Retrofitting Controlled Dynamic Reconfiguration into the Architecture Description Language MontiArcAutomaton

Robert Heim[1], Oliver Kautz[1], Jan Oliver Ringert[2], Bernhard Rumpe[1,3], Andreas Wortmann[1]

[1] Software Engineering, RWTH Aachen University, Aachen, Germany
[2] School of Computer Science, Tel Aviv University, Israel
[3] Fraunhofer FIT, Aachen, Germany

Abstract. Component & connector architecture description languages (C&C ADLs) provide hierarchical decomposition of system functionality into components and their interaction. Most ADLs fix interaction configurations at design time while some express dynamic reconfiguration of components to adapt to runtime changes. Implementing dynamic reconfiguration in a static C&C ADL by encoding it into component behavior creates implicit dependencies between components and forfeits the abstraction of behavior paramount to C&C models. We developed a mechanism for retrofitting dynamic reconfiguration into the static C&C ADL MontiArcAutomaton. This mechanism lifts reconfiguration to an architecture concern and allows to preserve encapsulation and abstraction of C&C ADLs. Our approach enables efficient retrofitting by a smooth integration of reconfiguration semantics and encapsulation. The new dynamic C&C ADL is fully backwards compatible and well-formedness of configurations can be statically checked at design time. Our work provides dynamic reconfiguration for the C&C ADL MontiArcAutomaton.

1 Introduction

Component & connector (C&C) architecture description languages (ADLs) [1,2] combine the benefits of component-based software engineering with model-driven engineering (MDE) to abstract from the accidental complexities [3] and notational noise [4] of general-purpose programming languages (GPLs). They employ abstract component models to describe software architectures as hierarchies of connected components. This allows to abstract from ADL implementation details to a conceptual level applicable to multiple C&C ADLs.

In many ADLs, including MontiArcAutomaton [5], the configuration of C&C architectures is fixed at design time. The environment or the current goal of the system might however change during runtime and require dynamic adaptation of the system [6] to a new configuration that only includes a subset of already existing components and their interconnections as well as introduces new components and connectors. To support dynamic adaptation, a C&C architecture either has to adapt its configuration at runtime or it must encode adaptation in the

[HKRRW16] R. Heim, O. Kautz, J. O. Ringert, B. Rumpe, A. Wortmann:
Retrofitting Controlled Dynamic Reconfiguration into the Architecture Description Language MontiArcAutomaton.
In: Software Architecture - 10th European Conference, ECSA 2016, Copenhagen, Denmark, November 28 - December 2, 2016, Proceedings, volume 9839 of LNCS, pages 175-182. Springer, 2016.
www.se-rwth.de/publications/

behaviors of the related components. This encoding introduces implicit dependencies between components and forfeits the abstraction of behavior paramount to C&C models. It imposes co-evolution requirements on different levels of abstraction and across components. Dynamic reconfiguration mechanisms and their formulation in ADLs help to mitigate these problems by formalizing adaptation as structural reconfiguration. This ensures that components keep encapsulating abstractions over functionality.

We develop a concept for retrofitting controlled dynamic adaptation into the static C&C ADL MontiArcAutomaton. The concept lifts reconfiguration to the conceptual level of components and connectors to preserve the fundamental abstraction and encapsulation mechanisms of C&C ADLs. It is *controlled* in the sense that it enables a restricted dynamism to benefit from greater run-time flexibility without loosing the validation properties of static configurations and their testability. Our concept enables efficient retrofitting by a smooth integration of reconfiguration semantics and encapsulation. It is implemented in the C&C ADL MontiArcAutomaton and its code generation framework. Our design for retrofitting reconfiguration kept changes to the language and code generation local and the resulting dynamic C&C ADL is fully backwards compatible.

Sect. 2 gives an example to demonstrate benefits of dynamic reconfiguration. Afterwards, Sect. 3 introduces our concept of controlled dynamic reconfiguration for MontiArcAutomaton and describes its implementation. Sect. 4 discusses our approach and compares it to related work and Sect. 5 concludes.

2 Example

Automatic transmission is a commonly used type of vehicle transmission, which can automatically change gear ratios as a vehicle moves. The driver may choose from different transmission operating modes (TOMs) such as Park, Reverse, Neutral, Drive, Sport, or Manual while driving. Depending on the chosen TOM, a transmission control system decides when to shift gears.

