesis [MRR14a], refactoring [Rum12, PR03], translating models from one language into another [MRR11c, Rum12], and systematic model transformation language development [Wei12]. [Rum04] describes how comprehensible sets of such transformations support software development and maintenance [LRSS10], technologies for evolving models within a language and across languages, and mapping architecture descriptions to their implementation [MMR10]. Automaton refinement is discussed in [PR94, KPR97], refining pipe-and-filter architectures is explained in [PR99]. Refactorings of models are important for model driven engineering as discussed in [PR01, PR03, Rum12]. Translation between languages, e.g., from class diagrams into Alloy [MRR11c] allows for comparing class diagrams on a semantic level.

Variability and Software Product Lines (SPL)

Products often exist in various variants, for example cars or mobile phones, where one manufacturer develops several products with many similarities but also many variations. Variants are managed in a Software Product Line (SPL) that captures product commonalities as well as differences. Feature diagrams describe variability in a top down fashion, e.g., in the automotive domain [GHK⁺08a] using 150% models. Reducing overhead and associated costs is discussed in [GRJA12]. Delta modeling is a bottom up technique starting with a small, but complete base variant. Features are additive, but also can modify the core. A set of commonly applicable deltas configures a system variant. We discuss the application of this technique to Delta-MontiArc [HRR⁺11, HRR⁺11] and to Delta-Simulink [HKM⁺13]. Deltas can not only describe spacial variability but also temporal variability which allows for using them for software product line evolution [HRRS12]. [HHK⁺13] and [HRW15] describe an approach to systematically derive delta languages. We also apply variability modeling languages in order to describe syntactic and semantic variation points, e.g., in UML for frameworks [PFR02] and generators [GMR⁺16]. Furthermore, we specified a systematic way to define variants of modeling languages [CGR09], leverage features for compositional reuse [BEK⁺18b], and applied it as a semantic language refinement on Statecharts in [GR11].

Modeling for Cyber-Physical Systems (CPS)

Cyber-Physical Systems (CPS) [KRS12] are complex, distributed systems which control physical entities. In [RW18], we discuss how an elaborated theory can be practically applied for the development of CPS. Contributions for individual aspects range from requirements [GRJA12], complete product lines [HRRW12], the improvement of engineering for distributed automotive systems [HRR12], autonomous driving [BR12a, KKR19], and digital twin development [BDH⁺20] to processes and tools to improve the development as well as the product itself [BBR07]. In the aviation domain, a modeling language for uncertainty and safety events was developed, which is of interest for the European airspace [ZPK⁺11]. A component and connector architecture description language suitable for the specific challenges in robotics is discussed in [RRW13b, RRW14]. In [RRW13a], we describe a code generation framework for this language. Monitoring for smart and energy efficient buildings is developed as Energy Navigator toolset [KPR12, FPPR12, KLPR12].

Model-Driven Systems Engineering (MDSysE)

Applying models during Systems Engineering activities is based on the long tradition on contributing to systems engineering in automotive [GHK⁺08b], which culminated in a new comprehensive model-driven development process for automotive software [KMS⁺18, DGH⁺19]. We leveraged SysML to enable the integrated flow from requirements to implementation to integration. To facilitate modeling of products, resources, and processes in the context of Industry 4.0, we also conceived a multi-level framework for machining based on these concepts [BKL⁺18]. Research within the excellence cluster Internet of Production considers fast decision making at production time with low latencies using contextual data traces of production systems, also known as Digital Shadows (DS) [SHH⁺20]. We have investigated how to derive Digital Twins (DTs) for injection molding [BDH⁺20], how to generate interfaces between a cyber-physical system and its DT [KMR⁺20] and have proposed model-driven architectures for DT cockpit engineering [DMR⁺20].

State Based Modeling (Automata)

Today, many computer science theories are based on statemachines in various forms including Petri nets or temporal logics. Software engineering is particularly interested in using statemachines for modeling systems. Our contributions to state based modeling can currently be split into three parts: (1) understanding how to model object-oriented and distributed software using statemachines resp. Statecharts [GKR96, BCR07b, BCGR09b, BCGR09a], (2) understanding the refinement [PR94, RK96, Rum96, RW18] and composition [GR95, GKR96, RW18] of statemachines, and (3) applying statemachines for modeling systems. In [Rum96, RW18] constructive transformation rules for refining automata behavior are given and proven correct. This theory is applied to features in [KPR97]. Statemachines are embedded in the composition and behavioral specification concepts of Focus [GKR96, BR07]. We apply these techniques, e.g., in MontiArcAutomaton [RRW13a, RRW14, RRW13a, RW18] as well as in building management systems [FLP⁺11b].

Model-Based Assistance and Information Services (MBAIS)

Assistive systems are a special type of information system: they (1) provide situational support for human behaviour (2) based on information from previously stored and real-time monitored structural context and behaviour data (3) at the time the person needs or asks for it [HMR⁺19]. To create them, we follow a model centered architecture approach [MMR⁺17] which defines systems as a compound of various connected models. Used languages for their definition include DSLs for behavior and structure such as the human cognitive modeling language [MM13], goal modeling languages [MRV20] or UML/P based languages [MNRV19]. [MM15] describes a process how languages for assistive systems can be created.

We have designed a system included in a sensor floor able to monitor elderlies and analyze impact patterns for emergency events [LMK⁺11]. We have investigated the modeling of human contexts for the active assisted living and smart home domain [MS17] and user-centered privacy-driven systems in the IoT domain in combination with process mining systems [MKM⁺19], differential privacy on event logs of handling and treatment of patients at a hospital [MKB⁺19], the mark-up of online manuals for devices [SM18] and websites [SM20], and solutions for privacy-aware environments for cloud services [ELR⁺17] and in IoT manufacturing [MNRV19]. The user-centered view on the system design allows to track who does what, when, why, where and how with personal data, makes information about it available via information services and provides support using assistive services.

Modelling Robotics Architectures and Tasks

Robotics can be considered a special field within Cyber-Physical Systems which is defined by an inherent heterogeneity of involved domains, relevant platforms, and challenges. The engineering of robotics applications requires composition and interaction of diverse distributed software modules. This usually leads to complex monolithic software solutions hardly reusable, maintainable, and comprehensible, which hampers broad propagation of robotics applications. The MontiArcAutomaton language [RRW13a] extends the ADL MontiArc and integrates various implemented behavior modeling languages using MontiCore [RRW13b, RRW14, RRRW15, HR17] that perfectly fit robotic architectural modeling. The LightRocks [THR⁺13] framework allows robotics experts and laymen to model robotic assembly tasks. In [AHRW17a, AHRW17b], we

define a modular architecture modeling method for translating architecture models into modules compatible to different robotics middleware platforms.

Automotive, Autonomic Driving & Intelligent Driver Assistance

Introducing and connecting sophisticated driver assistance, infotainment and communication systems as well as advanced active and passive safety-systems result in complex embedded systems. As these feature-driven subsystems may be arbitrarily combined by the customer, a huge amount of distinct variants needs to be managed, developed and tested. A consistent requirements management that connects requirements with features in all phases of the development for the automotive domain is described in [GRJA12]. The conceptual gap between requirements and the logical architecture of a car is closed in [GHK⁺07, GHK⁺08a]. [HKM⁺13] describes a tool for delta modeling for Simulink [HKM⁺13]. [HRRW12] discusses means to extract a well-defined Software Product Line from a set of copy and paste variants. [RSW⁺15] describes an approach to use model checking techniques to identify behavioral differences of Simulink models. In [KKR19], we introduce a framework for modeling the dynamic reconfiguration of component and connector architectures and apply it to the domain of cooperating vehicles. Quality assurance, especially of safety-related functions, is a highly important task. In the Carolo project [BR12a, BR12b], we developed a rigorous test infrastructure for intelligent, sensor-based functions through fully-automatic simulation [BBR07]. This technique allows a dramatic speedup in development and evolution of autonomous car functionality, and thus enables us to develop software in an agile way [BR12a]. [MMR10] gives an overview of the current state-of-the-art in development and evolution on a more general level by considering any kind of critical system that relies on architectural descriptions. As tooling infrastructure, the SSElab storage, versioning and management services [HKR12] are essential for many projects.

