Component Story Diagrams: A Transformation Language for Component Structures in Mechatronic Systems*

Matthias Tichy, Stefan Henkler, and Jörg Holtmann
Software Engineering Group, University of Paderborn,
Warburger Str. 100, D-33098 Paderborn, Germany

Simon Oberthür
Design of Distributed Embedded Systems Research Group,
University of Paderborn,
Warburger Str. 100, D-33098 Paderborn, Germany

E-mail: [mtt|shenkler|chrome|oberthuer]@uni-paderborn.de

Abstract

To cope with the high complexity of software in advanced technical systems, the software of these systems is often built in a component-based fashion. The growing usage of self-adaptive techniques leads to sophisticated reconfigurations of the software component structures during runtime. Current modeling approaches for component-based software systems do not include a transformation language for the specification of component structure reconfigurations. In this paper we therefore introduce an extension of a component-based modeling approach. This extension enables (1) the specification of hierarchical component structures and (2) the specification of structural transformations based on the specified hierarchical component structures. Further, as we consider mechatronic systems, we also show the predictable runtime behavior of the transformations.

1. Introduction

Software has become the driving force in the evolution of mechatronic systems. It grows in domains, like the automotive, at an exponential rate and results in a high complexity due to the cooperation of previously isolated functions (e.g., automotive software [10, 12]). Therefore, these systems are often built in a component-based fashion to counter the effect of growing complexity. The system is then a specific structure of those reusable components.

Self-adaptiveness is a second general trend for mechatronic systems [18]. Self-adaptive systems are robust to changes in their environment by self-monitoring and adaptation. Adaptations can be as simple as changing one parameter of a single component and as complex as a structural transformation on a hierarchical component structure.

Component structures as proposed in current modeling or architecture description languages like UML [26] and its variant SysML [27] as well as in older ones like Darwin [21] can be employed to model such flexible component-based software. However, those approaches do not support the specification of complex structural transformations.

The commercial development environment for mechatronic systems, MATLAB with Simulink and Stateflow, is extensively used for the design of embedded control systems. Simulink supports the notion of enabled subsystems. Enabled subsystems are only executed if a positive value is available at a specific input. It is in principle possible to model complex transformations of the component structures solely with enabled subsystems by enumerating all resulting component structures as distinct configurations. The resulting model becomes very complex for non-trivial transformations. Furthermore, it is only possible to specify

*This work was developed in the course of the Special Research Initiative 614 - Self-optimizing Concepts and Structures in Mechanical Engineering - University of Paderborn, and was published on its behalf and funded by the Deutsche Forschungsgemeinschaft.

a fixed set of distinct component structures. It is not possible to describe transformations which create new configurations.

Other approaches for building adaptive component based systems have been published in [15, 32, 6, 22, 11]. Kacem et al. specify single transformations for component structures using a graphical language based on component diagrams [15]. Similar languages are presented in [32, 6, 11]. The approach by Métayer [22] allows the specification of single transformations as well as sequences but is not based on component diagrams. Graph transformation languages (e.g. [16]) are used in metamodeling approaches to specify transformation on component structures. The specification on the metamodel level typically results in complex and difficult to understand models since they do not use the concrete syntax of component diagrams. We refer to [2] for a recent survey which discusses other approaches.

We postulate the following requirements for a transformation language on component structures based on these related approaches: (1) The language should align with well-known specification languages for component structures, (2) it should support specification of single transformations, and (3) it should support transformation sequences including the specification of loops, decisions, etc. due to our experience with modeling languages for structural transformations (cf. [17, 7, 23]).

We employ the MECHATRONIC UML for modeling mechatronic systems, a variant of the UML [26]. The MECHATRONIC UML approach enables the development of complex mechatronic systems [5]. It supports the component-based specification of software structure and its changes [1], as well as complex real-time coordination behavior and formal verification of safety properties [9]. In addition, the MECHATRONIC UML integrates the modeling of control behavior by embedding controllers in the states of the real-time behavior without forfeiting the verification results [8].

The MECHATRONIC UML supports in principle the required modeling notions. Nevertheless, the current approach has two disadvantages from the modeling perspective.

First, the definition of component types does not allow arbitrary amounts of embedded components on the instance level. It is not possible to model a component type which can carry in principle n embedded components on the instance level. Because of this drawback a more variable type definition is required.

Second, since every possible instance situation has to be modeled in the component type as configuration, it would be helpful for the modeling of systems which possess many configurations to describe the differences between the configurations instead of modeling each possible configuration itself. So there is a need for a transformation language on components.

In this paper we present an extension of the MECHATRONIC UML which overcomes the outlined limitations of current approaches and fulfills the requirements stated above.

1.1. Application Example

A small excerpt from the RailCab project[1] and its safety-critical software is used in the following as our application example. The overall project aims at using a passive track system with intelligent RailCabs that operate autonomously and make independent and decentralized operational decisions. RailCabs either transport goods or up to approx. 10 passengers.

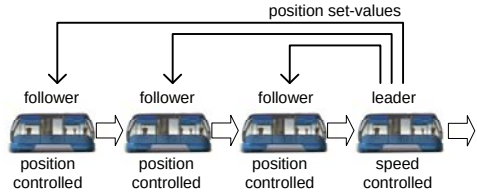


Figure 1. Communication structure in convoy

RailCabs can build convoys in order to reduce energy consumption due to utilization of the slipstream. Information is distributed between the RailCabs in convoy mode [13]. We focus in this paper on the position set values which are sent from the convoy leader to the following RailCabs. The safe coordination between different RailCabs to build and break a convoy has been presented in [9]. The resulting changes on the component structure of the first RailCab in the convoy with respect to the addition of following RailCabs has been sketched in [3]. We extend these sketches by a more thorough structural definition of the component structures and the structural transformations. Note that the presented component structures are kept very simple to serve as an example in this paper.

