Semantic Differencing for Message-Driven Component & Connector Architectures

Arvid Butting, Oliver Kautz, Bernhard Rumpe, Andreas Wortmann
Software Engineering, RWTH Aachen University, Aachen, Germany, www.se-rwth.de

***Abstract*—Stepwise refinement is a development methodology in which software components progressively evolve under strict adherence of proven properties. This requires means to check whether a new version of a component – with potentially different interface and behavior implementation – refines the behavior of its predecessor. Where architecture description languages (ADLs) support refinement checking, the complexity of their semantic domain requires (partially) manual proving to establish refinement between component versions. We identified a subset of the FOCUS semantics for describing distributed systems as stream processing functions that is powerful enough to model complex and realistic systems, yet sufficiently powerful to support fully automated refinement checking. Leveraging this, we present a refinement checking method for ADLs yielding semantics that can be expressed as stream processing functions. This method relies on transforming architectures into composed port automata and translating these to Büchi automata prior to proving refinement using RABIT for language inclusion checking. This method enables to compare the behaviors of component versions with minimal effort, yields witnesses for non-refining component pairs, and, thus, ultimately facilitates stepwise component refinement.**

I. INTRODUCTION

Stepwise refinement [3], [4] is a development methodology for continuous architecture modeling based on controlled evolution and progressive improvement of components: each successor component version must adhere to properties already proven for its predecessors. To this effect, checking whether successor component versions *refine* their predecessors in terms of observable input/output behavior is crucial.

Architecture description languages (ADLs) [20] leverage the potential of model-driven engineering [32] for the description of software architectures. Research has produced over 120 ADLs [19] for different domains, such as automotive [9], avionics [11], consumer electronics [31], or robotics [28].

Similar to UML [21], the specific semantics of many ADL details are encoded in their infrastructures and tools only. Where fully detailed denotational or operational semantics are available, such as FOCUS [5], these are usually too complex for fully automated refinement checking and typically require to (partially) manually prove refinement between two component versions. This impedes stepwise refinement so severely that it becomes a "highly idealistic" [3] idea. However, enabling stepwise refinement for software architecture models would greatly facilitate development in domains where component adherence to certain properties is crucial.

We identified a subset of the FOCUS [5] semantics for time-synchronous, distributed, interactive systems that is powerful enough to model complex and realistic systems and yet enables fully automated refinement checking between components. Based on this, we present an approach to transform software component models into a variant of port automata [12], compose these syntactically, and translate these into Büchi automata, where their refinement can be checked via language inclusion. This approach is realized with the MontiArcAutomaton component & connector ADL [23], [25] and the RABIT [1], [2] tool for fully automated language inclusion checking between Büchi automata. It enables modeling software architectures with powerful ADLs and checking refinement on a push-button basis. To this effect, the contributions of this paper are:

- Formulation of the semantics domain of time-synchronous [5] stream processing functions (TSSPFs) inspired by the notion of stream processing function [24].
- Presentation of a time-synchronous variant of port automata (TSPA) [12] with operational semantics based on execution traces and denotational semantics based on sets of TSSPFs.
- A semantically compositional syntactic composition operator for TSPAs: The semantics of the syntactic composition of two TSPAs is equal to the composition of the semantics of the individual TSPAs.
- A transformation from finite TSPAs to Büchi automata.
- A proof showing the operational semantics of a finite TSPA and the language accepted by the Büchi automaton resulting from such a transformation coincide.
- The result that refinement checking and disproof generation in form of semantic difference witnesses for software architectures where components can be mapped to finite TSPAs can be reduced to language inclusion checking and counterexample generation for Büchi automata.
- An implementation based on MontiArcAutomaton [23], [25] and RABIT [1], [2].

In the following, Sec. II sketches the idea of stepwise refinement, before Sec. III presents the FOCUS subset used as semantics domain for components. Afterwards, Sec. IV presents semantic differencing based on this subset and Sec. V presents the implementation of our approach with MontiArcAutomaton and RABIT and evaluates its applicability. Sec. VI discusses observations and Sec. VII highlights related work. Sec. VIII concludes.

II. EXAMPLE

Consider the model-driven development of an elevator control system (ECS) as presented in [29]. The `ECS` depicted

[BKRW17] A. Butting, O. Kautz, B. Rumpe, A. Wortmann:
Semantic Differencing for Message-Driven Component & Connector Architectures.
In: International Conference on Software Architecture (ICSA'17), pages 145-154. IEEE, 2017.
www.se-rwth.de/publications/

in Fig. 1 comprises two hierarchically composed components representing the three floors the elevator serves (component `Floors`) and the elevator cabin (component `Elevator`) itself. Whenever a button on a floor (indicated, for example, by a message on the incoming port `btn1`) is pressed, the `ECS` should activate the light (by sending a message via outgoing port `li1`) on the corresponding floor and instruct the elevator cabin to visit that floor. The control logic of the elevator is modeled via a statechart variant embedded into the `Elevator`'s subcomponent `Control`. This component receives messages upon arriving at a specific floor (*e.g.,* via incoming port `at1`) and sends messages to `Door` and `Motor` to operate its door and to move between the floors. The latter two embed models of compact action languages to describe their respective behavior.

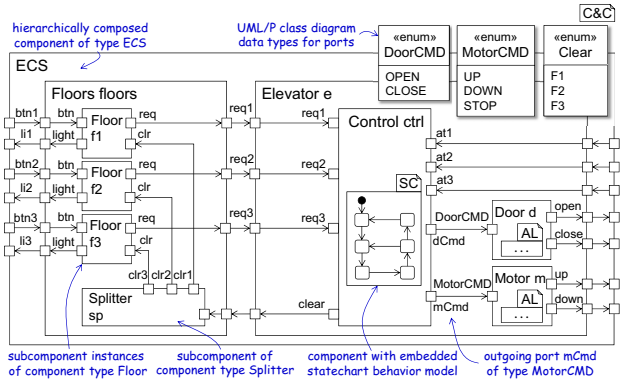

Fig. 1. The elevator control system `ECS` comprises subcomponents to manage serving elevation requests on up to three floors.

For this version of `ECS`, the company has proven that certain properties hold (*e.g.,* that it cannot produce blocking situations). Now the company aims to replace the `Elevator` component with an improved version that reacts only to elevator requests on a floor if there is no such request yet. To this effect, the company employs stepwise refinement to avoid proving the properties of `Elevator` again for its successor version `NewElevator`. Therefore, the behavior descriptions of all subcomponents are translated into port automata. For composed components, the behavior descriptions of their subcomponents are translated also and merged iteratively. This ultimately eliminates all hierarchy levels but the last. The result of this transformation is depicted in Fig. 2, where the behavior descriptions of all three subcomponents have been transformed accordingly and merged into a single port automaton. The same is performed for the improved `NewElevator` component before both are transformed into nondeterministic Büchi automata as presented in Sec. V.

Using this transformation reduces semantic component refinement to language inclusion on Büchi automata and can be solved automatically using RABIT. Hence, with this infrastructure in place, the company now can fully automated ensure whether the `NewElevator`, and its potential successors, actually refine their predecessors or require further adjusting. Where refinement is refuted, difference witnessing

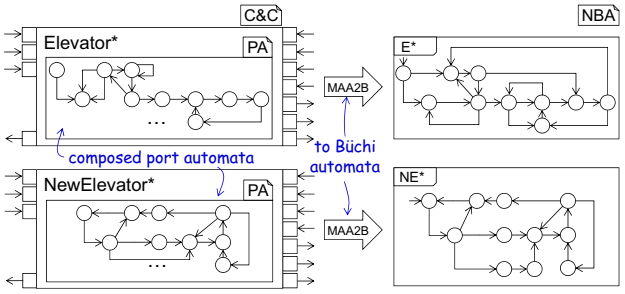

Fig. 2. The composed components `Elevator` and `NewElevator` each are transformed into flat components with a single port automaton prior to being transformed into Büchi automata and checked for language inclusion.

input/output pairs are produced. This automation of stepwise refinement can increase the pace of each refinement step and, hence, overall development efficiency.