A C&C architecture might provide one component for each different shifting behavior. If the architecture is static, components must exchange control information at runtime to decide whether they take over the shifting behavior. The architect then has to define and implement inter-component protocols for switching between different behaviors. Dynamic reconfiguration enables to model structural flexibility in composed software components explicitly. Here, the transmission control system's architecture uses only components related to the selected transmission operating mode by reconfiguring connections between components as well as by dynamic component activation and instantiation.

Fig. 1 (top) depicts a C&C model showing the composed component `Shift-Controller`. It contains the three subcomponents `manual`, `auto`, and `sport` for the execution of different gear shifting behaviors and the subcomponent `scs` for providing sensor data comprising the current revolutions per minute (rpm), the vehicle inclination (vi), and the throttle pedal inclination (tpi) encoded as integers. The component `ShiftController` has an interface of type `TOM` to

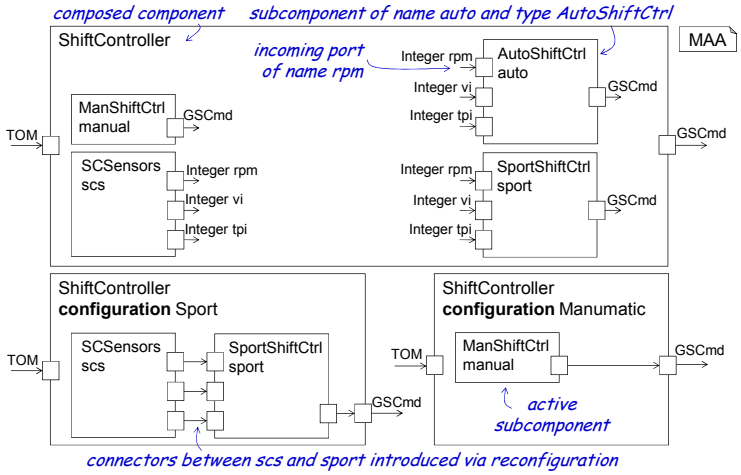

Fig. 1. Three configurations of the `ShiftController` component. Top: Initial configuration. Middle: Configuration for shifting gears during the transmission operating mode `Sport`. Bottom: Configuration for shifting gears during the transmission operating mode `Manumatic`.

receive the currently selected TOM and one interface of type `GSCmd` to emit commands for shifting gears. Immediately after engine start up all subcomponents are neither active nor connected (top configuration). Once the currently selected TOM is known to component `ShiftController`, it changes its configuration accordingly and starts the contributing subcomponents (bottom configurations). While the currently selected TOM is `Sport` (bottom left configuration), only subcomponents `scs` and `sport` are active to emit sensor data and commands for shifting gears, whereas only subcomponent `manual` is active when the currently selected TOM is `Manumatic` (bottom right configuration). Making the active components and connectors explicit increases comprehensibility of the architecture. The deactivation of components at runtime has further practical benefits, such as saving computation time and power consumption.

3 Retrofitting Controlled C&C Reconfiguration

We present a concept for retrofitting dynamic reconfiguration into the MontiArc-Automaton ADL [5]. All possible component configurations and their transitions are defined at design time, which allows static analysis to prevent malformed configurations from being deployed. At runtime, the reconfiguration is applied when pre-defined conditions for reconfiguration are met. No configuration validity changes are required at runtime. The reconfiguration mechanism is self-directed and pre-defined: Initiation and application of dynamic reconfiguration can only be applied by a component itself. This allows independent and reusable

specifications of composed components. All reconfiguration possibilities are specified and fully available in the reconfiguring component. This facilities analysis and application. In addition, our approach enables component instantiation and removal to gain greater flexibility.

This section describes preliminaries on the MontiArcAutomaton ADL, an overview of our concept, and its implementation within MontiArcAutomaton.