We first present the specification language for the component structures in Section 2. In Section 3, we introduce the graphical transformation language. Thereafter, we present the employed approach for the formal definition of the semantics in Section 4. Predictable runtime behavior of the presented transformation language in the mechatronic domain is discussed

[1] www.railcab.de/en/

in Section 5. We conclude the paper in Section 6 and provide an outlook on planned future work.

2. Component Models

In the following, we present the extensions of the specification of hierarchical component structures to the MECHATRONIC UML which are required for the specification of transformations. Similar to the UML [26] and as indicated in the last paragraphs, we distinguish component types and their instances. We start by presenting non-hierarchical component types. Thereafter, we present how hierarchical component types are specified.

2.1. Simple Component Types

Simple component types are the building blocks of component structures. They do not contain any sub components but are used as sub components in hierarchical component types. A simple component type has ports and interfaces for connections with other components. In contrast to UML, we define port types for component types. A multiplicity can be associated with these port types. When the classifying component type is instantiated, the multiplicities define the allowed number of instances of the corresponding port type.

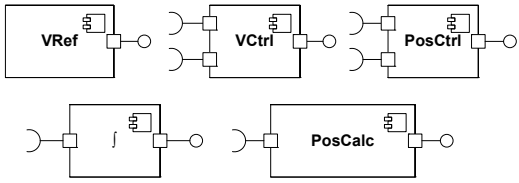


Figure 2. Simple component types used in the RailCab software

Figure 2 shows the component types which are used in our convoy example. The VCtrl and VRef components are used to control the velocity of a RailCab which is driving alone or as the first RailCab in a convoy. The PosCtrl is in contrast used for controlling the RailCab's position in a convoy.

The $\int$ and PosCalc components are used by the convoy-leader RailCab to compute the position of the following RailCabs in the convoy. For each following RailCab, one PosCalc component instance is employed. This component computes the required position of a RailCab in a convoy based on the position of the RailCab directly ahead.

2.2. Hierarchical Component Types

The power of component-based software development stems from the independent reuse of components by third parties [31], i.e. larger components are built from smaller ones by connecting them via contractually specified interfaces.

The UML supports this approach by their concept of composite structures [26]. Structured classes of the composite structures contain references to other classifiers which are called *parts*. An instance of the superior classifier contains instances of the embedded parts by composition. The parts as well as the connectors which connect these parts have multiplicity attributes. Thus, a part can express multiple instances (of the same type). The same applies for the connector or, to be more precisely, for the connector ends which bind the connector to the ports. Parts are also referred to as roles [26], so different parts of the same type can describe different roles. For the specification of hierarchical component types in the MECHATRONIC UML, we simply reuse these UML concepts.

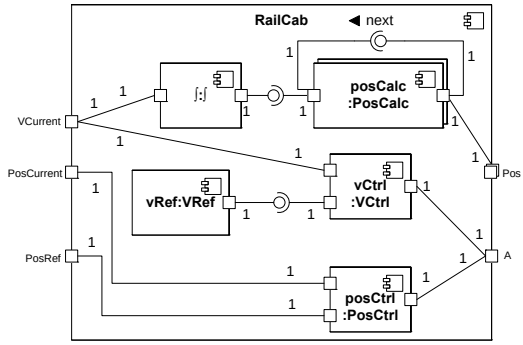


Figure 3. Specification of a RailCab's software structure

Figure 3 shows the hierarchical RailCab component type. This component type contains different parts of the aforementioned simple component types. All those parts have the default multiplicity of 0..1 except the posCalc part which can be instantiated multiple times. This is denoted by the second frame. All parts of the RailCab component type are only used once, thus their names are the same as their types except for the first lower case letter.

The Pos port type has a multiplicity of $*$ (in contrast to the port types of the simple component types from figure 2), and thus is called a *multiport*. Each instance of this multiport is used to connect to a different following RailCab. The A port type is used to send acceleration values to other parts of the software to control the engine.

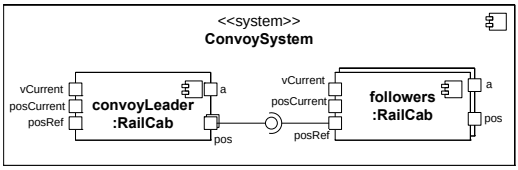


Figure 4. Specification of a convoy system

Figure 4 depicts the system level ConvoySystem, modeled as a hierarchical component type. The system level represents the top level of all hierarchies, where the actual deployment of the several component instances takes place. The system consists of the two parts (or roles) convoyLeader and followers, which are classified by the component type RailCab. The convoyLeader gets the current velocity as input and sends reference positions to all subsequent followers. Note that the port figures represent port parts, classified by the corresponding port type, and thus begin with a lower case letter.

2.3. Component Instances

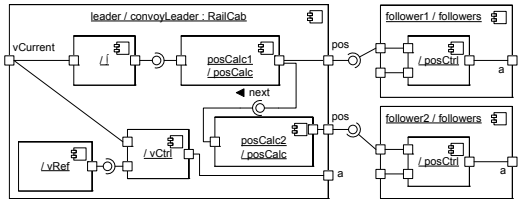


Figure 5. Component instance structure for a convoy of three RailCabs.

