Component Story Diagrams in Fujaba4Eclipse*

Jörg Holtmann, Matthias Tichy
Software Engineering Group
University of Paderborn
Warburger Str. 100
D-33098 Paderborn, Germany
[chrome|mtt]@uni-paderborn.de

ABSTRACT

A current trend in Software Engineering is the development of self-adaptive component-based software architectures. Our modeling approach for component-based software systems MECHATRONIC UML, integrated in the FUJABA Real-Time Tool Suite plugin for Fujaba4Eclipse, supports self-adaptiveness only to a certain degree. This paper presents tool support for an extension of MECHATRONIC UML, which facilitates initialization and reconfiguration of a MECHATRONIC UML system based on Story Diagrams and thus enables a step towards self-adaptiveness on a structural level.

1. INTRODUCTION

Today's software systems are mainly build in a component-based fashion to reduce their complexity. Furthermore, some software systems have to be self-adaptive, which means that they adapt their behavior in response to event occurrences or changes in the environment. The adaption can be realized by changing system parameters or by a structural reconfiguration. MECHATRONIC UML [1] supports, among other things, the modeling of components with real-time behavior, the coordination between distributed components as well as its verification, but only a limited structural reconfiguration by enumerating all different configurations.

In [6], we presented concepts for an extension to MECHATRONIC UML, which overcomes this limitation. This paper presents the implementation of these concepts in Fujaba4Eclipse. The general idea is to use a formalism based on Story Diagrams [2] to model system initialization and reconfiguration for a component-based architecture. Component instance structures consist in contrast to object structures of elements such as component and port instances and different connector types. Furthermore, conventional Story Diagrams operate only on flat object structures and do not provide the possibility to traverse a hierarchy, which is induced by a component-based system. So traditional Story Diagrams are not well suited for component-based architectures, but their variant *Component Story Diagrams* [6] make use of both the Story Diagrams' formal foundations and sophisticated control structures as well as the extensions required by component-based architectures. The developed formalism reuses the graphical syntax of component diagrams for a tight integration of the modeling of structure as well as reconfiguration behavior. We refer to [3, 6] for a discussion of the limitations of related approaches.

The running example used in this paper is taken from [6] and describes the communication structure of autonomous railway vehicles from the RailCab project[1]. These RailCabs can build convoys to reduce the energy consumption by utilizing the slipstream. Their communication structure is build by MECHATRONIC UML components. The leading RailCab computes reference positions for all subsequent RailCabs in a convoy. For each following RailCab, one position calculation component is deployed in the leading RailCab. This component sends the reference position to its connected following RailCab. Thus, a structural reconfiguration is required for the leading RailCab's embedded components when a RailCab joins or leaves a convoy.

Section 2 sketches a flexible component type definition as a base for the modeling of Component Story Diagrams. While Section 3 presents the Component Story Diagrams themselves, Section 4 introduces the code generation plugin, which is a prerequisite for the execution of the Component Story Diagrams. Section 5 concludes this paper and adduces current and future work on Component Story Diagrams.

2. COMPONENT TYPE DEFINITION

Like conventional Story Diagrams, also Component Story Diagrams require a type definition as a basis for modeling. Instead of classes, component types are employed as classifying model. In [6] we presented a variable component type definition based on the UML Composite Structures [4]. Component types consist of *parts*, which are also classified by a component type and have a multiplicity. So parts represent a set of instances classified by the part's component type, which an instance of the superordinate component type may contain by composition. Additionally, ports are provided with a multiplicity.

Figure 1 shows the embedded component structure for the component type RailCab, holding the behavior concerning computation of reference positions for subsequent RailCabs and adaptation of the own position to a reference position. Besides some parts with a default multiplicity of 0..1 for component instances which control and determine the position or the velocity of a RailCab, the part posCalc—classified by the component type PosCalc which contains the behavior for calculating reference positions for following RailCabs—is deployed within RailCab. This part has a multiplicity of

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

[1] www.railcab.de/en

$*$ (denoted by the second frame), so it stands for an arbitrary amount of embedded component instances of the type PosCalc. One of its regular ports is connected via a delegation connector type PosRef to a so-called *multiport* with multiplicity $*$ attached to the superordinate component type RailCab. Instances of this multiport are used for connections between the embedded position calculation component instances with component instances representing the following RailCabs. Since the delegation connector type PosRef has a source and target multiplicity of 1, each component instance of the part posCalc may be connected via a delegation connector :PosRef exactly to one instance of the corresponding multiport. In the same manner, component instances of the part posCalc may be connected among each other via assembly connectors :Next. That way, a position calculation component instance gets the reference position of the preceding RailCab as input for the calculation of the reference position of the subsequent RailCab.