III. A Semantics Domain for Components

This section introduces the semantics domain for components based on the FOCUS framework [3], [5], [12], [24], [27] and recaps the most important results from [12], which underlie the approach presented in this paper.

We interpret software architectures as networks of autonomously acting components communicating in a time-synchronous manner via directed, typed channels connecting the components' interfaces. A time-synchronous architecture can be interpreted as a system where component computations are performed concurrently and controlled by a global clock that splits runtime into discrete and equidistant time units. In every time unit, each component receives finitely many input messages via its interfaces and outputs finitely many messages to its environment. The computations of each component in every time unit must terminate.

In the remainder, we denote by $[X \to Y]$ the set of all functions from a set X to a set Y. For a function $f \in [X \to Y]$ and a set $Z \subseteq X$, the restriction of f to Z is the function $f|_Z \in [Z \to Y]$ that satisfies $f|_Z(x) = f(x)$ for all $x \in Z$. Given two functions $f \in [X \to A]$ and $g \in [Y \to B]$, the overriding union of f with g is the function $f + g \in [(X \cup Y) \to (A \cup B)]$ that satisfies $(f + g)(x) = g(x)$ if $x \in Y$ and $(f + g)(x) = f(x)$ if $x \in X \setminus Y$ for all $x \in X \cup Y$.

A. Streams, Messages, Types, and Communication Histories

The history of messages a component receives or sends via an interface is formally described as a stream that contains messages in order of their transmission. Let M be an arbitrary alphabet. A stream over the set M is a finite or infinite sequence of elements from M. Following [5], we denote by

- M^* the set of all finite streams over M,
- M^∞ the set of all infinite streams over M,
- $\langle\rangle$ the empty stream, which is an element of M^*,
- $s^\frown t$ the concatenation of two streams s and t such that $((M^* \cup M^\infty), ^\frown, \langle\rangle)$ is a monoid. If $s \in M^\infty$ then $s^\frown t = s$.
- $\sqsubseteq$ the prefix relation over streams, which is a partial order defined by: $\forall s, t \in (M^* \cup M^\infty) : s \sqsubseteq t \Leftrightarrow \exists u : s^\frown u = t$,

- $s.t$ the t-th element of a stream $s \in M^\infty$,
- $s{\downarrow_t}$ the prefix of a stream $s \in M^\infty$ of length $t \in \mathbb{N}$.

In the remainder, let M denote an arbitrary but fixed set of data elements, such as messages, and let $Type$ be a set of data types such that each $t \in Type$ satisfies $t \subseteq M$. Types facilitate restricting the set of messages a component may emit or receive via an interface. We assume a discrete model of time where component computation is divided into discrete time units of equal and finite duration. In each time unit each component receives at most one message via each incoming interface, may perform finitely many state changes and emits at most one message via each outgoing interface. We use the special symbol $\varepsilon \in M$ to denote the absence of a message during a time unit and require $\varepsilon \in t$ for each $t \in Type$.

A *channel* is an identifier for a communication link between interface elements of components. In the following we denote by C a set of typed channel names. The function $type \in [C \to Type]$ maps each channel in the set C to its type. Let $B \subseteq C$ be an arbitrary set of channel names. A *communication history* is an element of the set B^Ω defined as follows:

$B^\Omega \stackrel{\text{def}}{=} \{h \in [B \to M^\infty] \mid \forall b \in B : h(b) \in type(b)^\infty\}$.

A communication history $h \in B^\Omega$ is used to model the history of messages emitted via the channels in the set B.

Let $h \in B^\Omega$ be a communication history, $H \subseteq B^\Omega$ a set of communication histories, and $t \in \mathbb{N}$ a natural number. We lift the operator $\downarrow$ to communication histories and sets of communication histories in a point-wise manner, *i.e.,* $b{\downarrow_t} \in [B \to M^*]$ denotes the function that satisfies $b{\downarrow_t}(i) = b(i){\downarrow_t}$ for all $i \in B$ and $H{\downarrow_t} \stackrel{\text{def}}{=} \bigcup_{h \in H} h{\downarrow_t}$ denotes the set resulting from applying the operator to each element in H.

B. Time-Synchronous Stream Processing Functions

We model the semantics of distributed interactive systems as sets of time-synchronous stream processing functions (TSSPFs). The notion of TSSPFs is inspired by the notion of timed SPFs [5], [12], [24], [27]. The major and crucial difference between the two notions is that TSSPFs process exactly one message per channel per time unit, whereas SPFs process a stream of messages per channel per time unit. The key idea is to treat components as black-boxes having an observable behavior characterized by the interactions on channels between systems and subsystems while hiding internal implementation details. A component is mapped to a set of functions describing the component's possible behaviors. Such a function maps communication histories over the set of input channels of a component to communication histories over the set of the component's output channels. Thus, each function in the semantics of a component with input channels $I \subseteq C$ and output channels $O \subseteq C$ is of the form $f \in [I^\Omega \to O^\Omega]$. However such functions are not always realizable in the sense that they can be implemented [5], [22]. Intuitively, the characterizing properties for realizability are that a component cannot change messages received or sent in the past and cannot react to messages received in the future [5], [22], [24], [27]. Thus, the output of a behavior describing function until time t must be completely determined by its input until time t:

Definition 1 (Time-Synchronous Stream Processing Function)**.** *Let $I, O \subseteq C$ be two disjoint sets of input and output channels. A function $f \in [I^\Omega \to O^\Omega]$ is called (weakly causal) time-synchronous stream processing function iff*

$\forall i, i' \in I^\Omega : \forall t \in \mathbb{N} : i{\downarrow_t} = i'{\downarrow_t} \Rightarrow f(i){\downarrow_t} = f(i'){\downarrow_t}$.

We denote by $[I^\Omega \xrightarrow{wc} O^\Omega]$ the set of all TSSPFs mapping input histories in I^Ω to output histories in O^Ω. The semantics of components are modeled as closed sets of TSSPFs.

Definition 2 (Component Describing)**.** *Let $I, O \subseteq C$ be two disjoint sets of channels. A set of TSSPFs $F \subseteq [I^\Omega \xrightarrow{wc} O^\Omega]$ is called component (semantics) describing iff it satisfies $\forall g \in [I^\Omega \xrightarrow{wc} O^\Omega] : ((\forall i \in I^\Omega : \exists f \in F : g(i) = f(i)) \Rightarrow g \in F)$.*

The definition above makes the semantics domain of components fully abstract [12], [13] in the sense of [15] and allows to handle unbounded nondeterminism [12]. Full abstraction is achieved by the closeness property, which requires that each TSSPF resulting from a combination of TSSPFs included in the set F is also included in F. The closeness property is also important to make component semantics as little distinguishing as possible. This is illustrated by the fact that two different arbitrary sets of TSSPFs may encode the same component behaviors. The reason for this is that one may find a TSSPF $g \notin F$ that is not included in a set of TSSPFs F, which can be interpreted as a combination of different TSSPFs contained in F. It thus does not induce a new behavior not already covered by a TSSPF in F but, for instance, induces a semantic difference between a component with semantics described by F and a component with semantics described by $F \cup \{g\}$. As a result the semantics of two components that have the exact same observable behaviors may be considered unequal. Consequently, full abstraction is not achieved. Thereby, the closeness property is necessary.

1) Composition of TSSPFs: Composition is an important concept to achieve modularity. Composing the semantics of the individual components of a system leads to the semantics of the whole system. Composing arbitrary sets of TSSPFs can lead to realizability problems in delay-free feedback loops where the output of a component in time unit t depends on its input in time unit t and vice versa. Thus, composition is only defined for TSSPFs where causality between inputs and outputs on channels connected via a feedback loop is ensured. This is the case if one of the TSSPFs participating in a composition is strongly causal with respect to its channels connected by the composition. Intuitively, a set of TSSPFs F is strongly causal with respect to (J, P), if the output of at least one TSSPF $f \in F$ on the channels in P until time unit $t+1$ is not influenced by the function's inputs received on the channels in J after time unit t.