After presenting the component type specification, Figure 5 shows the component instance structure for three RailCabs in convoy mode. The component instance structure for the leading RailCab is shown on the left hand side whereas the component instances of the following RailCabs are shown on the right side.

An instance is typed over its part, and thus is implicitly classified by the part's component type. The first relationship is indicated by a "/" between instance name and part name, the latter relationship (like normal type relations) by a ":". A full example which includes instance, part and type name is given by the leader instance, the other instances use short notations. We refer to the UML composite structures [26] for the precise definition of this notation and [14] for the developed metamodels.

In the next section, we present the transformation specifications for the reconfiguration of the leading RailCab when a new RailCab enters such a convoy.

3. Structural Transformations

Our approach for the specification of structural transformations is twofold. Single transformations are specified by a graphical language which is based on a well proven variant of graph transformations [29], called Story Patterns [7, 33]. These single transformations are embedded into the activities of UML like activity diagrams. Activity diagrams allow to combine single transformations for a sequential execution including if-expressions and loops.

Graph transformations [29] are a formalism for the specification of structural changes to graph like structures. Graph transformation rules consist of structural specifications for a precondition and a postcondition. A graph transformation rule is applicable if each element of the precondition structure can be mapped onto one element of the host graph. No two elements of the precondition structure may be mapped onto the same element of the host graph (isomorphism check). Then, the host graph is changed in such a way that each element of the postcondition can be mapped onto an element of the host graph while retaining the element mappings created during the precondition mapping and removing elements which are part of the precondition but not part of the postcondition structure.

Story Patterns [7] are employed as concrete formalism. Story Patterns combine precondition and postcondition structures into a single diagram based on UML [26] collaboration diagrams. The difference between precondition and postcondition structures is denoted by annotating elements by ≪create≫ and ≪destroy≫ stereotypes. An element annotated by ≪create≫ will be created during application of the Story Pattern, whereas an element annotated by ≪destroy≫ will be destroyed. Among other things, these Story Patterns are embedded into Story Diagrams, which are extended UML activity diagrams. In contrast to [7], we employ graph transformations on component instance structures instead of object structures, which we refer to as *Component Story Diagrams*.

Since Component Story Diagrams are based on hierarchical component types, they are also defined and executed in a hierarchical way. We differ between *constructors* and *reconfiguration rules*. Constructors create the initial embedded instance structure of a component, while reconfiguration rules change these once created instance structures and thus allow also deletion of component instances. Like classical Story Diagrams, Component Story Diagrams can be associated with parameters. In the following exam-

ple, we demonstrate the component reconfiguration of the convoy system and the leader convoy shuttle residing one hierarchy level deeper, which is required when another RailCab enters a convoy of at least two RailCabs. In this case, an additional position calculation component instance has to be added to the component instance representing the leading RailCab. Figure 6 depicts the transformation of the convoy system.

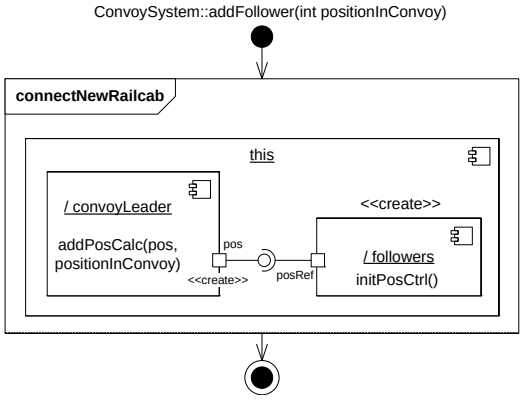


Figure 6. Adding a new RailCab to a convoy of at least two RailCabs.

Each Story Pattern of a Component Story Diagram contains a *component variable* with the name this, which represents an instance of the diagram's hierarchical component type in the host graph. Within this variable, the actual modeling of the Story Pattern takes place. See Figures 6 and 7 for examples on this notation.

The this-variable can contain several further variables, which are classified by the parts which the corresponding component type consists of. These variables represent component and port instances, which are connected by connector links representing the instances of the corresponding connector types. Each variable and each link can be assigned the aforementioned modifier stereotypes.

Figure 6 shows the transformation on the convoy system when a new RailCab enters the convoy. The transformation takes the position of the new RailCab in the convoy as parameter. Since there exists just one instance of the role convoyLeader in a convoy system, the convoy leader will be bound over its role. A new port instance pos of the corresponding multiport of /convoyLeader is created and connected to the following RailCab's software component instance, which is newly created as well.[2]

[2]We assume here that the underlying communication framework transparently handles the necessary inter node communication.

In order to respect the hierarchies of the different component types and to benefit from the reuse of several small transformations, we employ calls to other constructors and reconfiguration rules, which are modeled for the corresponding component type. Calls to constructors are applied for component variables with ≪create≫ modifier. If such a component variable is classified by a part of a hierarchical component, one wants to specify the instantiation of the embedded component instances at once. This can be done with constructors. In our example the component variable /followers is associated with a call to the constructor initPosCtrl, which will be called after the successful execution of the transformation. Calls to reconfiguration rules basically provide the same features, but they are specified on component variables without modifier stereotypes. An example of a reconfiguration call is addPosCalc of the component variable /convoyLeader, which takes the newly created port instance pos and positionInConvoy as arguments.