3.1 The MontiArcAutomaton ADL

MontiArcAutomaton [5] is an architecture modeling infrastructure comprising the MontiArcAutomaton C&C ADL [7] as well as model transformation and code generation capabilities. The MontiArcAutomaton ADL enables to model C&C architectures as hierarchies of connected components. Components are black-boxes that consume input messages and produce output messages. Atomic components employ embedded behavior models or attached GPL artifacts to perform computations. The behavior of composed components emerges from the interaction of their subcomponents. These interact via unidirectional connectors between the typed ports of their interfaces. Components and connectors cannot be instantiated, nor removed at runtime. The data types of ports are defined in terms of class diagrams. The MontiArcAutomaton ADL distinguishes component types from instances, supports component configuration, and components with generic type parameters. Its infrastructure supports transformation of platform-independent architecture models into platform-specific models and composition of code generators to reuse generation capabilities for different aspects.

3.2 Overview: Component Modes for Dynamic Reconfiguration

Our approach for modeling dynamic reconfiguration relies on explicit modes, which fully define possible configurations. A mode is a configuration of a composed component and components can only switch between their pre-defined modes. In modes, we distinguish subcomponent instantiation and activation: the lifecycle of instantiated subcomponents ends with any mode switch, while deactivated subcomponents retain their state between modes. Components switch between their modes via mode transitions, which are again fixed at design time. Each mode transition consists of a source mode, a target mode, and a guard expression (e.g., over ports of the composed component and its direct subcomponents). Intuitively, when the source mode equals the current mode of a corresponding component instance, and the guard is satisfied, reconfiguration to the target mode takes place. The mode transitions of a component define a state machine over the state space of component modes with input of data messages observable within the component.

MontiArcAutomaton distinguishes component types and instances. Modes and mode transitions are defined on the component type level of MontiArcAutomaton. However, at runtime each component instance reconfigures itself independently based on its current mode and observable messages. Thus there is no synchronization overhead induced by component types.

MontiArcAutomaton

```
1  component ShiftController {
2    port in TOM tom, out GSCmd cmd;
3
4    component ManShiftCtrl manual;
5    component AutoShiftCtrl auto;
6    component SCSensors scs;
7
8    mode Idle {}   mode Manumatic { /* ... */ }   mode Auto { /* ... */ }
9
10   mode Sport, Kickdown {
11     activate scs;
12     component SportShiftCtrl sport;
13     connect scs.rpm -> sport.rpm; connect scs.vi -> sport.vi;
14     connect scs.tpi - sport.tpi;  connect sport.cmd -> cmd;
15   }
16
17   modetransitions {
18     initial Idle;
19     Idle -> Auto [tom == DRIVE];
20     Auto -> Kickdown [scs.tpi > 90 && tom == DRIVE];
21     Kickdown -> Auto [scs.tpi < 90 && tom == DRIVE]; // further transitions
22   }
23 }
```

Listing 1. Excerpt of the `ShiftController` component type definition with five modes (ll. 8-15) and a mode transition automaton (ll. 17-22).

3.3 Defining Component Modes

The C&C core concepts identified in [1] and implemented in MontiArcAutomaton consist of components with interfaces, connectors, and architectural configurations (i.e., topologies of subcomponents). In MontiArcAutomaton, architectural configurations are defined locally within composed components. For dynamic reconfiguration we extend the existing single component configuration with multiple modes, where each mode expresses one configuration. We continue the `ShiftController` example depicted in Fig. 1 in MontiArcAutomaton syntax with support for modes shown in Listing 1.

Subcomponents with instances shared between multiple modes are defined in the body of the composed component and can be activated or deactivated in modes. As an example, the subcomponent `scs` of type `SCSensors` is defined in Listing 1, l. 6 and activated in modes `Sport` and `Kickdown` in l. 11. Subcomponents are deactivated by default, e.g., subcomponents `manual` and `auto`, ll. 4-5 are deactivated in mode `Sport`, ll.10-15. In addition, subcomponents can be instantiated when entering a mode and destroyed when switching to another mode. As an example, instances of subcomponent `sport` of type `SportShiftCtrl` as defined in l. 12 are unique to modes `Sport` and `Kickdown`. Connectors between components are defined for each mode.

For each mode we can determine at design time whether the expressed configuration is a valid MontiArcAutomaton component configuration. In addition some well-formedness rules need to be checked: (1) Each mode of each composed component type has a unique name. (2) Each subcomponent instantiated in a mode has a unique name in the context of the component containing the mode. (3) Each subcomponent instance referenced in a mode exists.

