ACM/IEEE 17th International Conference on
Model Driven Engineering Languages and Systems

September 29, 2014 • Valencia (Spain)

1st International Workshop on Model-Driven Engineering for Component-Based Software Systems (ModComp) 2014

Workshop Proceedings

Federico Ciccozzi, Massimo Tivoli, Jan Carlson (Eds.)

Published on October 2014 v2.0

Copyright © 2014 for the individual papers by the papers' authors. Copying permitted for private and academic purposes. Re-publication of material from this volume requires permission by the copyright owners.

Editors' addresses:

Federico Ciccozzi and Jan Carlson
School of Innovation, Design and Engineering – Mälardalen University (Sweden)

Massimo Tivoli
Department of Information Engineering Computer Science and Mathematics – University of L'Aquila (Italy)

Organizers

Federico Ciccozzi (co-chair)	Mälardalen University (Sweden)
Massimo Tivoli (co-chair)	University of L'Aquila (Italy)
Jan Carlson (co-chair)	Mälardalen University (Sweden)

Program Committee

Marco Autili	University of L'Aquila (Italy)
Steffen Becker	University of Paderborn (Germany)
Etienne Borde	Telecom ParisTech (France)
Antonio Cicchetti	Mälardalen University (Sweden)
Ivica Crnkovic	Mälardalen University (Sweden)
Guglielmo De Angelis	CNR IASI/ISTI (Italy)
David Garlan	Carnegie Mellon University (USA)
Sebastién Gerard	CEA List (France)
Jeff Gray	University of Alabama (USA)
Lucia Happe	Karlsruhe Institute of Technology (Germany)
Anne Koziolek	Karlsruhe Institute of Technology (Germany)
Ludovico Iovino	University of L'Aquila (Italy)
Patricia López Martinez	University of Cantabria (Spain)
Julio Luis Medina	University of Cantabria (Spain)
Raffaela Mirandola	Politecnino di Milano (Italy)
Henry Muccini	University of L'Aquila (Italy)
Marco Panunzio	Thales Alenia Space (France)
Patrizio Pelliccione	Chalmers University (Sweden)
Alfonso Pierantonio	University of L'Aquila (Italy)
Pascal Poizat	Universit Paris Ouest Nanterre la Defense (France)
Ansgar Radermacher	CEA List (France)
Salah Sadou	IRISA/University of South Brittany (France)
Lionel Seinturier	University of Lille/INRIA (France)
Severine Sentilles	Mälardalen University (Sweden)
Tullio Vardanega	University of Padua (Italy)

Table of Contents

Preface 1

Keynote title 4
Ivica Crnkovic

Using component frameworks for model transformations by an internal DSL ... 6
Georg Hinkel and Lucia Happe

Specifying Intra-Component Dependencies for Synthesizing Component Behaviors 16
Stefan Dziwok, Sebastian Goschin and Steffen Becker

Multi-Platform Generative Development of Component & Connector Systems using Model and Code Libraries 26
Jan Oliver Ringert, Bernhard Rumpe and Andreas Wortmann

Interaction Components Between Components based on a Middleware 36
Van Cam Pham, Önder Gürcan and Ansgar Radermacher

Towards a metamodel for the Rubus Component Model 46
Alessio Bucaioni, Antonio Cicchetti and Mikael Sjödin

Translating Timing Constraints during Vehicular Distributed Embedded Systems Development 57
Saad Mubeen, Mikael Sjödin and Jukka Mäki-Turja

Synthesizing an Automata-based Representation of BPMN2 Choreography Diagrams 67
Marco Autili, Davide Di Ruscio, Amleto Di Salle and Paola Inverardi

A Lightweight Framework for Testing Safety-critical Component-based Systems on Embedded Targets 78
Nermin Kajtazovic, Andrea Höller, Tobias Rauter and Christian Kreiner

Preface

The design of modern software systems requires support capable of properly dealing with their ever-increasing complexity. In order to account for such a complexity, the whole software engineering process needs to be rethought and, in particular, the traditional division among development phases to be revisited, hence moving some activities from design time to deployment and runtime. Model-Driven Engineering (MDE) and Component-Based Software Engineering (CBSE) can be considered as two orthogonal ways of reducing development complexity: the former shifts the focus of application development from source code to models in order to bring system reasoning closer to domain-specific concepts; the latter aims to organize software into encapsulated independent components with well-defined interfaces, from which complex applications can be built and incrementally enhanced.

When exploiting these development approaches, numerous different modelling notations and consequently several software models are involved during the software life cycle. On the one hand, effectively dealing with all the involved models and heterogeneous modelling notations that describe software systems needs to bring component-based principles at the level of the software model landscape hence supporting, e.g., the specification of model interdependencies, and their retrieval, as well as enabling interoperability between the different notations used for specifying the software. On the other hand, MDE techniques must become part of the CBSE process to enable the effective reuse of third-party software entities and their integration as well as, generally, to boost automation in the development process.

An effective interplay of CBSE and MDE approaches could help in handling the intricacy of modern software systems and thus reducing costs and risks by: (i) enabling efficient modelling and analysis of extra-functional properties, (ii) improving reusability through the definition and implementation of components loosely coupled into assemblies, (iii) providing automation where applicable (and favourable) in the development process. In the last fifteen years, such a cooperation has been recognized as extremely promising; tools and frameworks have been developed for supporting this kind of integrated development process. Nevertheless, when exploiting interplay of MDE and CBSE, clashes arise due to misalignments in the related terminology but also, and more importantly, due to differences in some of their basic assumptions and focal points.

The goal of the workshop on Model-Driven Engineering for Component-Based Software Systems 2014 (ModComp'14) was to gather researchers and practitioners to share opinions, propose solutions to open challenges and generally explore the frontiers of collaboration between MDE and CBSE. ModComp'14 aimed at attracting contributions related to the subject at different levels, from modelling to analysis, from componentization to composition, from consistency to versioning; foundational contributions as well as concrete application experiments were sought.

The workshop was co-located with ACM/IEEE 17th International Conference on Model Driven Engineering Languages & Systems, and represented a forum for practitioners and researchers. We received fifteen papers out of which eight papers were selected for inclusion in the proceedings. The accepted papers covers many different forms of evolution in modeling including, but not limited to:

- model transformations for analysis and code generation;
- model interoperability;
- modeling component interaction and component behaviors.

This was the first edition of the workshop and the high attention received in terms of submissions demonstrates that the topics are relevant both in practice and in theory of model-driven engineering of component-based software systems. Thus, we would like to thank the authors – without them the workshop simply would not have taken place – and the program committee for their hard and precious work.

September 2014 Federico Ciccozzi, Massimo Tivoli and Jan Carlson

Keynote

Component-Based and Model-Driven Engineering: what is the difference? A CBSE perspective

Ivica Crnkovic

Chalmers University - Mälardalen University

Model-driven Engineering (MDE) and component-based software engineering have many similarities but also their own specific. The goals are similar – by raising the abstraction level of software system development, the goal is increase the development efficiency, effectiveness, and quality. Reusability plays an important role in both approaches. Components (mostly assumed as software components) are the fundamental parts in both MDE and CBSE approaches. In many CBSE approaches the emphasis is on component models, and system modeling using component models. Yet there are some clear differences – while the ultimate result of MDE is generated executable code from models (using modeling languages), CBSE aims at reusability in general, and in particular reusability of the executable code. What are the common characteristics then? Which achievements from one approach can be utilised in the other approach? Is it possible to combine the results for each approach? In the modern era of ambiguous computing new challenges are imposed on software system development – dynamic architecture, heterogeneous platforms, energy and other resources constraints, real-time issues, etc.. How these challenges are reflected in CBSE and how they can be used in MDE? In this presentation we will address these questions and identify some possible directions in further research in CBSE & MDE.

Ivica Crnkovic *is a professor of industrial software engineering at Mlardalen University where he is the scientific leader of the industrial software engineering research. His research interests include component-based software engineering, software architecture, software configuration management, software development environments and tools, as well as software engineering in general. Professor Crnkovic is the author of more than 150 refereed articles and papers on software engineering topics and a co-author and co-editor of two books: "Building reliable component-based Systems" and "Implementing and integrating Product Data Management and Software Configuration Management". He has co-organized several conferences and workshops and related to software engineering (such as CBSE symposium, ESEC/FSE conference, Euromicro SEAA conference), and participated in Program Committees of software configuration management symposia and workshops. His teaching activities cover several courses in the area of Software Engineering undergraduate and graduate courses. From 1985 to 1998, Ivica Crnkovic worked at ABB, Sweden, where he was responsible for software development environments and tools. He was a project leader and manager of a group*

developing software configuration management systems and other software development environment tools and methods for distributed development and maintenance of real-time systems. From 1980 to 1984, he worked for the Koncar company in Zagreb, Croatia. Professor Crnkovic received an M.Sc. in electrical engineering in 1979, an M.Sc. in theoretical physics in 1984, and a Ph.D. in computer science in 1991, all from the University of Zagreb, Croatia.

Using component frameworks for model transformations by an internal DSL

Georg Hinkel and Lucia Happe

Karlsruhe Institute of Technology
Am Fasanengarten 5
Karlsruhe, Germany
{georg.hinkel,lucia.kapova}@kit.edu

Abstract. To increase the development productivity, possibilities for reuse, maintainability and quality of complex model transformations, modularization techniques are indispensable. Component-Based Software Engineering targets the challenge of modularity and is well-established in languages like Java or C# with component models like .NET, EJB or OSGi. There are still many challenging barriers to overcome in current model transformation languages to provide comparable support for component-based development of model transformations. Therefore, this paper provides a pragmatic solution based on NMF Transformations, a model transformation language realized as an internal DSL embedded in C#. An internal DSL can take advantage of the whole expressiveness and tooling build for the well established and known host language. In this work, we use the component model of the .NET platform to represent reusable components of model transformations to support internal and external model transformation composition. The transformation components are hidden behind transformation rule interfaces that can be exchanged dynamically through configuration. Using this approach we illustrate the possibilities to tackle typical issues of integrity and versioning, such as detecting versioning conflicts for model transformations.

1 Introduction

In Model-Driven Engineering (MDE), systems are designed in models that conform to a metamodel. This formal definition of the model makes it possible to transform models to other artifacts like code or other models by means of model transformations. As MDE is getting applied in more complex scenarios, these model transformations also get very complex. As a consequence, it gets more important to divide these transformations into parts, i.e. components, with the goal to reuse these components in new scenarios as much as possible.

However, Wimmer, Kusel et al. indicate that reuse functionality of current model transformations is hardly established in practice [1, 2], especially reuse among different metamodels.

On the other hand, the situation is entirely different in most object-oriented general purpose languages. Here, classes can hide implementation details even to

inheriting classes by use of private methods or fields. Functionality can be reused through these classes, even independent of domains through the usage of generic types. These classes are assembled to components that have well-defined public interfaces. Furthermore, these components are explicitly versioned, so that a system can automatically detect versioning conflicts.

To pick an example, in the .NET component model, components (assemblies) consist of a provided interface (the publicly visible types), references to required assemblies, an explicit version information and possibly a digital signature. References to other assemblies also include the referenced version and the digital signature where applicable. Assemblies can be reused, deployment is simplified as dependencies are explicitly specified, integrity can be ensured through the usage of digital signatures and the runtime can detect version conflicts. A version conflict is detected when a required component with the specified major and minor version number (but possibly higher build or revision number) cannot be found. A similar component model is also required for model transformations. Unlike other component models like OSGi, the .NET component model does not allow components to be exchanged, e.g. through configuration. This is typically solved by using dependency injection frameworks.

Solving this problem by inventing a new component model for model transformations yields all risks of duplicate concepts, as e.g. a high maintenance effort. Thus, we think that it is a better approach to adopt existing component models where possible to use the interface mechanism for model transformation. As interfaces of .NET components are the publicly visible types, i.e. classes, we only need a meaningful mapping from model transformation concepts like transformation rules to classes in order to be able to reuse such a component model for internal model transformation composition, i.e. composing a single model transformation of multiple reusable parts.

The .NET component model is a particularly interesting candidate, as it has been used for a wide range of languages, including originally imperative languages like C# or VB.NET as well as more recently functional languages like F#. This variety of language paradigms yields the question of whether it can also be reused for more tailored languages like model transformation languages.

One approach for such a mapping is the definition of an internal DSL where concepts of model transformations are represented in the host language. Such an internal DSL is provided e.g. by NMF TRANSFORMATIONS, which has been applied to several cases at the Transformation Tool Contest (TTC) 2013 [3, 4]. Furthermore, it is used within the .NET Modeling Framework (NMF)[1] to generate code for the model representation, quite similar to EMF[2]. Especially the latter transformation is rather large, so there is a need to make it more modular.

This embedding gives us a range of benefits regarding both internal and external model transformation composition. We can have model transformation components that specify interfaces, we can ensure their integrity through digital

[1] http://nmf.codeplex.com/
[2] http://www.eclipse.org/modeling/emf/

signatures and detect versioning conflicts. Here, integrity means that a model transformation component cannot be replaced by a forged component with the same public interface. These benefits come at no maintenance cost for the required infrastructure, as the infrastructure e.g. to ensure integrity through digital signatures is still maintained by the original owner, i.e. Microsoft. Furthermore, our embedding allows us to specify how model transformation rules can be dynamically exchanged through configuration by means of dependency injection.

Because model transformation concepts like transformation rules and a trace are represented directly by classes, NMF TRANSFORMATIONS can act as a baseline for mappings of other transformation languages to classes as well, e.g. by translating model transformations of other languages to NMF TRANSFORMATIONS.

The rest of this paper is structured as follows: Section 2 shows related work. Section 3 explains the running example of finite state machines and Petri Nets. Section 4 gives a brief overview of NTL, before Section 5 explains how NTL can be used to specify model transformation components mapped to assemblies. Finally, Section 6 concludes this paper.

2 Related Work

The idea of using internal DSLs [5] for model transformation has already been applied several times, but for different reasons. These reasons include a low implementation effort [6], type safety [7] or extensibility [8, 7, 9, 10]. However, their implications on reusing underlying component models have not been analyzed so far.

For external languages, experiences of reusing existing component models exist, as e.g. Xtend[3] is reusing the whole technology stack of Java, including the organization in Jar archives as the technical component model. However, unlike NTL, Xtend is a general-purpose language with support for model-to-text transformations through Xpand templates. Tailored model-to-model transformation languages implemented as external DSLs, like most commonly known QVT-O [11] or ATL [12], usually specify module reuse concepts (as surveyed by Wimmer, Kusel et al. [1, 2], plus the more recent approach from Rentschler et al. for modular QVT-O [13]), but cannot reuse component infrastructure that provides support regarding versioning conflicts or checking the integrity of model transformation components.

Rather, these languages are mainly organized in files that do not specify version information. References to other files are specified as import links. These links also do not specify a version information and possible version conflicts are not automatically detected. This is different for our approach where components of model transformations are represented by assemblies that specify references enriched with both version information and digital signatures in assemblies. This rather technical information has to be specified separately from the transformation specification and does not pollute the latter. Essentially, our work reuses

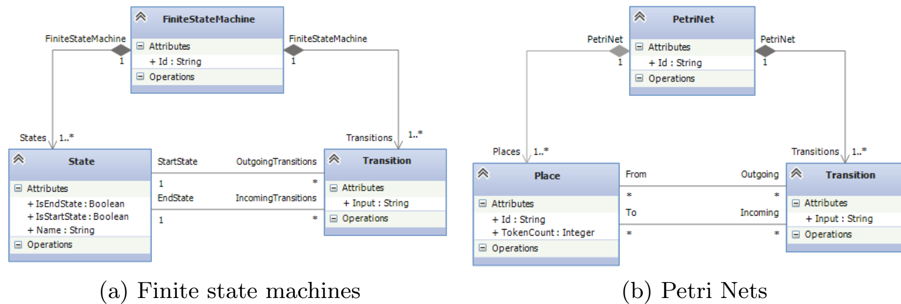
[3] http://www.eclipse.org/xtend/

the specification of component dependencies from existing component models as these dependency specifications are not specific to the domain of model transformation.

Model transformations can also be composed of multiple transformations through external composition such as chaining model transformations of multiple languages (see e.g. [1, 2] for a survey). Our embedding approach is also applicable for external composition, but yields the limitation that all model transformations must be embedded in the same platform. This limitation is typically avoided in specialized component models for model transformation [14], but these component models suffer from duplicated concepts and interoperability issues with other components.

3 Finite State Machines and Petri Nets

This section will introduce the running example of the transformation of finite state machines to Petri Nets. Both finite state machines and Petri Nets are popular formalization techniques to model processes, for example business processes. One is sometimes interested to transform finite state machines into Petri Nets because Petri Nets are more expressive.



(a) Finite state machines

(b) Petri Nets

Fig. 1. Metamodels of finite state machines and Petri Nets

The metamodels for finite state machines and Petri Nets are depicted in Figure 1. The transformation maps each state of the state machine to a corresponding place. Each transition is mapped to a transition with an according input and output. Places corresponding to start states have an initial token and those corresponding to end places have a transition without a target.

4 NMF Transformations

NMF Transformations consists of two parts: a model transformation framework and a DSL that provides an easy syntax for this framework. This language

is called NTL (NMF Transformations Language) and is an internal DSL embedded in C#. Its abstract syntax is depicted in Figure 2. In NMF TRANSFORMATIONS, model transformations consist of transformations and patterns (which we omit for brevity here). These rules can have dependencies to each other, letting computations of a transformation rule depend on one or multiple other computations.

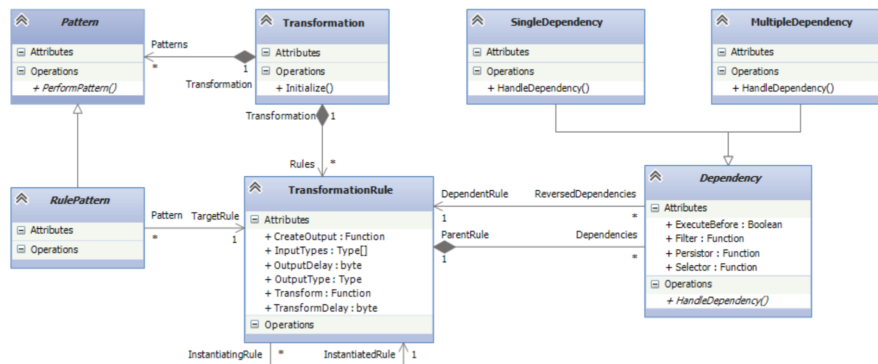


Fig. 2. The abstract syntax of NMF TRANSFORMATIONS

In NTL, model transformations are represented as classes with their transformation rules as public nested classes. The transformation rules then specify how model elements should be transformed. Listing 1 shows an example of a transformation transforming finite state machines to Petri Nets, accompanied with the start rule (implicitly the first rule that matches the transformation request). The `Transform` method is used to specify the actions to be done when a finite state machine is to be transformed, i.e. initializes the transformation rule output where it may access the transformation context for tracing purposes.

```
1  using NMF.Transformations;
2  using NMF.Transformations.Core;
3
4  public class FSM2PN : ReflectiveTransformation {
5   public class Automata2Net : TransformationRule<FSM.FiniteStateMachine, PN.
        PetriNet>
6   {
7    public override void Transform(...)
8    {
9     output.ID = input.ID;
10   }
11  }
12 }
```

Listing 1. The transformation rule FiniteStateMachine2PetriNet

To specify dependencies, transformation rule classes may override the method `RegisterDependencies` and call several methods that create such dependencies using lambda expressions. An example is shown in Listing 2. The first dependency is a special one, as it specifies when the rule is going to be called (instead

of what other rules should be called). The last argument specifies how dependent computations should be saved. For example, the place corresponding to the start state of a finite state machine transition should be added to the `From` collection of the corresponding Petri Net transition.

```
1  public class Transition2Transition : TransformationRule<FSM.Transition, PN.
       Transition>
2  {
3   public override void RegisterDependencies()
4   {
5    CallForEach(Rule<Automata2Net>(),
6      selector: fsm => fsm.Transitions,
7      persistor: (net, transitions) => net.Transitions.AddRange(transitions));
8
9    Require(Rule<State2Place>(),
10      selector: t => t.StartState,
11       persistor: (t, place) => t.From.Add(place));
12
13   Require(Rule<State2Place>(),
14      selector: t => t.EndState,
15       persistor: (t, place) => t.To.Add(place));
16  }
17 }
```

Listing 2. The rule Transition2Transition with multiple dependencies

The language also supports inheritance mechanisms that can be facilitated for model transformation components. This includes both inheritance of transformations (like superimposition in ATL) as well as two applicable concepts for transformation rules, inheritance and instantiation. Transformation rule inheritance really is inheritance of the transformation rule class (similar to masked rules in ATL), whereas instantiation is what most other transformation languages (including ATL) call inheritance, unfortunately. An inherited rule may really override the body of that rule, as well as its dependencies. An instantiating rule must not do so, but may instead take control over the creation of outputs. Whereas rule inheritance aims for extensibility, instantiation is rather to support inheritance hierarchies and thus omitted in this paper for brevity.

Let us, for example, extend the above transformation to use colored Petri Nets. Listing 3 shows the implementation with rule inheritance. The behavior of the *Transition2Transition* rule is simply overridden in that it now creates a *ColoredTransition* with the default color. The transformation engine will simply instantiate a *ColoredTransition2Transition* rule instead of a *Transition2Transition* rule, because a *ColoredTransition2Transition* rule **is a** *Transition2Transition* rule and marked as overriding.

```
1  public class FSM2ColoredPN : FSM2PN {
2   [OverrideRule]
3   public class ColoredTransition2Transition : Transition2Transition
4   {
5    public override PN.Transition CreateOutput(...) {
6       return new PN.ColoredTransition() { Color = DefaultColor };
7     }
8   }
9  }
```

Listing 3. Introducing colored Petri Nets through rule inheritance

Next to transformation rule inheritance, Listing 3 demonstrates the inheritance of transformations. The transformation *FSM2ColoredPN* inherits *FSM2PN* and thus inherits its transformation rules (except for *Transition2Transition* which is overridden).

5 Components in model transformations with NTL

One of the advantages of using an internal DSL for model transformation is the easy integration of arbitrary code of the host language. For NTL, this is C# (or in theory any other .NET language). The `Transform` method is just executed normally and can contain arbitrary C# code, e.g. also using third-party components referenced by the assembly that contains the model transformation. This embedding allows target model elements to depend on source model elements in more sophisticated forms than simple transformations such as adding a prefix but more complex analysis. Being an ordinary method call, such a call is of course possible through dependency injection, so targets of external calls can be replaced by means of configuration. Likewise, model transformations can be called from arbitrary .NET assemblies as a transformation in NTL can be triggered by a method call. This yields way to any means of external model transformation composition. This composition is however restricted only to model transformations written for a common platform, which in our case is .NET.

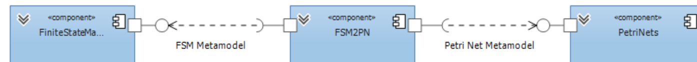


Fig. 3. Dependent metamodels as components

An example of this is the generated model representation code for the involved metamodels, as demonstrated in Figure 3. One usually wants to separate this model representation code in own assemblies. Because both metamodel code and transformation code lie in assemblies that carry a version information, the transformation has an explicit knowledge of which metamodel version it is using.

Being .NET classes, transformation rules in NTL are perfectly allowed to also split the implementation of their interface (which by default most importantly consists of the two methods `Transform` and `RegisterDependencies`) in as many private methods as they wish, hiding their implementation. More interestingly, they can also have virtual, abstract or final methods. Thus, transformation rules can decide how their behavior can (or must) be overridden. They can e.g. mark the overridden `RegisterDependencies` method final to prevent child classes to modify dependencies. Alternatively, transformation rules can specify new virtual methods that are called e.g. from their `Transform` implementation, enabling extension rules to override only parts of its behavior. Making the `Transform` method final, the extension is also limited to these parts.

With these virtual methods, such transformation rules do provide an interface for other components that wish to extend them. If they are additionally marked as abstract, the only remaining differences are that true .NET interfaces do not interfere the inheritance hierarchy and type parameters can be covariant or contravariant. The interference with the inheritance hierarchy is not relevant to transformation rules, as transformation rules in NTL must eventually inherit from `TransformationRule<,>` anyhow. The restriction of covariancy/contravariancy means that implementations of such a transformation rule interface must stick to the same signature, which we argue is an acceptable restriction. Implementations of such an interface are transformation rules that inherit from the interface transformation rule.

Since `TransformationRule<,>` itself is an abstract class, we can also use it as an interface. This is possible because in the .NET platform, the runtime is still aware of generic type arguments (unlike e.g. Java). This is even directly supported by NTL. So instead of specifying a concrete transformation rule in listing 2, one can also just specify the transformation rule signature (i.e. the input and output type of the transformation rule) having the transformation engine apply the dependency for all registered transformation rules that correspond to this signature (without having to know the exact transformation or its implementation).

Using generic transformation rule classes as transformation rule interfaces has another advantage, as this decouples a transformation rule from the metamodels used by this rule. Instead, the transformation rule interfaces can require the input and/or output type to adhere to some type constraints or require implementations to inject domain knowledge through abstract methods and generic type arguments.

Now consider the transformation to create a language-independent model representation code model. One of the challenges of this transformation is to cope with the fact that the metamodels may use multiple inheritance whereas .NET only supports single inheritance. This challenge is independent from the exact mapping how model elements are transformed to type members and is thus a good candidate for a reusable component.

Using external composition techniques, one would first transform the metamodel to a code model with multiple inheritance and chain a separate transformation that removes the multiple inheritance by introducing interfaces and merging the implementing classes. Since our embedding allows to treat the involved model transformations just like any other method call, we can put them into separate components and hide them behind interfaces as we wish (using platform standard interfaces).

On the other hand, we can also create a system of transformation rules in a separate component where these rules fix their output type but leave the input type open (using generic type parameters). This way, we compose the transformation by refining specialized rules for this task, i.e. in a manner of internal model transformation composition. These specialized rules would then contain domain-specific extension points that allow to alter the merging step in

many places. With this approach, the second merging step becomes more like a transformation framework in its own, based on the transformation language. The advantage of this is that the merge process can be controlled in much more detail as derived transformation rules may choose an extension point to override.