Figure 7 shows this transformation which specifies the reconfiguration of the leader RailCab which needs a new position calculation component for the newly added RailCab. This transformation uses the formerly created port instance pos as well as the position in the convoy as parameters.

Since the task of addPosCalc is a little bit more challenging than the previous transformation, it applies *constraints* to some component variables. A negative constraint for a variable means that no instance of the variable's part must occur in the host graph. Negative constraints are visualized by crossing out the corresponding variable, e.g. the component variable prev of Story Pattern get first PosCalc. An optional constraint means that an instance of the variable's part can exist in the host graph, but the existence is not required for a successful binding. These constraints are denoted by dashed lines, see the component variable next and the port variable of first in Story Pattern add newCalc.

The transformation is basically divided into two parts. In the activities get first PosCalc and get next PosCalc, the component structure of the leading RailCab is searched for the correct place for adding the position calculation component of the new RailCab. This is specified as a loop in the control flow. The search starts with the first position calculation component in the first activity. This component is identified on having no previous position calculation component connected over next. Then the list of position calculation components is traversed until the appropriate place between the positionInConvoy-1-th component and the next one is found.

There, the new position calculation component is

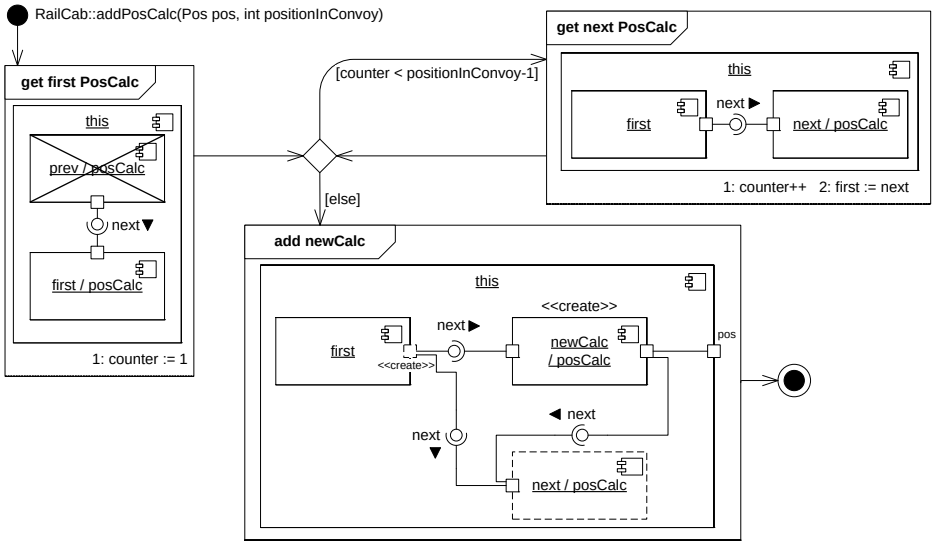


Figure 7. Adding a new RailCab to a convoy of at least two RailCabs.

inserted into the component structure as shown in activity add newCalc. This includes reconnecting the different components denoted by the various created connectors. Since the next connector type has source and target multiplicity of 1, the newly created connector replaces a possible connector between first and next. Note first, that the next component is optional, i.e. only if there is such a next component, the creation and deletion of the connectors associated with this component are executed. Note second, that also the outgoing port of first is optional and has to be created, if first is the last posCalc component. This allows adding a RailCab at the rear-end of the convoy.

Variables and links have compositional relationships among each other: Port variables are attached to component variables, and connector links are attached to port variables. This stands for more than just the fact that the deletion of the corresponding component instance implies the deletion of all attached port and connector instances. The modifier stereotypes and the constraints have to be taken over from the higher-level variable as well. In the Story Pattern add newCalc e.g. the ≪create≫ modifier needs just to be assigned to the component variable newRailcab, the modifier is automatically assigned to all attached port variables and connector links. This stems from the classical Story Patterns, where the destruction of an object destroys all its incident links and the instantiation of objects implicitly creates all in the Story Pattern modeled links of the corresponding object node [33] conforming to the single push out (SPO) approach [20].

The Story Pattern add newCalc completes the reconfiguration. We omit appropriate error handling activities due to space restrictions.

The presented transformations are used whenever a single RailCab enters a convoy of n RailCabs resulting in a convoy of length $n+1$. Therefore, this example transformation shows that our approach is not restricted to the specification of a predefined set of component configurations but can in principle be used to specify an infinite amount of configurations. See Section 5 for a discussion of the resulting issues when employing this approach in hard real-time systems. A precise definition including the complete metamodel of the described transformation language is given in [14].

4. Formal Semantics

The semantics of the transformation language must be rigorously defined. Based on these semantics, source code can be generated for the specified transformations and subsequently executed.