3.4 Defining Mode Transitions

Composed components with multiple modes change their configuration based on observable messages. The messages observable by a component are messages on its own ports and messages on ports of its subcomponent instances.

All mode transitions are defined locally within the composed component. An example is shown in Listing 1, ll. 17-22. Following the keyword `modetransitions`, the mode automaton contains a single initial mode declaration (l. 18) and multiple transitions (ll. 19-21). These describe mode switches and their conditions in guard expressions. Guard expression are written in a language resembling expressions in an object-oriented GPL (e.g., it uses dot-notation to reference messages on ports of components).

The following well-formedness rules apply for the definition of mode transitions: (4) Each composed component type has exactly one initial mode. (5) The subcomponent interface elements referenced in guards exist. (6) Modes referenced in transitions exist in the containing component.

3.5 Implementation Details of Retrofitting

We now highlight some implementation details of retrofitting dynamic reconfiguration into the MontiArcAutomaton infrastructure.

On the language level, modes reuse existing modeling elements for subcomponents, ports, and connectors. Mode transitions reuse the automata modeling elements presented in [7], which allowed us to reuse existing well-formedness rules of the MontiArcAutomaton ADL to describe the static semantics of dynamic reconfiguration. We added the well-formedness rules described above.

We extended the existing code generators [5] to enable integration of dynamic reconfiguration with the dynamic semantics of MontiArcAutomaton. Due to localizing the impact of reconfiguration in composed components only, retrofitting into code generation was straightforward. The extended MontiArcAutomaton ADL and the generated code are backwards compatible because we could transfer the encapsulation of reconfiguration from the model level to the code level.

4 Discussion and Related Work

The importance of dynamic reconfiguration has long been recognized [8] and is implemented for multiple ADLs [9–17]. Nonetheless, many ADLs focus on other aspects and support static architectures only (e.g., DiaSpec [18], Palladio [19], xADL [20]). Also, there is no consensus on how architectural models describe dynamic reconfiguration. Usually, specific modeling elements exist [10–13,15–17].

Similar to our concept, some ADLs (e.g, AADL [15], AutoFocus [16, 17]) enable dynamic reconfiguration in a controlled fashion. Here, composed components change between configurations (called "modes") predefined at design time only. Specific transitions control when components may change their configuration. While this restricts arbitrary reconfiguration (cf. π-ADL [11], ArchJava [10]), it increases comprehensibility and guarantees static analyzability.

Dynamic reconfiguration can be *programmed* or *ad-hoc* [21]. In programmed reconfiguration (e.g., ACME/Plastik [13], AADL [15], ArchJava [10]), conditions and effects specified at design time are applied at runtime. Ad-hoc reconfiguration (e.g., C2 SADL [9], Fractal [14], ACME/Plastik [13]) does not necessarily have to be specified at design time and takes place at runtime, e.g., invoked by reconfiguration scripts. It introduces greater flexibility, but component models do not reflect the reconfiguration options. This enables simulating unforeseen changes to test an architecture's robustness, but it complicates analysis and evolution. For the latter reason MontiArcAutomaton's concept solely includes programmed reconfiguration.

Besides modeling dynamic removal and establishment of connectors, MontiArcAutomaton supports dynamic instantiation and removal of components. In ACME/Plastik [13], so-called actions can remove and create connectors and components. ArchJava [10] embeds architectural elements in Java and, hence, enables instantiating corresponding component classes as Java objects. C2 SADL [9] supports ad-hoc instantiation and removal of components. Fractal [14] provides similar concepts in its aspect-oriented Java implementation. π-ADL's [11] language constructs enable instantiation, removal, and movement of components.

5 Conclusion

We have developed and presented a concept for retrofitting controlled dynamic reconfiguration into the static ADL MontiArcAutomaton. Our concept maintains important abstraction and encapsulation mechanisms. Dynamic reconfigurable components have modes and mode automata to switch between configurations declaratively programmed at design time. The state of components during runtime can be either retained between different configurations or components can be instantiated and removed. We implemented our concept within the MontiArcAutomaton architecture modeling infrastructure. The implementation includes an extended syntax, analysis tools, and a code generator realizing semantics of dynamic reconfigurable components with synchronous communication. Interesting future work could investigate the applicability of our concept for retrofitting dynamic reconfiguration into further ADLs.