Definition 3 (Strongly Causal Modulo)**.** *Let $f \in [I^\Omega \xrightarrow{wc} O^\Omega]$ be a TSSPF and let $J \subseteq I$ and $P \subseteq O$ be two subsets of input and output channels names. The TSSPF f is called strongly causal with respect to (J, P) iff*

$\forall i, i' \in I^\Omega : \forall t \in \mathbb{N} : (i|_J){\downarrow_t} = (i'|_J){\downarrow_t} \wedge i|_{I \setminus J} = i'|_{I \setminus J} \Rightarrow f(i)|_P{\downarrow_{t+1}} = f(i')|_P{\downarrow_{t+1}}$.

A set of TSSPFs F is called strongly causal with respect to (J, P) iff there exists a function $f \in F$ that is strongly causal with respect to (J, P). The causality complication is avoided, if causality between the inputs and outputs on the connected channels of a composition's participant is guaranteed:

Definition 4 (Composable)**.** *Two sets of TSSPFs $F_1 \subseteq [I_1^\Omega \xrightarrow{wc} O_1^\Omega]$ and $F_2 \subseteq [I_2^\Omega \xrightarrow{wc} O_2^\Omega]$ are called composable iff F_1 is strongly causal with respect to $(I_1 \cap O_2, I_2 \cap O_1)$ or F_2 is strongly causal with respect to $(I_2 \cap O_1, I_1 \cap O_2)$.*

Components communicate with each other via unidirected, typed channels established by connectors connecting component interfaces. Multiple components may read from the same channel, whereas only one component is permitted to write messages on a channel. This ensures that no merging of messages emitted from different components via the same channel is necessary. Thus the output channels of the functions of two sets of TSSPFs need to be disjoint to enable composition. The composition of two sets of TSSPFs yields a set of TSSPFs:

Definition 5 (Composition)**.** *Let $F_1 \subseteq [I_1^\Omega \xrightarrow{wc} O_1^\Omega]$ and $F_2 \subseteq [I_2^\Omega \xrightarrow{wc} O_2^\Omega]$ be two component describing and composable sets of TSSPFs with disjoint output channel sets $O_1 \cap O_2 = \emptyset$. Let $I = (I_1 \backslash O_2) \cup (I_2 \backslash O_1)$ and $O = O_1 \cup O_2$. The composition $F_1 \otimes F_2 \subseteq [I^\Omega \xrightarrow{wc} O^\Omega]$ of F_1 and F_2 is defined by*

$$F_1 \otimes F_2 \stackrel{\text{def}}{=} \{f \mid \forall i \in I^\Omega : \exists f_1 \in F_1 : \exists f_2 \in F_2 : f(i) = o + p \text{ where } o = f_1((i+p)|_{I_1}), p = f_2((i+o)|_{I_2})\}$$

The composition operator is defined similar as in [12], [13], [27] with the difference that we consider the time-synchronous system model instead of the more general timed system model [5]. The composition is well defined and thus results in a component semantics describing set of TSSPFs.

Theorem 1. *If F_1 and F_2 are two component describing and composable sets of TSSPFs with disjoint output channel sets, then $F_1 \otimes F_2$ is also component describing.*

Proof. Analogous to proof of Thm. 9 in [12] by replacing the set the function f is chosen from with $[I^\Omega \xrightarrow{wc} O^\Omega]$. □

C. Time-Synchronous Port Automata

A TSPA specifies the behavior (of parts) of an interactive system and represents a component semantics describing set of TSSPFs that is given by its semantics. TSPAs as introduced in this paper are strongly inspired by port automata as introduced in [12], I/O* automata as introduced in [27], [24], and MAA$_{ts}$ automata as defined in [22]. Port and I/O* automata consume and produce time slices of arbitrary but finitely many input messages in every transition step. In contrast, TSPAs and MAA$_{ts}$ automata consume and output at most one message per input channel in each time slice. Given the set of states and the channel types of an automaton are finite, MAA$_{ts}$ automata and the automata presented here are guaranteed to have finitely many transitions. This is not the case for I/O* and port automata since both have to define a transition for each state and each possible input communication history. Even if the type of a channel is finite, the number of communication histories (streams) of the channel's type is infinite. I/O* and MAA$_{ts}$ automata enforce causality between input and output histories by requiring initial outputs on all channels. In contrast, TSPAs do not require initial outputs. While the syntax of MAA$_{ts}$ automata treat variables explicitly, variables have to be represented implicitly in the state space of TSPAs. TSPAs can be treated as a special case of port automata as presented in [12]. Thereby the proofs of many theorems presented in the following are analog to proofs, which have already been carried out in [12]. In case we are stating an analogous theorem we refer to the appropriate corresponding proof in [12].

A TSPA consists of a set of states, an interface given by input and output channels, and transitions defining the TSPA's behavior. The interface is encoded by a port signature.

Definition 6 (Port Signature)**.** *Let $I, O \subseteq C$ be two disjoint sets of channel names (ports). A port signature is a tuple $\Sigma = (I, O)$. We denote by $C(\Sigma) \stackrel{\text{def}}{=} I \cup O$ the set of all ports in Σ. A port signature Σ is called finite iff $C(\Sigma)$ and $type(c)$ for all $c \in C(\Sigma)$ are finite.*

Let $B \subseteq C$. A *port assignment* is an element of the set $B^\rightarrow$ defined as $B^\rightarrow \stackrel{\text{def}}{=} \{a \in [B \to M] \mid \forall b \in B : a(b) \in type(b)\}$.

TSPAs must not block their environments and must be able to react to any possible well-typed input in any time unit. Therefore, a TSPA must define a reaction to every possible input for each of its states. The reactions of a TSPA are defined by its transitions. In each time unit, a TSPA performs exactly one state change by executing one transition enabled by its input and outputs exactly one message on each output channel.

Definition 7 (Time-Synchronous Port Automaton)**.** *A time-synchronous port automaton is a tuple $A = (\Sigma, S, \iota, \delta)$ where:*

- *$\Sigma = (I, O)$ is a port signature,*
- *S is a set of states,*
- *$\iota \in S$ is the initial state,*
- *$\delta \subseteq S \times C(\Sigma)^\rightarrow \times S$ is the transition relation, which is required to be reactive, i.e., $\forall s \in S : \forall i \in I^\rightarrow : \exists t \in S : \exists \theta \in C(\Sigma)^\rightarrow : (s, \theta, t) \in \delta \wedge \theta|_I = i$.*

A is called finite iff Σ and S are finite.

For convenience we sometimes write $s \xrightarrow{\theta}_\delta t$ instead of $(s, \theta, t) \in \delta$ and simply $s \xrightarrow{\theta} t$ if δ is clear from the context.

1) Execution and Behavior Semantics of TSPAs: This section formalizes the intuitive descriptions of a TSPA's behavior.

Definition 8 (Execution)**.** *Let $A = (\Sigma, S, \iota, \delta)$ be a TSPA. An execution σ of A is an infinite, alternating sequence of states and port assignments starting with the initial state ι:*

$\sigma = s_0, \theta_0, s_1, \theta_1, \ldots$ s.t. $s_0 = \iota$ and $\forall i \in \mathbb{N} : s_i \xrightarrow{\theta_i} s_{i+1}$. The set of all executions of A is denoted by $execs(A)$.

Executions comprise the state changes and interactions performed by a TPSA. Abstracting from state changes allows to treat TSPAs as black boxes with hidden internal details.