Finally, we can also load single transformation rules from other components, adding e.g. the merging step as a separate transformation rule that runs delayed. This way, the merging step runs in the very same transformation run and has access to all the trace. If we had multiple components that realize this functionality in different ways, we could swap the implementation by means of a dependency injector dynamically as NTL also allows to load transformation rules in a method. The only restriction here is that a transformation always requires fresh transformation rule instances (transformation rule instances cannot be shared across multiple transformations), but most dependency injectors can be configured to fit this requirement, i.e. creating a new instance per request.

Thus, the mapping of model transformation concepts to classes yields a clear and precise notion of interfaces for model transformation rules as well as model transformation components, where the components are the .NET assemblies each containing a subset of transformation rules. But as they are assemblies, they also inherit the version information of assemblies.

As a consequence, developers can (and have to) specify the version of their model transformation components explicitly as version of the assembly that contains the model transformation component and likewise the version of referenced components. These version numbers consist of four parts, major, minor, build and revision. Changes of build and revision number by default are interpreted as bug fixes, i.e. the runtime will load assemblies with higher build or revision numbers instead of the specified referenced assembly. Higher major or minor version numbers are interpreted as breaking changes regarding the assemblies public interface, which for NTL are publicly accessible model transformation rules (or other public e.g. helper types). If only a higher version of a referenced component is found (components are allowed to be present in multiple versions concurrently), the runtime raises an exception, detecting possible versioning issues.

6 Conclusion

In this paper, we have shown how a mapping from model transformation concepts to object-oriented general-purpose constructs can be used to reuse component models. We have achieved this goal through NMF Transformations, a framework and internal DSL embedded in C#. This embedding allows us to:

- Integrate existing .NET components (assemblies) into model transformations and vice versa
- Detect versioning conflicts of model transformation components
- Compose model transformations as extension of model transformations specified in other components
- Specify transformation rule interfaces

– Compose model transformations of instancies previously defined transformation rule interfaces loaded from referenced components
– Ensure integrity of model transformation components by means of digital signatures.

Reusing existing dependency injectors further yields the chance to reconfigure model transformations based on configuration files without new compilation. Thus, we showed that a mapping of model transformation concepts to general-purpose programming, as e.g. with an internal DSL such as NTL, can be used to adopt existing component models for model transformation and reuse a lot of concepts and tools.

References

1. M. Wimmer, G. Kappel, A. Kusel, W. Retschitzegger, J. Schönböck, and W. Schwinger, "Fact or fiction–reuse in rule-based model-to-model transformation languages," in *Theory and Practice of Model Transformations.* Springer, 2012, pp. 280–295.
2. A. Kusel, J. Schönböck, M. Wimmer, G. Kappel, W. Retschitzegger, and W. Schwinger, "Reuse in model-to-model transformation languages: are we there yet?" *Software & Systems Modeling*, pp. 1–36, 2013.
3. G. Hinkel, T. Goldschmidt, and L. Happe, "An NMF Solution for the Flowgraphs case study at the TTC 2013," in *Sixth Transformation Tool Contest (TTC 2013)*, ser. EPTCS, 2013.
4. ——, "A NMF solution for the Petri Nets to State Charts case study at the TTC 2013," in *Sixth Transformation Tool Contest (TTC 2013)*, ser. EPTCS, 2013.
5. M. Fowler, *Domain-specific languages.* Addison-Wesley Professional, 2010.
6. H. Barringer and K. Havelund, *TraceContract: A Scala DSL for trace analysis.* Springer, 2011.
7. L. George, A. Wider, and M. Scheidgen, "Type-Safe model transformation languages as internal DSLs in Scala," in *Theory and Practice of Model Transformations.* Springer, 2012, pp. 160–175.
8. J. S. Cuadrado, J. G. Molina, and M. M. Tortosa, "RubyTL: A practical, extensible transformation language," in *Model Driven Architecture–Foundations and Applications.* Springer, 2006, pp. 158–172.
9. T. Horn, "Model Querying with FunnyQT," in *Theory and Practice of Model Transformations.* Springer, 2013, pp. 56–57.
10. G. Hinkel, "An approach to maintainable model transformations using internal DSLs," Master thesis, 2013.
11. Object Management Group, "Meta Object Facility (MOF) 2.0 Query/View/ Transformation Specification," http://www.omg.org/spec/QVT/1.1/PDF/, 2011.
12. F. Jouault and I. Kurtev, "Transforming models with ATL," in *Satellite Events at the MoDELS 2005 Conference.* Springer, 2006, pp. 128–138.
13. A. Rentschler, D. Werle, Q. Noorshams, L. Happe, and R. Reussner, "Designing information hiding modularity for model transformation languages," in *Proceedings of the of the 13th international conference on Modularity.* ACM, 2014, pp. 217–228.
14. J. S. Cuadrado, E. Guerra, and J. de Lara, "A component model for model transformations," *Software Engineering, IEEE Transactions on*, vol. PP, no. 99, pp. 1–1, 2014.

Specifying Intra-Component Dependencies for Synthesizing Component Behaviors

Stefan Dziwok[1,2], Sebastian Goschin[3], and Steffen Becker[1]

[1] Software Engineering Group, Heinz Nixdorf Institute, University of Paderborn, Zukunftsmeile 1, 33102 Paderborn, Germany, stefan.dziwok@upb.de
[2] Fraunhofer IPT, Project Group Mechatronic Systems Design, Software Engineering, Zukunftsmeile 1, 33102 Paderborn, Germany
[3] Step2e Innovation GmbH, Rosstraenke 4, 94032 Passau, Germany

Abstract. Cyber-physical systems, e.g., cars, interact with their physical environment, underlie real-time constraints, and exchange messages with each other. An engineer can define their software using a component-based architecture. An approach to manage the complexity of this task is to separate concerns by specifying the behavior of each component's port independently and, afterwards, synthesizing the component behavior based on the port's behaviors and their dependencies. Though, such a synthesis requires to specify the intra-component dependencies formally. However, for several dependencies that are commonly used, no formal language exists. In this paper, we present a language that enables the specification of all commonly used dependencies in the domain of cyber-physical systems. Moreover, we define the requirements for an intra-component dependency language, provide an extended synthesis process, and introduce the dependency kinds the language shall support.

Keywords: Component-Based Software Systems, Model-Driven Engineering, Model Dependencies, Cyber-Physical Systems, MechatronicUML

1 Introduction

Cyber-physical or intelligent technical systems [7] are systems that interact with their physical environment. Their software development is especially difficult as these systems have to obey real-time constraints, are safety-critical, and coordinate with other systems by exchanging various messages. A cyber-physical system (CPS) might utilize a component-based software architecture, where components interact via ports. The interaction is either message-based or signal-based (e.g., to access sensors, actors, and feedback controllers). The behavior specification of a component is expressed via state machines. In addition, to ensure a safe and useful coordination between the components, each interaction between message ports has to adhere to communication protocols at application-level that define which message at which time span must be sent or received.

One particular example of CPSs are autonomously driving cars. In the scenario given in Fig. 1, the white car is faster than the gray car and, thus, wants to

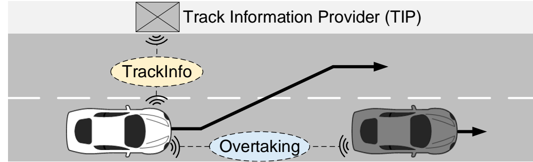


Fig. 1. Overtaking Scenario: The white car wants to overtake the gray car.

overtake it. Each car can measure its velocity and white can additionally measure its distance to gray. To reduce the risk of a crash, the involved systems shall coordinate the overtaking by adhering to two communication protocols. Both cars must adhere to the communication protocol Overtaking which formally defines that white needs permission from gray to start the overtaking and that gray does not accelerate while white is overtaking. Moreover, white and a track information provider (TIP) must adhere to the communication protocol TrackInfo which formally defines that the TIP informs white whether the track is safe (e.g., it is not safe if obstacles are on the street). To further increase the safety, white shall fulfill additional requirements: (d1) white may only request to overtake, if it reaches the safety distance to gray and the TIP has confirmed that the track is safe for more than 5s. (d2) The planned overtaking speed that white sends to grey depends on the current velocity of white. (d3) Before informing gray that the overtaking has finished, white must restore the safety distance to gray.

A software engineer that shall develop the component behavior for the white car's software can split this task into two steps to enable a separation of concerns and, therefore, to manage the task's complexity. In step 1, he specifies the externally visible component behavior. It consists of the behavior of the component's single ports to access physical information (here: the distance and the velocity) and to exchange messages according to the two communication protocols. The result is a set of independent port behavior specifications. Step 2 serves to additionally fulfill the intra-component requirements (here: d1-d3). To achieve this, the software engineer has to enhance the independently defined behavior models of the component. Thus, these requirements are intra-component dependencies. As errors in the component behavior are safety-critical, it is of vital importance that the software engineer considers *all* intra-component dependencies and implements them correctly such that the component behavior adheres to both, the single ports behaviors as well as the ports communication requirements. However, our experience shows that executing the second step manually is – despite this separation of concerns – still complex and error-prone.

A semi-automatic approach for step 2 is provided by Eckardt and Henkler [6]. They provide domain-specific languages (DSLs) a software engineer can use to formally describe the intra-component dependencies between the independent behaviors. Based on the independent behaviors and the formal dependencies, the component behavior can be synthesized automatically. The approach is integrated into the software engineering method MechatronicUML [4,8].

In this paper, we extend the approach of Eckardt and Henkler [6] as their formal dependency languages are not able to express all intra-component dependencies that are commonly used in the domain of CPSs. For example, their languages cannot express the mentioned dependencies d1-3. One reason is that their languages are not able to access the physical information (distance and velocity). As the related work also does not provide DSLs for the unsupported dependencies, we define a new DSL. In particular, the contribution of our paper is as follows: (i) we explicitly define the requirements for specifying intra-component dependencies that shall serve as an input for the synthesis, (ii) we extend the synthesis process defined by Eckardt and Henkler, (iii) we provide 20 kinds of intra-component dependencies, and (iv) we introduce a formal intra-component dependency language for MechatronicUML. We provide our implementations and the models of our running example online[4].

The paper is structured as follows. Section 2 introduces MechatronicUML and presents the models of the running example. Then, in Sect. 3, we define the requirements for an intra-component dependency language and discuss to what extent Eckardt and Henkler's approach fulfills them. We explain the adapted synthesis process in Sect. 5. Afterwards, we present the identified kinds of intra-component dependencies in Sect. 6 and introduce our new DSL in Sect. 7. We discuss related work in Sect. 8 and conclude the paper in Sect. 9.

2 MechatronicUML

MechatronicUML [4,8] is a method that is designed for the software development of CPSs. It provides a component-based modeling language and a development process. Thus, it is able to specify the models of the overtaking scenario.

MechatronicUML distinguishes between continuous and discrete components. Continuous components represent, among others, sensors, actors and time-continuous feedback-controllers. In contrary, the behavior of discrete components is specified via extended states machines called Real-Time Statecharts (RTSCs). A RTSC may contain data variables and continuous clocks (known from timed automata [3]) which are used to define real-time constraints. Clocks can be reset to zero and be compared with an expression. An example for a discrete component is WhiteSw (cf. Fig. 2). Discrete components may contain discrete ports to exchange messages with other discrete components and hybrid ports to send and receive signals from continuous components. A RTSC of a discrete component can specify to send and receive messages via its discrete ports and may read or write values from hybrid ports. In our example, WhiteSw contains the discrete ports Overtaker and Receiver and the hybrid ports Distance and Velocity. In particular, Distance is a hybrid in-port that periodically reads an incoming signal measuring the distance between the cars while Velocity is a hybrid out-port that writes an outgoing signal to set the car's target velocity.

For ensuring a correct message communication, each discrete port must adhere to application-level communication protocols called Real-Time Coordina-

[4] https://trac.cs.upb.de/mechatronicuml/wiki/ModComp2014

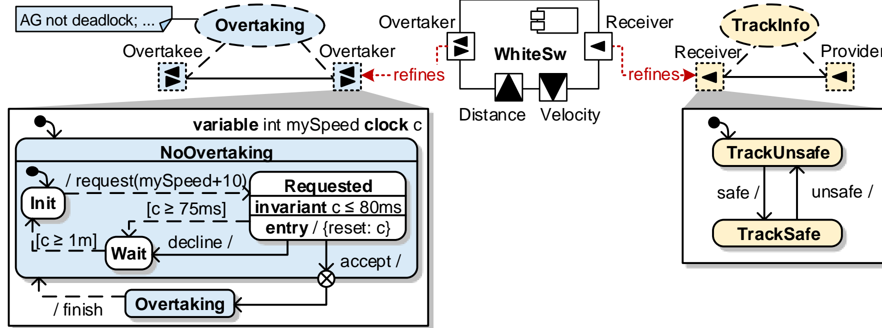


Fig. 2. Models of the Overtaking Scenario Specified in MechatronicUML

tion Protocols (RTCPs). In particular, a RTCP consists of two communicating partners, called roles, whereby each role represents a discrete port. As depicted in Fig. 2, in our example, the discrete port Overtaker must adhere to and may refine the same-named role of RTCP Overtaking, which coordinates the overtaking. The discrete port Receiver must adhere to and may refine the same-named role of RTCP TrackInfo, which coordinates the information whether the track is safe. RTCPs may be formally verified via timed model checking [3] concerning formal properties, e.g., that a deadlock may never occurs. Due to the separation of components from RTCPs, MechatronicUML enables a scalable timed model checking of the software under development for large and complex systems [8].

The behavior of a RTCP depends on the behavior of each roles which we separately specify using RTSCs. The lower part of Fig. 2 shows the RTSCs for the roles Overtaker and Receiver. The informal behavior description of the role Overtaker is as follows: Initially, the role is in state NoOvertaking.Init and may at any time send an overtaking request to role Overtakee including its planned overtaking speed as a message parameter which is the current speed plus 10 km/h. Within 75 ms, it expects an accept or decline message. If no message or a decline message is received, the role Overtaker has to wait until 1 minute in total is over before it may send a new request. However, if the message accept is received, it switches to state Overtaking and may now start the overtaking process. After the overtaking, it sends the message finish and switches back to NoOvertaking.Init. In contrast, the behavior of the role Receiver is quite simple: Initially, the role Receiver assumes that the track is unsafe. As soon as the track is safe, the role Receiver will be informed by the role Provider about this change. Analogously, the role Provider will inform the role Receiver when the track becomes unsafe.

3 Existing Dependency Languages for MechatronicUML

Eckardt and Henkler [6] provide for MechatronicUML two formal languages to describe intra-component dependencies: *state composition rules* (SCR) and *event composition automata* (ECA).

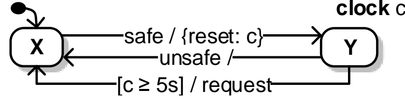


Fig. 3. Event Composition Automata for Synchronizing the Message Exchange

SCRs are able to synchronize the concurrent executed behavior of discrete ports based on their states and clocks. In particular, SCRs can forbid that a set of states may be active together at any point in time or at least for a specific time interval (by defining constraints that reference existing clocks of a port). An exemplary SCR is as follows: $\neg((A, true) \wedge (B, c < 30))$. It defines that the states A and B may not be active together when the clock c is less than 30.

ECAs are expressed using timed automata [3]. They are able to synchronize the behavior of two discrete ports based on sequences of messages the ports exchange with other components. Moreover, timing restrictions on the message sequences may be defined using auxiliary clocks. In particular, the automaton can enforce the order and the delay of messages that the ports can send or receive. We show an example in Fig. 3. The depicted automata constraints the sending of message request by enforcing that message safe shall be received at least 5 seconds before and that message unsafe is not received in between.

4 Requirements for the Dependency Language

Before we adapt the existing dependency languages, we explicitly define requirements and discuss the current fulfillment by Eckardt and Henkler [6].

Based on our experience, the dependency language shall (r1) be formal, (r2) be able to reference existing modeling elements of the design language that are used in the independent behaviors, (r3) include analyses to identify errors and conflicts within the specification, (r4) cover the most commonly used dependencies, (r5) be acceptable by all stakeholders, (r6) only enable useful dependencies, and (r7) have a good tool support with a high usability.

The requirements r1 and r2 are essential to enable an automated synthesis, while r3 shall improve the correctness of the dependencies and thus avoid senseless syntheses. Concerning r4, in our opinion, the goal should not be to support all dependencies that are theoretically possible. This would lead to a language that is complex to use and may result in incorrectly formalized dependencies. Moreover, it is hard (maybe even impossible) to prove that all possibilities are covered. Having a less complex language should benefit r5 because - in our opinion - software engineers define the formal dependencies but requirements and test engineers need to be able to understand them as they have to check if the informal requirements are covered. Via r6, we want to prevent that the engineer can define dependencies that the synthesis does not support or that he can define senseless expressions (e.g., tautologies). Concerning r7, good tool support reduces the development time and avoids mistakes by the user. Examples for usability enhancements are live syntax checks and auto suggestions.

The approach of Eckardt and Henkler [6] only fulfills the requirement r1. The other requirements are partially or not fulfilled. The reason for this is that their focus was to prove that their general synthesis approach is applicable rather than concentrating on its usability or its completeness. In particular, they only partially fulfill r2 as their languages cannot reference all syntax elements of RTSCs, e.g., hierarchical states, concurrent state machines, variables, actions, and entry/do/exit state events. Moreover, hybrid ports cannot be referenced, too. Requirement r3 is not fulfilled as they only focus to identify conflicts after the synthesis. Concerning requirement r4, their two dependency languages already support a lot of dependencies. However, the languages do not support commonly used dependencies, e.g., dependencies concerning data variables. Without an evaluation, we can not answer in general if requirement r5 is fulfilled. However, in our opinion, it is not obvious which dependencies they describe. This leads to the conclusion that r5 can still be improved. Requirement r6 is only partially fulfilled as they only enable dependencies that the synthesis supports but their language allows superfluous dependencies (e.g., tautologies). Concerning requirement r7, good tool support exists for ECA in form of a graphical editor. However, SCR dependencies are defined as a plain string only and, therefore, do not provide usability enhancements like the ones mentioned above.

5 Adaptions to the Synthesis Process

As most of the requirements from Sect. 4 were not in Eckhardt's and Henkler's focus, adaptation to their syntheses process [6] are necessary. Figure 4 shows the new process for semi-automatically defining the component behavior. In particular, we added the steps 1b and 2c and the analyses for the steps 2a and 2c. In the following, we briefly describe the new process.

Based on (informal) requirements and the component structure, in the Steps 1a and 1b, software engineers concurrently specify *all* independent behaviors that form the external component behavior, i.e., RTCPs for discrete ports and hybrid ports. For each RTCP, a developer has to apply timed model checking to ensure their correctness concerning the specified properties (e.g., no deadlock occurs). Then, in Step 2a, a software engineer formalizes the dependencies using a DSL that fulfills the requirements from Sect. 4. Dependencies that cannot be formalized using the DSL are postponed to Step 2c. While specifying, analyses identify errors and conflicts (e.g., contradictions and tautologies) to prevent superfluous executions of the synthesis. Afterwards, in Step 2b, the developer executes the automatic synthesis and the succeeding refinement checks [11] which verifies if the discrete ports still adhere to their RTCP. In Step 2c, a software engineer may adapt the synthesized component behavior to integrate the dependencies that were not formalizable in Step 2a. In that case, another refinement check is necessary. To prevent that the manual changes are lost after a new synthesis run, we propose to automatically log all manual changes and to enable a semi-automatic reapplication. The final result of this process is the complete component behavior that respects all requirements.

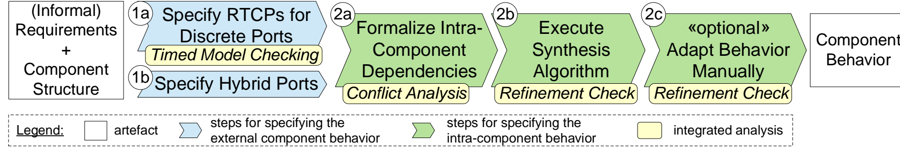


Fig. 4. Adapted Process for Synthesizing the Component Behavior

As MechatronicUML already fully enables the Steps 1a-b (including the timed model checking) as well as the refinement check [11], we only have to adapt the concepts of Steps 2a and 2b. In this paper, we only focus on improving Step 2a without the conflict analysis. Thus, in the following two sections, we present the most commonly used dependency kinds of MechatronicUML components and propose to exchange MechatronicUML's formal dependency languages.

6 Kinds of Intra-Component Dependencies

By analyzing the modeling language and existing example models as well as interviewing MechatronicUML users, we identified 20 commonly used dependency kinds and list them in a catalog [9, Appx. B]. We classify them concerning the involved modeling elements (data, time, transition, state, message) and their direction (uni- or bidirectional). We describe each dependency kind via an uniform description format that defines six attributes: (1) the name of the kind, (2) an informal description, (3), the involved modeling elements, (4) the direction, (5) an informal example, and (6) the formalization of this example. In the following, we briefly introduce a selection of ten kinds due to space restrictions.

Synchronization The RTSCs of two discrete ports shall synchronize their behavior by enforcing that their transitions can only fire together.

Required State Constraint A transition of a discrete port shall be constrained by the status (inactive / active) of a state of another discrete port, e.g., a transition may only fire as long as another state is active.

Data Pull A discrete port shall have read access to local data variables of another discrete port.

Data Push A discrete port needs write access to an existing local data variable of another discrete port.

Data Merge Two or more data variables are in fact the same data variable and shall be merged into one. If one of them is a hybrid port, all data variables are merged into that port.

Clock Merge Two or more local clocks of different discrete ports are in fact the same clock and shall be merged into one.

Forbidden State Combination A particular set of states distributed over several discrete ports shall not be active at the same time.

Allowed State Combination States of different discrete ports may only be active at the same time if they are listed in one common set.

Allowed Message Sequence A certain message sequence shall be allowed, but subsets of these message sequences are not allowed.

Forbidden Message Sequence A sequence of messages shall be excluded from the combined state space of the independent behaviors. However, subsets of these message sequences are allowed.

In a small survey [9, Appx. C], we verified the completeness of our catalog. The results were that no commonly used kinds were left out and most of the identified kinds are commonly used. However, the survey showed that two dependency kinds, e.g., *Allowed State Combination*, are not as common as expected. If these results are confirmed in the future, we will remove them from the catalog.

7 Dependency Languages for MechatronicUML

The dependency languages of Eckardt and Henkler [6] (partially) support 12 dependency kinds. However, among others, the kinds *Synchronization*, *Data Push*, *Data Pull*, *Data Merge*, *Clock Merge*, and *Forbidden Message Sequence* are not supported. One reason for this is that the languages do not consider data variables. Thus, we either have to extend the languages, select existing languages, or define new languages.

Instead of ECAs, we propose to use Modal Sequence Diagrams (MSD) [10] to describe message-based dependencies as they do not only support the dependencies that an ECA may specify (e.g., *Allowed Message Sequence*) but also further dependency kinds like *Forbidden Message Sequence*.

Moreover, we replace SCRs by a new DSL named *MechatronicUML Intra-Component Dependency Language* (MIDL). MIDL shall support all dependency kinds of SCR and all dependency kinds that MSDs do not support. We provide a detailed definition of MIDL in [9, Sect. 4.4]. Below, we list three formal dependencies for component WhiteSw specified using MIDL that relate to the informal dependencies d1-d3 given in Sect. 1. The dependencies d1 and d3 are of kind *Transition Firing Constraint* while dependency d2 is of kind *Data Merge*. The formalized dependency d1 (line 1) states that the port Overtaker may only fire the transition from NoOvertaking.Init to Requested (and send the message request) if port Receiver is in state TrackSafe for more than 5s and if the hybrid port Distance has a value below 20. The formalized dependency d2 (line 2) states that the local variable mySpeed shall be merged into the hybrid port variable Velocity. The formalized dependency d3 (line 3) states that the port Overtaker may only fire the transition from NoOvertaking to Overtaking (and send the message finish) if the hybrid port Distance has a value higher than 20.

```
1  if [Receiver.TrackSafe is active longer than 5s] and [
     Distance < 20] {enable transition Overtaker.NoOvertaking.
     Init-->Requested};
2  merge variable Overtaker.mySpeed into Velocity;
3  if [Distance > 20] {enable transition Overtaker.Overtaking-->
     NoOvertaking};
```

We realize our language by defining a formal representation using Xtext's LL(*) attribute grammar (requirement r1). We modified the generated meta

model by referencing elements of the MechatronicUML meta model (r2) and by adding OCL constraints that prevent erroneous and superfluous dependencies (r3,r7). Furthermore, we provide an editor for our DSL which has usability features like syntax highlighting, live syntax check, and auto completion (r7).

In a small case study about trains that may enter various track sections [9, Appx. A], we were able to formalize 29 intra-component dependencies in total from three discrete component where each component consist of three to five ports. This is a first indicator that our language fulfills r4.

8 Related Work

In the related field of controller synthesis for discrete and timed systems, formal languages are required as an input. Via scenarios, the software engineer has to define possible interactions between the controller (the software under development) and the environment, e.g., using timed game automaton [1] or using modal sequence diagrams [10]. However, the scenarios are not independent. Thus, in contrary to us, a developer does not need to specify dependencies explicitly. However, these related approaches have other negative side effects, e.g., in contrary to our approach that requires independent behaviors, they cannot enable a scalable model checking of the system which is mandatory in CPSs.

Other synthesis approaches require temporal logic expression as input, e.g., Letier et al. [12] require the Fluent Linear Temporal Logic (FLTL) to specify event-based dependencies and Attie et al. [2] propose to specify inter-task dependencies using the temporal logic CTL. Uchitel et al. [14] even support scenarios as well as FLTL as inputs for the synthesis. However, none of these logics is able to express all intra-component dependency kind that we identified for the domain of CPSs. One reason for this is that the partial behavior models of these approaches do not support all properties of a CPS (e.g., real-time, data variables, accessing physical information).

Donatelli [5] proposes to directly insert formal dependencies within the automata. This is contrary to us, as we specify our dependencies in an additional model to separate concerns. Moreover, Donatelli only enables the specification of event-based dependencies. Thus, only a small subset of our required dependency kinds is covered.

9 Conclusion and Future Work

In this paper, we defined the requirements for specifying intra-component dependencies to enable the synthesis of discrete component behaviors for CPSs. We improve an existing synthesis process and identified the most commonly used intra-component dependencies for discrete components of MechatronicUML. Moreover, we propose the usage of MSDs as well as a new defined DSL called MIDL to formally describe these dependencies.