Smart Energy Management

In the past years, it became more and more evident that saving energy and reducing CO₂ emissions is an important challenge. Thus, energy management in buildings as well as in neighbourhoods becomes equally important to efficiently use the generated energy. Within several research projects, we developed methodologies and solutions for integrating heterogeneous systems at different scales. During the design phase, the Energy Navigators Active Functional Specification (AFS) [FPPR12, KPR12] is used for technical specification of building services already. We adapted the well-known concept of statemachines to be able to describe different states of a facility and to validate it against the monitored values [FLP⁺11b]. We show how our data model, the constraint rules, and the evaluation approach to compare sensor data can be applied [KLPR12].

Cloud Computing & Enterprise Information Systems

The paradigm of Cloud Computing is arising out of a convergence of existing technologies for web-based application and service architectures with high complexity, criticality, and new application domains. It promises to enable new business models, to lower the barrier for web-based innovations and to increase the efficiency and cost-effectiveness of web development [KRR14]. Application classes like Cyber-Physical Systems and their privacy [HHK⁺14, HHK⁺15b], Big

Data, App, and Service Ecosystems bring attention to aspects like responsiveness, privacy and open platforms. Regardless of the application domain, developers of such systems are in need for robust methods and efficient, easy-to-use languages and tools [KRS12]. We tackle these challenges by perusing a model-based, generative approach [NPR13]. The core of this approach are different modeling languages that describe different aspects of a cloud-based system in a concise and technology-agnostic way. Software architecture and infrastructure models describe the system and its physical distribution on a large scale. We apply cloud technology for the services we develop, e.g., the SSELab [HKR12] and the Energy Navigator [FPPR12, KPR12] but also for our tool demonstrators and our own development platforms. New services, e.g., collecting data from temperature, cars etc. can now easily be developed.

Model-Driven Engineering of Information Systems

Information Systems provide information to different user groups as main system goal. Using our experiences in the model-based generation of code with MontiCore [KRV10, HR17], we developed several generators for such data-centric information systems. *MontiGem* [AMN⁺20] is a specific generator framework for data-centric business applications that uses standard models from UML/P optionally extended by GUI description models as sources [GMN⁺20]. While the standard semantics of these modeling languages remains untouched, the generator produces a lot of additional functionality around these models. The generator is designed flexible, modular and incremental, handwritten and generated code pieces are well integrated [GHK⁺15a], tagging of existing models is possible [GLRR15], e.g., for the definition of roles and rights or for testing [DGH⁺18].

- [AHRW17a] Kai Adam, Katrin Hölldobler, Bernhard Rumpe, and Andreas Wortmann. Engineering Robotics Software Architectures with Exchangeable Model Transformations. In *International Conference on Robotic Computing (IRC'17)*, pages 172–179. IEEE, April 2017.
- [AHRW17b] Kai Adam, Katrin Hölldobler, Bernhard Rumpe, and Andreas Wortmann. Modeling Robotics Software Architectures with Modular Model Transformations. *Journal of Software Engineering for Robotics (JOSER)*, 8(1):3–16, 2017.
- [AMN⁺20] Kai Adam, Judith Michael, Lukas Netz, Bernhard Rumpe, and Simon Varga. Enterprise Information Systems in Academia and Practice: Lessons learned from a MBSE Project. In *40 Years EMISA: Digital Ecosystems of the Future: Methodology, Techniques and Applications (EMISA '19)*, LNI P-304, pages 59–66. Gesellschaft für Informatik e.V., May 2020.
- [BBR07] Christian Basarke, Christian Berger, and Bernhard Rumpe. Software & Systems Engineering Process and Tools for the Development of Autonomous Driving Intelligence. *Journal of Aerospace Computing, Information, and Communication (JACIC)*, 4(12):1158–1174, 2007.
- [BCGR09a] Manfred Broy, María Victoria Cengarle, Hans Grönniger, and Bernhard Rumpe. Considerations and Rationale for a UML System Model. In K. Lano, editor, *UML 2 Semantics and Applications*, pages 43–61. John Wiley & Sons, November 2009.
- [BCGR09b] Manfred Broy, María Victoria Cengarle, Hans Grönniger, and Bernhard Rumpe. Definition of the UML System Model. In K. Lano, editor, *UML 2 Semantics and Applications*, pages 63–93. John Wiley & Sons, November 2009.
- [BCR07a] Manfred Broy, María Victoria Cengarle, and Bernhard Rumpe. Towards a System Model for UML. Part 2: The Control Model. Technical Report TUM-I0710, TU Munich, Germany, February 2007.
- [BCR07b] Manfred Broy, María Victoria Cengarle, and Bernhard Rumpe. Towards a System Model for UML. Part 3: The State Machine Model. Technical Report TUM-I0711, TU Munich, Germany, February 2007.
- [BDH⁺20] Pascal Bibow, Manuela Dalibor, Christian Hopmann, Ben Mainz, Bernhard Rumpe, David Schmalzing, Mauritius Schmitz, and Andreas Wortmann. Model-Driven Development of a Digital Twin for Injection Molding. In Schahram Dustdar, Eric Yu, Camille Salinesi, Dominique Rieu, and Vik Pant, editors, *International Conference on Advanced Information Systems Engineering (CAiSE'20)*, Lecture Notes in Computer Science 12127, pages 85–100. Springer International Publishing, June 2020.
- [BDL⁺18] Arvid Butting, Manuela Dalibor, Gerrit Leonhardt, Bernhard Rumpe, and Andreas Wortmann. Deriving Fluent Internal Domain-specific Languages from Grammars. In *International Conference on Software Language Engineering (SLE'18)*, pages 187–199. ACM, 2018.
- [BEK⁺18a] Arvid Butting, Robert Eikermann, Oliver Kautz, Bernhard Rumpe, and Andreas Wortmann. Controlled and Extensible Variability of Concrete and Abstract Syntax with Independent Language Features. In *Proceedings of the 12th International*

Workshop on Variability Modelling of Software-Intensive Systems (VAMOS'18), pages 75–82. ACM, January 2018.

- [BEK⁺18b] Arvid Butting, Robert Eikermann, Oliver Kautz, Bernhard Rumpe, and Andreas Wortmann. Modeling Language Variability with Reusable Language Components. In *International Conference on Systems and Software Product Line (SPLC'18)*. ACM, September 2018.
- [BEK⁺19] Arvid Butting, Robert Eikermann, Oliver Kautz, Bernhard Rumpe, and Andreas Wortmann. Systematic Composition of Independent Language Features. *Journal of Systems and Software*, 152:50–69, June 2019.
- [BGH⁺97] Ruth Breu, Radu Grosu, Christoph Hofmann, Franz Huber, Ingolf Krüger, Bernhard Rumpe, Monika Schmidt, and Wolfgang Schwerin. Exemplary and Complete Object Interaction Descriptions. In *Object-oriented Behavioral Semantics Workshop (OOPSLA'97)*, Technical Report TUM-I9737, Germany, 1997. TU Munich.
- [BGH⁺98] Ruth Breu, Radu Grosu, Franz Huber, Bernhard Rumpe, and Wolfgang Schwerin. Systems, Views and Models of UML. In *Proceedings of the Unified Modeling Language, Technical Aspects and Applications*, pages 93–109. Physica Verlag, Heidelberg, Germany, 1998.
- [BGRW17] Arvid Butting, Timo Greifenberg, Bernhard Rumpe, and Andreas Wortmann. Taming the Complexity of Model-Driven Systems Engineering Projects. Part of the Grand Challenges in Modeling (GRAND'17) Workshop, July 2017.
- [BGRW18] Arvid Butting, Timo Greifenberg, Bernhard Rumpe, and Andreas Wortmann. On the Need for Artifact Models in Model-Driven Systems Engineering Projects. In Martina Seidl and Steffen Zschaler, editors, *Software Technologies: Applications and Foundations*, LNCS 10748, pages 146–153. Springer, January 2018.
- [BHK⁺17] Arvid Butting, Robert Heim, Oliver Kautz, Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. A Classification of Dynamic Reconfiguration in Component and Connector Architecture Description Languages. In *Proceedings of MODELS 2017. Workshop ModComp*, CEUR 2019, September 2017.
- [BHP⁺98] Manfred Broy, Franz Huber, Barbara Paech, Bernhard Rumpe, and Katharina Spies. Software and System Modeling Based on a Unified Formal Semantics. In *Workshop on Requirements Targeting Software and Systems Engineering (RTSE'97)*, LNCS 1526, pages 43–68. Springer, 1998.
- [BJRW18] Arvid Butting, Nico Jansen, Bernhard Rumpe, and Andreas Wortmann. Translating Grammars to Accurate Metamodels. In *International Conference on Software Language Engineering (SLE'18)*, pages 174–186. ACM, 2018.
- [BKL⁺18] Christian Brecher, Evgeny Kusmenko, Achim Lindt, Bernhard Rumpe, Simon Storms, Stephan Wein, Michael von Wenckstern, and Andreas Wortmann. Multi-Level Modeling Framework for Machine as a Service Applications Based on Product Process Resource Models. In *Proceedings of the 2nd International Symposium on Computer Science and Intelligent Control (ISCSIC'18)*. ACM, September 2018.