Definition 9 (Behavior)**.** *Let $A = (\Sigma, S, \iota, \delta)$ be a TSPA with port signature $\Sigma = (I, O)$. The behavior of an execution $\sigma = s_0, \theta_0, s_1, \theta_1, \ldots$ of A is defined as the sequence $beh(\sigma) \stackrel{\text{def}}{=}$*

$\theta_0, \theta_1, ...$ *containing only port assignments. We denote by* $behs(A) \stackrel{\text{def}}{=} \bigcup_{\sigma \in execs(A)} beh(\sigma)$ *the set of all behaviors of all executions of A. The named communication history* h_α *induced by a behavior* $\alpha \in behs(A)$ *with* $\alpha = e_0, e_1, ...$ *is defined as the function* $h_\alpha \in (I \cup O)^\Omega$ *that satisfies* $h_\alpha(x).t = e_t(x)$ *for all* $x \in I \cup O$ *and* $t \in \mathbb{N}$.

Given a TSPA $A = (\Sigma, S, \iota, \delta)$ with $\Sigma = (I, O)$ and an input history $i \in I^\Omega$, we denote the set of communication histories induced by a behavior of A with input i by

$A[i] \stackrel{\text{def}}{=} \{o \in O^\Omega \mid \exists \alpha \in behs(A) : o = h_\alpha|_O \wedge h_\alpha|_I = i\}$.

2) Composition of TSPAs: As for TSSPFs, causality expresses the dependency between the inputs and outputs of a TSPA. A TSPA's output in time t must be completely determined by its input until time t. Thus it cannot change messages sent in the past and cannot predict messages it receives in the future (*cf.* pulse drivenness in [12]):

Definition 10 (Weakly Causal TSPA)**.** *A TSPA* $A = (\Sigma, S, \iota, \delta)$ *with* $\Sigma = (I, O)$ *is called weakly causal iff*

$\forall i, i' \in I^\Omega : \forall t \in \mathbb{N} : i{\downarrow_t} = i'{\downarrow_t} \Rightarrow A[i]{\downarrow_t} = A[i']{\downarrow_t}$.

Weak causality states that for every two inputs i, i' having a common prefix of length t and for every behavior $\alpha \in A[i]$ there is a behavior $\beta \in A[i']$ having a common prefix of length t with α. Similar as for TSSPFs, weak causality can lead to composition complications, which are avoidable analogously.

Definition 11 (Strongly Causal Modulo)**.** *Let* $A = (\Sigma, S, \iota, \delta)$ *be a TSPA with port signature* $\Sigma = (I, O)$ *and let* $J \subseteq I$ *and* $P \subseteq O$ *be two sets of input and output ports of A. The TSPA A is called strongly causal with respect to* (J, P) *iff*

$$\forall i, i' \in I^\Omega : \forall t \in \mathbb{N} : (i|_J){\downarrow_t} = (i'|_J){\downarrow_t} \wedge i|_{I \setminus J} = i'|_{I \setminus J} \Rightarrow (A[i]|_P){\downarrow_{t+1}} = (A[i']|_P){\downarrow_{t+1}}.$$

Intuitively, a TSPA is strongly causal with respect to (J, P), if its outputs on the channels in P until time $t + 1$ are not influenced by its inputs on the channels in J after time t.

TSPAs communicate with each other via their input and output ports. Multiple automata may read from the same channel, whereas only one automata is permitted to write messages on a channel. This ensures no merging of messages on channels emitted by different automata is necessary.

Definition 12 (Compatible Port Signatures)**.** *Two port signatures* $\Sigma_1 = (I_1, O_1)$ *and* $\Sigma_2 = (I_2, O_2)$ *are called compatible iff* $O_1 \cap O_2 = \emptyset$.

By composing two TSPAs, the output ports of one automaton are connected to the input ports with the same name of the other automaton. The connected input channels are hidden implicitly. The set of output channels of the new automaton is the union of the sets of the output channels of the two original TSPAs. The input channels of the new automaton are the input channels of the two automata that do not share a common name with the output channels of the other automaton.

Definition 13 (Composition of Signatures)**.** *The composition of two compatible port signatures* $\Sigma_1 = (I_1, O_1)$ *and* $\Sigma_2 = (I_2, O_2)$ *is defined as* $\Sigma_1 \otimes \Sigma_2 \stackrel{\text{def}}{=} (I, O)$ *where* $I = (I_1 \setminus O_2) \cup (I_2 \setminus O_1)$ *and* $O = (O_1 \cup O_2)$.

The following defines the composition operator for TSPAs.

Definition 14 (Composition of TSPA)**.** *Let* $A_1 = (\Sigma_1, S_1, \iota_1, \delta_1)$ *and* $A_2 = (\Sigma_2, S_2, \iota_2, \delta_2)$ *be two TSPAs with compatible port signatures* $\Sigma_1 = (I_1, O_1)$ *and* $\Sigma_2 = (I_2, O_2)$. *The composition of* A_1 *and* A_2 *is defined as* $A_1 \otimes A_2 \stackrel{\text{def}}{=} (\Sigma_1 \otimes \Sigma_2, S_1 \times S_2, (\iota_1, \iota_2), \delta)$ *where the transition relation* δ *is defined by the following rule:*

$$\frac{s_1 \xrightarrow{\theta|_{C(\Sigma_1)}}_{\delta_1} t_1 \ \wedge \ s_2 \xrightarrow{\theta|_{C(\Sigma_2)}}_{\delta_2} t_2}{(s_1, s_2) \xrightarrow{\theta}_{\delta} (t_1, t_2)}$$

TPSAs can block each other if they simultaneously require an input emitted by another TSPA to produce the next output. Composing such TSPAs results in a structure with an empty transition relation, which is no TSPA since the requirement for reactiveness in Def. 7 implies that the transition relation of a TSPA is not empty. However, there is a sufficient condition ensuring the resulting transition relation is reactive.

Definition 15 (Composability of TSPAs)**.** *Two TSPAs* $A_1 = (\Sigma_1, S_1, \iota_1, \delta_1)$ *and* $A_2 = (\Sigma_2, S_2, \iota_2, \delta_2)$ *with port signatures* $\Sigma_1 = (I_1, O_1)$ *and* $\Sigma_2 = (I_2, O_2)$ *are called composable iff* A_1 *is strongly causal with respect to* $(I_1 \cap O_2, I_2 \cap O_1)$ *or* A_2 *is strongly causal with respect to* $(I_2 \cap O_1, I_1 \cap O_2)$.

The following theorem states that composing two composable TSPAs always results in a well-formed TSPA.

Theorem 2. *If* A_1 *and* A_2 *are composable TSPAs with compatible port signatures, then* $A_1 \otimes A_2$ *is a TSPA.*

Proof. Analogous to proof of Thm. 3 in [12] by replacing the set the function i is chosen from with $I^\rightarrow$. □

3) TSSPF semantics of TSPAs: This section defines the semantics of TSPAs by sets of TSSPFs and reveals an important relation between the composition operators: The semantics of the syntactic composition of two TSPAs A and B is equal to the composition of the semantics of the individual automata.

Definition 16 (TSSPF Semantics of a TSPA)**.** *The TSSPF semantics* $[\![A]\!]$ *of a TSPA* $A = (\Sigma, S, \iota, \delta)$ *with port signature* $\Sigma = (I, O)$ *is defined as follows:*

$$[\![A]\!] \stackrel{\text{def}}{=} \{f \in [I^\Omega \xrightarrow{wc} O^\Omega] \mid \forall i \in I^\Omega : \exists \alpha \in behs(A) : i = h_\alpha|_I \wedge f(i) = h_\alpha|_O\}$$

For each behavior, the semantics contain a function that maps inputs to outputs as encoded by the history induced by the behavior, *i.e.,* no behavior is lost in the semantic mapping.

Theorem 3. *Let* A *be a TSPA. For each* $\alpha \in behs(A)$ *there is a function* $f \in [\![A]\!]$ *such that* $f(h_\alpha|_I) = h_\alpha|_O$.