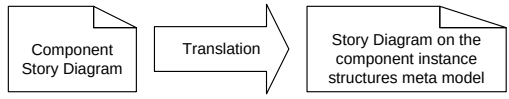


Figure 8. Semantics definition of the transformation language by translation

We define the semantics of the transformation language formally by translating them to Story Diagrams

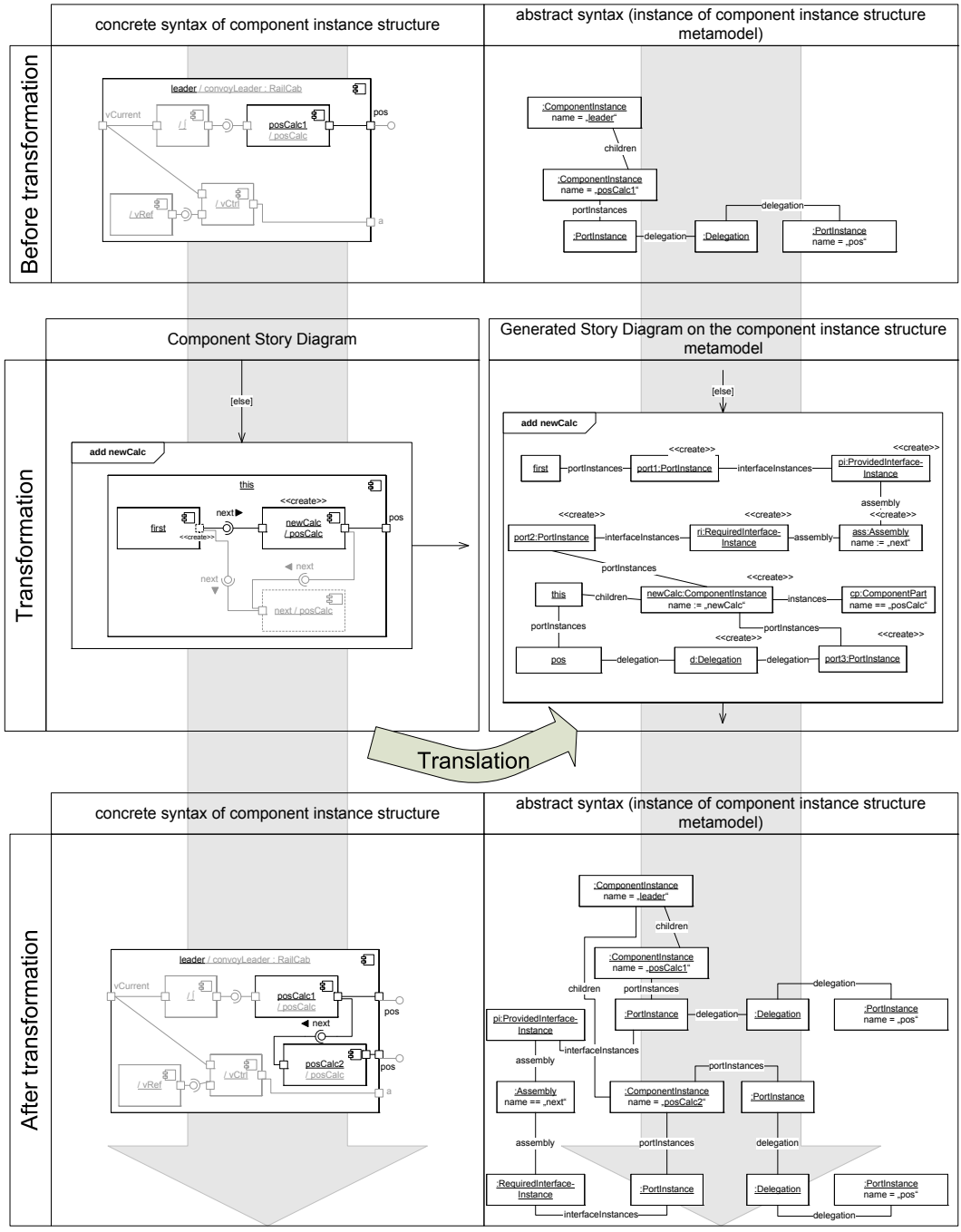


Figure 9. Translation from Component Story Diagrams to Story Diagrams on the component instance structure metamodel.

[7, 33] (cf. Section 3) on the component instance structure metamodel [14] (cf. Figure 8).

The translation of the presented transformation language onto Story Diagrams on the metamodel level enables us to reuse the existing techniques and tools for Story Diagrams like code generation.

Figure 9 shows the component structure of the leading RailCab before and after the transformation add newCalc has been executed. In the middle part of the figure, the transformation part concerning the addition of the new PosCalc component instance is depicted. The left hand side of Figure 9 shows the component structures and transformation in concrete syntax as presented in the previous sections. On the right side, the component instance structures are shown in abstract syntax, i.e. as instances of the component instance structure metamodel. The Story Diagram on the metamodel level in the middle of the right side is the result of several translation rules.[3]

For example, the shown PosCalc component instance on the left hand side is an instance of the ComponentInstance metaclass and the pos Port is an instance of the PortInstance metaclass. The specified transformation for creating the newCalc:PosCalc component instance inside the leading RailCab component instance is then executed by the creation of an instance of the ComponentInstance metaclass and the appropriate link creation to the component instance representing the leading RailCab. The creation of the connector between the newCalc:PosCalc component instance and the created pos port instance is realized by the instantiation of the Delegation metaclass. Finally, the creation of the next connector between the first component instance and the newCalc:PosCalc is implemented by a creation of the required and provided interface instances and the ass:Assembly metaobjects.

Figures 10 to 12 show a more detailed view on the translation of the this-variable, of the creation of newCalc/posCalc and of one of its port instances. We refer the reader to [14] for the details of the metamodels as well as the translation rules for the component story diagrams.