Using our proposed DSL, software engineers are able to formally define intra-component dependencies. In contrast to informally described dependencies, this

should avoid misunderstandings between the engineers. Moreover, MSDs and MIDL can serve as an input for a component behavior synthesis that only requires rare manual changes afterwards. Component models like Sofa 2 [13], which define the external component behavior in separate parts and have no component behavior synthesis yet, can apply our concepts to define their own DSL for specifying intra-component dependencies. We think that a lot of the dependency kinds should still be valid and that their DSL could be similar to MIDL.

In the future, we will complete the realization of our synthesis method. First, we will further evaluate MIDL. Then, we will develop a detection of dependencies conflicts. Afterwards, we will adapt the synthesis algorithm and perform a thorough evaluation of the complete method.

Acknowledgement We thank Tobias Eckardt, Christian Heinzemann, and Hendrik Kassner for their feedback on earlier versions of this paper.

References

1. Asarin, E., Maler, O., Pnueli, A.: Symbolic controller synthesis for discrete and timed systems. In: Antsaklis, P., Kohn, W., Nerode, A., Sastry, S. (eds.) Hybrid Systems II, pp. 1–20. LNCS 999, Springer Berlin Heidelberg (1995)
2. Attie, P.C., Singh, M.P.: Specifying and enforcing intertask dependencies. In: In Proceedings of the 19th VLDB Conference. pp. 134–145 (1993)
3. Baier, C., Katoen, J.P.: Principles of model checking. MIT Press (2008)
4. Becker, S., et al.: The MechatronicUML design method - process and language for platform-independent modeling. Tech. Rep. tr-ri-14-337, Heinz Nixdorf Institute, University of Paderborn (Mar 2014), version 0.4
5. Donatelli, S.: Dependent automata for the modelling of dependencies. In: Setola, R., Geretshuber, S. (eds.) Critical Information Infrastructure Security, pp. 311–318. LNCS 5508, Springer Berlin Heidelberg (2009)
6. Eckardt, T., Henkler, S.: Component behavior synthesis for critical systems. In: Giese, H. (ed.) ISARCS 2010. pp. 52–71. LNCS 6150, Springer (June 2010)
7. Gausemeier, J., Rammig, F.J., Schäfer, W. (eds.): Design Methodology for Intelligent Technical Systems. Lecture Notes in Mechanical Engineering, Springer (2014)
8. Giese, H., Tichy, M., Burmester, S., Schäfer, W., Flake, S.: Towards the compositional verification of real-time uml designs. pp. 38–47. ESEC/FSE'03 (Sep 2003)
9. Goschin, S.: Synthesis of Extended Hierarchical Real-Time Behavior. Master's thesis, University of Paderborn (Feb 2014), http://tinyurl.com/MA-Gos14
10. Greenyer, J.: Scenario-based Design of Mechatronic Systems. Ph.D. thesis, University of Paderborn (Oct 2011)
11. Heinzemann, C., Brenner, C., Dziwok, S., Schäfer, W.: Automata-based refinement checking for real-time systems. Computer Science - Research and Development (2014), published Online June 2014
12. Letier, E., Kramer, J., Magee, J., Uchitel, S.: Deriving event-based transition systems from goal-oriented requirements models. Automated Software Engineering 15(2), 175–206 (Jun 2008)
13. Malohlava, M., Hnetynka, P., Bures, T.: Sofa 2 component framework and its ecosystem. Electr. Notes Theor. Comput. Sci. 295, 101–106 (2013)
14. Uchitel, S., Brunet, G., Chechik, M.: Synthesis of partial behavior models from properties and scenarios. IEEE Transactions on SE 35(3), 384–406 (2009)

Multi-Platform Generative Development of Component & Connector Systems using Model and Code Libraries

Jan Oliver Ringert[1,2]*, Bernhard Rumpe[1], and Andreas Wortmann[1]

[1] Software Engineering
RWTH Aachen University
http://www.se-rwth.de/
[2] School of Computer Science
Tel Aviv University
http://www.cs.tau.ac.il/

Abstract. Component-based software engineering aims to reduce software development effort by reusing established components as building blocks of complex systems. Defining components in general-purpose programming languages restricts their reuse to platforms supporting these languages and complicates component composition with implementation details. The vision of model-driven engineering is to reduce the gap between developer intention and implementation details by lifting abstract models to primary development artifacts and systematically transforming these into executable systems. For sufficiently complex systems the transformation from abstract models to platform-specific implementations requires augmentation with platform-specific components. We propose a model-driven mechanism to transform platform-independent logical component & connector architectures into platform-specific implementations combining model and code libraries. This mechanism allows to postpone commitment to a specific platform and thus increases reuse of software architectures and components.

1 Introduction

Component-based software engineering (CBSE) [16] ultimately aims to compose complex systems from off-the-shelf components. Usually, components are provided as general-purpose programming language (GPL) source code. This restricts reuse to certain platforms and requires domain experts to become programming experts. Model-driven engineering (MDE) pursues to reduce the conceptual gap [7] between domain and implementation concepts by describing software systems as abstract models. These models can be systematically transformed into implementations for potentially multiple target platforms. Component & connector (C&C) architecture description languages (ADLs) [15] are

* J. O. Ringert acknowledges support from a postdoctoral Minerva Fellowship, funded by the German Federal Ministry for Education and Research.

modeling languages with high potential to combine the benefits of MDE and CBSE. Software architectures can be modeled platform-independently, enriched with platform-specific information, and transformed into an implementation.

We have developed the C&C ADL and framework MontiArcAutomaton [19,20] to facilitate MDE in robotics. MontiArcAutomaton supports the integration of the most suitable modeling languages and the composition and orchestration of independently developed code generators. Modeling software components and their behavior reduces the need for GPL components and liberates developers from implementation details. However, some components still require manual implementations or the integration of legacy code. As components models are tied to implementations by MontiArcAutomaton convention, architectures containing components with platform-specific implementations (PSIs) are tied to specific platforms as well. This poses challenges when generating PSIs from models.

We present a mechanism implemented in MontiArcAutomaton to enable modeling of logical, platform-independent C&C architectures and their transformation into PSIs for different platforms. This mechanism relies on a combination of model and code libraries as well as an application specific configuration that regulates the transition from models to PSIs. To illustrate the toolchain and its benefits, the next section introduces the MontiArcAutomaton modeling language and framework (Sect. 2). Afterwards, Sect. 3 explains the transformation toolchain itself and illustrates its application. Section 4 discusses related work and Sect. 5 concludes this contribution with an outlook on future work.

2 MontiArcAutomaton

MontiArcAutomaton [19,20] is a modeling language family and framework for generative MDE of robotics applications. Logical architectures are modeled as the hierarchical composition of components that provide the system's functionality. Components posses a stable interface comprising their type, configuration parameters, generic type parameters, and sets of typed input ports and output ports. A component is either atomic or composed. Atomic components specify behavior directly. The behavior of a composed component emerges from the interaction of its subcomponents. Components interact by sending and receiving messages over directed connectors between their ports. The types of ports are defined via class diagrams (CDs) or a GPL. Encapsulation of components with stable interfaces facilitates logically distributed development and physically distributed computation models. It enables component composition independent of their behavior description. MontiArcAutomaton exploits this encapsulation mechanism to allow the embedding of behavior modeling languages into atomic components [20]. Component developers may use the most suitable behavior description languages instead of GPLs.

MontiArcAutomaton is developed with the domain-specific language workbench MontiCore [13] which provides frameworks for language integration [14,26] and code generator development [23]. MontiCore languages are textual and defined by context free grammars with additional well-formedness rules. From these

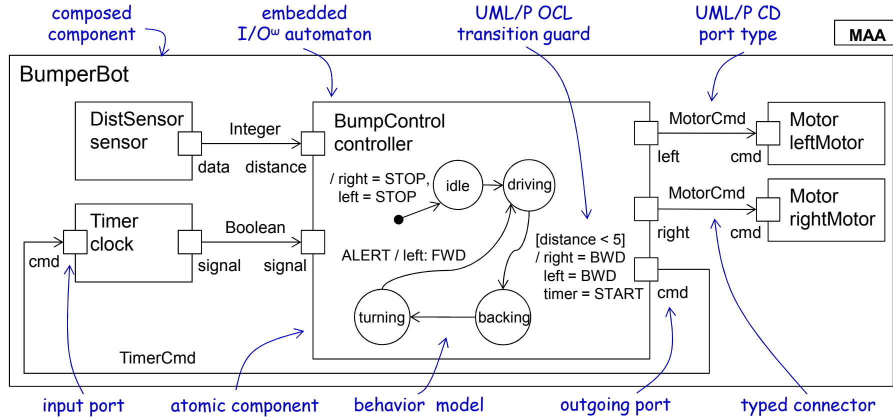


Fig. 1. Platform-independent software architecture of the composed component type `BumperBot` with five subcomponent instances.

grammars, MontiCore generates infrastructure to parse complying models into their corresponding abstract syntax tree (AST). MontiCore supports language inheritance, language embedding, and language aggregation (referencing and using models from other languages) [14,26] to compose new languages from existing ones and MontiArcAutomaton uses all three mechanisms: it extends the MontiArc [11] ADL, component behavior languages are embedded into the base ADL, and port types may use UML/P [22] CD models. The MontiCore code generation framework facilitates development of code generators using the FreeMarker[3] template engine to generate code from ASTs and code templates written in a target language [18,23]. MontiArcAutomaton comprises modeling languages, code generators, generator composition mechanisms, model-transformations, language integration support, and libraries.

Consider a robot that comprises a distance sensor to measure the distance to the closest obstacle ahead and motors to control its left and right wheel. The robot drives forward until it approaches an obstacle, then backs up, rotates, and continues to drive forward. Figure 1 depicts the logical software architecture of this robot which consists of the composed component `BumperBot` with five subcomponent instances: `sensor` of type `DistSensor`, `clock` of type `Timer`, `controller` of type `BumpControl`, and two instances `leftMotor` and `rightMotor` of type `Motor`. The subcomponent `sensor` has the single outgoing port `data` of type `Integer`, which is connected to the incoming port `distance` of `controller`. Based on the inputs received, the controller sends messages of type `MotorCmd` to the motors. This type is defined in a CD. The behavior of `controller` is modeled as an automaton following the I/O$^\omega$ automaton paradigm [17,21].

Executable code for the C&C architecture of the system requires some platform dependent component implementations. To execute the system on a Lego

[3] Website of the FreeMarker Java template engine: `http://freemarker.org/`.

NXT robot using the Lego Java Operating System (leJOS)[4] the component instances `leftMotor` and `rightMotor` require Java wrappers for the leJOS API. Executing the same system on a NXT robot using the Robot Operating System (ROS)[5] requires a Python implementation controlling ROS nodes. These platform specific components cannot easily be modeled and are among existing legacy components examples for the need of integrating GPL code in MDE.

3 Platform-Independent Model and Multi-Platform Code

To facilitate reuse of the same logical architecture model with different platforms, it is favorable to postpone commitment to a specific platform as long as possible. With MontiArcAutomaton this commitment is expressed as *binding* component instances to PSIs. We distinguish two kinds of components: *fully modeled components* are composed components or atomic components with an embedded behavior model. *Abstract components* are atomic components without a behavior model. The interfaces of abstract component types may refer only to types provided by the MontiArc type system and types defined in CDs. The port types depicted in Fig. 1 are such types. Fully modeled components require no binding as their implementation is generated by the combination of code generators for component structure and behavior.

Integrating existing code: Abstract components require GPL behavior implementations compatible with the generated code of the surrounding architecture. Integration of generated code with manual implementations can follow different patterns (e.g., generation gap [6] or delegation [8]). MontiArcAutomaton does not prescribe a pattern. Instead, MontiArcAutomaton code generators specify which *runtime environment* (RTE) they are compatible with. Such a RTE may employ appropriate patterns to integrate generated and manually implemented code, define how communication between components and scheduling are realized, and contain common domain functionality [24]. Technical details and requirements for the integrated code are RTE specific. Our RTE for Java component implementations defines an abstract class `Component` and factories [8] which enable utilization of the generation gap pattern. The code generator transforms component models into subclasses of `Component`, which realize the component behavior. For abstract components, the generator only creates the according factory and expects the component developer to provide an according component implementation in the RTE's GPL Java, i.e., to bind the component model to a PSI. A manual binding is error-prone and requires knowledge of implementation details of the generated code. Modeling the binding reduces these "accidental complexities" [7].

Model and code libraries: Enabling component developers to efficiently develop software architectures with abstract components requires to enable component and component implementation reuse. MontiArcAutomaton therefore

[4] Website of the Lego Java Operating System (leJOS): `http://www.lejos.org/`.
[5] Website of the Robot Operating System (ROS): `http://www.ros.org/`.

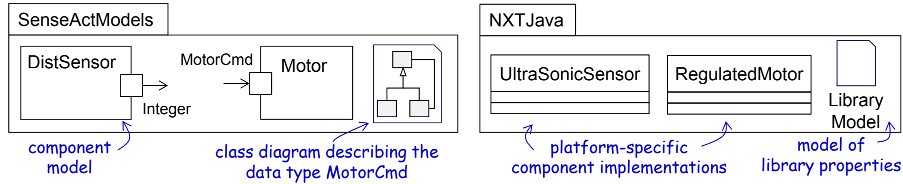


Fig. 2. Model library and code library used with the `BumperBot` software architecture.

distinguishes platform-independent *model libraries* and platform-specific *code libraries*. Model libraries contain fully modeled components, abstract components, and CDs. Code libraries contain component behavior implementations and port types formulated in a GPL. Furthermore, each code library contains a library properties model, describing the RTE of the contained implementations and component types each implementation conforms to. This is necessary to ensure compatibility of the generated and provided implementations for different RTEs.

The left part of Fig. 2 shows the model library `SenseActModels` used by the platform-independent `BumperBot` architecture depicted in Fig. 1. The right part shows a corresponding code library. The model library contains the abstract component models `DistSensor`, `Motor` and a CD modeling the data type `MotorCmd` used by component `Motor`. The `NXTJava` code library contains PSIs and a library properties model which describes the RTE `UltraSonicSensor` and `RegulatedMotor` are compatible with.

Binding PSIs: Retaining platform-independent architectures prohibits to model component binding in the logical architecture itself. Instead, MontiArcAutomaton applications may provide application configuration models. These describe the selected code generators and binding information. A binding describes a mapping of component instances of the architecture model to implementations. The mapping augments the architecture's AST before any code is generated and thus can be reused with arbitrary generator combinations.

The MontiArcAutomaton generator toolchain parses the application configuration and passes the binding information to a transformation which adds information about component implementations to the architecture. The generation framework considers this information and, e.g., generates factories instantiating the bound implementations accordingly.

Listing 1 shows the application configuration used to bind component instances `sensor`, `leftMotor`, `rightMotor`, and `clock`. First, the required implementation library is imported (ll. 1). Model libraries are imported by the architecture and made available to the application configuration. Afterwards (l. 4) code generators are selected. The generators declare which runtime environments they are compatible with and thus restrict which implementations can be bound. Finally, ll. 5-9 describe the actual bindings of the application. Here, component instances, identified by the name between `map` and `to`, are mapped to imported implementations, identified by the name after `to`. Please note that the two instances of the component motor are mapped to the same implementations `RegulatedMotor`.

ApplicationConfiguration

```
1  import NXTJava.*;
2
3  application NXTJavaBumperBot {
4    generators ComponentJava, AutomatonJava, CDJava;
5    bindings
6      map BumperBot.sensor      to UltraSonicSensor,
7      map BumperBot.leftMotor   to RegulatedMotor,
8      map BumperBot.rightMotor  to RegulatedMotor,
9      map BumperBot.clock       to JavaTimer;
10 }
```

Listing 1. Application configuration model for the BumperBot selecting code generators and binding component instances sensor, leftMotor, rightMotor, and clock.

Application configuration models are checked at design time whether all components are bound and whether the binding is compatible by reading the libraries' property models, which map the contained implementations to component types.

Implementation in MontiArcAutomaton: Figure 3 illustrates how the MontiArcAutomaton code generation framework integrates applications, code generators, libraries, and transformations of platform-independent architecture models into PSIs. The GenerationTool parses architecture and application models, which reference model libraries and code libraries, respectively. The result is passed to the BindingTransformation which augments the architecture before code generation. Architecture AST, binding, and imported libraries are passed to the BindingTransformation which transforms the AST accordingly. With the transformed AST, the GenerationTool starts the GeneratorOrchestration process which instantiates and executes the selected code generators as selected. Both, code library and code generators have to comply to the same runtime environment (RTE) to ensure an executable implementation of the architecture. The RTE provides interfaces manually implemented and generated components have to implement to ensure compatibility. Data types are translated into PSIs using the selected CD generator, which maps the basic types of the MontiArc type system onto platform-specific representations.

With help of the MontiArcAutomaton transformation toolchain, application configuration, and libraries the logical BumperBot architecture (Fig. 1) can be transformed into an intermediate platform-specific architecture where the subcomponents sensor, leftMotor, rightMotor, and clock are bound to PSIs. This resulting software architecture is passed to the code generation framework and ultimately transformed into implementations executable on robotic platforms.

An excerpt of the resulting implementations for two different platforms is shown in Fig. 4. The left panel shows the project structure of the BumperBot application containing two application configuration models. The first maps sensor, leftMotor, rightMotor and clock to Java implementations based on leJOS, the second maps them to Python implementations based on ROS. The bottom

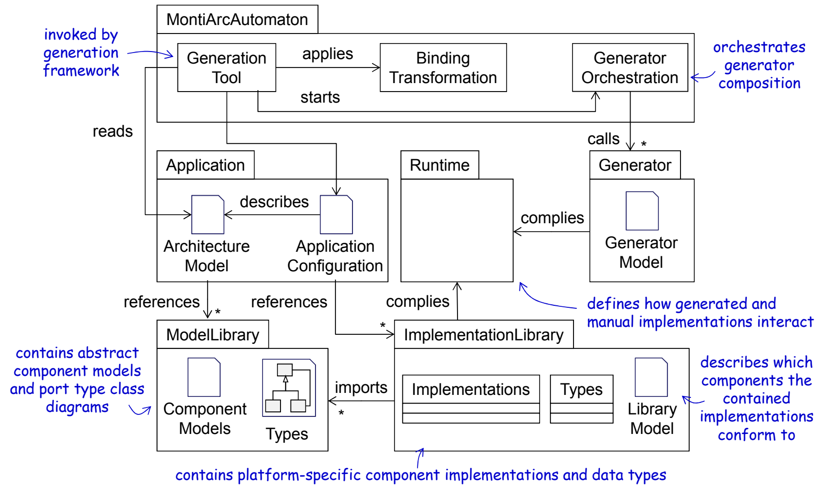


Fig. 3. Elements of the MontiArcAutomaton transformation toolchain and their dependencies.

panels show part of the generated implementations for component `BumperBot` where subcomponents are instantiated. The leJOS Java implementation uses the implementations `UltraSonicSensor` and `RegulatedMotor` and the ROS python implementation uses `RangeSensor` and `JointMotor` as defined in the respective application configuration models (depicted in the top panels).

4 Related Work

Related approaches are toolchains enabling platform-independent modeling and automated creation of source code implementations — especially ADL frameworks with code creation capabilities, e.g., the Architecture Analysis & Design Language [5] (AADL), AutoFocus [12], Simulink [25], and SysML [27].

AADL is modeling language for systems consisting of software components and hardware components. While AADL models could be subjected to late binding as well, AADL architectures models component implementations explicitly – thus hampering reuse. We are not aware of an integrated binding modeling language and framework for AADL. AutoFocus is a C&C ADL and modeling tool for the development of distributed systems based on the semantics of Focus [3]. Behavior is modeled as state transition diagrams similar to I/O$^\omega$ automata. In contrast to MontiArcAutomaton, AutoFocus lacks a distinction between component types and instances. This prohibits component reuse by instantiation and bindings as introduced above. MathWorks Simulink features a block diagram language to describe of components and connectors. Stateflow[6] extends blocks with state transition diagrams. Simulink relies on M2T code generation without

[6] Website of Stateflow: `http://www.mathworks.de/products/stateflow/`.

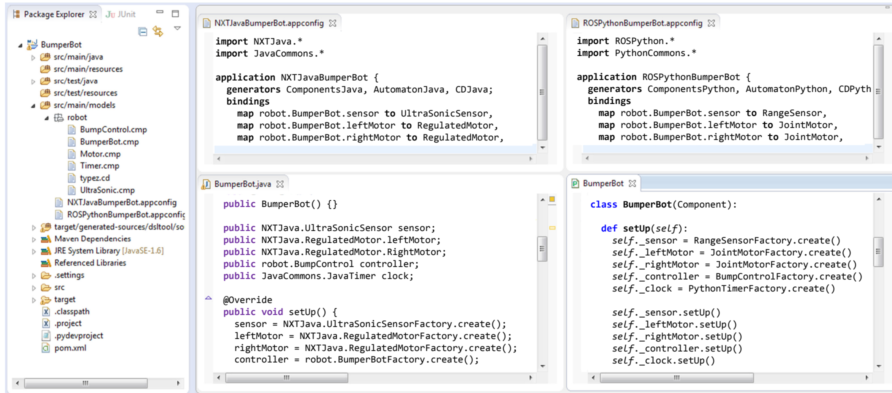


Fig. 4. Application configuration models and generated implementations for execution of the logical `BumperBot` architecture on two different platforms.

intermediate model transformations. SysML is a set of modeling languages based on a subset of extended UML [10]. The SysML language for internal block diagrams resembles MontiArcAutomaton and component behavior can be modeled with state machine diagrams, thus SysML enables to express architectures similar to MontiArcAutomaton. Modeling with SysML focuses on the requirements phase and thus provides "only models on the PIM level" [9]. In most approaches manually written code (if required) is typically integrated after code generation.

While we propose a binary notion of platform-independence compared to a continuous notion where "abstract platforms" [1] may add and refine platform-properties, e.g., an abstract-platform for the `BumperBot` could describe that it requires two motors. It is an interesting future work to evaluate these differences.

Other approaches to transform PIMs into PSIs focus different issues: the authors of [4], for example, transform platform-independent statecharts with real-time properties into PSMs via complex model analysis. Such languages and transformations are beyond the scope of this contribution.

5 Discussion and Conclusion

We presented a model-driven integrated, automated transformation toolchain, modeling languages, and library concepts for the transformation of platform-independent C&C software architectures into PSIs for multiple platforms. This transformation is defined as the binding of subcomponents to platform-specific component implementations. Abstract components are provided in model libraries while their implementations are provided in platform specific code libraries. To separate binding information for the architecture, we extended MontiArcAutomaton's application configuration modeling language to contain bindings. This separation enables reuse of logical architecture models with different source code implementations without modifications to the software architecture

Currently, bindings specify unconditional mappings. Different distribution scenarios might require to bind components under certain conditions (e.g., target platform properties). An extension of the application configuration language with conditions is easily possible due to MontiCore's language integration mechanisms. We currently explore different notions of interface compatibility as it might be feasible to bind components where a port's type might be a subtype of the abstract component's respective port. Another notion of interface compatibility is, that the replacing component extends the component of the replaced component instance in the sense of component inheritance [11]. While interface compatibility ensures syntactic well-formedness, it does not ensure that bound component implementations behave similarly. Securing this could be achieved by employing component behavior contracts. We are working on such mechanisms based on assumptions and guarantees [2].

Overall the MontiArcAutomaton toolchain integrates transformations and code generation seamlessly and enables easy reuse of the same software architecture on different platforms. In the future we plan to work on the issues mentioned above and evaluation of the toolchain.