Proof. Analogous to proof of Thm. 11 in [12] by replacing the definition of maximality with $\forall i \in I^\Omega : i \in S|_I$. □

The semantics of TSPAs are well formed, *i.e.,* TSPAs can be used to specify component behavior because the semantics of every TSPA is component semantics describing.

Theorem 4. *The semantics $[\![A]\!]$ of a TSPA A is component semantics describing.*

Proof. Analogous to proof of Thm. 12 in [12] by replacing the set the function f is chosen from with $[I^\Omega \xrightarrow{wc} O^\Omega]$. □

The semantics of the composition of two TSPAs is equal to the composition of their individual semantics:

Theorem 5. *For two composable TSPAs A and B with compatible signatures the following holds: $[\![A \otimes B]\!] = [\![A]\!] \otimes [\![B]\!]$.*

Proof. Analogous to proof of Thm. 13 in [12] by replacing the applications of $[\![\cdot]\!]$ for PAs and $\otimes$ for SPFs by applications of the corresponding definitions for TSPAs and TSSPFs. □

An important implication of the theorem is that we can first syntactically compose the individual automata of an architecture and then perform analysis on the semantics of the automaton encoding the behavior of the whole system. This gives another basis for analysis that does not necessarily require to compose the semantics of the individual components of a system as, for example, done in [26].

IV. Semantic Differencing of Component Behavior: From TSPAs to BAs

After introducing the notations for Büchi Automata (BAs) used in this paper, this section presents a theorem stating that there is a nondeterministic BA for each finite TSPA that accepts exactly the behaviors of the TSPA. Afterwards, it is shown that refinement checking and semantic difference witness generation for TSPAs can be reduced to language inclusion checking and counterexample generation for BAs.

A. Büchi Automata

Büchi automata [2] are a variant of finite automata that are acceptors for infinite words and thus induce languages consisting of infinite words. They are well known and much used in the model checking domain. Infinite words over an alphabet Π are infinite sequences of symbols in Π. The set of all infinite words over an alphabet Π is denoted by Π^ω.

Definition 17 (Büchi Automaton)**.** *A BA is a tuple* (Π, Q, I, F, δ) *where* Π *is a finite alphabet, Q is a finite set of states, $I \subseteq Q$ is a set of initial states, $F \subseteq Q$ is a set of accepting states, and $\delta \subseteq Q \times \Pi \times Q$ is the transition relation.*

Let $\mathcal{B} = (\Pi, Q, I, F, \delta)$ be a BA. A run of $\mathcal{B}$ on a word $w = \sigma_1, \sigma_2 ... \in \Pi^\omega$ starting in a state $q_0 \in Q$ is an infinite sequence $q_0, q_1, ...$ such that $q_{j-1} \xrightarrow{\sigma_j}_\delta q_j$ for all $j > 0$. A run $q_0, q_1, ...$ is accepting if $q_0 \in I$ and $q_i \in F$ for infinitely many $i > 0$. The accepted language of $\mathcal{B}$ is defined as $\mathcal{L}(\mathcal{B}) \stackrel{\text{def}}{=} \{w \in \Pi^\omega \mid \text{there exists an accepting run for } w \text{ in } \mathcal{B}\}$.

Checking language inclusion between two Büchi automata is PSPACE-complete [18], though decidable. Although the computational complexity is large, several approaches for checking language inclusion and counterexample (diff witness) generation have been implemented and produce promising results in practice [2]. In the next section, we present a translation from finite TSPAs to BAs and thereby reduce semantic differencing and refinement checking for finite TSPAs to the language inclusion problem for Büchi automata.

B. From TSPAs to BAs

We consider semantic differencing and refinement checking for architectures where the individual components have a finite state space, communicate over finitely many communication channels, and where the types of messages emitted via component interfaces are finite. There exists a nondeterministic BA for each finite TSPA that accepts exactly the TSPA's behaviors.

Theorem 6. *For any finite TSPA A there exists a BA $\mathcal{B}$ such that $behs(A) = \mathcal{L}(\mathcal{B})$.*

Proof. Let $A = (\Sigma, S, \iota, \delta)$ with $\Sigma = (I, O)$ be a finite TSPA. Let $\mathcal{B} = (\Pi, S, \{\iota\}, S, \Delta)$ where

- $\Pi = [(I \cup O) \to \bigcup_{c \in I \cup O} type(c)]$ and
- $\Delta = \{(s, l, t) \mid \exists \theta \in C(\Sigma)^\to : (s, \theta, t) \in \delta \wedge \theta = l\}$.

The TSPA A is finite. Thus, S, I, O, and $\bigcup_{c \in I \cup O} type(c)$ are finite. As therefore Π and Δ are finite, $\mathcal{B}$ is a well-defined BA. It remains to show that $behs(A) = \mathcal{L}(\mathcal{B})$.

$\subseteq$: Let $s_0, \theta_1, s_1, \theta_2, s_2, ... \in execs(A)$ be an execution of A. By definition of execution $s_{j-1} \xrightarrow{\theta_j} s_j$ for all $j > 0$ and $s_0 = \iota$. By definition of $\mathcal{B}$ we have that $(s_{j-1}, \theta_j, s_j) \in \Delta$ for all $j > 0$. Thus, $s_0, s_1, s_2, ...$ is a run of $\mathcal{B}$ on the word $\theta_1, \theta_2, ...$ Since all states $s \in S$ are accepting, the run is accepting. Thus, $beh(s_0, \theta_1, s_1, \theta_2, s_2, ...) = \theta_1, \theta_2, ... \in \mathcal{L}(\mathcal{B})$.

$\supseteq$: Assume that $\sigma = \sigma_1, \sigma_2, \sigma_3, ... \in \mathcal{L}(\mathcal{B})$ and let $q_0, q_1, q_2, ...$ be an accepting run of $\mathcal{B}$ on σ. By definition of run we have $q_{j-1} \xrightarrow{\sigma_j} q_j$ for all $j > 0$. Thus, by definition of Δ we have that there are $\theta_j \in C(\Sigma)^\to$ with $(q_{i-1}, \theta_j, q_j) \in \delta$ and $\theta_j = \sigma_j$ for each $j > 0$. Thus $\tau = q_0, \theta_1, q_1, \theta_2, ...$ is an execution of A. Therefore, by definition of behavior we have that $beh(\tau) = \sigma_1, \sigma_2, ... \in behs(A)$ is a behavior of A. □

C. Semantic Differencing for Component Behavior

The semantics of components are defined as sets of TSSPFs. We denote the semantics of a component c by $[\![c]\!]$. Each function $f \in [\![c]\!] \setminus [\![c']\!]$ in the semantics of one component c that is no member of the semantics of another component c' is a representative for the difference between the components' semantics. However, such a representative defines an output for each possible component input, even if the semantic difference is only given by a single input/output pair. Thus, such a TSSPF does not effectively reveal the differences between the component semantics. In contrary, the exact input/output pairs for which there is a function in the semantics of one component that maps the input to the output and for which there is no function in the semantics of the other component mapping the input to the output clearly reveals a difference. If two components have different interfaces, *i.e.,* they read and write from and to different channels, each input/output pair of

the first component indicates a difference to the semantics of the other component. However, if the components have channels of the same types one can easily avoid this problem by channel renaming and hiding [3]. Thus, we define the semantic difference for components having the same interfaces, only.

Definition 18 (Diff Witness)**.** *Let $F_1, F_2 \subseteq [I^\Omega \xrightarrow{wc} O^\Omega]$ be two sets of TSSPFs. A diff witness distinguishing F_1 from F_2 is a communication history $w \in (I \cup O)^\Omega$ satisfying*

$\exists f_1 \in F_1 : f_1(w|_I) = w|_O \wedge \forall f_2 \in F_2 : f_2(w|_I) \neq w|_O$.