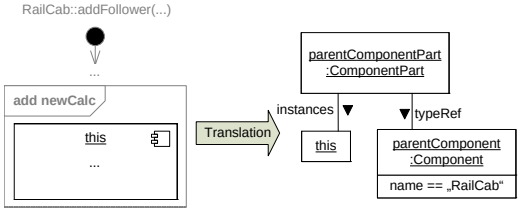


Figure 10. Translation of the this-variable

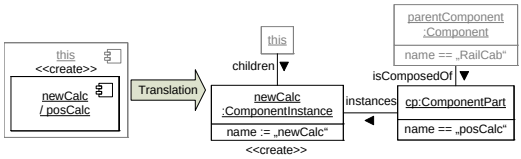


Figure 11. Translation of a component variable

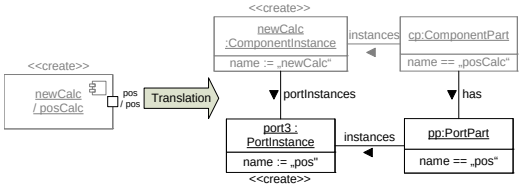


Figure 12. Translation of a port variable

5. Predictable Runtime Behavior

In the previous sections, we describe a modeling approach for adaptive component structures with graph transformation rules. In this section, we present the required techniques for using this approach in the domain of mechatronic systems. As these systems have real-time characteristics, the basic question is: how can we guarantee that all tasks are finished within their deadlines.

In order to guarantee to meet the required deadlines, the worst-case execution times (WCETs) of implementation artifacts must be known. In addition, WCETs are required for a schedulability analysis which is used to check whether concurrent processes are executable on a given processor meeting the required deadlines.

5.1. Determining Worst Case Execution Times

To determine the WCET of Story Diagrams, we have to eliminate modeling constructs which are unpredictable. The presented RailCab's software structure in Figure 3 shows an unbounded number of PosCalc instances. As the number of PosCalc runtime instances is not known while developing the system, the WCET of this model is not determinable.

To overcome this limitation, we define a maximum of multiplicities as discussed in [4]. This enables an implementation with an upper bound of the required data structures. These upper bounds determine a worst-case number of iterations (WCNIs)

[3] Parts of the left hand side figures are grayed out since they are not shown on the metamodel side of the figure due to space restrictions.

when searching in these data structures which lead to predictable real-time behavior.

The WCET of a Story Pattern does not only depend on the WCETs of its single code fragments and on the WCNIs when searching in data structures. The problem of WCET determination for Story Patterns is more complicated, because the order in which the elements of a Story Pattern are matched has significant impact on the resulting WCET as nested iterations can occur. The reason why different matching sequences usually lead to different WCETs is that the two multiplicities of a connector can be different. Another reason why different matching sequences usually lead to different WCETs is that the matching process explores in the worst-case a path for each existing instance when binding an instance that is connected to a bound instance via a to-many connector. To obtain an optimal WCET, the number of such paths has to be minimized. This is achieved by first respecting the path via connectors with low multiplicities.

As we specified exact multiplicities and thus know the upper bounds of the corresponding data structures, we can determine the WCET of a matching sequence. As usually multiple matching sequences exist, we will determine one of the matching sequences that will lead to the optimal WCET. In order to calculate the WCET of a matching sequence for a Story Pattern and in order to determine a matching sequence that optimizes the WCET of a Story Pattern, we use the algorithm presented in [4].

5.2. Scheduling

Transformation at run-time changes the used resources (e.g. cpu utilization or memory) of the application tasks online. In the literature, these changes are known as mode changes [28]. Additionally the transformation itself requires some resources. To guarantee that no timing constrains of the transformed task or other tasks of the system are violated due to the transformation itself or the afterwards changed task set a schedulability analysis is required before transformation.

The schedulability analysis in general is a NP-hard problem. However restricting the task set $\tau_1, ..., \tau_n$ to periodic tasks with fixed periods T_i, deadlines equal to period, an WCET C_i for each instance and using the Earliest Deadline First (EDF) scheduling strategy the schedulability analysis has only linear complexity regarding to the number of tasks n [19]. The ratio $U_i = C_i/T_i$ is called the utilization factor of τ_i and represents the fraction of processor time used by that task and $U_p = \sum_{k=1}^{n} U_i$ is the total processor utilization of the task set. If $U_p \leq 1$ the task set is schedulable with EDF. Even aperiodic tasks can be integrated without changing the complexity class of the schedubility analysis by integrating them with the Total Bandwidth Server (TBS) [30]. The basic idea of the TBS is to reserve some processor utilization U_s for aperiodic events, assign them a deadline and than schedule them in the same way as periodic tasks with EDF. As long as for the sum of utilization holds $U = U_p + U_s \leq 1$, the periodic tasks and the aperiodic tasks can be feasible scheduled together with no deadline violation.

The transformation is an aperiodic event and can be modeled as an aperiodic task. Through reservation of the TBS approach, the transformation can be feasible scheduled. If no transformation is used the reserved processor utilization can be put at other applications disposal by the Flexible Ressource Manager [24]. The Flexible Ressource Manager was developed to allow the distribution of unused but reserved resources. Criteria have been developed for the manager [25], which allow a immediate and not interrupted reconfiguration, which likewise can be applied to the transformation process if immediate transformation is required during run-time.

6. Conclusions and Future Work

We presented an approach which facilitates the component-based modeling of reconfigurable systems. The approach supports the definition of hierarchical component types as well as a transformation language to specify the reconfiguration of component structures. In contrast to related work, the approach supports a powerful specification language for transformations on component structures combined with a control flow specification to express if- and loop-constructs.

The sketched translation of component structure transformations based on the component instance structure metamodel formally defines the semantics of the transformation language. It additionally enables the execution of the transformations during design time by executing the Story Diagrams which are the result of the translation.