- [BKRW17] Arvid Butting, Oliver Kautz, Bernhard Rumpe, and Andreas Wortmann. Semantic Differencing for Message-Driven Component & Connector Architectures. In *International Conference on Software Architecture (ICSA '17)*, pages 145–154. IEEE, April 2017.
- [BKRW19] Arvid Butting, Oliver Kautz, Bernhard Rumpe, and Andreas Wortmann. Continuously Analyzing Finite, Message-Driven, Time-Synchronous Component & Connector Systems During Architecture Evolution. *Journal of Systems and Software*, 149:437–461, March 2019.
- [BR07] Manfred Broy and Bernhard Rumpe. Modulare hierarchische Modellierung als Grundlage der Software- und Systementwicklung. *Informatik-Spektrum*, 30(1):3–18, Februar 2007.
- [BR12a] Christian Berger and Bernhard Rumpe. Autonomous Driving - 5 Years after the Urban Challenge: The Anticipatory Vehicle as a Cyber-Physical System. In *Automotive Software Engineering Workshop (ASE'12)*, pages 789–798, 2012.
- [BR12b] Christian Berger and Bernhard Rumpe. Engineering Autonomous Driving Software. In C. Rouff and M. Hinchey, editors, *Experience from the DARPA Urban Challenge*, pages 243–271. Springer, Germany, 2012.
- [CBCR15] Tony Clark, Mark van den Brand, Benoit Combemale, and Bernhard Rumpe. Conceptual Model of the Globalization for Domain-Specific Languages. In *Globalizing Domain-Specific Languages*, LNCS 9400, pages 7–20. Springer, 2015.
- [CCF⁺15] Betty H. C. Cheng, Benoit Combemale, Robert B. France, Jean-Marc Jézéquel, and Bernhard Rumpe, editors. *Globalizing Domain-Specific Languages*, LNCS 9400. Springer, 2015.
- [CEG⁺14] Betty Cheng, Kerstin Eder, Martin Gogolla, Lars Grunske, Marin Litoiu, Hausi Müller, Patrizio Pelliccione, Anna Perini, Nauman Qureshi, Bernhard Rumpe, Daniel Schneider, Frank Trollmann, and Norha Villegas. Using Models at Runtime to Address Assurance for Self-Adaptive Systems. In *Models@run.time*, LNCS 8378, pages 101–136. Springer, Germany, 2014.
- [CGR08] María Victoria Cengarle, Hans Grönniger, and Bernhard Rumpe. System Model Semantics of Class Diagrams. Informatik-Bericht 2008-05, TU Braunschweig, Germany, 2008.
- [CGR09] María Victoria Cengarle, Hans Grönniger, and Bernhard Rumpe. Variability within Modeling Language Definitions. In *Conference on Model Driven Engineering Languages and Systems (MODELS'09)*, LNCS 5795, pages 670–684. Springer, 2009.
- [CKM⁺18] Benoit Combemale, Jörg Kienzle, Gunter Mussbacher, Olivier Barais, Erwan Bousse, Walter Cazzola, Philippe Collet, Thomas Degueule, Robert Heinrich, Jean-Marc Jézéquel, Manuel Leduc, Tanja Mayerhofer, Sébastien Mosser, Matthias Schöttle, Misha Strittmatter, and Andreas Wortmann. Concern-Oriented Language Development (COLD): Fostering Reuse in Language Engineering. *Computer Languages, Systems & Structures*, 54:139 – 155, 2018.

- [DEKR19] Imke Drave, Robert Eikermann, Oliver Kautz, and Bernhard Rumpe. Semantic Differencing of Statecharts for Object-oriented Systems. In Slimane Hammoudi, Luis Ferreira Pires, and Bran Selić, editors, *Proceedings of the 7th International Conference on Model-Driven Engineering and Software Development (MODEL-SWARD'19)*, pages 274–282. SciTePress, February 2019.
- [DGH⁺18] Imke Drave, Timo Greifenberg, Steffen Hillemacher, Stefan Kriebel, Matthias Markthaler, Bernhard Rumpe, and Andreas Wortmann. Model-Based Testing of Software-Based System Functions. In *Conference on Software Engineering and Advanced Applications (SEAA'18)*, pages 146–153, August 2018.
- [DGH⁺19] Imke Drave, Timo Greifenberg, Steffen Hillemacher, Stefan Kriebel, Evgeny Kusmenko, Matthias Markthaler, Philipp Orth, Karin Samira Salman, Johannes Richenhagen, Bernhard Rumpe, Christoph Schulze, Michael Wenckstern, and Andreas Wortmann. SMArDT modeling for automotive software testing. *Software: Practice and Experience*, 49(2):301–328, February 2019.
- [DHH⁺20] Imke Drave, Timo Henrich, Katrin Hölldobler, Oliver Kautz, Judith Michael, and Bernhard Rumpe. Modellierung, Verifikation und Synthese von validen Planungszuständen für Fernsehausstrahlungen. In Dominik Bork, Dimitris Karagiannis, and Heinrich C. Mayr, editors, *Modellierung 2020*, pages 173–188. Gesellschaft für Informatik e.V., February 2020.
- [DKMR19] Imke Drave, Oliver Kautz, Judith Michael, and Bernhard Rumpe. Semantic Evolution Analysis of Feature Models. In Thorsten Berger, Philippe Collet, Laurence Duchien, Thomas Fogdal, Patrick Heymans, Timo Kehrer, Jabier Martinez, Raúl Mazo, Leticia Montalvillo, Camille Salinesi, Xhevahire Tërnav, Thomas Thüm, and Tewfik Ziadi, editors, *International Systems and Software Product Line Conference (SPLC'19)*, pages 245–255. ACM, September 2019.
- [DMR⁺20] Manuela Dalibor, Judith Michael, Bernhard Rumpe, Simon Varga, and Andreas Wortmann. Towards a Model-Driven Architecture for Interactive Digital Twin Cockpits. In Gillian Dobbie, Ulrich Frank, Gerti Kappel, Stephen W. Liddle, and Heinrich C. Mayr, editors, *Conceptual Modeling*, pages 377–387. Springer International Publishing, October 2020.
- [EFLR99] Andy Evans, Robert France, Kevin Lano, and Bernhard Rumpe. Meta-Modelling Semantics of UML. In H. Kilov, B. Rumpe, and I. Simmonds, editors, *Behavioral Specifications of Businesses and Systems*, pages 45–60. Kluwer Academic Publisher, 1999.
- [EJK⁺19] Rolf Ebert, Jahir Jolianis, Stefan Kriebel, Matthias Markthaler, Benjamin Pruenster, Bernhard Rumpe, and Karin Samira Salman. Applying Product Line Testing for the Electric Drive System. In Thorsten Berger, Philippe Collet, Laurence Duchien, Thomas Fogdal, Patrick Heymans, Timo Kehrer, Jabier Martinez, Raúl Mazo, Leticia Montalvillo, Camille Salinesi, Xhevahire Tërnav, Thomas Thüm, and Tewfik Ziadi, editors, *International Systems and Software Product Line Conference (SPLC'19)*, pages 14–24. ACM, September 2019.