References

1. Almeida, J.P., Dijkman, R., Van Sinderen, M., Pires, L.F.: Platform-independent modelling in mda: supporting abstract platforms. In: Model Driven Architecture, pp. 174–188. Springer (2005)
2. Broy, M.: Towards a theory of architectural contracts:-schemes and patterns of assumption/promise based system specification. In: Broy, M., Leuxner, C., Hoare, T. (eds.) Software and Systems Safety—Specification and Verification. NATO Science for Peace and Security Series—D: Information and Communication Security, vol. 30, pp. 33–87. IOS Press (2011)
3. Broy, M., Stølen, K.: Specification and Development of Interactive Systems. Focus on Streams, Interfaces and Refinement. Springer Verlag Heidelberg (2001)
4. Burmester, S., Giese, H., Schäfer, W.: Model-driven architecture for hard real-time systems: From platform independent models to code. In: Hartman, A., Kreische, D. (eds.) Model Driven Architecture – Foundations and Applications, Lecture Notes in Computer Science, vol. 3748, pp. 25–40. Springer Berlin Heidelberg (2005)
5. Feiler, P.H., Gluch, D.P.: Model-Based Engineering with AADL - An Introduction to the SAE Architecture Analysis and Design Language. SEI series in software engineering, Addison-Wesley (2012)
6. Fowler, M.: Domain-Specific Languages (Addison-Wesley Signature Series (Fowler)). Addison-Wesley Professional (2010)
7. France, R., Rumpe, B.: Model-driven development of complex software: A research roadmap. In: Future of Software Engineering 2007 at ICSE. pp. 37–54 (2007)
8. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Professional (1995)
9. Giese, H., Henkler, S.: A survey of approaches for the visual model-driven development of next generation software-intensive systems. Journal of Visual Languages & Computing 17(6), 528–550 (2006)
10. 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/

11. Haber, A., Ringert, J.O., Rumpe, B.: MontiArc - Architectural Modeling of Interactive Distributed and Cyber-Physical Systems. Tech. Rep. AIB-2012-03, RWTH Aachen (February 2012)
12. Hölzl, F., Feilkas, M.: 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, pp. 317–322. Springer (2011)
13. Krahn, H., Rumpe, B., Völkel, S.: MontiCore: a framework for compositional development of domain specific languages. STTT 12(5), 353–372 (2010)
14. Look, M., Perez, A.N., Ringert, J.O., Rumpe, B., Wortmann, A.: Black-box Integration of Heterogeneous Modeling Languages for Cyber-Physical Systems. In: Proceedings of the 1st Workshop on the Globalization of Modeling Languages (GEMOC). Miami, Florida, USA (2013)
15. Medvidovic, N., Taylor, R.: A Classification and Comparison Framework for Software Architecture Description Languages. IEEE Transactions on Software Engineering (2000)
16. Naur, P., Randell, B. (eds.): Software Engineering: Report of a conference sponsored by the NATO Science Committee, Garmisch, Germany, 7-11 Oct. 1968, Brussels, Scientific Affairs Division, NATO (1969)
17. Ringert, J.O., Rumpe, B.: A Little Synopsis on Streams, Stream Processing Functions, and State-Based Stream Processing. International Journal of Software and Informatics 5(1-2), 29–53 (July 2011)
18. Ringert, J.O., Rumpe, B., Wortmann, A.: A Case Study on Model-Based Development of Robotic Systems using MontiArc with Embedded Automata. In: Giese, H., Huhn, M., Philipps, J., Schätz, B. (eds.) Dagstuhl-Workshop MBEES: Modellbasierte Entwicklung eingebetteter Systeme. pp. 30–43 (2013)
19. Ringert, J.O., Rumpe, B., Wortmann, A.: From Software Architecture Structure and Behavior Modeling to Implementations of Cyber-Physical Systems. In: Software Engineering 2013 Workshop Proceedings. pp. 155–170. LNI (2013)
20. Ringert, J.O., Rumpe, B., Wortmann, A.: MontiArcAutomaton : Modeling Architecture and Behavior of Robotic Systems. In: Workshops and Tutorials Proceedings of the International Conference on Robotics and Automation (ICRA). Karlsruhe, Germany (2013)
21. Rumpe, B.: Formale Methodik des Entwurfs verteilter objektorientierter Systeme. Doktorarbeit, Technische Universität München (1996)
22. Rumpe, B.: Modellierung mit UML. Xpert.press, Springer Berlin, 2nd edn. (September 2011)
23. Schindler, M.: Eine Werkzeuginfrastruktur zur agilen Entwicklung mit der UML/P. Aachener Informatik-Berichte, Software Engineering, Band 11, Shaker Verlag (2012)
24. Sun, Y., Gray, J., Bulheller, K., von Baillou, N.: A model-driven approach to support engineering changes in industrial robotics software. In: France, R.B., Kazmeier, J., Breu, R., Atkinson, C. (eds.) Model Driven Engineering Languages and Systems, Lecture Notes in Computer Science, vol. 7590, pp. 368–382. Springer Berlin Heidelberg (2012)
25. Tyagi, A.K.: MATLAB and SIMULINK for Engineers. Oxford University Press (2012)
26. Völkel, S.: Kompositionale Entwicklung domänenspezifischer Sprachen. Aachener Informatik-Berichte, Software Engineering Band 9. 2011, Shaker Verlag (2011)
27. Weilkiens, T.: Systems Engineering mit SysML/UML. Dpunkt.Verlag GmbH (2008)

Interaction Components Between Components based on a Middleware

Văn Cam Phạm, Önder Gürcan, Ansgar Radermacher

CEA, LIST, Laboratory of Model driven engineering for embedded systems,
Point Courrier 174, Gif-sur-Yvette, F-91191 France
name.surname@cea.fr

Abstract. One of the problems of systems based on distributed architectures is the communication between applications running on different platforms on a network. The appearance of middleware reduces the complexity in transferring data between heterogeneous platforms of such systems. Up until now, various middleware have been proposed to facilitate the distributed system construction. In the context of component-based development, connectors represent links that realize the communication between application components. However, from the modeling perspective, the transition from the behavior of connectors to middleware implementation is still not clear.
This paper reports how to model the interaction components that define the behavior of connectors by using the ZeroMQ middleware due to several advantages it offers such as effective asynchronous communication patterns. In order to test our approach, we designed and implemented several different examples. Based on these examples, we observed that implementing interaction components between components based on a middleware simplifies the connection between components in a distributed system.

1 Introduction

A distributed system consists of multiple different *application components* that connect together to exchange data. These components usually run on heterogeneous platforms and thus have to handle platform differences such as byte-order. In model-driven approaches, this problem is often tackled by abstracting the communication logic from its implementation. In the UML specification, *connectors* illustrate such abstract communication links between the application components. However, the UML specification does not define the behavior of connectors. Therefore, an additional refinement is required on the model level.

On the implementation level, it is possible to integrate the connection code into application components directly. In other words, the connection code is integrated into the application components. Nevertheless, the management of application components becomes more difficult as their number increases and the embedded connection code cannot be reused. It is therefore necessary to

separate interaction components[1] from application components; hence developers can focus on application components without taking the communication into account.

In case of heterogeneous platforms, the implementation of connections needs to take several issues into account, notably different conventions for the ordering of bytes within a word[2]. In addition, it is also difficult to directly manage complicated connections from the application using socket connections since many sockets need to be created. Middleware is a way to overcome such difficulties since it offers a higher level of abstraction and does not depend on the underlying operating system.

The presented paper is based on previous work in this area, notably the support of connectors [9] for the UML profile MARTE and the support of simple socket interactions in [10]. The Qompass designer tool chain has been developed in the context of this work. It is a code generation and deployment extension of the UML modeler Papyrus[3]. The novelties of this paper are (1) the presentation of an additional interaction component based on the ZeroMQ middleware and (2) the support of asynchronous requests with return values (also called deferred synchronous calls).

The remaining of this paper is organized as follows. Section 2 outlines the methods and tools. Section 3 presents the modeling of ZeroMQ interaction components. Section 4 shows examples to test our implementation. Section 5 gives the related work and Section 6 concludes the paper.

2 Background

In this section, we introduce the method and tools used for our study, in particular Qompass Designer. It is used to transform models and deploy an application. Besides a model of the application software, the input model consists of a library of interaction components (and container services), a platform description and a deployment description that declares, configures and allocates instances. In the context of this paper, we only focus on application and interaction components. The component model is enriched by means of the Flex-eware Component Model (FCM) profile. It provides (among other extensions) a means to enrich ports of components. An FCM port has an additional port kind (extensible) that denotes whether the port is for instance a client/server, data-flow or event port. From an implementation perspective, the port kind determines the required and provided interfaces of this port.

There are basically two main steps for using interaction components[4] (1) transforming the UML application model into an intermediate model, and (2)

[1] As a common terminology, components that implement a UML connector are called *interaction components*.

[2] Orderding of bytes, http://www.gnu.org/software/libc/manual/html_node/Byte-Order.html, accessed on 07/07/2014.

[3] Papyrus, `http://www.eclipse.org/papyrus/`, accessed on 17/07/2014.

[4] It is basically a UML component (class) tagged as interaction component.

generating the implementation code from the intermediate model. The first transformation step replaces UML connectors with interaction components, as detailed later (see Fig. 2).

From the perspective of a developer who wants to incorporate new interaction components, a preliminary step is the modeling of this interaction component. This is done in form of a stereotyped class. The interaction component has ports to connect to the ports of application components. To be able to allocate these ports on different platforms, interaction components are logically decomposed into several fragments [10] (fragment per node). For example, a uni-directional communication interaction component has a sending fragment and a receiving fragment. These logically connected fragments are physically connected by using programming languages such as C++, Java in the implementation level. In this work, a new interaction component on top of the ZeroMQ (also known as ZMQ) middleware is developed[5] since we want to apply the AMI callback pattern and ZeroMQ offers a set of asynchronous socket APIs that transfers messages quickly and efficiently over the network. These sockets run on top of the standard sockets of operating systems and carry atomic messages across various transports such as in-process, inter-process, TCP, and multicast. The modeling of the interaction component is detailed in the next section.

In this study, we focus on *asynchronous method invocation (AMI) callback communication pattern* [11] since it allows clients to achieve high performance. For example, in a client/server application, a client sends a request to a server. Instead of blocking and waiting for a reply from the server (as synchronous calls), it provides callback functions to be invoked in order to process results received. These callback functions are called once replies are received. In the sense of component-based development, we use ports dedicated to the AMI callback pattern that are used by applying the AMI callback element of the FCM profile (see Fig. 1).

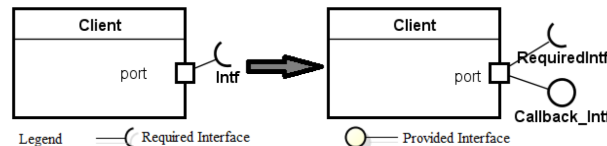


Fig. 1: The AMI port has two interfaces (right), one required and one provided, derived from a original port interface (left). The provided interface is needed since it contains callback functions that are invoked through the AMI callback port.

During application deployment, the modeled UML connectors are transformed into interaction components in an intermediate model (Fig. 2) by using Qompass Designer. The FCM *Connector* stereotype references the interaction

[5] ZeroMQ, `http://zeromq.org/`, accessed on 18/07/2014.

component that should be used. The transformation adapts the interaction component automatically to the application components that are connected, e.g. the ports of the interaction component need to be compatible with the ports of the connected application components. The ports of application components then connect to the ports of the generated interaction components instead of the end points of the UML connectors.

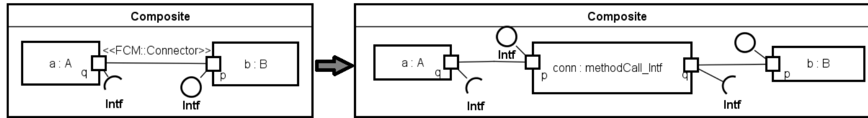


Fig. 2: Transformation from a system with (a) line of connector to (b) a composite structure of connector

The implementation code is generated from the intermediate model (a UML2 model with expanded interaction components). The code generator is basically generating C++ code from the UML model.

3 Interaction components modeling based on ZeroMQ

In this section, we present the decomposition of connectors and how AMI callback ports are used for modeling the asynchronous communication pattern. The AMI ports are dedicated for asynchronous requesting components such as clients in Client/Server applications.

This interaction component (see Fig. 3) contains fragments that define the behavior of connectors, provides interfaces to connect to application components through its ports and are co-located with appropriate application components on specific nodes of platforms. Interaction components often have two ports to connect two application components, but the concept is not limited to this case.

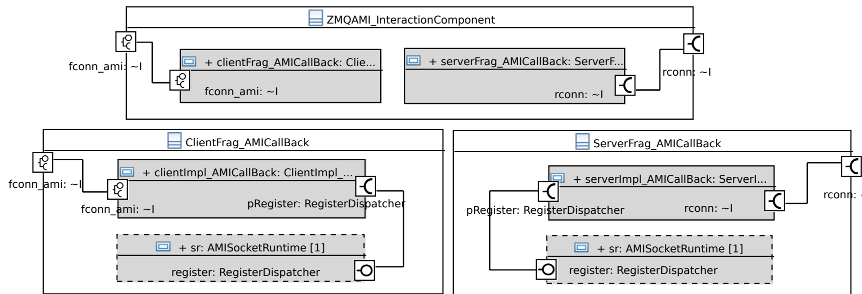


Fig. 3: Interaction component composite for AMI Callback model

The client fragment (ClientFrag_AMICallback) is asynchronous and the server fragment (ServerFrag_AMICallback) is synchronous. The interaction component needs to be reusable in other applications; hence the interfaces of the ports of the interaction component must match with the interfaces of different application components. In other words, when the interfaces of a port change, the interaction component has to adapt with the new interfaces.

To do this, we use an interface I as a formal parameter in a template and the ports of the interaction component are typed with this template. I is then bound to a specific interface when it is in use. The template binding defined here is realized by model to model transformations in Qompass Designer.

The inside of each of these two fragments is divided into two parts to differentiate between dispatching (xImpl) and communication (SocketRuntime) tasks. For the client and the server, there are ClientImpl and ServerImpl respectively that *dispatch* the requests or callbacks to right addresses. SocketRuntime, on the other hand, permits the dispatching component to register the dispatch interface (RegisterDispatcher) to the corresponding port (pRegister). RegisterDispatcher is called when the SocketRuntime receives some data. To realize this mechanism, SocketRuntime uses a set of ZeroMQ sockets to connect to the application components.

When a requesting component (e.g., client) calls a function through the AMI method invocation, the in/inout parameters of the function are marshalled into a chain of bytes. These parameters are stored in a buffer of the interaction component, ClientImpl in particular. These parameters are then passed as the parameters of the callbacks. This storage is essential to distinguish callbacks from multiple invocations since different callbacks corresponding to different input parameters may process results received in different ways. The parameters marshaling is generated from an Acceleo template. An example of an interface with two operations (int sum(int a, int b) and int square(int value)) is shown in Fig. 4. The chain of bytes also includes an operation ID and a handler ID. The operation ID is used by the server to determine the right processing function and the hander ID to find again the input parameters saved corresponding to the right results received. The callbacks therefore execute with its results and input parameters. The called function returns immediately after saving its parameters. The requesting component can go ahead without waiting for results. Data are actually sent and received in background threads. The socket runtime at the server side receives the request, calls dispatch to de-marshal parameters and then execute the right function to get the result. The de-marshaling of parameters is also generated from Acceleo.

The maximum number of input data has to be configured by users. For network applications with high calls number density or high computational time on servers, this number should be large enough to prevent the data of previous requests from overwriting. ClientFragment (sender) has a DEALER[6] socket of ZeroMQ to send requests and a ROUTER socket to asynchronously receive replies

[6] See the ZeroMQ web site for more information.

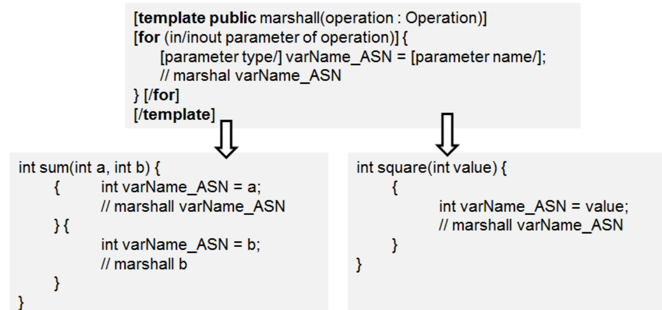


Fig. 4: Transformation from Acceleo to C++ code

from ServerFragment (recipient). The DEALER socket connects to a ROUTER socket of the recipient. These sockets offer asynchronous data transfers.

4 Examples

In this section, we present two examples to testify our interaction component implementation. The first example is about a client/server application. The second one is about a simple load balancing application.

4.1 Client/Server application using AMI callback

This system consists of a client and a server. The client is asynchronous and requests to the server through AMI callback communication with the interface ICompute as shown in Fig. 5. The interface has two operations: add(in a:Long, in b:Long):Long and mult(in a:Long, in b:Long):Long. The client needs to initiate requests. For this need, the FCM provides a simple convention: the client possesses a port start providing the interface IStart. This interface contains a run method that is automatically called pending the system start-up. The connector between the client and the server refers the interaction component implemented.

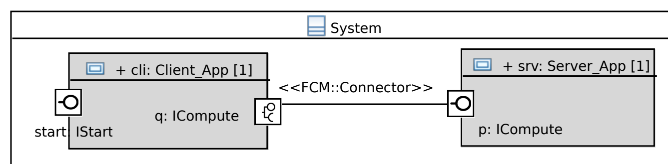


Fig. 5: Client/Server using AMI Callback example

For the deployment, the system is distributed on two different nodes. The client is deployed on ClientNode, the server on ServerNode as exposed in Fig. 6.

The fragments of the connector are co-located with the application components. The model transformed by Qompass Designer is then the input of the code generation process. This process is realized by using Acceleo[7].

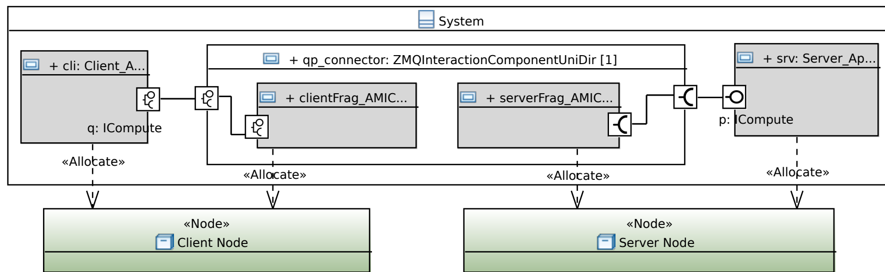


Fig. 6: Client/Server AMI callback example deployment

4.2 Interactions between components in the load balancing model

The Client/Server model is widely used because of its simplicity and facility of implementation. However, the model presents some issues, i.e. it is difficult to scale since the server must always run or the server can be a bottleneck since it has to treat all requests. Load balancing model[8] has been proposed as a solution to overcome these issues.

Load balancing is offered by ZeroMQ for distributing workloads of an application onto several servers called workers. Workloads distribution is performed by a broker component. The workers have the computational responsibility. They expedite the result to the broker.

In this example, there are one client, one broker and one worker (on the type level). The client needs to implement call back functions. AMI callback port kind is used. The ZMQAMI_InteractionComponent interaction component is applied to the connectors between components. The workers act synchronously. Information about the address and listening ports of the broker is configured. Clients need to know the front end port number and the broker's IP address and workers know about the back end's.

The application components of the system are allocated onto three nodes, client node, worker node, broker node. Many instances of client and worker can be run in different platforms. The broker has to start firstly and listen on the worker side (back end). When a worker begins, it sends a ready signal to the broker and the broker sets it as an available worker. The broker only actives on the client (front end) side if there is one available worker at least. Requests are forwarded from the broker and arrive to the workers alternatively.

[7] Acceleo, http://www.eclipse.org/acceleo/, accessed on 17/07/2014.
[8] Load Balanced Cluster, http://msdn.microsoft.com/en-us/library/ff648960.aspx, accessed on 12/09/2014.

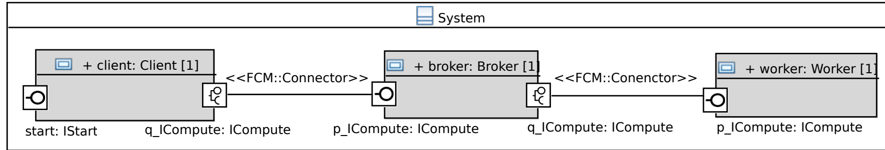


Fig. 7: Simple application follows load balancing model

5 Related Work

The concept of interaction components presented in this paper is supported by multiple component models and tools in other terminologies. The following sketches several works that are categorized into AMI callback implementations, component models supporting interaction components and implementations based on ZeroMQ.

Arulanthu et al. [1] provide the implementation of AMI callback for CORBA. Their implementation is in TAO [11]. They use the IDL (interface definition language) compiler to generate callbacks from the original interface. However, this does not resolve asynchronous messaging in MDE. At a higher level of abstraction, asynchronous messaging has been integrated into the CORBA component model (CCM), called AMI4CCM [7]. An AMI4CCM connector (analogous to the interaction component described in this paper) is responsible for managing the interaction. The connector is part of the extensibility mechanism in CCM, providing a so-called generic interaction support (GIS). The major difference between AMI4CCM and the work presented here is to address the concepts directly at the modeling level and the support for the middleware ZeroMQ. Please note that Qompass designer is inspired by CCM and supports similar concepts.

The interest for a further standardization of component models with extensible interaction support is expressed by a request-for-proposal of a Unified Component Model (UCM) [8] that the Object Management Group (OMG) has issued recently. In the sequel we reference two older component models with this ability, before we talk about a different approach to build higher level services on top of ZeroMQ: ZeroRPC.

SOFA 2[9] [5, 6, 4] is a component system employing hierarchically composed components. SOFA connectors are automatically generated. A connector might support a transport mechanism such as CORBA or low level mechanisms. In this context, they are responsible for marshaling and de-marshaling. The proposed connector architecture consists of a distributor deployment unit and several sender/recipient units. The sender/recipient unit allows sending messages to attached components. The sender/recipient units connect to the distributor unit in a similar way. Connector configurations and deployment models are also shown. However, SOFA 2 does not support UML.

Fractal is a hierarchical and reflective component model [3]. It is intended to implement, deploy, and manage complex software systems, including in partic-

[9] SOFA, `http://sofa.ow2.org/`, accessed on 15/09/2014.

ular operation systems and middleware. Fractal connectors are Fractal binding components with behavior [2].A composite binding component is a communication path between an arbitrary numbers of component interfaces, of arbitrary language types. These bindings are represented as a set of primitive bindings and binding components (stubs, skeletons, adapters in the con-text of remote method calls).

ZeroRPC[10] is a light-weight, reliable and modern communication library for distributed systems. ZeroRPC builds on top of ZeroMQ and MessagePack. ZeroRPC is more than a typical Remote Procedure Call (RPC) engine and supports multiple ZeroMQ socket types, streaming, heartbeat and more. ZeroRPC is created to satisfy requirements such as exposing arbitrary code with minimal modification, self-document systems, propagate exceptions, trace nested calls and provide brokerless, highly available, fast fan-in/fan-out. However, ZeroRPC focuses on communications between server-side processes and so far is only implemented in Python and Node.js that are not suitable to distributed embedded applications

6 Conclusion and Future Work

In this paper, we have shown the modeling in UML of the AMI interaction component that defines the behavior of connectors. We used the stereotypes of the FCM profile to apply UML connectors and ports for the modeling. A UML connector applying the *Connector* stereotype of the FCM profile is transformed to a composite structure. We used Papyrus to model and Qompass Designer to transform models. At the physical connection level, we used the ZeroMQ middleware due to the several advantages it offers.

After the modeling of the interaction component, we tested it with two examples. One is a simple Client/Server application with asynchronous client and synchronous server; the other one is a simple load balancing application[11]. The separation between interaction and application components simplifies the development process of distributed systems. The interaction component can be reused in other applications. Application components developers can therefore focus on data processing at application level.

As future work, we will enrich the properties of quality of service for the interaction components to provide more reliable communications. We will also further study systems with dynamic adaptation which are currently poorly supported by our approach.

The work presented in this paper is supported by the European project SafeAdapt, grant agreement No. 608945, see `http://www.SafeAdapt.eu`.

[10] ZeroRPC, `http://zerorpc.dotcloud.com/`, accessed on 15/09/2014.

[11] We also support publisher/subscriber and producer/consumer patterns, but we are unable to present them here due to space limitation.

References

[1] Alexander B. Arulanthu, Carlos O'Ryan, Douglas C. Schmidt, Michael Kircher, and Jeff Parsons. The design and performance of a scalable orb architecture for cobra asynchronous messaging. In IFIP/ACM International Conference on Distributed Systems Platforms, Middleware '00, pages 208–230, Secaucus, NJ, USA, 2000. Springer-Verlag New York, Inc.

[2] Tomás Barros, Rabea Boulifa, Antonio Cansado, Ludovic Henrio, and Eric Madelaine. Behavioural Models for Distributed Fractal Components. Rapport de recherche RR-6491, INRIA, 2008.

[3] E. Bruneton, T. Coupaye, and J.B. Stefani. The Fractal Component Model, February 2004. Version 2.0-3.

[4] Lubomir Bulej and Tomas Bures. Using connectors for deployment of heterogeneous applications in the context of omg d&c specification. In Dimitri Konstantas, Jean-Paul Bourrières, Michel Léonard, and Nacer Boudjlida, editors, Interoperability of Enterprise Software and Applications, pages 349–360. Springer London, 2006.

[5] Tomas Bures and Frantisek Plasil. Communication style driven connector configurations. In LNCS3026, ISBN 3-540-21975-7, ISSN 0302-9743, pages 102–116. Springer-Verlag, 2004.

[6] Ondrej Galik and Tomás Bures. Generating connectors for heterogeneous deployment. In Elisabetta Di Nitto and Amy L. Murphy, editors, Proceedings of the 5th International Workshop on Software Engineering and Middleware, SEM 2005, Lisbon, Portugal, September 5-6, 2005, pages 54–61. ACM, 2005.

[7] Object Management Group. Asynchronous method invocation for ccm. Specification Version 1.0, Object Management Group, April 2013.

[8] OMG. OMG Unified Component Model for Distributed, Real-Time and Embedded Systems. Request for proposal, OMG, May 2014. http://www.omgwiki.org/ucm/doku.php.

[9] Ansgar Radermacher, Arnaud Cuccuru, Sebastien Gerard, and François Terrier. Generating Execution Infrastructures for Component-oriented Specifications with a Model Driven Toolchain: A Case Study for MARTE's GCM and Real-time Annotations. SIGPLAN Not., 45(2):127–136, October 2009.

[10] Ansgar Radermacher, Önder Gürcan, Arnaud Cuccuru, Sebastien Gerard, and Brahim Hamid. Split of composite components for distributed applications. In Torsten Maehne and Marie-Minerve Louërat (eds), editors, Languages, Design Methods, and Tools for Electronic System Design, chapter 14, pages 255–267. Springer, Septembre 2014. doi:10.1007/978-3-319-06317-1_14.

[11] Douglas C. Schmidt, David L. Levine, and Sumedh Mungee. The Design of the TAO Real-time Object Request Broker. Computer Communications, 21(4):294–324, April 1998.

Towards a metamodel for the Rubus Component Model

Alessio Bucaioni, Antonio Cicchetti, and Mikael Sjödin

Mälardalen Real-Time Research Centre (MRTC)
School of Innovation, Design and Engineering (IDT)
Mälardalen University, Västerås, Sweden
{alessio.bucaioni,antonio.cicchetti,mikael.sjodin}@mdh.se

Abstract. *Component-Based Software Engineering has been recognized as an effective practice for dealing with the increasing complexity of the software for vehicular embedded systems. Despite the advantages it has introduced in terms of reasoning, design and reusability, the software development for vehicular embedded systems is still hampered by constellations of different processes, file formats and tools, which often require manual ad hoc translations. By exploiting the crossplay of Component-Based Software Engineering and Model-Driven Engineering, we take initial steps towards the definition of a seamless chain for the structural, functional and execution modeling of software for vehicular embedded systems. To this end, one of the entry requirements is the metamodels definition of all the technologies used along the software development. In this work, we define a metamodel for an industrial component model, Rubus Component Model, used for the software development of vehicular real-time embedded systems by several international companies. We focus on the definition of metamodeling elements representing the software architecture.*

Keywords: Component-Based Software Engineering, Model-Driven Engineering, Component-Based Software Systems, Vehicular Embedded Systems, Rubus Component Model

1 Introduction

During the last decades, industrial requirements on vehicular embedded systems have been constantly evolving causing an enlargement of the related software complexity: it has been estimated that current vehicles have more than 70 embedded systems running up to 100 million lines of code [14]. In this context, traditional software development processes have revealed strong limitations. On the one hand, industry needs efficient processes for reducing software development cost and time-to-market. On the other hand, most of the vehicular embedded systems present real-time properties, which have to be taken into account from the early stages of the development.