We sketched in Section 5 how we deal in principle with real-time issues like hard deadlines and schedulability. We are currently working on generating code from the transformations for a real-time operating system.

We presented in [8] an approach for the specification of real-time state-based behavior integrated with the reconfiguration of component structures. Component structures are embedded into the states of a state machine with real-time annotations. We are currently working to integrate the presented transformation lan-

guage into this state machine approach. The basic idea is that calls of the transformations can be attached to the transitions and states of the state machine.

In [14], tool support for the presented concepts and graphical specification languages was implemented and integrated into the eclipse version of our case tool Fujaba Real-Time Tool Suite[4].

References

[1] Basil Becker, Dirk Beyer, Holger Giese, Florian Klein, and Daniela Schilling. Symbolic Invariant Verification for Systems with Dynamic Structural Adaptation. In *Proc. of the* 28^{th} *International Conference on Software Engineering (ICSE), Shanghai, China*. ACM Press, 2006.

[2] Jeremy S. Bradbury, James R. Cordy, Juergen Dingel, and Michel Wermelinger. A survey of self-management in dynamic software architecture specifications. In *Proceedings of the 1st ACM SIGSOFT workshop on Self-managed systems (WOSS '04)*, pages 28–33, New York, NY, USA, 2004. ACM Press.

[3] Sven Burmester and Holger Giese. Visual Integration of UML 2.0 and Block Diagrams for Flexible Reconfiguration in Mechatronic UML. In *Proc. of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'05), Dallas, Texas, USA*, pages 109–116. IEEE Computer Society Press, September 2005.

[4] Sven Burmester, Holger Giese, Andreas Seibel, and Matthias Tichy. Worst-case execution time optimization of story patterns for hard real-time systems. In *Proc. of the 3rd International Fujaba Days 2005, Paderborn, Germany*, pages 71–78, September 2005.

[5] Sven Burmester, Holger Giese, and Matthias Tichy. Model-Driven Development of Reconfigurable Mechatronic Systems with Mechatronic UML. In Uwe Assmann, Arend Rensink, and Mehmet Aksit, editors, *Model Driven Architecture: Foundations and Applications*, volume 3599 of *Lecture Notes in Computer Science*, pages 47–61. Springer Verlag, August 2005.

[6] Frank Drewes, Berthold Hoffmann, and Detlef Plump. Hierarchical graph transformation. *J. Comput. Syst. Sci.*, 64(2):249–283, 2002.

[7] T. Fischer, J. Niere, L. Torunski, and A. Zündorf. Story Diagrams: A new Graph Rewrite Language based on the Unified Modeling Language. In G. Engels and G.Rozenberg, editors, *Proc. of the* 6^{th} *International Workshop on Theory and Application of Graph Transformation (TAGT), Paderborn, Germany*, LNCS 1764. Springer Verlag, 1998.

[8] Holger Giese, Sven Burmester, Wilhelm Schäfer, and Oliver Oberschelp. Modular Design and Verification of Component-Based Mechatronic Systems with Online-Reconfiguration. In *Proc. of 12th ACM SIGSOFT Foundations of Software Engineering 2004 (FSE 2004), Newport Beach, USA*, pages 179–188. ACM Press, November 2004.

[9] Holger Giese, Matthias Tichy, Sven Burmester, Wilhelm Schäfer, and Stephan Flake. Towards the Compositional Verification of Real-Time UML Designs. In *Proc. of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering (ESEC/FSE-11)*, pages 38–47. ACM Press, September 2003.

[10] Klaus Grimm. Software technology in an automotive company: major challenges. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, pages 498–503, Washington, DC, USA, 2003. IEEE Computer Society.

[11] Lars Grunske. Transformational Patterns for the Improvement of Safety. In *Proc. of the The Second Nordic Conference on Pattern Languages of Programs (VikingPLoP 03)*. Microsoft Buisness Press, 2003.

[12] Bernd Hardung, Thorsten Kölzow, and Andreas Krüger. Reuse of software in distributed embedded automotive systems. In *EMSOFT '04: Proceedings of the 4th ACM international conference on Embedded software*, pages 203–210, New York, NY, USA, 2004. ACM Press.

[13] Christian Henke, Henner Vöcking, Joachim Böcker, Norbert Fröhleke, and Ansgar Trächtler. Convoy operation of linear motor driven railway vehicles. In *Proceedings of the Fifth International Symposium on Linear Drives for Industry Applications, Japan*, 2005.

[14] Jörg Holtmann. Graphtransformationen für komponentenbasierte Softwarearchitekturen. Master's thesis, University of Paderborn, Germany, 2008.

[15] Mohamed Hadj Kacem, Ahmed Hadj Kacem, Mohamed Jmaiel, and Khalil Drira. Describing dynamic software architectures using an extended uml model. In *SAC '06: Proceedings of the 2006 ACM symposium on Applied computing*, pages 1245–1249, New York, NY, USA, 2006. ACM Press.

[16] Gabor Karsai, Aditya Agrawal, Feng Shi, and Jonathan Sprinkle. On the use of graph transformation in the formal specification of model interpreters. *Journal of Universal Computer Science*, 9(11):1296–1321, November 2003.

[17] H.J. Köhler, U. Nickel, J. Niere, and A. Zündorf. Integrating UML Diagrams for Production Control Systems. In *Proc. of the* 22^{nd} *International Conference on Software Engineering (ICSE), Limerick, Ireland*, pages 241–251. ACM Press, 2000.