- [ELR⁺17] Robert Eikermann, Markus Look, Alexander Roth, Bernhard Rumpe, and Andreas Wortmann. Architecting Cloud Services for the Digital me in a Privacy-Aware Environment. In Ivan Mistrik, Rami Bahsoon, Nour Ali, Maritta Heisel, and Bruce Maxim, editors, *Software Architecture for Big Data and the Cloud*, chapter 12, pages 207–226. Elsevier Science & Technology, June 2017.
- [FELR98] Robert France, Andy Evans, Kevin Lano, and Bernhard Rumpe. The UML as a formal modeling notation. *Computer Standards & Interfaces*, 19(7):325–334, November 1998.
- [FHR08] Florian Fieber, Michaela Huhn, and Bernhard Rumpe. Modellqualität als Indikator für Softwarequalität: eine Taxonomie. *Informatik-Spektrum*, 31(5):408–424, Oktober 2008.
- [FLP⁺11a] M. Norbert Fisch, Markus Look, Claas Pinkernell, Stefan Plessner, and Bernhard Rumpe. Der Energie-Navigator - Performance-Controlling für Gebäude und Anlagen. *Technik am Bau (TAB) - Fachzeitschrift für Technische Gebäudeausrüstung*, Seiten 36-41, März 2011.
- [FLP⁺11b] M. Norbert Fisch, Markus Look, Claas Pinkernell, Stefan Plessner, and Bernhard Rumpe. State-based Modeling of Buildings and Facilities. In *Enhanced Building Operations Conference (ICEBO'11)*, 2011.
- [FPPR12] M. Norbert Fisch, Claas Pinkernell, Stefan Plessner, and Bernhard Rumpe. The Energy Navigator - A Web-Platform for Performance Design and Management. In *Energy Efficiency in Commercial Buildings Conference (IEECB'12)*, 2012.
- [GHK⁺07] Hans Grönniger, Jochen Hartmann, Holger Krahn, Stefan Kriebel, and Bernhard Rumpe. View-based Modeling of Function Nets. In *Object-oriented Modelling of Embedded Real-Time Systems Workshop (OMER4'07)*, 2007.
- [GHK⁺08a] Hans Grönniger, Jochen Hartmann, Holger Krahn, Stefan Kriebel, Lutz Rothhardt, and Bernhard Rumpe. Modelling Automotive Function Nets with Views for Features, Variants, and Modes. In *Proceedings of 4th European Congress ERTS - Embedded Real Time Software*, 2008.
- [GHK⁺08b] Hans Grönniger, Jochen Hartmann, Holger Krahn, Stefan Kriebel, Lutz Rothhardt, and Bernhard Rumpe. View-Centric Modeling of Automotive Logical Architectures. In *Tagungsband des Dagstuhl-Workshop MBEES: Modellbasierte Entwicklung eingebetteter Systeme IV*, Informatik Bericht 2008-02. TU Braunschweig, 2008.
- [GHK⁺15a] Timo Greifenberg, Katrin Hölldobler, Carsten Kolassa, Markus Look, Pedram Mir Seyed Nazari, Klaus Müller, Antonio Navarro Perez, Dimitri Plotnikov, Dirk Reiß, Alexander Roth, Bernhard Rumpe, Martin Schindler, and Andreas Wortmann. Integration of Handwritten and Generated Object-Oriented Code. In *Model-Driven Engineering and Software Development*, Communications in Computer and Information Science 580, pages 112–132. Springer, 2015.
- [GHK⁺15b] Timo Greifenberg, Katrin Hölldobler, Carsten Kolassa, Markus Look, Pedram Mir Seyed Nazari, Klaus Müller, Antonio Navarro Perez, Dimitri Plotnikov, Dirk Reiß, Alexander Roth, Bernhard Rumpe, Martin Schindler, and Andreas Wortmann.

- A Comparison of Mechanisms for Integrating Handwritten and Generated Code for Object-Oriented Programming Languages. In *Model-Driven Engineering and Software Development Conference (MODELSWARD'15)*, pages 74–85. SciTePress, 2015.
- [GHR17] Timo Greifenberg, Steffen Hillemacher, and Bernhard Rumpe. *Towards a Sustainable Artifact Model: Artifacts in Generator-Based Model-Driven Projects*. Aachener Informatik-Berichte, Software Engineering, Band 30. Shaker Verlag, December 2017.
- [GKPR08] Hans Grönniger, Holger Krahn, Claas Pinkernell, and Bernhard Rumpe. Modeling Variants of Automotive Systems using Views. In *Modellbasierte Entwicklung von eingebetteten Fahrzeugfunktionen*, Informatik Bericht 2008-01, pages 76–89. TU Braunschweig, 2008.
- [GKR96] Radu Grosu, Cornel Klein, and Bernhard Rumpe. Enhancing the SysLab System Model with State. Technical Report TUM-I9631, TU Munich, Germany, July 1996.
- [GKR⁺06] Hans Grönniger, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. MontiCore 1.0 - Ein Framework zur Erstellung und Verarbeitung domänenspezifischer Sprachen. Informatik-Bericht 2006-04, CFG-Fakultät, TU Braunschweig, August 2006.
- [GKR⁺07] Hans Grönniger, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. Textbased Modeling. In *4th International Workshop on Software Language Engineering, Nashville*, Informatik-Bericht 4/2007. Johannes-Gutenberg-Universität Mainz, 2007.
- [GKR⁺08] Hans Grönniger, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. MontiCore: A Framework for the Development of Textual Domain Specific Languages. In *30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, May 10-18, 2008, Companion Volume*, pages 925–926, 2008.
- [GKRS06] Hans Grönniger, Holger Krahn, Bernhard Rumpe, and Martin Schindler. Integration von Modellen in einen codebasierten Softwareentwicklungsprozess. In *Modellierung 2006 Conference*, LNI 82, Seiten 67-81, 2006.
- [GLPR15] Timo Greifenberg, Markus Look, Claas Pinkernell, and Bernhard Rumpe. Energieeffiziente Städte - Herausforderungen und Lösungen aus Sicht des Software Engineerings. In Linnhoff-Popien, Claudia and Zaddach, Michael and Grahl, Andreas, Editor, *Marktplätze im Umbruch: Digitale Strategien für Services im Mobilen Internet*, Xpert.press, Kapitel 56, Seiten 511-520. Springer Berlin Heidelberg, April 2015.
- [GLRR15] Timo Greifenberg, Markus Look, Sebastian Roidl, and Bernhard Rumpe. Engineering Tagging Languages for DSLs. In *Conference on Model Driven Engineering Languages and Systems (MODELS'15)*, pages 34–43. ACM/IEEE, 2015.
- [GMN⁺20] Arkadii Gerasimov, Judith Michael, Lukas Netz, Bernhard Rumpe, and Simon Varga. Continuous Transition from Model-Driven Prototype to Full-Size Real-World Enterprise Information Systems. In Bonnie Anderson, Jason Thatcher, and Rayman Meservy, editors, *25th Americas Conference on Information Systems (AMCIS*