Component Based Software Engineering (CBSE) [15] has been acknowledged as an effective practice for dealing with the increasing software complexity. It promotes the development of the system at higher level of abstraction relying on the definition and reuse of atomic unit of composition, i.e., components. Also, CBSE allows to annotate components, at design time, with real-time properties

and constraints, e.g., worst-case execution time, enabling pre-run-time analysis, e.g., end-to-end response time and delay analysis [15].

Several component-based development processes have been introduced for improving the vehicular embedded systems software development. EAST-ADL, together with its follow-up initiatives, is the *de facto* standard for the software development of vehicular embedded systems. Among other contributions, EAST-ADL has standardized the terminology and promoted separation of concerns through a top-down development process, which makes use of four different abstraction layers. Despite the great initial reception, EAST-ADL is rarely adopted as it is. For instance, considering modern vehicle development, e.g., vehicles product line, it is very unlikely that vehicles are developed from scratch using top-down approaches. Contrariwise, they are mostly developed using a bottom-up strategy, reusing pre developed and tested components. In this context, the process defined by EAST-ADL finishes to hamper the software development, as the concepts used in each layer are designed for hiding non necessary information at higher and lower layers. While this can be effective within a top-down strategy - where the artifacts are enriched as they move forward towards the development layers - it is counterproductive when used within a bottom-up strategy - where low-level information, such as component's real time properties, need to be available at the earlier development stages. Also, the industrial vehicle software development is hindered by manual *ad hoc* translations, needed to integrate legacy systems and external tools: automation and tools integration have been acknowledged, by several projects [1], as key factors when dealing with extensive architectures as those for vehicular embedded systems. Information management, interoperability and traceability issues can not be fully solved using CBSE, as the discipline itself was not defined towards such aspects [15].

Model-Driven Engineering (MDE) is a discipline which promotes the separation of concerns by using different models for different concerns [16]. Unlike CBSE, MDE establishes precise relationships among models for the automatic generation of new models, change propagation and model-synchronization [16]. In this respect, MDE enhances software development targeting important development issues, such as information management, traceability, integration and interoperability [17].

We propose to exploit the crossplay of MDE and CBSE for realizing a seamless chain for the structural, functional and execution modeling of software for vehicular embedded systems. To this end, we believe one of the entry requirements is the metamodels definition of all the technologies used along the software development. In this work we define a metamodel for the Rubus Component Model (RCM), a component model (CM) used in the development of resource-constrained real-time vehicular embedded systems, focusing on the metamodeling elements representing the software architecture. As a proof of concept, we show a model transformation from RCM to AUTOSAR (RCM2AUTOSAR).

The rest of the paper is organized as follows. Section 2 presents the context of this work. Section 3 introduces the RCM metamodel. Section 4 shows the

[1] OSLC: http://open-services.net ; CRYSTAL: http://www.crystal-artemis.eu

RCM2AUTOSAR transformation while Section 5 discusses some related works. Finally, Section 6 draws conclusions and future works.

2 Context

In this section we present the context of this work by describing the four abstraction layers used in the software development of vehicular embedded systems. Additionally, we give some insights about RCM and the accompanying tool suite.

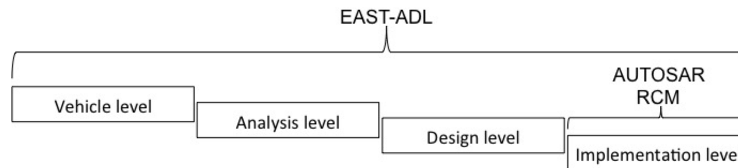


Fig. 1. The four abstraction layers as introduced by the EAST-ADL specification

2.1 Abstraction levels

EAST-ADL standardized a top-down development process composed of four different abstraction layers. Despite the top-down strategy is rarely used in industry, the abstraction layers and the related terminology have been fully adopted. The four abstraction levels are shown in Figure 1.

Vehicle level The vehicle level captures all the information regarding what the system is supposed to do. Feature models can be used for showing what the system provides and, eventually, how the product line is organized in terms of available assets. Feature models can be complemented with requirements. The vehicle layer is also known as End-to-End level as it serves to capture requirements and features on the end-to-end vehicle functionality.

Analysis level In the analysis level, vehicle functions are expressed using formal notations. The functionality are defined in terms of behaviors and interfaces. Yet, design and implementation details are omitted. At this stage, high level analysis for functional verification can be performed.

Design level In this level, the analysis-level artifacts are refined with more design-oriented details. While the analysis level does not differentiate among software, middleware abstraction and hardware architecture, the Design level explicitly separates this areas of the system implementation. Also, software functions to hardware allocation is expressed in this layer.

Implementation level In the implementation layer, the design-level artifacts are refined with implementation details. At this stage CMs, e.g., AUTOSAR, RCM, can be used to model the systems in terms of components and interactions among them. The output of this layer is a complete software architecture used for the code synthesis.

2.2 The Rubus Concept

Rubus [10] is a collection of methods, theories and tools for model- and component-based development of resource-constrained embedded real-time systems; it is developed by Arcticus Systems in collaboration with Mälardalen University. It is mainly used for the development of control functionality in vehicles by several international companies. The Rubus concept is based around the RCM[11] and its development environment Rubus-ICE [10]. Rubus-ICE includes:

- The Rubus Analysis Framework, for expressing real-time requirements and properties while modeling the system architecture;
- The Rubus Code Generator and Run-Time System, for synthesizing the code from the specified architecture;
- The Rubus SIMulation Model (RSIM), for simulating and testing the architecture at all the different hierarchical levels, e.g., components, Electronic Control Units (ECUs), subsystems, complete distributed system;
- The Rubus Execution Platform, for optimizing the run-time architecture.

With respect to the aforesaid four layers architecture, RCM is currently used in the implementation level as alternative/complement to AUTOSAR .

3 Providing a Metamodel to RCM

In this section, we present the RCM metamodel. [2] focusing on the metamodel definition of the architectural elements. For reading sake, we present the metamodel in three sections: Section 3.1 introduces the metamodel backbone, Section 3.2 introduces the metamodel elements for the data flow while Section 3.3 introduces the metamodel elements for the control flow.

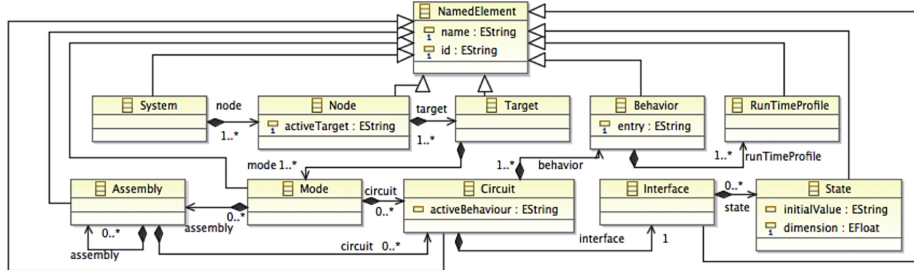


Fig. 2. Metamodel fragment for the backbone architectural elements

3.1 Backbone

Figure 2 shows the metamodel backbone. The top element is *System*, which acts as a container for the whole architecture. System, as all the elments in the metamodel, inherits from the abstract element *NamedElement*. A System element contains one or more *Node*(s). A Node is a hardware and operating-system independent abstraction of a *Target* and groups the software architecture elements which realize a certain function. Its attribute *activeTarget* defines which

[2] In this work, we do not seek to explain RCM; the interested reader may refer to [11]

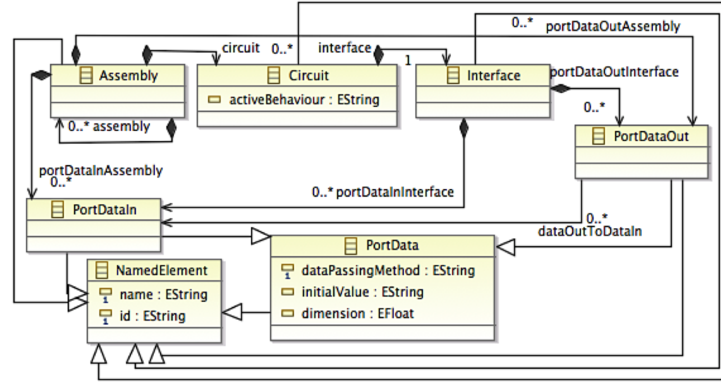
Target, among those specified, is active for a certain Node. A Target is a hardware and operating-system specific instance of a Node, which models the deployment of the software architecture, that is, it contains all the functions which have to be deployed on the same ECU. A Node can be realized by different Targets, depending from which hardware and operating system are considered for the deployment, i.e., PowerPC with Rubus Operating System, Simulated target with Windows operating system. A Target might contain one or more *Mode*(s). A Mode represents a specific application of the software, i.e., start-up mode, low power mode. A Mode might contains *Circuit*(s) and *Assembly*(ies). A Circuit is the lowest-level hierarchical element which encapsulates basic functions. It is composed by an *Interface*, which collects its data and triggering ports (Section 3.2 and Section 3.3), and one, or more, *Behavior*(s). A Circuit has the run-to-completion semantic, which means that, upon triggering, it reads data from the input ports, executes its behavior and writes data on the output ports. Its attribute *activeBehavior* specifies which Behavior, among those defined, is active. A Behavior represents the code to be executed. It has one attribute, *entry*, and it is composed by one or more *RunTimeProfile*(s), which define the Behavior execution and run-time properties for a specific platform. Interface might be composed by several *State*(s). A State is used for preserving data among the different executions of a behavior. A State has two attributes: *initialValue*, and *dimension*. An Assembly is used for grouping different Circuits or Assemblies; it does not add any semantic.

3.2 Data elements

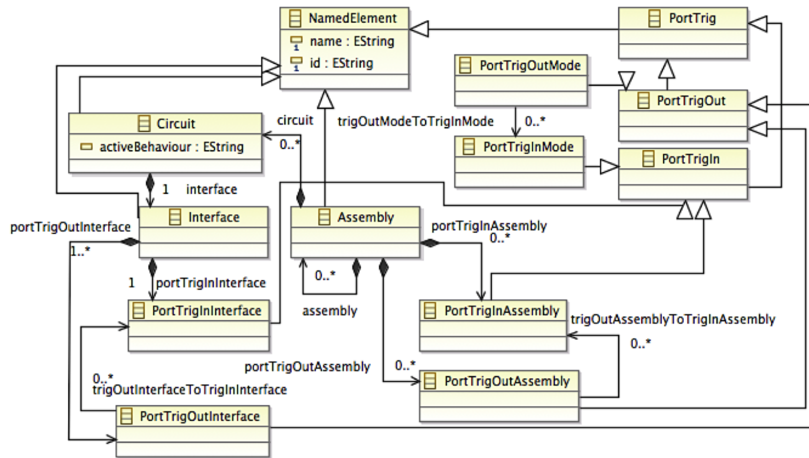
As aforesaid, RCM explicitly separates the data and the control flow. Figure 3(a) shows a metamodel fragment containing the architectural elements for modeling the data flow. *PortData* models a generic data port. Data ports are used for modeling data communication among Circuits or Assemblies. PortData is an abstract element. *PortDataIn* and *PortDataOut* specialize PortData and represent input and output data port, respectively. PortDataOut has a on-to-many relationship with PortDataIn, *dataOutToDataIn*, meaning that a value on the data output port can be fed to several input ports.

3.3 Triggering elements

Figure 3(b) shows a metamodel fragment containing the architectural elements for modeling the control flow. *PortTrig* models a generic trigger port. *PortTrigIn* and *PortTrigOut* specialize PortTrig; they represent trigger input and output ports, respectively. Trigger ports are used for specifying precedence and control over the architectural elements. A trigger output port generates trigger signal upon the completion of the related Circuit. When receiving a trigger signal, a trigger input port triggers the execution of the related Circuit's Behavior. Any trigger signal after the first received is meaningless, therefore ignored. PortTrig, PortTrigIn and PortTrigOut are abstract. PortTrigIn is specialized by *PortTrigInMode*, *PortTrigInAssembly* and *PortTrigInInterface*. Similarly, PortTrigOut is specialized by *PortTrigOutMode*, *PortTrigOutAssembly* and *PortTrigOutInterface*. A Mode is composed by, at least, one PortTrigInMode and one PortTrigOutMode; PortTrigOutMode has a one-to-many relationship with PortTrigInMode,



(a) Metamodel fragment for the data flow objects



(b) Metamodel fragment for the control flow objects

Fig. 3. RCM metamodel fragments

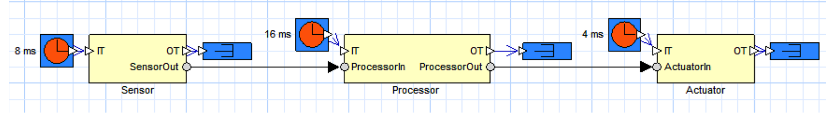
meaning that a trigger output port can trigger more than one trigger input port. Similarly, Interface is composed by exactly one PortTrigInInterface and, at least, one PortTrigOutInterface. Also, PortTrigOutInterface has a one-to-many relationship with PortTrigInInterface. Finally, an Assembly might contain PortTrigInAssembly and PortTrigOutAssembly, where a PortTrigOutAssembly has a one-to-many relationship with PortTrigInAssembly.

4 RCM2AUTOSAR transformation

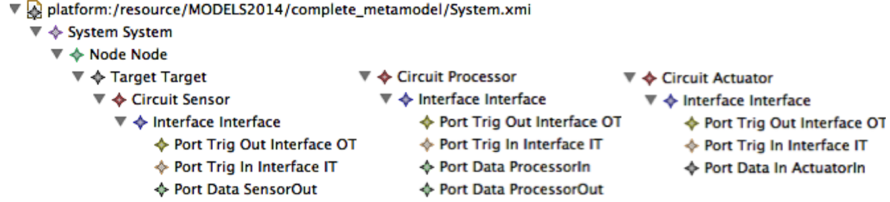
In this section, we describe the RCM2AUTOSAR transformation. Figure 4(a) depicts a RCM model of a single node real time system composed by three Circuits, *Sensor*, *Processor* and *Actuator*.

Sensor has one trigger input port, *IT*, one trigger output port, *OT*, and one data output port, *SensorOut*. Similarly, Processor has two trigger ports and two data ports, and Actuator has two trigger ports and a data port. Intuitively, the

model describes a vehicle function in which data are sensed, processed and fed to the actuator for the stimulation [3].



(a) RCM model of a single-node real time system



(b) RCM model serialization

Fig. 4. RCM model and its serialization

Although trivial, the transformation is used for miming typical scenarios in the software development of vehicle embedded systems. With models as that depicted in Figure 4(a), manual translations might still appear feasible; nevertheless, in reality, vehicle embedded systems are composed by over 70 embedded systems and thousands components [14]. Also, the transformation is used for proving the validity of the metamodel introduced in the Section 3. Figure 4(b) shows the textual serialization of the model.

Algorithm 1 shows the metacode for the RCM2AUTOSAR transformation. The algorithm mainly consists of two relationships between RCM and AUTOSAR elements, which are: Circuit to Software Component, and PortData to PortClientServer [4]. The former relationship exploits a naming convention for better translating the elements avoiding flattening the RCM model. The two involved metamodels do not have the same expressiveness, which means that the underneath relationship is partial. Indeed the are some elements of RCM which are ignored from the transformation. Figure 5 shows the serialization of the AUTOSAR model obtained as a result of the transformation.

5 Related Works

The embedded system research community and the vehicular industry have focused more and more on the definition of component-based technologies for embedded vehicular systems. Hereafter, we present and discuss some attempts targeted towards the development of resource-constrained vehicular real-time systems.

[3] The model contains triggering elements not presented in this work, i.e., clock and trigger terminator elements. The reader can assume they are responsible for triggering the circuits and terminating the control flow, respectively.

[4] The explanation of the AUTOSAR metamodel is outside the foucs of this work. The interested reader may refer to the [1].

Algorithm 1 RCM2AUTOSAR transformation

```
1: new VirtualFunctionBus VFB;
2: for each Circuit c in a Target t do
3:   switch c.name do
4:     case (1) //c.name ends in Sensor
5:       new SensorSoftwareComponent sc;
6:       sc.name = c.name;
7:     case (2) //c.name ends in Actuator
8:       new ActuatorSoftwareComponent sc;
9:       sc.name = c.name;
10:    case (default)
11:      new SoftwareComponent sc;
12:      sc.name = c.name;
13:  for each Interface i in c do
14:    for each PortDataIn di in i do
15:      new RequiredPortClientServer rp;
16:      rp.name = di.name;
17:    end for
18:    for each PortDataOut do in i do
19:      new ProvidedPortClientServer pp;
20:      pp.name = do.name;
21:      pp.receiver = do.dataOutToDataIn;
22:    end for
23:  end for
24: end for
```

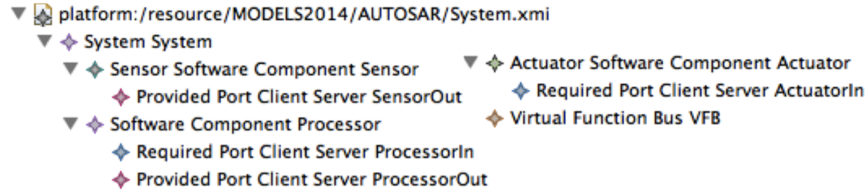


Fig. 5. Serialization of the obtained AUTOSAR model

5.1 EAST-ADL/AUTOSAR

AUTOSAR [1] is an industrial initiative to provide standardized software architecture for the development of embedded software for the vehicular domain. Within AUTOSAR, the software architecture is defined in terms of Software Components (SWCs) and Virtual Function Bus (VFB). VFB handles the virtual integration and communication among SWCs, hiding the low-level implementation details. Compared with RCM, EAST-ADL/AUTOSAR describes the software at a higher level of abstraction. It has no ability to specify and handle timing information at design time, such as component worst case execution time. AUTOSAR does not distinguish between data and control flow, as well as between inter and intra node communication. Contrariwise, RCM was specifically designed taking timing requirements into account. As shown in Section 3, RCM

clearly separates data and control flow; also, it has been recently extended with special network interface components for modeling inter-node communication [2]. The AUTOSAR sender receiver communication mechanism is very similar to the RCM pipe-and-filter communication mechanism. In short, AUTOSAR focuses on hiding the information which RCM highlights.

5.2 TIMMO/TIMMO-2-USE

TIMMO [3] is a large EU research project, which aims to provide AUTOSAR with a timing model [4]. To this end, it provides a predictable methodology and language TADL [5] for expressing timing requirements and constraints. TADL is inspired by MARTE [6], which is an UML profile for model-driven development of real-time and embedded systems. The TIMMO predictable methodology makes use of the EAST-ADL and AUTOSAR interplay, where the former is used for the software structural modeling, while the latter is used for the implementation. Although the TIMMO project has been evaluated upon prototype validators, from the best of our knowledge, there is no concrete industrial implementation of the TIMMO project. TIMMO-2-USE [7] follows-up on the TIMMO project. It presents a major redefinition of TADL and new functionality for supporting the AUTOSAR extensions regarding timing model. Arcticus Systems has been involved in TIMMO-2-USE project as one of the industrial partners. Both TIMMO and TIMMO-2-USE attempt to annotate AUTOSAR with a timing model. This may be hard to accomplish as AUTOSAR aims at hiding implementation details of execution environment and communication using the Virtual Function Bus, as shown in Section 4. That is, at the modeling level, there is no information in AUTOSAR to express low level details, e.g., linking information, which is necessary to extract the timing model from the software architecture. There is no focus in these initiatives on how to extract this information from the model or perform timing analysis or synthesize the run-time framework.

5.3 ProCom

ProCom [8] is a two-layered component model for the development of distributed embedded systems. It is the result of a research project conducted at Mälardalen University. The upper layer, ProSys, models the system and concurrent subsystems communicating by means of asynchronous messages. The lower layer, ProSave, models each subsystem in terms of functional components implemented as a piece of code. Being inspired by RCM, ProCom presents several similarities with it. Both CMs have passive components, clearly separate the control flow from the data flow and use the pipe-and-filter communication mechanism for components interconnection. However, ProCom does not differentiate between intra- and inter-node communication which is unlike RCM. As for AUTOSAR, also ProCom hides communication details, making hard the extraction of timing model and the execution of timing analysis [12].

6 Conclusions and Future Works

In the last decades, CBSE has enhanced the software development for vehicular embedded systems. Nevertheless, industry needs to move further towards a

seamless development chain for reducing software development costs and time-to-market. In this respect, one of the major challenge is the definition of a methodology and accompanying technologies. In this work we proposed the adoption of a methodology exploiting the crossplay of MDE and CBSE and took initial steps towards the realization of the aforesaid seamless chain. We i) motivated the usage of RCM within the vehicular domain, by highlighting its unique features against existing CMs, ii) formalized a metamodel based on RCM and ii) proved the metamodel validity by means of the RCM2AUTOSAR model transformation. The formalization of the metamodel not only serves as base for embracing the MDE vision, but it also aims in restoring the separation of concerns which has been lost during the evolution of the RCM. For sake of space we omitted a comparison between RCM and its metamodel. As future works, we will investigate further metamodel refinements targeting the enhancement of vehicular tool chaining while preserving the current expressive power. The RCM2AUTOSAR transformation outlines the potential benefits gained in having a proper metamodel for RCM, in terms of automation, interoperability and traceability. As future investigation direction we will also, together with our industrial partners, cover the identification of additional languages used along the software development for the vehicular embedded systems, with the aim of formalizing their metamodels and hence enable model transformations for supporting a more extensive tool chain.

Acknowledgment

The work in this paper is supported by the Swedish Knowledge Foundation (KKS) within the project FEMMVA and Swedish Research Council (VR) within the project SynthSoft. We thank our industrial partners Arcticus Systems and Volvo Construction Equipment, Sweden.

References

1. AUTOSAR Techincal Overview http://autosar.org
2. Mubeen, S., Mäki-Turja, J., Sjödin, M.: Communications-Oriented Development of Component-Based Vehicular Distributed Real-Time Embedded Systems. Journal of Systems Architecture. vol. 60, pp. 207-220 (2014)
3. TIMMO Methodology. DELIVERABLE 7 (2009)
4. Mastering Timing Information for Advanced Automotive Systems Engineering (2012)
5. TADL: Timing Augmented Description Language. Deliverable 6 (2009)
6. The UML Profile for MARTE: Modeling and Analysis of Real-Time and Embedded Systems (2010)
7. TIMMO-2-USE http://www.timmo-2-use.org/
8. Sentilles, S., Vulgarakis, A., Bures, T., Carlson, J., Crnkovic, I.: A Component Model for Control-Intensive Distributed Embedded Systems. In: 11th International Symposium on Component Based Software Engineering, pp. 310-317 (2008)
9. EAST-ADL Domain Model Specification. V 2.1.12 (2013)
10. Rubus models, methods and tools http://www.arcticus-systems.com
11. Hänninen, K., et.al.: The Rubus Component Model for Resource Constrained Real-Time Systems. In: 3rd IEEE International Symposium on Industrial Embedded Systems (2008)

12. Mubeen, S., Mäki-Turja, J., Sjödin, M.:Support for End-to-End Response-Time and Delay Analysis in the Industrial Tool Suite: Issues, Experiences and a Case Study. In: Computer Science and Information Systems, vol. 10, no. 1, pp. 453-482 (2013)
13. Bohlin, M., Hänninen, K., Mäki-Turja, J., Carlson, J., Sjödin, M.: Bounding Shared-Stack Usage in Systems with Offsets and Precedences. In: 20th Euromicro Conference on Real-Time Systems (2008).
14. Charette, R. N.: This Car Runs On Code. In: *Spectrum, IEE*, vol. 46, no. 2 (2009)
15. Crnkovic, I.: Component-Based Software Engineering for Embedded Systems. In: 27th International Conference on Software Engineering, pp. 712-713 (2005)
16. Bézivin, J.: On the unification power of models. Software & Systems Modeling, vol. 4, no. 2, pp. 171-188 (2005)
17. Kent, S.: Model Driven Engineering. Lecture Notes in Computer Science. vol. 2335, pp.286-298 (2002)

Translating Timing Constraints during Vehicular Distributed Embedded Systems Development

Saad Mubeen*†, Mikael Sjödin*, and Jukka Mäki-Turja*†

*Mälardalen Real-Time Research Centre (MRTC), Mälardalen University, Sweden
†Arcticus Systems AB, Järfälla, Sweden
{saad.mubeen, mikael.sjodin, jukka.maki-turja}@mdh.se

Abstract. The end-to-end response-time and delay analysis can verify timing requirements specified on vehicular distributed embedded systems without performing exhaustive testing. For this purpose, the timing requirements and constraints should be unambiguously translated among several models, methodologies and tools that are used at various abstraction levels and phases during the industrial development of these systems. Within this context, we translate timing constraints that are specified at higher abstraction levels using the Timing Augmented Description Language (TADL2) to an industrial model the Rubus Component Model (RCM). We also discuss corresponding extensions in RCM and perform a case study to validate our approach.

Keywords: Distributed embedded systems; component-based development; end-to-end timing analysis.

1 Introduction

With the recent advancement in computer controlled functionality, the software has immensely increased in size and complexity in vehicular embedded systems. For example, the embedded software in a modern premium car may consist of as many as 100 million lines of code that may occupy up to 1 GB of memory. This software may be realized by more than 2000 software functions. Moreover, it may be distributed over 100 Electronic Control Units (ECUs) [1, 2]. The model- and component-based development appears as a promising approach to deal with the complexity of these systems [3, 4]. This approach uses the principles of Model-Based Software Engineering (MBSE) and Component-Based Software Engineering (CBSE). It supports the use of models to describe functions, structures and other design artifacts. It also supports the development of large software systems by reuse and integration of software components and their architectures. Hence, it raises the level of abstraction for software development. The model- and component-based development of software architectures for vehicular embedded real-time systems has had a surge the last few years [5–8].

1.1 Objective and Paper Contribution

There are a number of models, methodologies and tools that are used at various abstraction levels[1] and phases during the development of vehicular distributed

[1] An abstraction level provides a complete definition of the system for a given purpose.

embedded systems. Intuitively, timing requirements and constraints can be specified using one modeling technology, whereas the detailed end-to-end timing analysis can be performed using the tools accompanying another. The end-to-end response-time and delay analyses [9–11] is one of the predominant techniques used in the industry to provide guarantees that the distributed embedded system is going to meet its deadlines when executed. In order to support the analysis, the timing information should be unambiguously translated among several modeling approaches, languages and tools.