[18] Jeff Kramer and Jeff Magee. Self-managed systems: an architectural challenge. In *FOSE '07: 2007 Future of Software Engineering*, pages 259–268, Washington, DC, USA, 2007. IEEE Computer Society.

[4] `http://wwwcs.uni-paderborn.de/cs/fujaba/projects/realtime/`

[19] C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 20(1):46–61, 1973.

[20] Michael Löwe and Hartmut Ehrig. Algebraic approach to graph transformation based on single pushout derivations. In Rolf H. Möhring, editor, *Graph-Theoretic Concepts in Computer Science, 16rd International Workshop, WG '90, Berlin, Germany, June 20-22, 1990, Proceedings*, volume 484 of *Lecture Notes in Computer Science*, pages 338–353. Springer, 1990.

[21] Jeff Magee, Naranker Dulay, and Jeff Kramer. Regis: A constructive development environment for distributed programs. *Distributed Systems Engineering Journal*, 1(5):304–312, September 1994.

[22] Daniel Le Métayer. Describing software architecture styles using graph grammars. *IEEE Transactions on Software Engineering*, 24(7):521–533, 1998.

[23] U. Nickel, J. Niere, and A. Zündorf. Tool demonstration: The FUJABA environment. In *Proc. of the* 22^{nd} *International Conference on Software Engineering (ICSE), Limerick, Ireland*, pages 742–745. ACM Press, 2000.

[24] Simon Oberthür and Carsten Böke. Flexible resource management – a framework for self-optimizing real-time systems. In Bernd Kleinjohann, Guang R. Gao, Hermann Kopetz, Lisa Kleinjohann, and Achim Rettberg, editors, *Proceedings of IFIP Working Conference on Distributed and Parallel Embedded Systems (DIPES'04)*. Kluwer Academic Publishers, 23 - 26 August 2004.

[25] Simon Oberthür and Hermann-Simon Lichte. Schedulability criteria and analysis for dynamic and flexible resource management. In *Proceedings of the DASMOD Workshop on Formal Verification of Adaptive Systems*, 2007.

[26] Object Management Group. *UML 2.0 Superstructure Specification*, October 2004. Document: ptc/04-10-02 (convenience document).

[27] Object Management Group. *Systems Modeling Language (SysML) Specification*, May 2006.

[28] Jorge Real and Alfons Crespo. Mode change protocols for real-time systems: A survey and a new proposal. *Real-Time Syst.*, 26(2):161–197, 2004.

[29] G. Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformation : Foundations*. World Scientific Pub Co, February 1997. Volume 1.

[30] Marco Spuri and Giorgio C. Buttazzo. Scheduling aperiodic tasks in dynamic priority systems. *Real-Time Systems*, 10(2):179–210, 1996.

[31] Clemens Szyperski. *Component Software, Beyond Object-Oriented Programming*. Addison-Wesley, 1998.

[32] Gabriele Taentzer, Michael Goedicke, and Torsten Meyer. Dynamic change management by distributed graph transformation: Towards configurable distributed systems. In *TAGT'98: Selected papers from the 6th International Workshop on Theory and Application of Graph Transformations*, pages 179–193, London, UK, 2000. Springer-Verlag.

[33] A. Zündorf. *Rigorous Object Oriented Software Development*. University of Paderborn, 2001.

A. Language Overview

For a short reference, the tables 1, 2, and 3 summarize the control flow and the Component Story Pattern elements of the transformation language and the supported modifiers and constraints, respectively.

type::name (parameter : parameterType, ...) : resultType	Start activity: Belongs to a component type and specifies parameters and result type.
return: a	Stop activity, which additionally allows the specification of a return value
	Decision/merge node
[success]	Transition which is taken, when the source transformation has been successfully executed.
[failure]	Transition which is taken, when the source transformation could not be executed.
[boolean expression]	Transition: Can have a guard with a boolean expression.
	Activity: Contains a Component Story Pattern or plain code.
[each time] [end]	forEach-activity: Contains a Component Story Pattern. Will be executed for each possible binding of the contained variables.

Table 1. Control flow elements

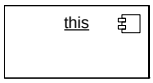
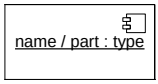
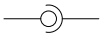
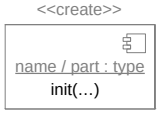
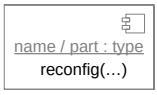
	this-variable: Classified by the component activity diagram's component type.
	Component variable: Classified by a component part. Can have modifier stereotypes and constraints.
	Expression for comparing the value of an attribute of the component instance with a specified value.
	Port variable: Attached to a component variable and classified by a port part. Can have modifier stereotypes and constraints.
	Assembly link: Connects two port variables of embedded component variables and is classified by an assembly type. Can have modifier stereotypes and constraints.
	Delegation link: Connects two port variables of the this-variable and an embedded component variable. Classified by a delegation type. Can have modifier stereotypes and constraints.
	Constructor call: Can be assigned to component variables with ≪create≫ modifier.
	Reconfiguration call: Can be assigned to component variables without modifier.
`1: code statement`	Code statement which is directly copied to the generated code.

Table 2. Component Story Pattern elements

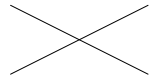
≪create≫	Create the attached component, port or connector variable
≪destroy≫	Destroy the attached component, port or connector variable
	Negative element: Element which must not exist.
	Optional element: Element which may or may not exist.

Table 3. Modifiers and constraints for Component Story Pattern elements