- 2020), AIS Electronic Library (AISeL), pages 1–10. Association for Information Systems (AIS), August 2020.
- [GMR⁺16] Timo Greifenberg, Klaus Müller, Alexander Roth, Bernhard Rumpe, Christoph Schulze, and Andreas Wortmann. Modeling Variability in Template-based Code Generators for Product Line Engineering. In *Modellierung 2016 Conference*, LNI 254, pages 141–156. Bonner Köllen Verlag, March 2016.
 - [GR95] Radu Grosu and Bernhard Rumpe. Concurrent Timed Port Automata. Technical Report TUM-I9533, TU Munich, Germany, October 1995.
 - [GR11] Hans Grönniger and Bernhard Rumpe. Modeling Language Variability. In *Workshop on Modeling, Development and Verification of Adaptive Systems*, LNCS 6662, pages 17–32. Springer, 2011.
 - [GRJA12] Tim Gülke, Bernhard Rumpe, Martin Jansen, and Joachim Axmann. High-Level Requirements Management and Complexity Costs in Automotive Development Projects: A Problem Statement. In *Requirements Engineering: Foundation for Software Quality (REFSQ’12)*, 2012.
 - [GRR10] Hans Grönniger, Dirk Reiß, and Bernhard Rumpe. Towards a Semantics of Activity Diagrams with Semantic Variation Points. In *Conference on Model Driven Engineering Languages and Systems (MODELS’10)*, LNCS 6394, pages 331–345. Springer, 2010.
 - [HHK⁺13] Arne Haber, Katrin Hölldobler, Carsten Kolassa, Markus Look, Klaus Müller, Bernhard Rumpe, and Ina Schaefer. Engineering Delta Modeling Languages. In *Software Product Line Conference (SPLC’13)*, pages 22–31. ACM, 2013.
 - [HHK⁺14] Martin Henze, Lars Hermerschmidt, Daniel Kerpen, Roger Häußling, Bernhard Rumpe, and Klaus Wehrle. User-driven Privacy Enforcement for Cloud-based Services in the Internet of Things. In *Conference on Future Internet of Things and Cloud (FiCloud’14)*. IEEE, 2014.
 - [HHK⁺15a] Arne Haber, Katrin Hölldobler, Carsten Kolassa, Markus Look, Klaus Müller, Bernhard Rumpe, Ina Schaefer, and Christoph Schulze. Systematic Synthesis of Delta Modeling Languages. *Journal on Software Tools for Technology Transfer (STTT)*, 17(5):601–626, October 2015.
 - [HHK⁺15b] Martin Henze, Lars Hermerschmidt, Daniel Kerpen, Roger Häußling, Bernhard Rumpe, and Klaus Wehrle. A comprehensive approach to privacy in the cloud-based Internet of Things. *Future Generation Computer Systems*, 56:701–718, 2015.
 - [HKM⁺13] Arne Haber, Carsten Kolassa, Peter Manhart, Pedram Mir Seyed Nazari, Bernhard Rumpe, and Ina Schaefer. First-Class Variability Modeling in Matlab/Simulink. In *Variability Modelling of Software-intensive Systems Workshop (VaMoS’13)*, pages 11–18. ACM, 2013.
 - [HKR⁺07] Christoph Herrmann, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. An Algebraic View on the Semantics of Model Composition. In *Conference on Model Driven Architecture - Foundations and Applications (ECMDA-FA’07)*, LNCS 4530, pages 99–113. Springer, Germany, 2007.

- [HKR⁺09] Christoph Herrmann, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. Scaling-Up Model-Based-Development for Large Heterogeneous Systems with Compositional Modeling. In *Conference on Software Engineering in Research and Practice (SERP'09)*, pages 172–176, July 2009.
- [HKR⁺11] Arne Haber, Thomas Kutz, Holger Rendel, Bernhard Rumpe, and Ina Schaefer. Delta-oriented Architectural Variability Using MontiCore. In *Software Architecture Conference (ECSA'11)*, pages 6:1–6:10. ACM, 2011.
- [HKR12] Christoph Herrmann, Thomas Kurpick, and Bernhard Rumpe. SSELab: A Plug-In-Based Framework for Web-Based Project Portals. In *Developing Tools as Plug-Ins Workshop (TOPI'12)*, pages 61–66. IEEE, 2012.
- [HLMSN⁺15a] Arne Haber, Markus Look, Pedram Mir Seyed Nazari, Antonio Navarro Perez, Bernhard Rumpe, Steven Völkel, and Andreas Wortmann. Composition of Heterogeneous Modeling Languages. In *Model-Driven Engineering and Software Development*, Communications in Computer and Information Science 580, pages 45–66. Springer, 2015.
- [HLMSN⁺15b] Arne Haber, Markus Look, Pedram Mir Seyed Nazari, Antonio Navarro Perez, Bernhard Rumpe, Steven Völkel, and Andreas Wortmann. Integration of Heterogeneous Modeling Languages via Extensible and Composible Language Components. In *Model-Driven Engineering and Software Development Conference (MOD-ELSWARD'15)*, pages 19–31. SciTePress, 2015.
- [HMR⁺19] Katrin Hölldobler, Judith Michael, Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. Innovations in Model-based Software and Systems Engineering. *The Journal of Object Technology*, 18(1):1–60, July 2019.
- [HMSNRW16] Robert Heim, Pedram Mir Seyed Nazari, Bernhard Rumpe, and Andreas Wortmann. Compositional Language Engineering using Generated, Extensible, Static Type Safe Visitors. In *Conference on Modelling Foundations and Applications (ECMFA)*, LNCS 9764, pages 67–82. Springer, July 2016.
- [HR04] David Harel and Bernhard Rumpe. Meaningful Modeling: What's the Semantics of "Semantics"? *IEEE Computer*, 37(10):64–72, October 2004.
- [HR17] Katrin Hölldobler and Bernhard Rumpe. *MontiCore 5 Language Workbench Edition 2017*. Aachener Informatik-Berichte, Software Engineering, Band 32. Shaker Verlag, December 2017.
- [HRR98] Franz Huber, Andreas Rausch, and Bernhard Rumpe. Modeling Dynamic Component Interfaces. In *Technology of Object-Oriented Languages and Systems (TOOLS 26)*, pages 58–70. IEEE, 1998.
- [HRR⁺11] Arne Haber, Holger Rendel, Bernhard Rumpe, Ina Schaefer, and Frank van der Linden. Hierarchical Variability Modeling for Software Architectures. In *Software Product Lines Conference (SPLC'11)*, pages 150–159. IEEE, 2011.
- [HRR12] Arne Haber, Jan Oliver Ringert, and Bernhard Rumpe. MontiArc - Architectural Modeling of Interactive Distributed and Cyber-Physical Systems. Technical Report AIB-2012-03, RWTH Aachen University, February 2012.