We denote by $\Delta(F_1, F_2)$ the set of all diff witnesses distinguishing F_1 from F_2.

A set of diff witnesses may be finite but is typically infinite. The following theorem reveals the relation between the differences of the behaviors and of the semantics of TSPAs.

Theorem 7. *Let $A_1 = (\Sigma, S_1, \iota_1, \delta_1)$ and $A_2 = (\Sigma, S_2, \iota_2, \delta_2)$ with $\Sigma = (I, O)$ be two TSPAs and let $w \in (I \cup O)^\Omega$ be a communication history. The following holds:*

$w \in \Delta([\![A_1]\!], [\![A_2]\!]) \Leftrightarrow$
$\exists \alpha \in behs(A_1) : w = h_\alpha \wedge \alpha \notin behs(A_2)$.

Proof. Let A_1, A_2, and w be given as above.

$\Rightarrow$: Assume $w \in \Delta([\![A_1]\!], [\![A_2]\!])$ is a diff witness. By definition of Δ, we have that there is a function $f_1 \in [\![A_1]\!]$ such that $f_1(w|_I) = w|_O$ and $f(w|_I) \neq w|_O$ for all $f \in [\![A_2]\!]$. In the following let f_1 be such a function that satisfies the above. By definition of $[\![\cdot]\!]$ we have that $\forall i \in I^\Omega : \exists \alpha \in behs(A_1) : i = h_\alpha|_I \wedge f_1(i) = h_\alpha|_O$. When substituting $w|_I$ for i, we get $\exists \alpha \in behs(A_1) : w|_I = h_\alpha|_I \wedge f_1(w|_I) = h_\alpha|_O$. Since $f_1(w|_I) = w|_O$ we can substitute $w|_O$ for $f_1(w|_I)$ and obtain $\exists \alpha \in behs(A_1) : w|_I = h_\alpha|_I \wedge w|_O = h_\alpha|_O$, which is equivalent to $\exists \alpha \in behs(A_1) : w = h_\alpha$.

In the following, let such an α with $w = h_\alpha$ be given. It remains to show $\alpha \notin behs(A_2)$. Towards a contradiction we assume $\alpha \in behs(A_2)$. By Thm. 3 we get there is a function $g \in [\![A_2]\!]$ such that $g(h_\alpha|_I) = h_\alpha|_O$. By definition of α we have $w = h_\alpha$ and thus $g(w|_I) = w|_O$. But since $w \in \Delta([\![A_1]\!], [\![A_2]\!])$, it holds that $\forall f \in [\![A_2]\!] : f(w|_I) \neq w|_O$. Substituting g for f yields a contradiction.

$\Leftarrow$: Assume there is an $\alpha \in behs(A_1)$ such that $w = h_\alpha$ and $\alpha \notin behs(A_2)$. Using Thm. 3 we get there is a function $f \in [\![A_1]\!]$ such that $f(h_\alpha|_I) = h_\alpha|_O$. By definition of w we have that $w = h_\alpha$ and thus obtain by substitution that $f(w|_I) = w|_O$. Thus there is a function $f \in [\![A_1]\!]$ such that $f(w|_I) = w|_O$. It remains to show that $g(w|_I) \neq w|_O$ for all $g \in [\![A_2]\!]$. Towards a contradiction we assume that there is a function $g \in [\![A_2]\!]$ such that $g(w|_I) = w|_O$. By definition of $[\![\cdot]\!]$ we get that $\forall i \in I^\Omega : \exists \beta \in behs(A_2) : i = h_\beta|_I \wedge g(i) = h_\beta|_O$. Substituting $w|_I$ for i we obtain $\exists \beta \in behs(A_2) : w|_I = h_\beta|_I \wedge g(w|_I) = h_\beta|_O$. Since by assumption $w|_I = h_\alpha|_I$ and $g(w|_I) = w|_O$ by definition of g, this is equivalent to $\exists \beta \in behs(A_2) : h_\alpha|_I = h_\beta|_I \wedge w|_O = h_\beta|_O$. By assumption we have $w = h_\alpha$ and thus obtain via substitution $\exists \beta \in behs(A_2) : h_\alpha|_I = h_\beta|_I \wedge h_\alpha|_O = h_\beta|_O$, which is equivalent to $\exists \beta \in behs(A_2) : h_\alpha = h_\beta$. Using the definitions of h_α and h_β, this is equivalent to $\exists \beta \in behs(A_2) : \alpha = \beta$, which is equivalent to $\alpha \in behs(A_2)$ and contradicts the assumption. □

In the previous section, we presented a translation from finite TSPAs to BAs. Each word accepted by a BA resulting from such a translation corresponds to a behavior of the input TSPA. Existing algorithms for checking language inclusion and counterexample generation for BAs can thus be used for refinement checking and diff witness generation of architectures as defined above: Given two TSPAs A_1 and A_2 we use the translation defined in proof of Thm. 6 to obtain two Büchi automata $\mathcal{B}_1$ and $\mathcal{B}_2$ such that $\mathcal{L}(\mathcal{B}_1) = behs(A_1)$ and $\mathcal{L}(\mathcal{B}_2) = behs(A_2)$. Using Thm. 7 and Thm. 6 we can transform a word accepted by $\mathcal{B}_1$ that is not accepted by $\mathcal{B}_2$ to a corresponding diff witness that semantically distinguishes the automata A_1 and A_2. By definition, if $\mathcal{L}(\mathcal{B}_1) = \mathcal{L}(\mathcal{B}_2)$ then the two TSPAs A_1 and A_2 are equivalent and if $\mathcal{L}(\mathcal{B}_1) \subseteq \mathcal{L}(\mathcal{B}_2)$ then the automaton A_1 refines the automaton A_2.

V. Implementation and Evaluation

This section recapitulates the MontiArcAutomaton ADL [23], [25], presents the application of refinement checking to its models and evaluates our approach.

A. The MontiArcAutomaton ADL

The MontiArcAutomaton ADL [23], [25] comprises the modeling elements common to many popular component & connector ADLs [20], *i.e.,* hierarchical components with interfaces of typed, directed ports and unidirectional connectors (typed FIFO channels) exchanging messages between these ports. The components are black-boxes and either atomic or composed: atomic components yield behavior descriptions in form of embedded automata (following the I/O$^\omega$ [27] paradigm) or in form of Java implementations. The behavior of composed components solely emerges from the interaction of their subcomponents. Components are scheduled by a global clock and perform cycles of 1.) read all messages on incoming ports; 2.) compute behavior (which might entail invoking subcomponents); 3.) produce a single message to each outgoing port. Each computation consumes a time slice, *i.e.,* the output for messages received at the global clock's i-th tick is produced at its i+1-th tick earliest. The MontiArcAutomaton ADL also distinguishes between component types and their instances, supports component type inheritance, generic type parameters for components (*e.g.,* to be used with generic port types), and constructor-like configuration of these instances.

The MontiArcAutomaton ADL is a textual modeling language implemented with the MontiCore [17] language workbench. The textual representation of the composed component type `Elevator` is illustrated in Listing 1. It begins with the keyword "component", followed by the component type's name and a body delimited by curly brackets (l. 1). The body contains an interface of typed ports (ll. 2-5), declares three subcomponents (ll. 7-9), and multiple connectors (ll. 11-13). The subcomponent declarations reference component types imported from artifacts (such as `Control`).

MontiArcAutomaton

```
component Elevator {
  port in Bool req1, in Bool at1,
       // ... further ports ...
       out Bool open, out Bool close,
       out Clear clear;

  component Control ctrl; // named
  component Motor m;      // subcomponent
  component Door d;       // instances

  connect req1 -> control.req1;
  // ... further connectors ...
  connect control.clear -> clear;
}
```

Listing 1. Textual representation of the component `Elevator`.