References

1. Medvidovic, N., Taylor, R.: A Classification and Comparison Framework for Software Architecture Description Languages. IEEE Transactions on Software Engineering (2000)
2. Malavolta, I., Lago, P., Muccini, H., Pelliccione, P., Tang, A.: What Industry Needs from Architectural Languages: A Survey. IEEE Transactions on Software Engineering (2013)
3. France, R., Rumpe, B.: Model-Driven Development of Complex Software: A Research Roadmap. In: Future of Software Engineering 2007 at ICSE. (2007)
4. Wile, D.S.: Supporting the DSL Spectrum. Computing and Information Technology (2001)

5. Ringert, J.O., Roth, A., Rumpe, B., Wortmann, A.: Language and Code Generator Composition for Model-Driven Engineering of Robotics Component & Connector Systems. Journal of Software Engineering for Robotics (JOSER) (2015)
6. Salehie, M., Tahvildari, L.: Self-Adaptive Software: Landscape and Research Challenges. ACM Transactions on Autonomous and Adaptive Systems (TAAS) (2009)
7. Ringert, J.O., Rumpe, B., Wortmann, A.: Architecture and Behavior Modeling of Cyber-Physical Systems with MontiArcAutomaton. Shaker Verlag (2014)
8. Lim, W.Y.P.: PADL–a Packet Architecture Description Language. Massachusetts Institute of Technology, Laboratory for Computer Science (1982)
9. Medvidovic, N.: ADLs and Dynamic Architecture Changes. In: Joint Proceedings of the Second International Software Architecture Workshop (ISAW-2) and International Workshop on Multiple Perspectives in Software Development (Viewpoints '96) on SIGSOFT '96 Workshops. (1996)
10. Aldrich, J., Chambers, C., Notkin, D.: ArchJava: Connecting Software Architecture to Implementation. In: Proceedings of the 24th International Conference on Software Engineering (ICSE). (2002)
11. Oquendo, F.: 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 (2004)
12. Cuesta, C.E., de la Fuente, P., Barrio-Solórzano, M., Beato, M.E.G.: An "abstract process" approach to algebraic dynamic architecture description. The Journal of Logic and Algebraic Programming (2005)
13. Joolia, A., Batista, T., Coulson, G., Gomes, A.T.: Mapping ADL Specifications to an Efficient and Reconfigurable Runtime Component Platform. In: 5th Working IEEE/IFIP Conference on Software Architecture, 2005. WICSA 2005. (2005)
14. Bruneton, E., Coupaye, T., Leclercq, M., Quéma, V., Stefani, J.: The FRACTAL component model and its support in java. Software - Practice and Experience (2006)
15. Feiler, P.H., Gluch, D.P.: Model-Based Engineering with AADL: An Introduction to the SAE Architecture Analysis & Design Language. Addison-Wesley (2012)
16. AutoFocus 3 Website. `http://af3.fortiss.org/` Accessed: 2016-01-18.
17. Aravantinos, V., Voss, S., Teufl, S., Hölzl, F., Schätz, B.: AutoFOCUS 3: Tooling Concepts for Seamless, Model-based Development of Embedded Systems. In: Joint proceedings of ACES-MB 2015 – Model-based Architecting of Cyber-physical and Embedded Systems and WUCOR 2015 – UML Consistency Rules. (2015)
18. Cassou, D., Koch, P., Stinckwich, S.: Using the DiaSpec design language and compiler to develop robotics systems. In: Proceedings of the Second International Workshop on Domain-Specific Languages and Models for Robotic Systems (DSLRob). (2011)
19. Becker, S., Koziolek, H., Reussner, R.: Model-Based Performance Prediction with the Palladio Component Model. In: Proceedings of the 6th International Workshop on Software and Performance. (2007)
20. Khare, R., Guntersdorfer, M., Oreizy, P., Medvidovic, N., Taylor, R.N.: xADL: Enabling Architecture-Centric Tool Integration with XML. In: Proceedings of the 34th Annual Hawaii International Conference on System Sciences. (2001)
21. Bradbury, J.S.: Organizing Definitions and Formalisms for Dynamic Software Architectures. Technical report, School of Computing, Queen's University (2004)