- [HRRS11] Arne Haber, Holger Rendel, Bernhard Rumpe, and Ina Schaefer. Delta Modeling for Software Architectures. In *Tagungsband des Dagstuhl-Workshop MBEES: Modellbasierte Entwicklung eingebetteter Systeme VII*, pages 1 – 10. fortiss GmbH, 2011.
- [HRRS12] Arne Haber, Holger Rendel, Bernhard Rumpe, and Ina Schaefer. Evolving Delta-oriented Software Product Line Architectures. In *Large-Scale Complex IT Systems. Development, Operation and Management, 17th Monterey Workshop 2012*, LNCS 7539, pages 183–208. Springer, 2012.
- [HRRW12] Christian Hopp, Holger Rendel, Bernhard Rumpe, and Fabian Wolf. Einführung eines Produktlinienansatzes in die automotive Softwareentwicklung am Beispiel von Steuergerätesoftware. In *Software Engineering Conference (SE’12)*, LNI 198, Seiten 181-192, 2012.
- [HRW15] Katrin Hölldobler, Bernhard Rumpe, and Ingo Weisemöller. Systematically Deriving Domain-Specific Transformation Languages. In *Conference on Model Driven Engineering Languages and Systems (MODELS’15)*, pages 136–145. ACM/IEEE, 2015.
- [HRW18] Katrin Hölldobler, Bernhard Rumpe, and Andreas Wortmann. Software Language Engineering in the Large: Towards Composing and Deriving Languages. *Computer Languages, Systems & Structures*, 54:386–405, 2018.
- [JWCR18] Rodi Jolak, Andreas Wortmann, Michel Chaudron, and Bernhard Rumpe. Does Distance Still Matter? Revisiting Collaborative Distributed Software Design. *IEEE Software*, 35(6):40–47, 2018.
- [KER99] Stuart Kent, Andy Evans, and Bernhard Rumpe. UML Semantics FAQ. In A. Moreira and S. Demeyer, editors, *Object-Oriented Technology, ECOOP’99 Workshop Reader*, LNCS 1743, Berlin, 1999. Springer Verlag.
- [KKP⁺09] Gabor Karsai, Holger Krah, Claas Pinkernell, Bernhard Rumpe, Martin Schindler, and Steven Völkel. Design Guidelines for Domain Specific Languages. In *Domain-Specific Modeling Workshop (DSM’09)*, Techreport B-108, pages 7–13. Helsinki School of Economics, October 2009.
- [KKR19] Nils Kaminski, Evgeny Kusmenko, and Bernhard Rumpe. Modeling Dynamic Architectures of Self-Adaptive Cooperative Systems. *The Journal of Object Technology*, 18(2):1–20, July 2019. The 15th European Conference on Modelling Foundations and Applications.
- [KLPR12] Thomas Kurpick, Markus Look, Claas Pinkernell, and Bernhard Rumpe. Modeling Cyber-Physical Systems: Model-Driven Specification of Energy Efficient Buildings. In *Modelling of the Physical World Workshop (MOTPW’12)*, pages 2:1–2:6. ACM, October 2012.
- [KMR⁺20] Jörg Christian Kirchhof, Judith Michael, Bernhard Rumpe, Simon Varga, and Andreas Wortmann. Model-driven Digital Twin Construction: Synthesizing the Integration of Cyber-Physical Systems with Their Information Systems. In *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*, pages 90–101. ACM, October 2020.

- [KMS⁺18] Stefan Kriebel, Matthias Markthaler, Karin Samira Salman, Timo Greifenberg, Stefan Hillemacher, Bernhard Rumpe, Christoph Schulze, Andreas Wortmann, Philipp Orth, and Johannes Richenhagen. Improving Model-based Testing in Automotive Software Engineering. In *International Conference on Software Engineering: Software Engineering in Practice (ICSE'18)*, pages 172–180. ACM, June 2018.
- [KNP⁺19] Evgeny Kusmenko, Sebastian Nickels, Svetlana Pavlitskaya, Bernhard Rumpe, and Thomas Timmermanns. Modeling and Training of Neural Processing Systems. In Marouane Kessentini, Tao Yue, Alexander Pretschner, Sebastian Voss, and Loli Burgueño, editors, *Conference on Model Driven Engineering Languages and Systems (MODELS'19)*, pages 283–293. IEEE, September 2019.
- [KPR97] Cornel Klein, Christian Prehofer, and Bernhard Rumpe. Feature Specification and Refinement with State Transition Diagrams. In *Workshop on Feature Interactions in Telecommunications Networks and Distributed Systems*, pages 284–297. IOS-Press, 1997.
- [KPR12] Thomas Kurpick, Claas Pinkernell, and Bernhard Rumpe. Der Energie Navigator. In H. Lichter and B. Rumpe, Editoren, *Entwicklung und Evolution von Forschungssoftware. Tagungsband, Rolduc, 10.-11.11.2011*, Aachener Informatik-Berichte, Software Engineering, Band 14. Shaker Verlag, Aachen, Deutschland, 2012.
- [KPRS19] Evgeny Kusmenko, Svetlana Pavlitskaya, Bernhard Rumpe, and Sebastian Stüber. On the Engineering of AI-Powered Systems. In Lisa O’Conner, editor, *ASE’19. Software Engineering Intelligence Workshop (SEI’19)*, pages 126–133. IEEE, November 2019.
- [KR18] Oliver Kautz and Bernhard Rumpe. On Computing Instructions to Repair Failed Model Refinements. In *Conference on Model Driven Engineering Languages and Systems (MODELS'18)*, pages 289–299. ACM, October 2018.
- [Kra10] Holger Krahn. *MontiCore: Agile Entwicklung von domänenspezifischen Sprachen im Software-Engineering*. Aachener Informatik-Berichte, Software Engineering, Band 1. Shaker Verlag, März 2010.
- [KRB96] Cornel Klein, Bernhard Rumpe, and Manfred Broy. A stream-based mathematical model for distributed information processing systems - SysLab system model. In *Workshop on Formal Methods for Open Object-based Distributed Systems*, IFIP Advances in Information and Communication Technology, pages 323–338. Chapman & Hall, 1996.
- [KRR14] Helmut Krcmar, Ralf Reussner, and Bernhard Rumpe. *Trusted Cloud Computing*. Springer, Schweiz, December 2014.
- [KRRS19] Stefan Kriebel, Deni Raco, Bernhard Rumpe, and Sebastian Stüber. Model-Based Engineering for Avionics: Will Specification and Formal Verification e.g. Based on Broy’s Streams Become Feasible? In Stephan Krusche, Kurt Schneider, Marco Kuhrmann, Robert Heinrich, Reiner Jung, Marco Konersmann, Eric Schmieders, Steffen Helke, Ina Schaefer, Andreas Vogelsang, Björn Annighöfer, Andreas Schweiger, Marina Reich, and André van Hoorn, editors, *Proceedings of the Workshops of the Software Engineering Conference. Workshop on Avionics Systems*

- and Software Engineering (AvioSE'19)*, CEUR Workshop Proceedings 2308, pages 87–94. CEUR Workshop Proceedings, February 2019.
- [KRRvW17] Evgeny Kusmenko, Alexander Roth, Bernhard Rumpe, and Michael von Wenckstern. Modeling Architectures of Cyber-Physical Systems. In *European Conference on Modelling Foundations and Applications (ECMFA'17)*, LNCS 10376, pages 34–50. Springer, July 2017.
- [KRS12] Stefan Kowalewski, Bernhard Rumpe, and Andre Stollenwerk. Cyber-Physical Systems - eine Herausforderung für die Automatisierungstechnik? In *Proceedings of Automation 2012, VDI Berichte 2012*, Seiten 113–116. VDI Verlag, 2012.
- [KRV06] Holger Krahn, Bernhard Rumpe, and Steven Völkel. Roles in Software Development using Domain Specific Modelling Languages. In *Domain-Specific Modeling Workshop (DSM'06)*, Technical Report TR-37, pages 150–158. Jyväskylä University, Finland, 2006.
- [KRV07a] Holger Krahn, Bernhard Rumpe, and Steven Völkel. Efficient Editor Generation for Compositional DSLs in Eclipse. In *Domain-Specific Modeling Workshop (DSM'07)*, Technical Reports TR-38. Jyväskylä University, Finland, 2007.
- [KRV07b] Holger Krahn, Bernhard Rumpe, and Steven Völkel. Integrated Definition of Abstract and Concrete Syntax for Textual Languages. In *Conference on Model Driven Engineering Languages and Systems (MODELS'07)*, LNCS 4735, pages 286–300. Springer, 2007.
- [KRV08] Holger Krahn, Bernhard Rumpe, and Steven Völkel. Monticore: Modular Development of Textual Domain Specific Languages. In *Conference on Objects, Models, Components, Patterns (TOOLS-Europe'08)*, LNBIP 11, pages 297–315. Springer, 2008.
- [KRV10] Holger Krahn, Bernhard Rumpe, and Stefan Völkel. MontiCore: a Framework for Compositional Development of Domain Specific Languages. *International Journal on Software Tools for Technology Transfer (STTT)*, 12(5):353–372, September 2010.
- [KRV14] Holger Krahn, Bernhard Rumpe, and Steven Völkel. Roles in Software Development using Domain Specific Modeling Languages. In *Proceedings of the 6th OOPSLA Workshop on Domain-Specific Modeling (DSM' 06)*. CoRR arXiv, 2014.
- [KRW20] Oliver Kautz, Bernhard Rumpe, and Andreas Wortmann. Automated semantics-preserving parallel decomposition of finite component and connector architectures. *Automated Software Engineering*, 27:119–151, April 2020.
- [LMK⁺11] Philipp Leusmann, Christian Möllering, Lars Klack, Kai Kasugai, Bernhard Rumpe, and Martina Zieffle. Your Floor Knows Where You Are: Sensing and Acquisition of Movement Data. In Arkady Zaslavsky, Panos K. Chrysanthos, Dik Lun Lee, Dipanjan Chakraborty, Vana Kalogeraki, Mohamed F. Mokbel, and Chi-Yin Chow, editors, *12th IEEE International Conference on Mobile Data Management (Volume 2)*, pages 61–66. IEEE, June 2011.