B. Semantic Differencing of MAA Components

The implementation comprises a translation from MontiArcAutomaton architectures to semantically equivalent TSPAs. TSPAs are only handled internally as representatives for sets of TSSPFs modeling component semantics and are not explicitly modeled by component developers. Each atomic component directly translates to a TSPA. The TSPA of a composed component is computed by composing the TSPAs of its subcomponents according to the architectural configuration defined by the composed component's connectors. The implementation further consists of a translation from TSPAs to BAs and generators that produce models in the "BA format", which is the input format of the tool RABIT [2]. In case a BA does not refine another BA, RABIT provides a counterexample serving as a concrete disproof for refinement. The counterexamples can be translated back to diff witnesses. Using the tool chain described above enables automated refinement checking and diff witness generation for MontiArcAutomaton architectures.

C. Evaluation

We evaluated the approach to semantic differencing with six MontiArcAutomaton architectures previously used for evaluation in [26]. We specifically chose these architectures for evaluation since the approach presented in [26] failed for some specifications, which we considered to be challenging, and to enable comparability. The architectures were slightly modified for this evaluation to resolve technical MontiArcAutomaton version compatibility issues. We reused the completion strategies [26] for completing the automata implementations of the architectures' atomic components.

The first architecture is given by an implementation of an elevator control system (ECS) (cf. Sec. II). It comprises 3 composed and 5 atomic components. The second example consists of four variants of a mobile robot. We only report on the evaluation of the most challenging variant. This variant comprises 4 components in total whereof 3 components are atomic. The last architecture implements a pump station consisting of 3 composed and 10 atomic components.

In [26], for each of the architectures three specification checks are executed: it is checked whether the semantics of a component is equal to itself, whether a component refines a component with the same interfaces that implements arbitrary behavior, *i.e.,* all possible behaviors, and whether the semantics of a component are equal to the semantics of a component implementing arbitrary behavior. We performed the same checks on a computer with 3.0 GHz Intel Core i7 CPU, 16 GB Ram, Windows 10, and RABIT 2.4 using our translation from MontiArcAutomaton architectures to BAs and the language inclusion checking tool RABIT [2] (cf Sec. V-B).

TABLE I
TIME FOR REFINEMENT CHECKING AND DIFF WITNESS CALCULATION.

	$\Delta([\![\cdot]\!],[\![\cdot]\!])$	$\Delta([\![\cdot]\!],Chaos)$	$\Delta(Chaos,[\![\cdot]\!])$
Floors	62ms	526ms	909ms
Elevator	75ms	2510ms	6064ms
ECS	463ms	7166ms	16537ms
SensorReading	94ms	764ms	1558ms
Controller	15ms	17ms	43ms
Pumpstation	119ms	334ms	486ms
MobileRobot	52ms	75ms	106ms

Table I summarizes the computation times of RABIT given the BAs resulting from the transformation as input. For component ECS, for instance, checking whether it refines itself took 463ms, checking refinement with arbitrary behavior took 7166ms, and calculating a diff witness distinguishing the component from arbitrary behavior took 16537ms. Table II depicts the sizes of the automata resulting from the translations. For component ECS, for instance, the TSPA and the BA resulting from the transformation have 746 states and 98496 transitions. RABIT reported the tool has reduced the BA to 8 states and 1728 transitions after internal preprocessing. For every component we modeled arbitrary behavior (Chaos) with a TSPA consisting of one state and a transition for every possible component input/output combination. The TSPA and the BA modeling arbitrary behavior for component ECS, for instance, comprise 472392 transitions (*cf.* Table II). In contrast to the translation from MontiArcAutomaton architectures to the model checker Mona [26], our implementation succeeded for all example architectures. The longest computation time of our evaluation (16537ms, *cf.* Table I) resulted from semantic differencing arbitrary behavior with the ECS component. We conclude that our translation provides promising results. Nevertheless the evaluation was only performed on a few specific architectures. Thus the results are not generalizable to all possible architectures: the time needed by our tool may vary strongly from system to system.

VI. DISCUSSION

If the semantics domain of an ADL is overly general, undecidability of the underlying mathematical problems renders automated formal verification impossible. Then, architecture properties have to be proven manually, which is too expensive to be carried out in continuous architecture modeling and thus hinders employing agile development in architecture modeling projects: little changes to requirements or implementations can

TABLE II
THE NUMBERS OF STATES AND TRANSITIONS OF THE TSPAS TRANSLATED FROM THE ARCHITECTURES AND OF THE GENERATED BAS.

	TSPA/BA		BA AP		Chaos
	#states	#trans.	#states	#trans.	#trans.
Floors	32	1024	32	1024	23328
Elevator	34	10206	1	729	236196
ECS	746	98496	8	1728	472392
SensorReading	2	1296	2	1296	69984
Controller	1	9	1	9	108
Pumpstation	6	3888	4	2592	17496
MobileRobot	150	2700	12	216	1152

entail changing many manually performed proofs. In contrast, where automated formal verification is possible, sound and complete proofs can be generated automatically, supporting agile implementation evolution.

FOCUS is a comprehensive framework that supports specifying the observable input/output behavior of interactive systems. Its complexity requires carrying out proofs for system behavior verification manually. FOCUS provides various constructs for describing the semantics of distributed systems [24]. Examples are relations, set-based functions, sets of functions, assumption/guarantee predicates, or state-based representations. As identified in [24], the most fine-grained domain for describing the semantics of distributed systems using FOCUS are sets of SPFs. Independent of the style, specifications can describe timed or untimed behavior. Untimed behavior only considers the causality regarding the order of inputs and outputs. Timed specifications additionally concern causality regarding the passage of time. Many requirements are not only concerned with the order of messages but also state requirements with respect to passage of time. Thus, we employ a variant of the timed subset of FOCUS and thereby use sets of TSSPFs as semantics domain [24], [27].

Our approach is limited to systems where the data types' domains are finite and is restricted to the time-synchronous model of computation. However, our system model fits well into the kinds of systems developed for embedded systems such as automotive or robotics applications. Thus, our results enable fully automated tool support for many systems in such domains. Emphasizing that our approach cannot be generalized to the timed model of FOCUS as, for example, used in [12], is important: Timed SPFs (*cf.* [12], [24], [27]), for instance, are too general to be applicable to our approach. A timed SPF processes infinite sequences of finite sequences (of arbitrary lengths) of messages. Each of the finite sequences represents a finite stream of messages received or sent by a component in a single time unit. In contrast, TSSPFs only process single messages per time unit. The set of finite streams of messages over a non-empty finite data type is already infinite. Thus, for each time unit, a timed SPF needs to define a possible behavior for infinitely many tuples of input streams, whereas a TSSPF needs to define a reaction for all possible tuples of input messages, which are finitely many if the messages' data types are finite. From a practical viewpoint it is rarely required to specify the reaction in a time unit in response to the receipt of an arbitrary number of messages. Usually it either requires to handle single messages (TSSPFs) or sequences of messages where the length of the sequence is bound by an arbitrary but fixed natural number. The latter can be reduced to the former by introducing lists of fixed length as message types.

The underlying theoretical problem for semantic differencing used in our approach is language inclusion checking between Büchi automata. Its complexity can be considered as another limitation of our approach. However, our main focus is not verifying a system's properties (*e.g.,* refinement or semantic differencing) within seconds, which is most often already rendered impossible due to the complex nature of the safety critical system under development. We believe that nonetheless the possibility to apply formal fully automated verification (*e.g.,* over night) greatly facilitates continuous architecture modeling.

VII. RELATED WORK

Studies on the verification techniques of ADLs have been conducted, *e.g.,* in [30] and [33]. The study in [33] surveys verification techniques supported by ADLs with formal semantics, the translation of architectures to inputs for model checkers, and tool support as well as usability, scalability, and expressiveness. As supported by our approach, the study states that architecture verification for practical applications requires tool-support and automation. The study in [30] compares different verification tools and applies them to various ADLs. All architectures are transformed into intermediate labeled transition systems before the verification tools are applied, hampering the direct comparison with our approach.