Within this context, we consider TIMMO methodology [12] at higher abstraction levels and an existing industrial model the Rubus Component Model (RCM) [13] at the lower level. The TIMMO methodology makes use of EAST-ADL [14] for modeling of software architecture and Timing Augmented Description Language (TADL2) [15] for annotation of timing constraints. TADL2 is intended to provide AUTOSAR with a timing model. At the lower abstraction level, we consider modeling and timing analysis support of RCM and accompanying tool suite Rubus-ICE [16]. We provide translation of those timing constraints, from TADL2 to RCM, that impose restrictions on end-to-end delays. We discuss corresponding extensions in RCM to carry out these translations. We also discuss the semantics and viewpoint of analysis engines about these constraints. In order to provide a proof of concept, we conduct a case study.

We choose RCM instead of AUTOSAR at the lower abstraction level because AUTOSAR lacks a complete timing model, e.g., control flow is not specified unambiguously. This work is first step towards a bigger goal, i.e., development of a seamless tool-chain for model-based development of vehicular real-time systems; and support for inter-operating various modeling and analysis tools [8].

2 Background and Related Works

There are several frameworks that support timing modeling such as AADL [17], SCADE [18], MARTE [19], MAST [20], SysML, CHESS [21, 22]. However, the focus in the vehicle industry today is on EAST-ADL and AUTOSAR; RCM is also being used. In this work, we focus only on the vehicular domain.

2.1 Abstraction Levels Considered by Various Methodologies

Various models and methodologies used for the development of vehicular distributed embedded systems [14, 15, 5, 23] consider four abstraction levels shown in Fig. 1. At the vehicle level, features and requirements on the end-to-end functionality of the vehicle are captured in an informal and solution-independent way. At the analysis level, the requirements are formally captured in allocation-independent way. Functionality of the system is defined based on the requirements and features without implementation details. A high-level analysis may also be performed for functional verification. The artifacts at this level are developed in an implementation-independent way. These artifacts also contain middleware abstraction, hardware architecture and software functions to hardware allocation. The implementation level contains software-based implementation of the system functionality. The EAST-ADL methodology defines a system at this level in terms of AUTOSAR elements. However, in this work, our focus is on using RCM. Hence, the artifact at this level consists of software architecture of the system defined in terms of Rubus components and their interactions. In this work, we focus on the design and implementation levels.

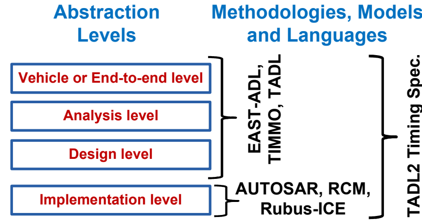


Fig. 1. Abstraction levels considered during the development

2.2 Models and Methodologies in the Vehicular Domain

Rubus Component Model (RCM) and Rubus-ICE Rubus [24] is a collection of methods and tools for model- and component-based development of dependable embedded real-time systems. It is developed by Arcticus Systems [2] in close collaboration with several industrial partners. Rubus is today mainly used for the development of control functionality in vehicles by several international companies, e.g., BAE Systems, Volvo Construction Equipment, Knorr-bremse, Mecel and Hoerbiger. The Rubus concept is based around RCM and its development environment Rubus-ICE which includes modeling tools, code generators, analysis tools and run-time infrastructure. The overall goal of Rubus is to be aggressively resource efficient and to provide means for developing predictable, timing analyzable and synthesizable control functions in resource-constrained embedded systems. The timing analysis supported by Rubus-ICE includes distributed end-to-end response-time and delay analyses [10]. Rubus methods and tools mostly focus at the implementation level in Fig. 1. The lowest-level hierarchical component in RCM is called Software Circuit (SWC). Its purpose is to encapsulate basic functions. Fig. 2 shows an example of a software architecture in RCM composed of SWCs; interconnections between SWCs; and their interactions with external events and actuators with regard to data and triggering.

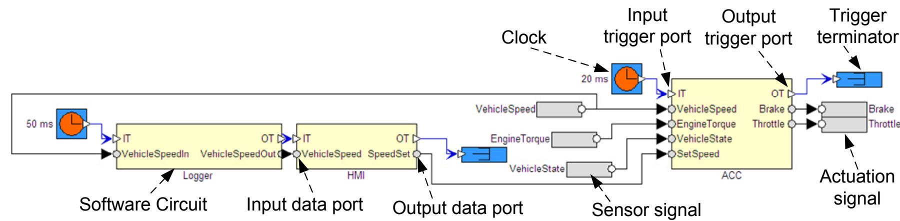


Fig. 2. Example of software architecture of a system modeled in RCM.

AUTOSAR AUTOSAR [25] is an industrial initiative to provide standardized software architecture for the development of embedded software. It is used at the implementation level in Fig. 1. It describes software development at a higher level of abstraction compared to RCM. The timing model in AUTOSAR is introduced fairly recently compared to RCM. AUTOSAR is more focussed on the functional and structural abstractions, hiding the implementation details about execution and communication. AUTOSAR hides the details that RCM highlights.

[2] http://www.arcticus-systems.com

EAST-ADL, MARTE, TIMMO, TIMMO-2-USE, TADL and TADL2 TIMMO [6] is an initiative to provide AUTOSAR with a timing model [26]. It is based on a methodology and TADL [23] language which expresses timing requirements and constraints. It is inspired by MARTE [19] which is a UML profile for model-driven development of real-time and embedded systems. TIMMO methodology uses EAST-ADL language [14] for structural modeling and AUTOSAR for the implementation. TIMMO and EAST-ADL focus on the top three levels in Fig. 1. In TIMMO-2-USE project [7], a major redefinition of TADL is done and released in TADL2 specification [15]. TADL2 can specify timing information at all the abstraction levels. Most of these initiatives lack the support on expressing low-level details at the higher levels such as linking information in distributed chains. These details are necessary to extract the end-to-end timing model from the architecture. Furthermore, there is no support on how to extract this information from the model or perform timing analysis. In our view, the end-to-end timing model includes enough information from the systems to be able to perform certain type of timing analysis, e.g., end-to-end response-time analysis.

Previous Works In [27], we presented a method to automatically extract the end-to-end timing models only at the implementation level. In [28], we extended our previous method to raise the models extraction at the design level[3]. In [29], we discussed the basic idea for translation of timing constraints.

3 Interpretation of TADL2 Timing Constraints in RCM

Timing requirements in TADL2 are modeled by means of timing constraints specified on events and event chains. We discuss those timing constraints that impose restrictions on the end-to-end delays, i.e., reaction and age constraints in Subsections 3.1 and 3.2 respectively. In each subsection, first we discuss semantics of the constraint according to TADL2 specification [15]. Then we provide its translation in RCM and propose corresponding extensions in RCM. Finally, we discuss it with the view point of analysis engines.

Definitions and Notations An event represents a distinct form of state change in a running system. It is used to trigger an analysis- or design-level function. When the function is triggered, input data is consumed followed by its processing, transformation and finally production at the output. An event takes place at distinct points in time which are called its occurrences. The occurrences of an event are denoted, e.g., by attributes s and r. These attributes are basically time points showing when instances of the event occur. They can be added, subtracted and compared with each other. A constraint often puts limits on the occurrences of events. These limits can be specified in terms of time distances using *minimum* and *maximum* attributes. The occurrences of the events are required to lie within these limits. We use object-oriented notation to define the attributes of a constraint, e.g., TC.response refers to the response event on which the timing constraint TC is specified. In multi-rate systems (see Fig. 3 and 4), components in an event chain can be triggered with independent clocks. Hence, there can be multiple response occurrences to a single occurrence of stimulus

[3] This is an unpublished work and is provided as a technical report for referencing.

in an event chain. In these chains, multiple response occurrences due to each consecutive stimulus occurrence are differentiated by means of colors.

3.1 Reaction Constraint

TADL2 Description It constrains the occurrence of a response event after the occurrence of a corresponding stimulus event in an event chain. It specifies "how long after the occurrence of a stimulus a corresponding response must occur" [15]. Both reaction and age constraints differ from the delay constraint in a way that they can only be applied to event chains and not to individual events. In order to satisfy the reaction constraint, the earliest occurrence of the response with same color as that of stimulus must take place within the limits specified by this constraint as shown in Fig. 3(a).

Semantics A system behavior satisfies the specified Reaction constraint denoted by ReaC if and only if for each occurrence s of the event ReaC.stimulus, there is an occurrence r of the event ReaC.response such that

$$(\text{r.color} = \text{s.color}) \text{ and } (\text{r is minimal in ReaC.response with that color})$$
$$\text{and}$$
$$(\text{minimum} \leq (\text{r- s}) \leq \text{maximum})$$

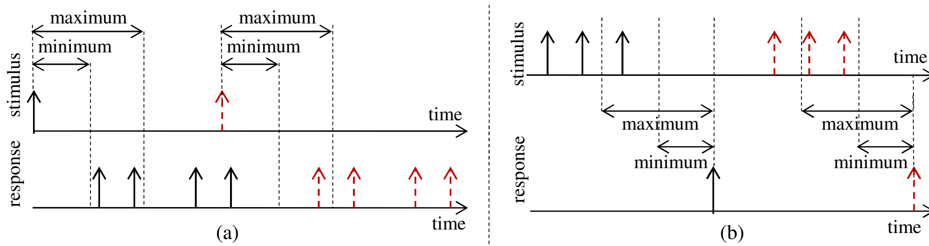


Fig. 3. Graphical illustration of (a) Reaction constraint, (b) Age constraint

Interpretation in RCM We introduce a new modeling entity in RCM denoted by DataReaction (DR for short) to specify the reaction constraint. This constraint can be specified on an event chain, event chain segment and distributed event chain (distributed over more than one node) by means of DR Start and DR End objects as shown in Fig. 4. The DR End object supports the specification of maximum" attribute by means of a deadline parameter associated to it. However, the minimum parameter is zero. In order to be consistent with TADL2 Reaction constraint, we propose to associate a parameter with DR End object to specify a non-zero minimum value of the constraint.

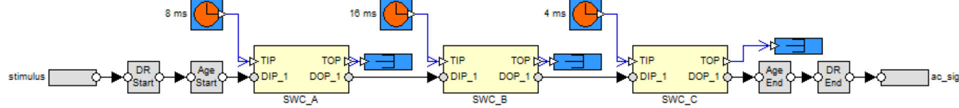


Fig. 4. Modeling objects in RCM to specify Reaction and Age constraint

Interpretation by the Analysis Engines The analysis engines provided by Rubus-ICE support calculations for the corresponding Reaction delay. Consider the example of an event chain in a multi-rate system in Fig. 4. Fig. 5 shows a time line depicting the execution of this chain (assuming each SWC corresponds to a task denoted by τ at run-time). τ_B is deliberately given an offset of 15 time units to maximize the delays. This delay is equal to the time elapsed between the previous non-overwritten release of task τ_A (input of the chain) and first response of task τ_C (output of the chain) corresponding to the current non-overwritten release of task τ_A. Assume that a new value of the input is available in the input buffer of task τ_A "just after" the release of the second instance of task τ_A (at time $8ms$). Hence, the second instance of task τ_A "just misses" the read of the new value from its input buffer. This new value has to wait for the next instance of task τ_A to travel towards the output of the chain. Therefore, the new value is read by the third and forth instances of task τ_A. The first output corresponding to the new value (arriving just after $8ms$) appears at the output of the chain at $34ms$. This results in the delay of $26ms$ as shown in Fig. 5. This phenomenon is more obvious in the case of distributed embedded systems where a task in the receiving node may just miss to read fresh signals from a message that is received from the network. The analysis engines calculate the Reaction delay as shown in Fig. 5 and compare it with the specified constraint parameters. We refer the reader to [10] for further details about the analysis.

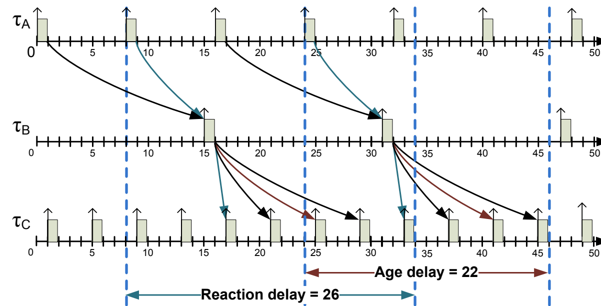


Fig. 5. Demonstration of Reaction and Age delay calculations by analysis engines

3.2 Age Constraint

TADL2 Description It constrains the occurrence of a stimulus from the occurrence of corresponding response looking back through the event chain. Basically it specifies "how long before each response a corresponding stimulus must have occurred" [15]. In order to satisfy this constraint, the latest occurrence of the stimulus with same color as that of the response must lie within the limits specified by this constraint as shown in Fig. 3(b).

Semantics A system behavior satisfies the specified Age constraint denoted by AgeC if and only if for each occurrence r of the event AgeC.response, there is an occurrence s of the event AgeC.stimulus such that

$$\text{(s.color = r.color) and (s is maximal in AgeC.stimulus with that color)}$$
$$\text{and}$$
$$\text{(minimum} \leq \text{(r- s)} \leq \text{maximum)}$$

Interpretation in RCM We introduce a new modeling entity in RCM denoted by DataAge. This constraint can be specified on an event chain, event chain segment and distributed event chain by means of Age Start and Age End objects as shown in Fig. 4. The Age End object supports the specification of "maximum" attribute by means of a deadline parameter associated to it. In order to be consistent with TADL2 Age constraint, we propose to associate a parameter with Age End object to specify non-zero minimum value of the constraint.

Interpretation by the Analysis Engines The analysis engines support the calculations for the corresponding Age delay. Consider the example of an event chain in a multi-rate system shown in Fig. 4. Fig. 5 shows the time line when this chain is executed. The analysis engines calculate the Age delay as shown in Fig. 5 and compare it with the specified constraint parameters.

4 Automotive-application case study

In order to show the applicability of our approach, we conduct the Steer-By-Wire (SBW) system case study. It is an automotive feature that substitutes most of the mechanical and hydraulic components with electronic components in the steering system of a vehicle. A partial architecture of the SBW system is shown in Fig. 6(a). There are two ECUs (rest of the ECUs are not shown for simplicity) that are connected to a single CAN network. The Steering Control (SC) ECU receives inputs from steering angle, steering torque and vehicle speed sensors. It also receives a CAN message from the Wheel Control (WC) ECU. It sends two CAN messages: one caries steer angle and torque signals; while the other carries feedback signals. Based on all the inputs, it calculates the feedback steering torque and sends it to the feedback torque actuator which is responsible for producing the turning effect of the steering. Similarly, the WC ECU receives inputs from wheel angle and torque sensors. Depending upon these signals and CAN messages received from the SC ECU, it calculates the wheel torque and produces actuation signals for the wheel actuators. It also sends one CAN message carrying wheel torque signal.

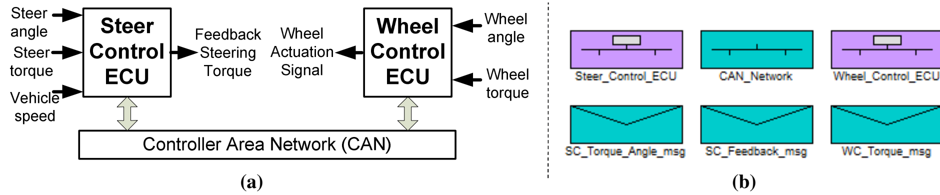


Fig. 6. Partial architecture of the steer-by-wire system

The internal software architecture of the two ECUs is modeled with EAST-ADL using EAST-ADLrubusDesigner[4] as shown in Fig. 7. There are two timing constraints namely age and reaction that are specified using TADL2. They put a restriction of 50 ms on the time between the production of steer angle and torque by Steer Controller component and their consumption by Wheel Controller component. These constraints are internally referenced to the components on which they are specified. For convenience, the start and end points for these constraints are identified using the solid-line arrow as shown in Fig. 7.

[4] http://www.arcticus-systems.com

The top level model of the SBW system consisting of models of the two ECUs, a CAN bus and CAN messages in RCM is shown in Fig. 6(b). Whereas, the internal software component architectures of SC and WC ECUs translated from the EAST-ADL model in Fig. 7 to RCM are shown in Fig. 8 and Fig. 9 respectively. The clocks corresponding to trigger flows are translated from the periodic constraints in TADL2 that are specified on the software components in Fig. 7. We use a one-to-one mapping between the software component (both functional as well as sensor and actuator components) in EAST-ADL and software circuit in RCM. The specified TADL2 timing constraints in Fig. 7 are also translated to RCM timing constraints as shown in Fig. 8 and Fig. 9.

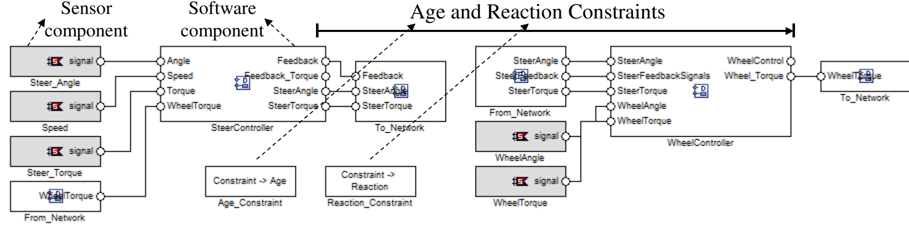


Fig. 7. Software architecture of SBW system modeled with EAST-ADL and TADL2

The run-time allocation of the SBW system results in three distributed chains (distributed over more than one node) and thirteen tasks. The analysis engines calculate the age and reaction delays for the distributed chains on which the timing constraints are specified. The calculated age and reaction delays are 30320 μs and 40320 μs respectively. A comparison between the specified constraints and calculated delays shows the system satisfies the specified timing constraints.

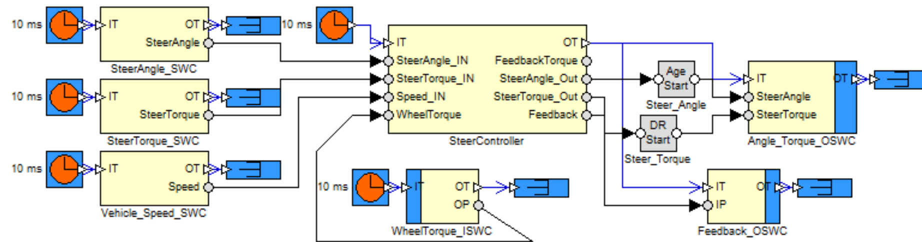


Fig. 8. Translated software architecture of SC sub-system in RCM

5 Conclusion

We discussed translation of timing constraints that are specified at higher abstraction levels using the TADL2 language to an existing industrial component model RCM. These translations along with our analysis engines enable the application of end-to-end response-time and delay analysis at a higher level of abstraction and early phases during the development of vehicular distributed embedded systems. In order to show the applicability of our approach, we conducted an automotive-application case study. We modeled the software architecture using

EAST-ADL and specified the age and reaction time constraints using TADL2. The software architecture and specified timing constraints were translated into the corresponding models in RCM. The analysis engines automatically analyzed end-to-end delays for the chains on which timing constraints were specified. Currently the translations of timing constraints and corresponding timing analysis is done automatically, however, the conversion of software architecture from the design model (using EAST-ADL) to the implementation model (using RCM) is done manually. In the future, we plan to support automatic conversion of the design-level models to the implementation-level models. Moreover, we plan to implement and validate automatic translations of other timing constraints from TADL2 to RCM including synchronization, repetition and pattern constraints.

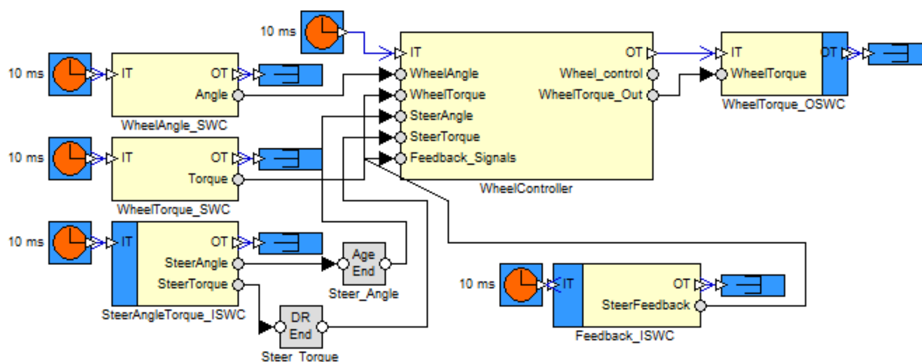


Fig. 9. Translated software architecture of WC sub-system in RCM

Acknowledgement

This work is supported by the Swedish Knowledge Foundation (KKS) and Swedish Research Council (VR) within the project FEMMVA and SynthSoft. We thank our industrial partners Arcticus Systems and Volvo CE, Sweden.

References

1. R. N. Charette, "This Car Runs on Code," *Spectrum, IEEE*, vol. 46, no. 2, 2009, http://spectrum.ieee.org/green-tech/advanced-cars/this-car-runs-on-code.
2. M. Broy, I. Kruger, A. Pretschner, and C. Salzmann, "Engineering automotive software," *Proceedings of the IEEE*, vol. 95, no. 2, pp. 356 –373, feb. 2007.
3. T. A. Henzinger and J. Sifakis, "The Embedded Systems Design Challenge," in *Proceedings of the 14th International Symposium on Formal Methods (FM), Lecture Notes in Computer Science*. Springer, 2006, pp. 1–15.
4. I. Crnkovic and M. Larsson, *Building Reliable Component-Based Software Systems*. Norwood, MA, USA: Artech House, Inc., 2002.
5. "AUTOSAR Techincal Overview, Release 4.1, Rev. 2, Ver. 1.1.0., The AUTOSAR Consortium, Oct., 2013," http://autosar.org.
6. "TIMMO Methodology, Ver. 2," *TIMMO (TIMing MOdel), Deliverable 7*, Oct. 2009, The TIMMO Consortium.
7. "TIMMO-2-USE," http://www.timmo-2-use.org/.

8. CRYSTAL - CRitical sYSTem engineering AcceLeration, http://www.crystal-artemis.eu, accessed May, 2014.
9. K. Tindell and J. Clark, "Holistic schedulability analysis for distributed hard real-time systems," *Microprocess. Microprogram.*, vol. 40, pp. 117–134, Apr. 1994.
10. S. Mubeen, J. Mäki-Turja, and M. Sjödin, "Support for end-to-end response-time and delay analysis in the industrial tool suite: Issues, experiences and a case study," *Computer Science and Information Systems, ISSN: 1361-1384*, vol. 10, no. 1, 2013.
11. R. Henia, A. Hamann, M. Jersak, R. Racu, K. Richter, and R. Ernst, "System level performance analysis - the symta/s approach," *Computers and Digital Techniques*, vol. 152, no. 2, pp. 148–166, March 2005.
12. TIMMO-2-USE Methodology Description, Ver. 2, Del. 13, Jul., 2012.
13. K. Hänninen et.al., "The Rubus Component Model for Resource Constrained Real-Time Systems," in *3rd IEEE International Symposium on Industrial Embedded Systems*, Jun. 2008.
14. "EAST-ADL Domain Model Specification, Deliverable D4.1.1," http://www.atesst.org/home/liblocal/docs/ATESST2_D4.1.1_EAST-ADL2-Specification_2010-06-02.pdf.
15. Timing Augmented Description Language (TADL2) syntax, semantics, metamodel Ver. 2, Deliverable 11, Aug. 2012.
16. "Rubus ICE-Integrated Development Environment," http://www.arcticus-systems.com.
17. P. Feiler, B. Lewis, S. Vestal, and E. Colbert, "An Overview of the SAE Architecture Analysis & Design Language (AADL) Standard: A Basis for Model-Based Architecture-Driven Embedded Systems Engineering," in *Architecture Description Languages*, ser. The International Federation for Information Processing (IFIP). Springer US, 2005, vol. 176, pp. 3–15.
18. SCADE Suite, http://www.esterel-technologies.com/products/scade-suite, accessed May, 2014.
19. "The UML Profile for MARTE: Modeling and Analysis of Real-Time and Embedded Systems," Jan. 2010. [Online]. Available: http://www.omgmarte.org/
20. "MAST–Modeling and Analysis Suite for Real-Time Applications," http://mast.unican.es/.
21. CHESS Project, CHESS consortium. Avialable at: http://www.chess-project.org, accessed May, 2014.
22. A. Cicchetti, F. Ciccozzi, S. Mazzini, S. Puri, M. Panunzio, T. Vardanega, and A. Zovi, "Chess: a model-driven engineering tool environment for aiding the development of complex industrial systems," in *27th International Conference on Automated Software Engineering (ASE 2012)*, Sep. 2012.
23. TADL: Timing Augmented Description Language, Ver. 2, Deliverable 6, Oct., 2009.
24. "Rubus models, methods and tools," http://www.arcticus-systems.com.
25. "AUTOSAR Techincal Overview, Version 2.2.2. AUTOSAR – AUTomotive Open System ARchitecture, Release 3.1, The AUTOSAR Consortium, Aug., 2008," http://autosar.org.
26. Mastering Timing Information for Advanced Automotive Systems Engineering. In the TIMMO-2-USE Brochure, 2012. Available at: http://www.timmo-2-use.org/pdf/T2UBrochure.pdf.
27. S. Mubeen, J. Mäki-Turja, and M. Sjödin, "Communications-Oriented Development of Component- Based Vehicular Distributed Real-Time Embedded Systems," *Journal of Systems Architecture*, 2013.
28. S. Mubeen, J. Mäki-Turja, and M. Sjödin, "Component-based vehicular distributed embedded systems: End-to-end timing models extraction at various abstraction levels," Tech. Rep., May 2014. [Online]. Available: http://www.es.mdh.se/publications/3545-
29. S. Mubeen, J. Mäki-Turja, and M. Sjödin, "Towards Translation of Timing Constraints during Vehicular Embedded Systems Development," in *CompArch Work-in-progress Session*, July 2014.