- [LRSS10] Tihamer Levendovszky, Bernhard Rumpe, Bernhard Schätz, and Jonathan Sprinkle. Model Evolution and Management. In *Model-Based Engineering of Embedded Real-Time Systems Workshop (MBEERTS'10)*, LNCS 6100, pages 241–270. Springer, 2010.
- [MKB⁺19] Felix Mannhardt, Agnes Koschmider, Nathalie Baracaldo, Matthias Weidlich, and Judith Michael. Privacy-Preserving Process Mining: Differential Privacy for Event Logs. *Business & Information Systems Engineering*, 61(5):1–20, October 2019.
- [MKM⁺19] Judith Michael, Agnes Koschmider, Felix Mannhardt, Nathalie Baracaldo, and Bernhard Rumpe. User-Centered and Privacy-Driven Process Mining System Design for IoT. In Cinzia Cappiello and Marcela Ruiz, editors, *Proceedings of CAiSE Forum 2019: Information Systems Engineering in Responsible Information Systems*, pages 194–206. Springer, June 2019.
- [MM13] Judith Michael and Heinrich C. Mayr. Conceptual modeling for ambient assistance. In *Conceptual Modeling - ER 2013*, LNCS 8217, pages 403–413. Springer, 2013.
- [MM15] Judith Michael and Heinrich C. Mayr. Creating a domain specific modelling method for ambient assistance. In *International Conference on Advances in ICT for Emerging Regions (ICTer2015)*, pages 119–124. IEEE, 2015.
- [MMR10] Tom Mens, Jeff Magee, and Bernhard Rumpe. Evolving Software Architecture Descriptions of Critical Systems. *IEEE Computer*, 43(5):42–48, May 2010.
- [MMR⁺17] Heinrich C. Mayr, Judith Michael, Suneth Ranasinghe, Vladimir A. Shekhovtsov, and Claudia Steinberger. *Model Centered Architecture*, pages 85–104. Springer International Publishing, 2017.
- [MNRV19] Judith Michael, Lukas Netz, Bernhard Rumpe, and Simon Varga. Towards Privacy-Preserving IoT Systems Using Model Driven Engineering. In Nicolas Ferry, Antonio Cicchetti, Federico Ciccozzi, Arnor Solberg, Manuel Wimmer, and Andreas Wortmann, editors, *Proceedings of MODELS 2019. Workshop MDE4IoT*, pages 595–614. CEUR Workshop Proceedings, September 2019.
- [MRR10] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. A Manifesto for Semantic Model Differencing. In *Proceedings Int. Workshop on Models and Evolution (ME'10)*, LNCS 6627, pages 194–203. Springer, 2010.
- [MRR11a] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. ADDiff: Semantic Differencing for Activity Diagrams. In *Conference on Foundations of Software Engineering (ESEC/FSE '11)*, pages 179–189. ACM, 2011.
- [MRR11b] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. An Operational Semantics for Activity Diagrams using SMV. Technical Report AIB-2011-07, RWTH Aachen University, Aachen, Germany, July 2011.
- [MRR11c] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. CD2Alloy: Class Diagrams Analysis Using Alloy Revisited. In *Conference on Model Driven Engineering Languages and Systems (MODELS'11)*, LNCS 6981, pages 592–607. Springer, 2011.

- [MRR11d] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. CDDiff: Semantic Differencing for Class Diagrams. In Mira Mezini, editor, *ECOOP 2011 - Object-Oriented Programming*, pages 230–254. Springer Berlin Heidelberg, 2011.
- [MRR11e] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. Modal Object Diagrams. In *Object-Oriented Programming Conference (ECOOP’11)*, LNCS 6813, pages 281–305. Springer, 2011.
- [MRR11f] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. Semantically Configurable Consistency Analysis for Class and Object Diagrams. In *Conference on Model Driven Engineering Languages and Systems (MODELS’11)*, LNCS 6981, pages 153–167. Springer, 2011.
- [MRR11g] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. Summarizing Semantic Model Differences. In Bernhard Schätz, Dirk Deridder, Alfonso Pierantonio, Jonathan Sprinkle, and Dalila Tamzalit, editors, *ME 2011 - Models and Evolution*, October 2011.
- [MRR13] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. Synthesis of Component and Connector Models from Crosscutting Structural Views. In Meyer, B. and Baresi, L. and Mezini, M., editor, *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE’13)*, pages 444–454. ACM New York, 2013.
- [MRR14a] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. Synthesis of Component and Connector Models from Crosscutting Structural Views (extended abstract). In Wilhelm Hasselbring and Nils Christian Ehmke, editors, *Software Engineering 2014*, LNI 227, pages 63–64. Gesellschaft für Informatik, Köllen Druck+Verlag GmbH, 2014.
- [MRR14b] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. Verifying Component and Connector Models against Crosscutting Structural Views. In *Software Engineering Conference (ICSE’14)*, pages 95–105. ACM, 2014.
- [MRV20] Judith Michael, Bernhard Rumpe, and Simon Varga. Human behavior, goals and model-driven software engineering for assistive systems. In Agnes Koschmider, Judith Michael, and Bernhard Thalheim, editors, *Enterprise Modeling and Information Systems Architectures (EMSIA 2020)*, pages 11–18. CEUR Workshop Proceedings, June 2020.
- [MS17] Judith Michael and Claudia Steinberger. Context modeling for active assistance. In Cristina Cabanillas, Sergio España, and Siamak Farshidi, editors, *Proc. of the ER Forum 2017 and the ER 2017 Demo Track co-located with the 36th Int. Conference on Conceptual Modelling (ER 2017)*, pages 221–234, 2017.
- [MSNRR16] Pedram Mir Seyed Nazari, Alexander Roth, and Bernhard Rumpe. An Extended Symbol Table Infrastructure to Manage the Composition of Output-Specific Generator Information. In *Modellierung 2016 Conference*, LNI 254, pages 133–140. Bonner Köllen Verlag, March 2016.