The following surveys concrete approaches for formally analyzing hierarchical architecture descriptions. AutoFOCUS 3 [14] is a tool for the development of reactive embedded systems that also bases its semantics on FOCUS [5]. Although AutoFOCUS 3 supports model checking architectures against LTL and CTL formulas that specify properties concerning component behavior [6], we are not aware of a fully automated refinement checking method for AutoFOCUS 3. The π-ADL supports statistical model checking for verifying dynamic software architectures against DynBLTL properties [7]. To this effect, a statistical model of finite system executions is built and the probability of satisfying a property within a confidential bound is calculated. This approach is particularly tailored to dynamic architectures and is only concerned with finite traces. In contrast, our approach deals with infinite traces, static architectures, and full certainty. Refinement of architectures specified with timed I/O is described in [16]. Similar to behaviors of TSPAs, the semantics of a timed automaton is given by a set of traces. Refinement between timed I/O automata is defined similar as in our approach by trace inclusion. However, timed I/O automata are only marked with one message per transition and composition is defined differently. Further, the timing concept of I/O automata is more powerful and complicated than the one

of our approach [12]. A game-based extension of the timed I/O automaton model enabling tool supported refinement checking has been proposed in [8]. Another approach to automated refinement checking based on the time synchronous frame of FOCUS is described in [22], [26]. This approach is based on a relational semantics domain where the semantics of a component is given as a relation between the component's possible inputs and outputs. In contrast, our approach uses a more fine grained [24] semantics domain consisting of sets of functions. Refinement checking in [22], [26] relies on translating component semantics into WS1S and is implemented using the model checker Mona [10]. The approach suffers from the tool's high computational complexity, which is grounded in the non-elementary complexity of solving W1S1 problems. In contrast, we define a translation to Büchi automata and thereby obtain a PSPACE-complete complexity for refinement checking. While the relational approach is based on analyzing the result from composing the semantics of the individual components of a system, our approach first syntactically composes the individual components and bases analysis on the semantics of the compound.

VIII. Conclusion

We have presented an implementation of stepwise refinement for ADLs using a subset of the FOCUS semantics for distributed systems. This subset consists of time-synchronous stream processing functions, and hence the corresponding software architecture models, can be translated to a variant of port automata. Via a transformation from port automata to Büchi automata, we can reduce component refinement to language inclusion. As the evaluation has shown, the fully automated implementation supports checking refinement for MontiArcAutomaton architecture models in reasonable time. While this might be improved further, we believe our approach facilitates continuous architecture modeling.

References

[1] RABIT Tool Homepage, 2016. http://http://www.languageinclusion.org/ [accessed 2016-12-31].

[2] 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 *International Conference on Concurrency Theory, CONCUR 2011*, 2011.

[3] Manfred Broy. A Logical Basis for Component-Oriented Software and Systems Engineering. *The Computer Journal*, 2010.

[4] Manfred Broy and Max Fuchs. The Design of Distributed Systems - An Introduction to FOCUS. Technical report, TU Munich, 1992.

[5] Manfred Broy and Ketil Stølen. *Specification and Development of Interactive Systems. Focus on Streams, Interfaces and Refinement*. Springer Verlag Heidelberg, 2001.

[6] Alarico Campetelli, Florian Hölzl, and Philipp Neubeck. User-friendly Model Checking Integration in Model-based Development. In *International Conference on Computer Applications in Industry and Engineering*, 2011.

[7] Everton Cavalcante, Jean Quilbeuf, Louis-Marie Traonouez, Flávio Oquendo, Thaís Batista, and Axel Legay. Statistical Model Checking of Dynamic Software Architectures. In *European Conference on Software Architecture*, 2016.

[8] 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 *ACM International Conference on Hybrid Systems: Computation and Control*, 2010.

[9] Vincent Debruyne, Françoise Simonot-Lion, and Yvon Trinquet. EAST-ADL - An architecture description language. In *Architecture Description Languages*. Springer, 2005.

[10] Jacob Elgaard, Nils Klarlund, and Anders Møller. MONA 1.x: New techniques for WS1S and WS2S. In *Computer-Aided Verification*, 1998.

[11] Peter H. Feiler and David P. Gluch. *Model-Based Engineering with AADL: An Introduction to the SAE Architecture Analysis & Design Language*. Addison-Wesley Professional, 2012.

[12] Radu Grosu and Bernhard Rumpe. Concurrent Timed Port Automata. Technical report, TU Munich, 1995.

[13] Radu Grosu, Ketil Stølen, and Manfred Broy. A Denotational Model for Mobile Point-to-Point Data-flow Networks with Channel Sharing, 1997.

[14] Florian Hölzl and Martin Feilkas. AutoFocus 3 - A Scientific Tool Prototype for Model-Based Development of Component-Based, Reactive, Distributed Systems. In *Model-Based Engineering of Embedded Real-Time Systems*, 2007.

[15] Bengt Jonsson. A Fully Abstract Trace Model for Dataflow and Asynchronous Networks. *Distributed Computing*, 1994.

[16] 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 *IEEE Real-Time Systems Symposium (RTSS 2003)*, 2003.

[17] Holger Krahn, Bernhard Rumpe, and Steven Völkel. MontiCore: Modular Development of Textual Domain Specific Languages. In *Proceedings of Tools Europe*, 2008.

[18] Orna Kupferman and Moshe Y. Vardi. Verification of Fair Transition Systems. In *International Conference on Computer Aided Verification*, 1996.

[19] Ivano Malavolta, Patricia Lago, Henry Muccini, Patrizio Pelliccione, and Antony Tang. What Industry Needs from Architectural Languages: A Survey. *IEEE Transactions on Software Engineering*, 2013.

[20] Nenad Medvidovic and Richard N. Taylor. A Classification and Comparison Framework for Software Architecture Description Languages. *IEEE Transactions on Software Engineering*, 2000.

[21] Object Management Group. OMG Unified Modeling Language (OMG UML), Superstructure Version 2.3 (10-05-05), May 2010. http://www.omg.org/spec/UML/2.3/Superstructure/PDF/ [accessed 2017-01-13].

[22] Jan Oliver Ringert. *Analysis and Synthesis of Interactive Component and Connector Systems*. Shaker Verlag, 2014.

[23] 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)*, 2015.

[24] 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.

[25] Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. *Architecture and Behavior Modeling of Cyber-Physical Systems with MontiArcAutomaton*. Shaker Verlag, 2014.

[26] Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. Model-Based Specification of Component Behavior with Controlled Underspecification. In *Modellbasierte Entwicklung eingebetteter Systeme (MBEES'16)*, 2016.

[27] Bernhard Rumpe. *Formale Methodik des Entwurfs verteilter objektorientierter Systeme*. Doktorarbeit, TU Munich, 1996.

[28] Christian Schlegel, Andreas Steck, and Alex Lotz. Model-Driven Software Development in Robotics: Communication Patterns as Key for a Robotics Component Model. In *Introduction to Modern Robotics*. iConcept Press, 2011.

[29] Frank Strobl and Alexander Wisspeintner. Specification of an Elevator Control System. Technical report, TU Munich, 1999.

[30] Jeffrey J.P. Tsai and Kuang Xu. A comparative study of formal verification techniques for software architecture specifications. *Annals of Software Engineering*, 2000.

[31] Rob Van Ommering, Frank Van Der Linden, Jeff Kramer, and Jeff Magee. The Koala Component Model for Consumer Electronics Software. *Computer*, 2000.

[32] Markus Völter, Thomas Stahl, Jorn Bettin, Arno Haase, Simon Helsen, and Krzysztof Czarnecki. *Model-Driven Software Development: Technology, Engineering, Management*. Wiley, 2013.

[33] Pengcheng Zhang, Henry Muccini, and Bixin Li. A Classification and Comparison of Model Checking Software Architecture Techniques. *Journal of Systems and Software*, 2010.