Synthesizing an Automata-based Representation of BPMN2 Choreography Diagrams*

Marco Autili, Davide Di Ruscio, Amleto Di Salle, and Paola Inverardi

Università degli Studi di L'Aquila, Italy
{marco.autili,davide.diruscio,amleto.disalle,paola.inverardi}@univaq.it

Abstract. Choreographies are an emergent Service Engineering approach to compose together and coordinate distributed services. They represent a global specification of the interactions between the participant services. BPMN2 provides a dedicated notation, called Choreography Diagrams, to define choreographies. This paper presents a model transformation to automatically transform a BPMN2 choreography specification into an automata-based representation called Choreography LTS (CLTS). The latter is a LTS suitably extended to, on one side model the complex interactions that can be specified by choreography diagrams, on the other provide modelers with a means to precisely extract the not-easy-to-grasp coordination logic "hidden" into BPMN2 Choreography Diagrams. Dedicated Eclipse plugins, within the CHOReO*Synt* tool, have been developed to support the presented transformation.

1 Introduction

Choreographies are an emergent Service Engineering approach to compose together and coordinate distributed services. They describe the interactions between the participant services by specifying the way business participants coordinate their interactions from a global perspective. The OMG BPMN2 [18] Choreography Diagrams are the standard de facto for specifying service choreographies by providing powerful constructs to specify complex interactions where message exchanges between participants go far beyond simple request-response interactions that follow a sequential flow. Choreography diagrams permit to specify inclusive and exclusive conditional branches, parallel branches to be joined at later execution points, looping tasks, and so on.

BPMN2 specifications can be very complex and the standard specification introduces constraints that a choreography designer shall obey to achieve well-formed choreography specifications. Unfortunately, the standard only provides a textual description for these constraints, hence making their correct understanding difficult. In the literature many approaches have been proposed to deal with the problems of choreography realizability, conformance, and enforcement, e.g., [21,8,4,23,9,2]). These approaches are based on different interpretations of the choreography interaction semantics in terms of both the subset of considered choreography constructs and the used formal notation. Moreover, the adopted notations, although powerful and well known in the formal community,

* This work is partially supported by the EU FP7/2007-2013 under grant agreement number 257178 (project CHOReOS - www.choreos.eu).

do not completely satisfy requirements related to usability and pragmatism that BPMN2 choreography modelers usually require.

In this paper we overview the model transformation we have adopted within the CHOReOS EU project (`www.choreos.eu`) to generate from a BPMN2 choreography diagram an automata-based specification of the coordination logic "implied" by the choreography. The transformation constitutes the core of the overall methodology we apply in CHOReOS to solve the problem of automatic choreography enforcement. The core objective of CHOReOS is to leverage model-based methodologies [5] and relevant SOA standards, while making *choreography development* a systematic process to the reuse and the assembling of services discovered within the Internet. Our approach has two main advances: (i) most of the complex constructs of BPMN2 choreography diagrams, e.g., inclusive and parallel gateways, are handled; (ii) an extension of LTSs, called *Choreography LTS* (CLTS), is provided to enable explicit descriptions of the coordination logic that must be applied to enforce the choreography, by adopting a notation that is closer to the BPMN2 choreography one; (iii) CLTS makes explicit coordination-related information that in BPMN2 is implicit. This allows to statically infer the information needed for enabling distributed coordination that, otherwise, should be calculated at run time for each choreography instance and for each execution of it. For instance, the CLTS model specifies the source and target state from which a task is initiated and terminated, the corresponding transition and enabling condition.

The paper is structured as follows. Section 2 introduces the considered BPMN2 choreography constructs. Section 3 briefly outlines the model transformation we have developed and how the introduced BPMN2 constructs are mapped to CLTS constructs. Related works are discussed in Section 4 and conclusions and future directions are given in Section 5.

2 BPMN2 choreography diagram constructs

In the following we leverage the in-depth study of the "meanders" of the BPMN2 standard specification document, and introduce the considered BPMN2 choreography diagram constructs by concisely describing their crucial characteristics. In Figures 1 and 2, besides the BPMN2 constructs on the left side, we report the corresponding CLTS translation that will be then discussed in Section 3.

The selection of the considered BPMN2 constructs has been performed by analysing the intrinsic aspects related to the choreography enforcement problem and by fulfilling the requirements of all the CHOReOS use cases.
With reference to Figure 1 **(a)..(d)**, a choreography `Task` is an atomic activity that represents an interaction by means of one or two (request and optionally response) message exchanges between two participants. Graphically, BPMN2 diagrams uses rounded-corner boxes to denote choreography tasks. Each of them is labeled with the roles of the two participants involved in the task, and the name of the service operation performed by the initiating participant and provided by the other one. A role contained in the white box denotes the initiating participant. In particular, we recall that the BPMN2 specification employs the

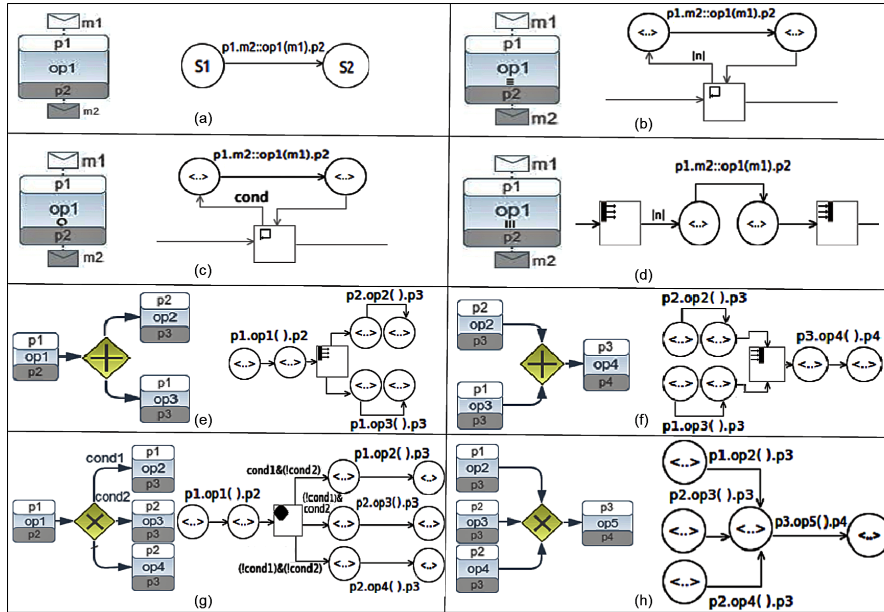


Fig. 1. From BPMN2 choreography to CLTS

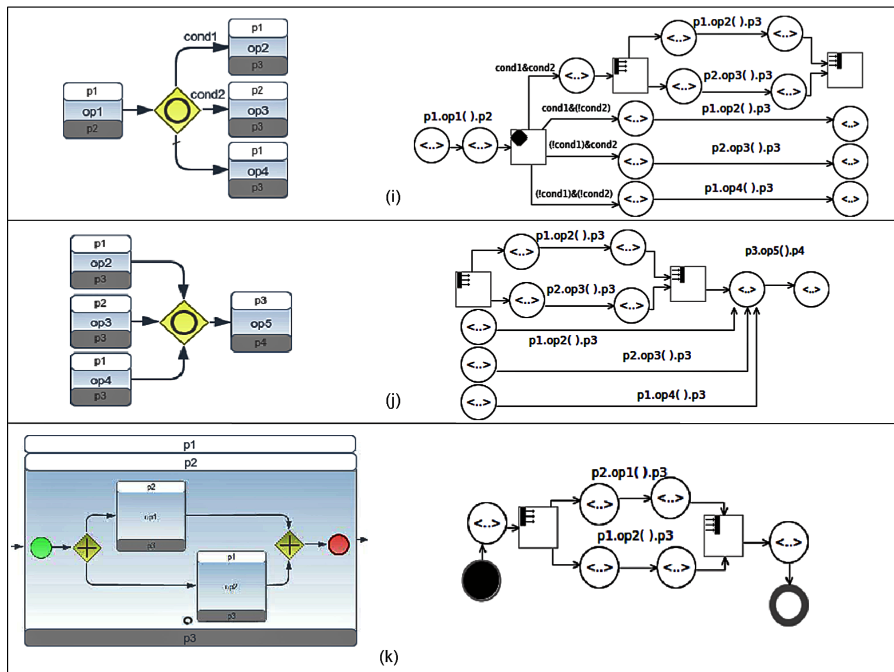


Fig. 2. From BPMN2 choreography to CLTS (Cont'd)

theoretical concept of a token that, traversing the sequence flows and passing through the elements in a process, aids to define its behavior. The start event generates the token that must eventually be consumed at an end event.

Depending on its type (i.e., `ChoreographyLoopType` in the BPMN2 specification), a task may have one of the three markers: sequential multi-instance (b), standard loop (c), and parallel multi-instance (d).

With reference to Figure 1 **(e) & (f)**, a Parallel Gateway is used to (e) create and/or (f) synchronize parallel flows without checking any condition. Each outgoing flow receives a token upon execution of this gateway. For incoming flows, this gateway will wait for all incoming flows before triggering the flow through its outgoing arrow. They create parallel paths of the choreography that all Participants are aware of. With respect to the constraints imposed by the BPMN2 official specification, the initiator participant(s) of all the tasks after the gateway must be involved in all tasks that immediately precede such gateway. The task that precedes the chain must also satisfy this constraint in the case where there is a chain of gateways with no tasks in between.

With reference to Figure 1 **(g) & (h)**, a Diverging (Decision) Exclusive Gateway (g) is used to create alternative paths within a choreography. If none of the conditional expressions (see `cond1` and `cond2`) evaluate to true, a default path can optionally be specified (see task `op4`). A Converging Exclusive Gateway (h) is used to merge alternative paths. Each incoming flow token is routed to the outgoing flow without synchronization. Being in a fully decentralized setting, there is no central mechanism to store the data that will be used in the condition expressions of the outgoing flows. The gateway's conditions may have natural language descriptions but, as clarified by the BPMN2 official specification, such choreographies would be underspecified and would not be enforceable. To create an enforceable choreography, the gateway conditions must be formal expressions that can be precisely (and automatically for tool supported approaches) checked. Still according to the BPMN2 official specification, the initiating participants of the choreography tasks that follow the gateway must have sent or received the message that provided the data upon which the conditional decision is made. In addition, the message that provides the data for the gateway conditional decision may be in any choreography task prior to the gateway (i.e., it does not have to immediately precede the gateway). Thus, for the gateway to be automatically enforced, we assume to have the specification of what messages provide the data upon which the conditional decision can be actually made.

With reference to Figure 2 **(i) & (j)**, a Diverging Inclusive Gateway (i) can be used to create alternative but also parallel paths. Unlike the Exclusive Gateway, all condition expressions are evaluated. All flows that evaluate to true will be traversed by a token. Since each path is considered to be independent, any combination of the paths may be taken, from zero to all. However, it should be designed so that at least one path is taken. If none of the conditional expressions (see `cond1` and `cond2`) evaluate to true, a default path can optionally be specified. A converging Inclusive Gateway (j) is used to merge a combina-

tion of alternative and parallel paths. A control flow token entering an Inclusive Gateway may be synchronized with some other token that arrives later.

With reference to Figure 2 **(k)**, a Sub-Choreography is a compound activity task that defines a flow of other tasks. Each sub-choreography involves two or more participants.

3 BPMN2-to-CLTS

In this section we discuss how it is possible to derive CLTS models out of BPMN2 choreography specifications. Such a translation can be done in different manners including the adoption of general purpose programming languages. However, as previously said, the presented work has been done in the context of the CHOReOS EU project where model-driven principles and techniques [5] have been employed to support the development of choreography-based systems.

Before describing the model transformation, we introduce the *CLTS* metamodel defined for extending LTSs, and constitutes the foundation of the coordination logic extracted by the synthesis process. The definition of these metamodels is the result of a survey we have conducted within CHOReOS. Specifically, in the literature a number of valuable approaches have been proposed to transform different kinds of choreography notations into more formal specifications. For instance, by leveraging the concept of token, alternative models, such as free-choice Petri Nets, might have been adopted (see Section 4). However, the deep study we have initially conducted within CHOReOS to precisely define, at the project level, the integration architecture for the CHOReOS Integrated Development and Run-time Environment[1] (IDRE), led the whole consortium to agree on the definition of the CLTS model, which best met the (both formal and technical) requirements of all the software tools now integrated by the IDRE. Indeed, to the purposes of defining an integrated suite of tools to support the whole choreography life cycle, the CLTS model brings together many features of already existing formalisms and notations in the literature, and filters out those ones not strictly needed. Last but not least, the main requirement was to have a notation as close as possible to the BPMN2 choreography diagrams, while enabling formal reasoning and automatic treatment by all the IDRE components.

A fragment of the CLTS metamodel is shown in Figure 3. The metamodel extends the basic notion of LTS state by introducing new elements to model complex states, i.e., `initial` and `final` states, `fork` and `join` states, as well as, `activity loop` and `alternative` states. The basic notion of labeled transition has been extended to have the possibility of specifying `participants roles`, `service operations request/response/fault` messages and related `types`, as well as, `conditions`.

As discussed later in the section, the BPMN2-to-CLTS transformation has been implemented in a model-driven setting by means of the ATLAS Transformation Language (ATL) [13]. A model transformation takes as input one or more models conforming to the source metamodels and generates one or more models conforming to the target metamodels. Thus, to develop the BPMN2-to-CLTS

[1] `http://www.choreos.eu/bin/view/Documentation/WebHome`

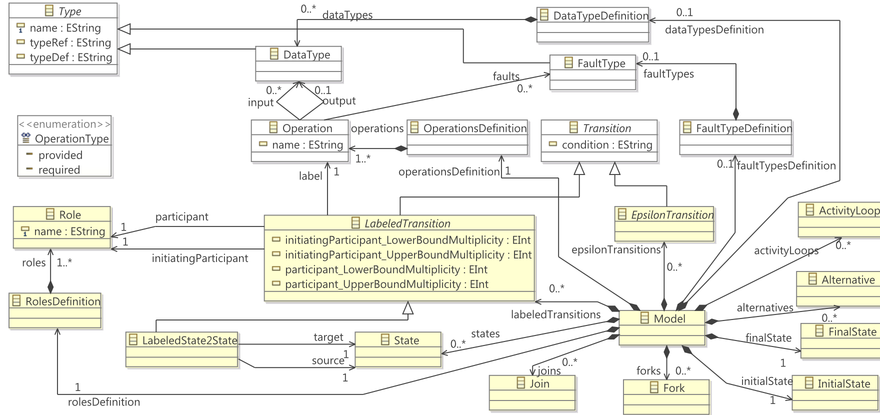


Fig. 3. CLTS metamodel

transformation, both the BPMN2 and CLTS metamodels are required. The former is available in the Eclipse ecosystem; the latter consists of the following elements (see Figure 3):

▷ the metaclass `Model` and its composition relations represent a choreography;

▷ plain states are represented by means of the metaclass `State`;

▷ initial and final states are represented by means of the metaclasses `InitialState` and `FinalState`, respectively;

▷ loop and alternative elements are represented by means of the metaclasses `ActivityLoop` and `Alternative`, respectively;

▷ fork and join elements are represented by means of the metaclasses `Fork` and `Join`, respectively;

▷ the set or roles played by the different participants in the choreography are represented by means of the metaclass `RoleDefinition`, which consists of a number of `Role` elements.

▷ the set of transition labels is represented by means of the metaclass `OperationDefinition`, which contains `Operation` elements.

▷ transition relations are represented by means of the metaclass `LabeledTransition`. The references **`initiatingParticipant`**, and **`participant`** are defined to represent the participants involved in the considered interactions;

In the remaining of the section we discuss how BPMN2 choreography diagram constructs can be mapped to CLTS model element. The discussion is based on the representative cases shown in Figure 1 and Figure 2.

(a)..(d) – As previously said, depending on its type (i.e., `ChoreographyLoopType` in the BPMN2 specification), a task may have one of the three markers: sequential multi-instance (b), standard loop (c), and parallel multi-instance (d). Accordingly, a task is transformed into a basic state-to-state transition if no marker is specified (a), a CLTS `ActivityLoop` transition with a fixed number |n| of possible iterations if a BPMN2 sequential multi-instance marker is specified (b), a conditional CLTS `ActivityLoop`

transition if a BPMN2 standard loop marker is specified (c), and a CLTS state-to-state transition that can be forked (and then joined) a fixed number |n| of times if a BPMN2 parallel multi-instance marker is specified (d). In the corresponding CLTS fragments, the transition label $p1.m2{::}op1(m1).p2$ specifies that the participant $p1$ initiates the task $op1$ by sending the message $m1$ to the receiving participant $p2$ which, in turn, returns the message $m2$.

Note that the BPMN2 graphical elements do not show, neither the condition expressions nor the specified fixed number, which can only be internally specified. In/out messages (i.e., request/response messages) are reported in the CLTS transitions on the right-hand and left-hand sides of the operation name, respectively. To bound the number of times a loop is repeated, either the condition expressions must be evaluated (based on the data contained in the exchanged messages) in the case of a standard loops, or counters must be employed (updated upon the observed message exchanges) in the case of sequential and parallel multi-instance loops. In any case, the task must be performed at least once, before checking the condition or the counter. In this respect, it is worth to mention that one of the reported critical issues (issue number 16554 - published in the official web site) is about "underspecification of `ChoreographyLoopType`". The reported issue basically says that it is mandatory to specify either the number of loop repetitions or the expression that must be evaluated, on what messages and (for reasons of enforceability) by which participant(s). In our model transformation we have anticipated the resolution of this issue.

(e) & (f) – When used to create parallel flows, the parallel diverging gateway is transformed using a CLTS `Fork` state that splits into all the outgoing flows. Note that, in order to enforce the coordination logic implied by a parallel gateway, the `Fork` state is used in a CLTS to model real parallelism (and not abstract parallelism by means of interleaving). Complementarily, when a parallel converging gateway is used to join parallel flows, a CLTS `Join` state is used.

The sequences for the remaining cases **(g)..(k)** can be easily obtained by following the same method as for the previous cases.

(g) & (h) – When used to create alternative paths, a diverging exclusive gateway is transformed using a CLTS `Alternative` state. Note that the conditions `cond1` and `cond2` are suitably combined to achieve exclusivity. When used to merge alternative paths, a converging exclusive gateway is transformed using state-to-state transitions that, by modeling the flows immediately preceding the gateway, collapse into a further state-to-state transition that models the flow immediately following the gateway. As a further clarification, the very same converging exclusive gateway behavior can be equivalently specified in BPMN2 without using the gateway construct. That is, with reference to the figure, it is sufficient to have three arrows that directly connect the tasks on the left to the task on the right.

(i) & (j) – Similarly to a diverging exclusive gateway, diverging inclusive gateway is transformed using a CLTS `Alternative` state. However, to model that all combinations of the paths may be taken, combined forking and joining paths are used. To conform with this characteristic, the conditions `cond1` and `cond2` are

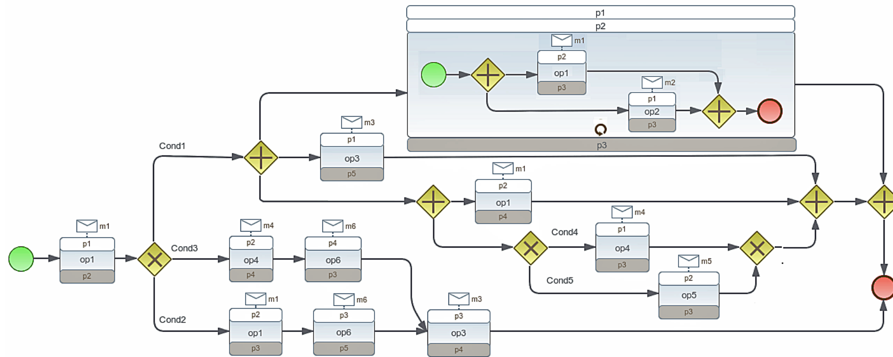


Fig. 4. BPMN2 choreography diagram example

suitably combined to achieve exclusivity between not only single paths, but also between the combined forking and joining paths and single paths. Considering the previous explanations for the converging exclusive gateway and the diverging inclusive gateway, the transformation for a converging exclusive gateway, when merging combinations of alternative and parallel paths, is rather intuitive.

(k) Compound activities tasks are transformed by recursively applying the previous rules. In Figure 2, only a very simple case is shown.

All the previously discussed mappings have been automatized by means an ATL [13] model transformation consisting of about 4.000 lines of code. By applying the transformation rules described above, the BPMN2 choreography diagram of Figure 4 is transformed to the corresponding CLTS diagram in Figure 5 (the CLTS diagram has been drawn by means of the GMF-based editor we have developed in CHOReOS). It is worth to clarify that the choreography in the figure has been aptly created to highlight, in one choreography, most of the crucial subtitles the transformation needs to handle. Therefore, it looks artificial from a use case point of view. For a set of realistic use cases, provided by the CHOReOS industrial partners, and for downloading the Eclipse plug-

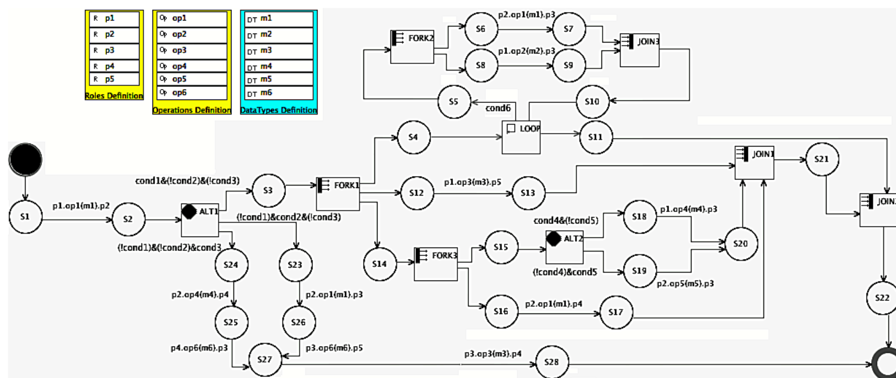


Fig. 5. CLTS derived from the BPMN2 choreography diagram in Figure 4

ins implementing the model transformation the interested reader can refer to `http://choreos.disim.univaq.it/`.

4 Related Work

The approach presented in this paper is related to a number of other valuable approaches in the literature that, to different purposes, transform different kinds of choreography notations into more formal specifications.

The definition of our approach required the consideration of valuable approaches in the literature (most notably [21,8,4,23,9,2]) and state-of-the-art languages, systems, and techniques that have emerged in different contexts including SOA, model-transformations, and seminal work on distributed coordination [15]. Their consideration within the same research and development space [1] is so far being representing the opportunity for us to harness our knowledge towards the systematic development of choreography-based systems.

The common idea underlying the approaches in [6,7,16,17,20] is to assume a high-level specification of the requirements that the choreography has to fulfill and a behavioral specification of the services participating in the choreography. From these two assumptions, by applying data and control-flow analysis, a BPEL, WSCI or WS-CDL description of a centralized choreographer specification is automatically derived. This description is derived in order to satisfy the specified choreography requirements.

The works described in [22,24,3,11,4] address the problem of checking whether a choreography can be realized by a set of interacting services, each of them synthesized by projecting the choreography specification on the role to be played. This problem is known as choreography realizability check. The focus is on verifying whether the set of services, required to realize a given choreography, can be easily implemented by simply considering the role-based local views of the specified choreography. That is, this verification does not aim at extracting the global coordination logic that, as we do in CHOReOS, is needed to check whether the collaboration among the discovered services leads to global interactions that violate the choreography behavior.

In [9] the authors presents a framework for verifying choreographies using model and equivalence checking techniques. Leveraging a translation of the choreography into LOTUS NT algebra, the framework enables the verification of some analysis tasks, i.e., repairability, realizability, conformance, synchronizability, and control for enforcing the choreography. In order to check in sequence the system synchronizability and realizability using equivalence checking, distributed controllers are generated through an iterative process presented in [10].

In [19], the authors show how to automatically generate a partial DAML-S process model out of a WSDL description. The generated process model can be then possibly completed manually by the developer. Although the aim of this work is completely different from ours, it shows that DAML-S can be considered as another possible notation BPMN2 choreographies could be mapped to.

The work in [14] presents an approach that generates a set of related orchestrations from a choreography specification. Specifically, a transformation to derive a set of BPEL specifications out of a CDL specification of the choreogra-

phy is formalized. Similarly to us, the transformation mappings is implemented as transformation rules in ATL.

In [12] the authors propose a model-driven method to develop collaborative systems. The methods uses a graph transformation to derive a flow-local choreography from a flow-global choreography. UML Activity Diagrams are used to specify both the source model and the target model.

Most of the previous approaches consider as input choreography specified by using different notations and formalisms. Only few of them uses BPMN2 Choreography Diagrams, notably, [9] and [21]. Depending on the specific purposes, the different approaches transform the choreography into different (formal) representations such as Petri Net, LOTUS NT, various state machines, etc. Moreover, all these approaches are based on different interpretations of the choreography interaction semantics and consider different subsets of choreography constructs. A weakness here resides on the fact that all the adopted formal notations are distant from BPMN2.

5 Conclusions and Future Work

This paper presents a model transformation to extract from a BPMN2 choreography specification the global coordination logic and codify it into an extended LTS, called *Choreography LTS* (CLTS). The expressiveness of the CLTS metamodel allows us to fully automate the approach and to transform very complex choreography specifications into rigorous descriptions. The presented approach is implemented as a REST service and it part of a model-based tool chain (named CHOReO*Synt*[2]) released to support the development of choreography-based systems in the CHOReOS EU project.

The approach has been applied to real-world use cases, provided by the CHOReOS industrial partners, in the *Airport*, *Marketing and Sale*, and *Taxy Transportation* domains. Interested readers can refer to the CHOReOS project and CHOReO*Synt* web sites for documentation. The application of our approach to these use cases has shown that the method is viable and practical. However, although our preliminary validation has been carried out in a context in which the presence of relevant to the approach stackholders are present, a real quantitative assessment of the method needs further investigation. This is part of our ongoing work together with an industrial partner of the CHOReOS project. Another direction of future work will address the extension of the implemented transformations to transform also BPMN2 choreography specifications containing events, as well as event-based and complex gateways.