- [NPR13] Antonio Navarro Pérez and Bernhard Rumpe. Modeling Cloud Architectures as Interactive Systems. In *Model-Driven Engineering for High Performance and Cloud Computing Workshop*, CEUR Workshop Proceedings 1118, pages 15–24, 2013.
- [PFR02] Wolfgang Pree, Marcus Fontoura, and Bernhard Rumpe. Product Line Annotations with UML-F. In *Software Product Lines Conference (SPLC'02)*, LNCS 2379, pages 188–197. Springer, 2002.
- [PR94] Barbara Paech and Bernhard Rumpe. A new Concept of Refinement used for Behaviour Modelling with Automata. In *Proceedings of the Industrial Benefit of Formal Methods (FME'94)*, LNCS 873, pages 154–174. Springer, 1994.
- [PR99] Jan Philipps and Bernhard Rumpe. Refinement of Pipe-and-Filter Architectures. In *Congress on Formal Methods in the Development of Computing System (FM'99)*, LNCS 1708, pages 96–115. Springer, 1999.
- [PR01] Jan Philipps and Bernhard Rumpe. Roots of Refactoring. In Kilov, H. and Baclavski, K., editor, *Tenth OOPSLA Workshop on Behavioral Semantics. Tampa Bay, Florida, USA, October 15*. Northeastern University, 2001.
- [PR03] Jan Philipps and Bernhard Rumpe. Refactoring of Programs and Specifications. In Kilov, H. and Baclavski, K., editor, *Practical Foundations of Business and System Specifications*, pages 281–297. Kluwer Academic Publishers, 2003.
- [Rin14] Jan Oliver Ringert. *Analysis and Synthesis of Interactive Component and Connector Systems*. Aachener Informatik-Berichte, Software Engineering, Band 19. Shaker Verlag, 2014.
- [RK96] Bernhard Rumpe and Cornel Klein. Automata Describing Object Behavior. In B. Harvey and H. Kilov, editors, *Object-Oriented Behavioral Specifications*, pages 265–286. Kluwer Academic Publishers, 1996.
- [RKB95] Bernhard Rumpe, Cornel Klein, and Manfred Broy. Ein strombasiertes mathematisches Modell verteilter informationsverarbeitender Systeme - Syslab-Systemmodell. Technischer Bericht TUM-I9510, TU München, Deutschland, März 1995.
- [RR11] Jan Oliver Ringert and Bernhard Rumpe. A Little Synopsis on Streams, Stream Processing Functions, and State-Based Stream Processing. *International Journal of Software and Informatics*, 2011.
- [RRRW15] Jan Oliver Ringert, Alexander Roth, Bernhard Rumpe, and Andreas Wortmann. Language and Code Generator Composition for Model-Driven Engineering of Robotics Component & Connector Systems. *Journal of Software Engineering for Robotics (JOSER)*, 6(1):33–57, 2015.
- [RRSW17] Jan Oliver Ringert, Bernhard Rumpe, Christoph Schulze, and Andreas Wortmann. Teaching Agile Model-Driven Engineering for Cyber-Physical Systems. In *International Conference on Software Engineering: Software Engineering and Education Track (ICSE'17)*, pages 127–136. IEEE, May 2017.

- [RRW13a] Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. From Software Architecture Structure and Behavior Modeling to Implementations of Cyber-Physical Systems. In *Software Engineering Workshopband (SE'13)*, LNI 215, pages 155–170, 2013.
- [RRW13b] Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. MontiArcAutomaton: Modeling Architecture and Behavior of Robotic Systems. In *Conference on Robotics and Automation (ICRA'13)*, pages 10–12. IEEE, 2013.
- [RRW14] Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. *Architecture and Behavior Modeling of Cyber-Physical Systems with MontiArcAutomaton*. Aachener Informatik-Berichte, Software Engineering, Band 20. Shaker Verlag, December 2014.
- [RSW⁺15] Bernhard Rumpe, Christoph Schulze, Michael von Wenckstern, Jan Oliver Ringert, and Peter Manhart. Behavioral Compatibility of Simulink Models for Product Line Maintenance and Evolution. In *Software Product Line Conference (SPLC'15)*, pages 141–150. ACM, 2015.
- [Rum96] Bernhard Rumpe. *Formale Methodik des Entwurfs verteilter objektorientierter Systeme*. Herbert Utz Verlag Wissenschaft, München, Deutschland, 1996.
- [Rum02] Bernhard Rumpe. Executable Modeling with UML - A Vision or a Nightmare? In T. Clark and J. Warmer, editors, *Issues & Trends of Information Technology Management in Contemporary Associations*, Seattle, pages 697–701. Idea Group Publishing, London, 2002.
- [Rum03] Bernhard Rumpe. Model-Based Testing of Object-Oriented Systems. In *Symposium on Formal Methods for Components and Objects (FMCO'02)*, LNCS 2852, pages 380–402. Springer, November 2003.
- [Rum04] Bernhard Rumpe. Agile Modeling with the UML. In *Workshop on Radical Innovations of Software and Systems Engineering in the Future (RISSEF'02)*, LNCS 2941, pages 297–309. Springer, October 2004.
- [Rum11] Bernhard Rumpe. *Modellierung mit UML, 2te Auflage*. Springer Berlin, September 2011.
- [Rum12] Bernhard Rumpe. *Agile Modellierung mit UML: Codegenerierung, Testfälle, Refactoring, 2te Auflage*. Springer Berlin, Juni 2012.
- [Rum13] Bernhard Rumpe. Towards Model and Language Composition. In Benoit Combe-male, Walter Cazzola, and Robert Bertrand France, editors, *Proceedings of the First Workshop on the Globalization of Domain Specific Languages*, pages 4–7. ACM, 2013.
- [Rum16] Bernhard Rumpe. *Modeling with UML: Language, Concepts, Methods*. Springer International, July 2016.
- [Rum17] Bernhard Rumpe. *Agile Modeling with UML: Code Generation, Testing, Refactoring*. Springer International, May 2017.

- [RW18] Bernhard Rumpe and Andreas Wortmann. Abstraction and Refinement in Hierarchically Decomposable and Underspecified CPS-Architectures. In Lohstroh, Marten and Derler, Patricia Sirjani, Marjan, editor, *Principles of Modeling: Essays Dedicated to Edward A. Lee on the Occasion of His 60th Birthday*, LNCS 10760, pages 383–406. Springer, 2018.
- [Sch12] Martin Schindler. *Eine Werkzeuginfrastruktur zur agilen Entwicklung mit der UML/P*. Aachener Informatik-Berichte, Software Engineering, Band 11. Shaker Verlag, 2012.
- [SHH⁺20] Günther Schuh, Constantin Häfner, Christian Hopmann, Bernhard Rumpe, Matthias Brockmann, Andreas Wortmann, Judith Maibaum, Manuela Dalibor, Pascal Bibow, Patrick Sapel, and Moritz Kröger. Effizientere Produktion mit Digitalen Schatten. *ZWF Zeitschrift für wirtschaftlichen Fabrikbetrieb*, 115(special):105–107, April 2020.
- [SM18] Claudia Steinberger and Judith Michael. Towards Cognitive Assisted Living 3.0 (Extended Abstract): Integration of non-smart resources into cognitive assistance systems. *EMISA Forum*, 38(1):35–36, Nov 2018.
- [SM20] Claudia Steinberger and Judith Michael. *Using Semantic Markup to Boost Context Awareness for Assistive Systems*, pages 227–246. Computer Communications and Networks. Springer International Publishing, 2020.
- [SRVK10] Jonathan Sprinkle, Bernhard Rumpe, Hans Vangheluwe, and Gabor Karsai. Meta-modelling: State of the Art and Research Challenges. In *Model-Based Engineering of Embedded Real-Time Systems Workshop (MBEERTS’10)*, LNCS 6100, pages 57–76. Springer, 2010.
- [THR⁺13] Ulrike Thomas, Gerd Hirzinger, Bernhard Rumpe, Christoph Schulze, and Andreas Wortmann. A New Skill Based Robot Programming Language Using UML/P Statecharts. In *Conference on Robotics and Automation (ICRA’13)*, pages 461–466. IEEE, 2013.
- [Völ11] Steven Völkel. *Kompositionale Entwicklung domänenspezifischer Sprachen*. Aachener Informatik-Berichte, Software Engineering, Band 9. Shaker Verlag, 2011.
- [Wei12] Ingo Weisemöller. *Generierung domänenspezifischer Transformationssprachen*. Aachener Informatik-Berichte, Software Engineering, Band 12. Shaker Verlag, 2012.
- [ZPK⁺11] Massimiliano Zanin, David Perez, Dimitrios S Kolovos, Richard F Paige, Kumardev Chatterjee, Andreas Horst, and Bernhard Rumpe. On Demand Data Analysis and Filtering for Inaccurate Flight Trajectories. In *Proceedings of the SESAR Innovation Days*. EUROCONTROL, 2011.