References

1. M. Aksit, I. Kurtev, and J. Bézivin. Technological Spaces: an Initial Appraisal. International Federated Conf. (DOA, ODBASE, CoopIS), Industrial Track, Los Angeles, 2002.
2. M. Autili, D. Ruscio, A. Salle, P. Inverardi, and M. Tivoli. A model-based synthesis process for choreography realizability enforcement. In *FASE*, volume 7793 of *LNCS*, pages 37–52. 2013.

[2] `http://choreos.disim.univaq.it`

3. S. Basu and T. Bultan. Choreography conformance via synchronizability. In *Proc. of WWW '11*, pages 795–804, 2011.
4. S. Basu, T. Bultan, and M. Ouederni. Deciding choreography realizability. In *Procs. of POPL 2012*, pages 191–202. ACM, 2012.
5. J. Bézivin. On the Unification Power of Models. *Jour. on Software and Systems Modeling (SoSyM)*, 4(2):171–188, 2005.
6. A. Brogi and R. Popescu. Automated Generation of BPEL Adapters. In *Proc. of ICSOC'06, volume 4294 of LNCS*, 2006.
7. D. Calvanese, G. D. Giacomo, M. Lenzerini, M. Mecella, and F. Patrizi. Automatic service composition and synthesis: the roman model. *IEEE Data Eng. Bull.*, 31(3):18–22, 2008.
8. G. Gössler and G. Salaün. Realizability of choreographies for services interacting asynchronously. In *FACS*, volume 7253 of *LNCS*, pages 151–167. 2012.
9. M. Güdemann, P. Poizat, G. Salaün, and A. Dumont. Verchor: A framework for verifying choreographies. In *FASE*, volume 7793 of *LNCS*, pages 226–230. 2013.
10. M. Güdemann, G. Salaün, and M. Ouederni. Counterexample guided synthesis of monitors for realizability enforcement. In *ATVA*, LNCS, pages 238–253, 2012.
11. S. Hallé and T. Bultan. Realizability analysis for message-based interactions using shared-state projections. In *FSE*, pages 27–36, 2010.
12. F. Han, S. Kathayat, H. Le, R. Brek, and P. Herrmann. Towards choreography model transformation via graph transformation. In *Procs. of ICSESS 2011*, pages 508–515, 2011.
13. F. Jouault, F. Allilaire, J. Bézivin, and I. Kurtev. ATL: A model transformation tool. *Science of Computer Programming*, 72(1-2):31–39, 2008.
14. R. Khadka. *Model-Driven Development of Service Compositions: Transformation from Service Choreography to Service Orchestrations*. Thesis for a degree in master of science in computer science, University of Twente, Enschede, The Netherlands.
15. L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM*, 21:558–565, July 1978.
16. A. Marconi, M. Pistore, and P. Traverso. Automated Composition of Web Services: the ASTRO Approach. *IEEE Data Eng. Bull.*, 31(3):23–26, 2008.
17. T. Melliti, P. Poizat, and S. B. Mokhtar. Distributed behavioural adaptation for the automatic composition of semantic services. In *In Procs. of FASE'08/ETAPS'08*, pages 146–162, 2008.
18. OMG. Business Process Model And Notation (BPMN) Version 2.0. `http://www.omg.org/spec/BPMN/2.0/`.
19. M. Paolucci, N. Srinivasan, K. Sycara, and T. Nishimura. Towards a semantic choreography of web services: from wsdl to daml-s. In *In Procs. of ICWS'03*, pages 22–26, 2003.
20. J. Pathak, R. Lutz, and V. Honavar. Moscoe: An approach for composing web services through iterative reformulation of functional specifications. *International Journal on Artificial Intelligence Tools*, 17:109–138, 2008.
21. P. Poizat and G. Salaün. Checking the Realizability of BPMN 2.0 Choreographies. In *Proc. of SAC 2012*, pages 1927–1934, 2012.
22. G. Salaün. Generation of service wrapper protocols from choreography specifications. In *Proc. of SEFM*, 2008.
23. A. Stefanescu, S. Wieczorek, and M. Schur. Message choreography modeling. *Software & Systems Modeling*, pages 1–25, 2012.
24. J. Su, T. Bultan, X. Fu, and X. Zhao. Towards a theory of web service choreographies. In *WS-FM*, pages 1–16, 2007.

A Lightweight Framework for Testing Safety-critical Component-based Systems on Embedded Targets

Nermin Kajtazovic, Andrea Höller, Tobias Rauter, and Christian Kreiner

Institute for Technical Informatics, Graz University of Technology,
Inffeldgasse 16, Graz, Austria
{nermin.kajtazovic,tobias.rauter,andrea.hoeller,christian.kreiner}@tugraz.at

Abstract. Rigorous development and quality assurance are inherent parts in the engineering of safety-critical systems. Many standards that address the development and certification of these systems provide a collection of various types of tests that have to be conducted to achieve the desired level of quality. Further, they recommend to perform most of these tests on the target embedded system, rather than on development hosts for example. For specific architectures, such as those used in component-based systems, this requirement is often difficult to achieve, mostly because of lack of available test frameworks that can support such specific architectures.
In this paper, we propose a framework for testing component-based systems on their embedded targets. The test framework allows to deploy software components in their binary form onto such targets. Further, it allows to build compositions out of deployed components so that complete applications can be tested. The compositions are build using the techniques for webservice composition, since interfaces to deployed software components are exposed as webservices. With this lightweight framework, it is possible to conduct some relevant tests required by the safety standards in early development phase, because it only requires software components to be implemented to test complete applications.

Keywords: safety-critical embedded systems; component-based systems; software testing

1 Introduction

Safety-critical systems can cause serious consequences such as harm on humans or equipment and environmental damages, if they malfunction. A rigorous development and quality assurance are therefore required to reduce the risk of such malfunctioning. To this end, standards for functional safety such as IEC 61508 and ISO26262 provide guidelines and methods on how to reduce the risk of failures and how to evaluate the quality of systems [8]. One set of these methods are

tests that have to be performed in order to provide an evidence about the operational profile of the system i.e., to show the conformance with the functional, safety, and other non-functional requirements. These tests are usually aligned with the well-known V development model, which comprises tests on different levels, i.e., from tests on module/unit level, integration on software and system level, to the final systems validation. One important aspect of this test chain within a V-model is that the evidence provided in reports has to conform to the real context in which the system shall operate, i.e., the safety standards require to perform tests on the real target hardware and considering the real environmental conditions [2], [8]. In many cases, this is difficult to realize, because of a variety of used target processors, used systems and software architecture and also because of a lack of specific test platforms for embedded systems. Especially, for component-based systems, which have separated development for the (software) components and for the system[1], this can be a tedious task. Many features have to be prepared in order to reach the point where software integration can be tested. For example, communication mechanisms, middleware for coordination of components and adequate interfaces to the development host are required to just test the integration between components, i.e., their composition.

In this paper, we describe a lightweight framework to perform tests of software components and their compositions on an embedded target. The distinguishing advantage of our approach is that only software components have to be provided to perform such tests. This allows developers to perform certain types of tests on complete component-based applications in the early development phases, i.e., before any middleware service for the coordination and communication of software components is developed. The framework utilizes the technique for webservice composition in order to build applications out of such components. To allow for building compositions, software components are deployed within an embedded target as standalone webservices. One of the major contributions in this paper is the mapping between specific component technology and webservices. In the end of paper, we discuss the applicability of the framework to different component technologies used in the industry.

Section 2 provides a brief overview of studies related to testing component-based systems on embedded targets. Section 3 summarizes a motivation behind the work. The proposed framework is described in Section 4, and its concrete implementation and used tools are described in Section 5. A brief discussion and concluding remarks are given in Sections 6 and 7 respectively.

2 Related Work

Now we turn to a brief overview of related studies. We outline here some relevant articles that describe test frameworks for component-based embedded systems.

Currently, the introduced problem of testing software components and their compositions on embedded targets is very important topic for automotive systems. In the last decade, several frameworks have been developed to support

[1] In the context of Component-based Software Engineering (CBSE) [1]

rapid prototyping, simulation and testing of automotive systems which implement a complex component-based architecture – AUTOSAR [3], [6], [9]. Among these frameworks, the DaVinci Component Tester [3] is the one with most features to test complete component-based systems. It is an emulated environment, with unit testing facilities for atomic and composite AUTOSAR software components. An emulated runtime environment (RTE) of the framework implements basic communication and coordination services for software components so that all necessary interaction scenarios between those components can be emulated. For the purpose of testing, the components have to be compiled for the target where the RTE is operating. Usually, the development host is used to execute RTE, rather than an embedded target. Similar to DaVinci Component Tester, the framework in [6] provides test support for compositions. In contrast to previous work, fault injection tests are applied here to evaluate the reliability of AUTOSAR systems. Finally, the Artop ARUnit framework [9] provides AUTOSAR RTE services to perform unit testing for single software components.

In general, described frameworks perform the testing on development host only. Therefore, for systems qualification, safety-relevant software components and related compositions need to be tested again on their real embedded target. As mentioned in the previous section, to realize this deployment and a support for building compositions such as RTE of the DaVinci Component Tester for embedded targets may require much effort. One option to overcome this issue is to use the standardized techniques for the deployment and composition, such as those provided by service-oriented architectures (SOA) for example. Currently, there are many approaches that use SOA to expose embedded devices with the purpose of testing [7] or integrating devices to implement certain business processes [4]. Another advantage of using SOA for this purposes is that many mature tools and methods exist that provide various generation facilities or test frameworks. Similar to work in [4], we use SOA to build compositions, but instead of exposing device functions as webservices, we expose the deployed software components. Thus, for every software component, there is a single container, a server, which provides a webservice interface to its function.

3 Motivation

Mastering complexity of today's embedded systems is one of the major challenges in safety engineering. In addition to rapid increase of software complexity, many application fields are confronted with the issues coming from the concurrent engineering, where different organizations are contributing to systems development. In the automotive industry for example, many system parts are delivered by the suppliers, including devices, OS services and libraries, while the automotive companies, i.e. manufacturers, are focusing on software applications. Developing such applications is challenging for manufacturers, because for testing purposes they need a system in which all required parts from suppliers are integrated.

Providing a test support for the application-level software without a need to consider supplier's parts would leverage rapid development for the manufactur-

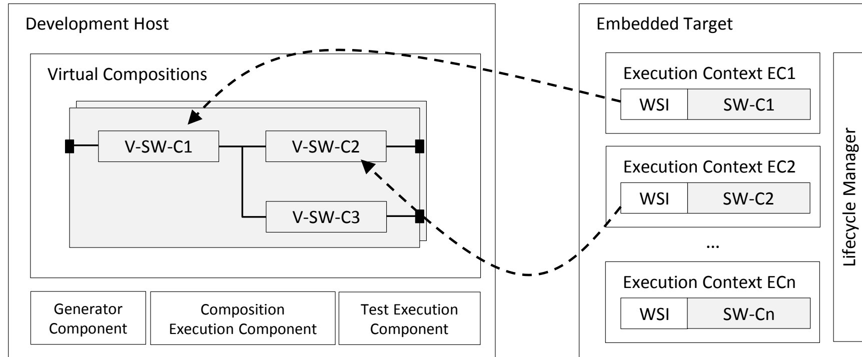


Fig. 1. Test framework architecture: a Development Host – modelling component-based system, definition and execution of tests (left) and an Embedded Target – execution of software components (right)

ers, and would allow them to achieve many test objectives earlier. In summary, the manufacturers would be able to (i) qualify the application-level software components according to regulations of safety standards, and (ii) to early conduct the functional tests, which can be used later in the verification and validation part of the V-model to complement the remaining test activities.

4 Proposed Framework

In this section, we introduce the proposed test framework. We first give an overview of its main components, and then we describe how the test process is conducted using the framework.

Fig. 1 shows the simplified architecture of the framework. Basically, the framework consists of the two main components: the Development Host and the Embedded Target, which has to be used in the operation of the safety-critical system. This Embedded Target provides services to deploy software components and to manage their lifecycle during the tests. The Lifecycle Manager component shown in Fig. 1 is responsible for these purposes. One of the main characteristics of the framework is that software components are executed in isolation and do not interfere with each other within the Embedded Target. That means, the interaction between those components is not possible within the Embedded Target. Thus, software components can just execute their functions, based on input data provided by the Execution Context component, which also collects the results from that component after the execution. The Execution Context corresponds to a simple middleware or a container that implements the lifecycle management for a single software component. To communicate with the rest of the framework, it exposes the interfaces of its software component as a webservice (the Webservice Interface WSI in figure).

On the other side, the Development Host comprises a collection of various tools in order to (i) to map a particular component technology on webservices, i.e., to build the Execution Context (see Section 5 for more details), (ii) to build a component-based application by composing software components running on the Execution Context, and (iii) to conduct the test process on those applications.

4.1 Framework Components

Virtual Composition corresponds to a modelled component-based application. In its simplest form, it corresponds to a concrete software component, which runs on the Embedded Target. Therefore, for every software component, there is a corresponding Virtual Composition which runs on the Development Host. From the technical viewpoint, a Virtual Composition is a webservice instance which points to a concrete software component on the Embedded Target. In a more complex form, the Virtual Composition can comprise multiple software components, i.e., a composite of multiple Virtual Compositions, so that complete component-based applications can be modelled. For this purpose, a technique for webservice composition is used (see Section 5 for more details).

Composition Execution Component. This component of the framework executes Virtual Compositions. Based on their descriptions, it simulates the application behavior while executing the involved components on the Embedded Target.

Test Execution Component is a bundle consisting of the test cases, stubs and drivers to conduct the complete test on modelled Virtual Compositions. It executes Virtual Compositions against provided test cases and reports the test status.

Execution Context. The Execution Context is a standalone component container that provides the lifecycle management for a single component. Furthermore, it exposes the component interfaces through webservices (WSI) in order to allow to build Virtual Compositions. It also provides the state isolation functionality for stateful components, which is relevant if a particular component is used multiple times by the Virtual Composition (see Section 4.2). Fig. 2 shows the architecture of Execution Context. In short, a software component is wrapped by the webservice, which is generated and linked with the standalone SOAP[2] server that hosts the service. The code on the left in figure shows the concrete method body of the webservice and its link with the concrete software component. We explain this link in Section 5 in more detail.

Generator Component is a part of the deployment process. It comprises a collection of tools (i) to generate the required artifacts for the modelled Virtual

[2] Simple Object Access Protocol: application-layer protocol for webservices.

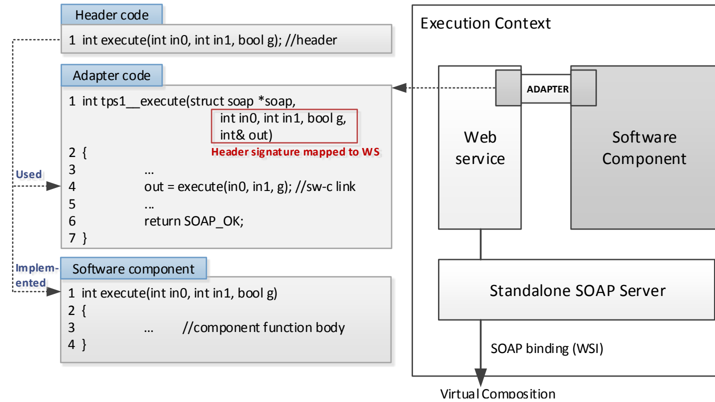


Fig. 2. Execution Context: artifacts used for the mapping between CBSE and SOA (left) and the architecture (right)

Compositions and to deploy them onto the Development Host, and (ii) to generate the Execution Context, i.e., to map CBSE on SOA, and to deploy it onto the Embedded Target.

Lifecycle Management manages the lifecycle of all Execution Contexts within the Embedded Target. It provides services for the deployment, removal and roll-back[3] of software components.

4.2 State Isolation

The presence of the stateful components is an issue if multiple instances within a single Virtual Composition exist and if several Virtual Compositions share the software components simultaneously. It is therefore necessary to protect such software components from transition into an inconsistent state, i.e., the state influenced by one Virtual Composition shall not be used or compromised by another Virtual Composition. This feature is a part of the Execution Context. It isolates the state by identifying the Virtual Composition that owns the currently active request. For this purpose, the context-dependent state is queued within the Execution Context for each stateful component. Based on the incoming request, the corresponding state is accordingly restored.

4.3 Test Workflow

After software components are developed, they are compiled for the Embedded Target and deployed there, using the Generator Component. To perform the tests on software components or on their compositions, a component-based system is modelled first, by defining Virtual Compositions and by deploying them onto the

[3] Required to reset the component state for new test iteration.

Development Host. Finally, test cases are defined and executed on the modelled Virtual Compositions, using the Test Execution Component.

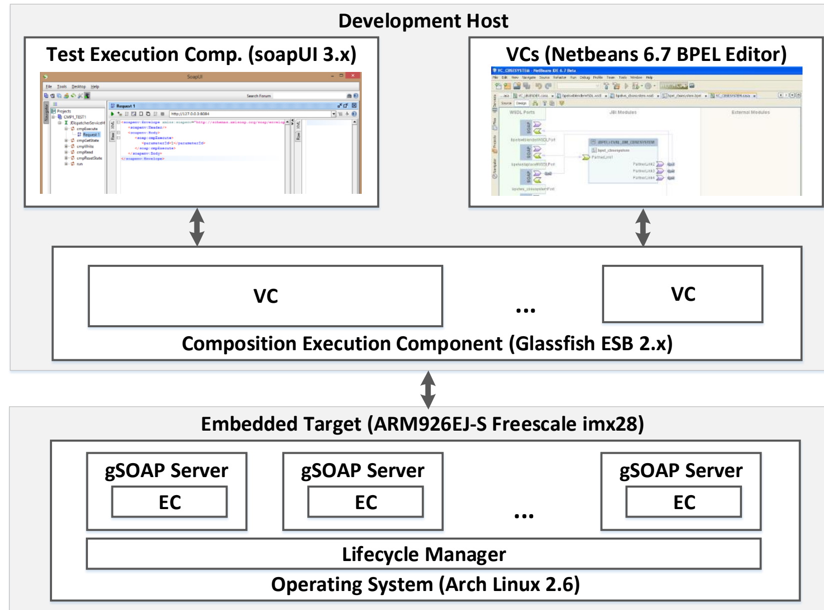


Fig. 3. Implementation of the framework and used tools and configurations (EC - Execution Context, VC - Virtual Composition)

In the following section, we introduce the concrete implementations and tools used to realize the mentioned components (see Fig. 3).

5 Implementation

In order to apply the framework in the domain of embedded systems, we use the gSOAP[4] webservice library, which offers the SOAP stack dedicated to resource-constrained systems. The gSOAP webservices host the Execution Context and the Lifecycle Manager directly on the Embedded Target. For every software component, there is a dedicated gSOAP server that hosts that particular component. All gSOAP servers are running as Unix processes within an Arch Linux operating system. In our test setup shown in Fig. 3, we deploy software components on an ARM9 target (Freescale imx28 with 454MHz, 128MB of RAM).

For modeling and executing the Virtual Compositions, we use an XML-based WS-BPEL (WS Business Process Execution Language) [5]. This technology is widely applied in enterprise applications to seamlessly integrate webservices or

[4] gSOAP Homepage: `http://gsoap2.sourceforge.net/`

legacy applications wrapped by webservices into a business process. Thus, every Virtual Composition is represented as a BPEL process and therefore consists of a workflow, which describes the sequence on how the software components have to be executed. In addition to the workflow, every Virtual Composition describes the integration between software components as a structure, i.e., in a composite diagram (see Fig. 3, Netbeans 6.7 BPEL editor).

From the front-end viewpoint, we use the soapUI Tool[5] to define the test cases and to drive the test process. This tool allows to perform various test strategies on webservices, such as functional tests, load tests and security tests. It also enables the automated test execution for given test suites. In our context, it plays the role of the Test Execution Component. We manually specify the test cases, define a test suite and execute it on the deployed Virtual Compositions.

Another tool which is used as part of the front-end is the Netbeans BPEL Editor. We used it to graphically define Virtual Compositions in terms of the structure and the workflow and to generate the necessary artifacts for the deployment, such as WSDLs and assembly descriptions for Virtual Compositions. In order to deploy Virtual Compositions as webservices on the Development Hosts, we use the Sun Glassfish Enterprise Service Bus (ESB).

5.1 Webservice Interfaces for Software Components

As illustrated in Fig. 3 webservice interfaces of the Execution Context are hosted by the gSOAP servers. Except of the SOAP stack, the gSOAP library consists of a generator toolchain, which allows to build webservice stubs and skeletons from the WSDL specifications. We use the *wsdl2h* tool to generate the header files that are in turn used to link the object code of software components with the Execution Context. The generation of the Execution Context is supported by the *soapcpp2* skeleton compiler (see Section 5.2). This is one of the most challenging parts of the framework, because here, a mapping from the used component technology to webservices is performed. An excerpt of this mapping is depicted in Fig. 2. Here, a software component is represented using just a single C/C++ method. This method is used by the Adapter, which routes the data from the Development Host to the component, and returns the results from that component. To establish this link, we define a header file, which is implemented by the software component, and which is used by the Adapter to find a proper symbol after linking the adapter with the component. Both the adapter code and the headers are generated based on interface description of a software component. In the following, we describe the process of generating artifacts used to build the Execution Context.

5.2 Deployment Process

The essential part of the deployment process in which the Execution Context is generated is depicted in Fig. 4. The process starts by submitting the software

[5] soapUI Homepage: `http://www.soapui.org/` – in this work used for specifying functional tests only.

component in form of the object code and its interface description to the Lifecycle Manager on the Embedded Target. The Lifecycle Manager in turn starts the deployment by generating the necessary skeleton code based on component interface description, i.e., it generates a header file that describes the component interface for the linking with the Execution Context, and the required libraries for the SOAP stack. In the next step, the Execution Context is generated. At this point, all artifacts for the deployment are ready. They are, in the final step, compiled and linked to a single image of the Execution Context, which is then bootstrapped by the Lifecycle Manager. The Execution Context in turn takes the control over the component lifecycle and publishes its WSI. After this last step is completed, the software component is ready for tests.

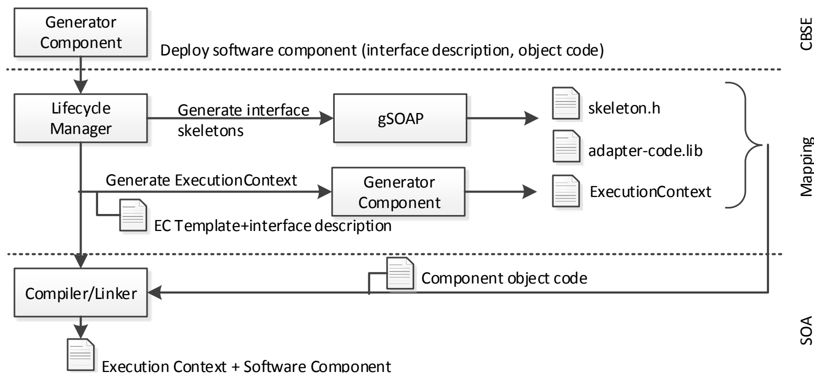


Fig. 4. Generation process for the Execution Context

6 Discussion

We showed in this paper how component-based systems can be tested on embedded targets. With the introduced framework, the functional tests can be performed on a level of software components and their compositions. Although the framework allows testing just on a functional level (compared to introduced frameworks where also middleware is part of a test), it allows to conduct the early qualification of software components on embedded targets and to test potential component-based applications.

We also described the link between SOA and CBSE, using plain C/C++ methods as component technology. To apply the framework to other component-based systems, similar adapters to SOA have to be realized. For instance, to test AUTOSAR systems, the adapters for Runnables have to be implemented. Runnables are execution units within AUTOSAR software components, and are triggered by specific events by the AUTOSAR RTE. The events have to be realized with BPEL, and AUTOSAR software component have to be realized as

Virtual Compositions. In contrast to CBSE to SOA mapping introduced here, a composition of AUTOSAR software components would be represented as a Virtual Composition that consists of further Virtual Compositions, each representing a single AUTOSAR software component. On the other side, for data-flow synchronous systems such as Matlab Simulink and IEC61131, the mapping to SOA can be realized as described in this paper. That means, in case of specific execution semantics such as request-response and sender-receiver interaction styles in AUTOSAR, additional layers of Virtual Compositions are required.

7 Conclusion

Testing safety-critical systems in their real context, i.e., on embedded targets and under real environmental conditions, is recommended by safety standards. However, there are many challenging factors to perform this task, such as variety of available target processors, lack of test frameworks for embedded systems and used specific systems architecture.

In this paper, we introduced a framework to test safety-critical component-based systems on embedded targets. With the framework, the functional tests can be performed on a level of software components and their compositions. The distinguishing advantage of our approach is that only software components have to be provided to perform such tests. This allows developers to perform functional tests on component-based applications in the early development phases.

Currently, the framework can host software components with the primitive data types on their interfaces only. As part of the ongoing work, we will provide a support for specific complex data types, to enable to host some existing component-based systems such as AUTOSAR or IEC61131 systems for example.

References

1. Crnkovic, I., Larsson, M.: Building Reliable Component-Based Software Systems. Artech House Publishers, ISBN 1-58053-327-2 (2002)
2. Grünfelder, S.: Software-Test for Embedded Systems. dpunkt.verlag (2013)
3. Informatik, V.: Davinci component tester - user manual. Tech. rep., VI GmbH (2011)
4. Karnouskos, S., Baecker, O., de Souza, L., Spiess, P.: Integration of soa-ready networked embedded devices in enterprise systems via a cross-layered web service infrastructure. In: IEEE ETFA. pp. 293–300 (Sept 2007)
5. Louridas, P.: Orchestrating Web Services with BPEL. IEEE Softw. (Mar 2008)
6. Piper, T., Winter, S., Manns, P., Suri, N.: Instrumenting autosar for dependability assessment: A guidance framework. In: 2012 42nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN). pp. 1–12 (June 2012)
7. Rusli, H.M., Ibrahim, S., Puteh, M.: Testing Web Services Composition: A Mapping Study. Communications of the IBIMA 2011(598357) (2011)
8. Smith, D., Simpson, K.: A Straightforward Guide to Functional Safety, IEC 61508 (2010 Edition) and Related Standards, Including Process IEC 61511 and Machinery IEC 62061 and ISO 13849. Elsevier Science (2010)
9. Wong, D., Wengler, T., Asmus, R., Rudorfer, M.: Artop: Developing autosar tools in the community. ATZextra worldwide 18(9), 34–36 (2013)