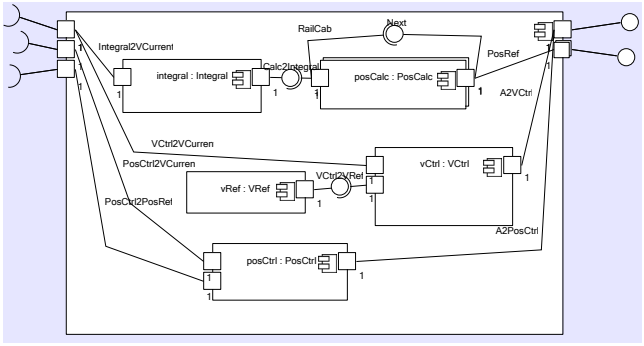

Figure 1: Component type RailCab [3]

Figure 2 depicts the system level ConvoySystem, which describes all allowed system configurations with interconnected RailCab parts: the assembly connector type PosRef connects the afore mentioned multiport of the part convoyLeader with multiplicity 0..1 to a regular port of the part convoyFollowers with multiplicity of $*$. This means that there is at most one instance of the role convoyLeader, which is connected by n :PosRef connectors to n convoyFollowers instances. In this way, the latter ones can receive the reference positions calculated by the corresponding posCalc instances, which are deployed within convoyLeader. Together with the internal definition of the component type RailCab, this builds the component-based communication structure for RailCab convoys.

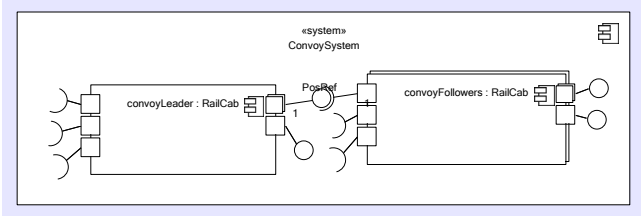

Figure 2: System level ConvoySystem [3]

For a more detailed view on the scenario and the semantics of the component types the reader is referred to [6]. For a precise description of the MECHATRONIC UML component type definition based on the UML Composite Structures and the differences to the previously used type modeling see [3].

3. COMPONENT STORY DIAGRAMS

Story Diagrams [2], a graph transformation formalism, offer a wide variety of features for the model-based specification of object-oriented behavior such as object/link creation/deletion, control flow structures and an appropriate visual representation. However, the structures conventional Story Diagrams operate on are simple objects connected by links. Components have ports connected among each other by assembly and delegation connectors. Secondly, Story Diagrams cannot traverse hierarchies of objects. In contrast to that, components explicitly are defined by composition and are arranged by other, more granular components and thus span a hierarchy. So the traditional Story Diagram approach does not align well with component-based architectures.

In [6], we introduced the new transformation language Component Story Diagrams, which combine the intuitive but formal notation of conventional Story Diagrams with the previously sketched, extended MECHATRONIC UML type definition. Instead of objects and links, component and port instances are created or destroyed and interconnected via assembly and delegation connector links. Calls to other Component Story Diagrams are supported to traverse the different hierarchy levels of a system configuration and to respect the encapsulation of components at the same time. The callee is defined either for the same component type or for a component type residing one hierarchy level below.

Figure 3 shows the Component Story Diagram initConvoySystem associated with ConvoySystem. This Component Story Diagram initializes a convoy system consisting of one component instance leader classified by convoyLeader (denoted by leader/convoyLeader) and several component instances /convoyFollowers (cf. Figure 2). The value of the parameter followers:Integer specifies the amount of /convoyFollowers instances to be created.

To achieve this behavior, firstly in Story Pattern createLeader the component instance leader/convoyLeader including two port instances is created by defining a corresponding component variable with ≪create≫ modifier and two attached port variables. Afterwards, the Component Story Diagram initConvoyLeader is called for the newly created component instance. This Component Story Diagram is defined for the component type RailCab residing one hierarchy level below and initializes the embedded configuration for leader. The subsequent statement activity creates a counter variable to create the required /convoyFollowers instances in a loop. The Story Pattern createNextFollower is executed as long as the loop condition holds. In this Story Pattern, a new component instance follower/convoyFollowers including three port instances is created and connected via the assembly connector :PosRef to a new port instance of the already bound leader. Whereas the call to initConvoyFollower initializes the embedded configuration of follower, the call to insertPosCalc changes the existing configuration of leader by deploying a new component instance /posCalc. To obtain the latter behavior, a name, the actual position, and the newly created port instance pos of leader are assigned as arguments.

The Fujaba4Eclipse editor for Component Story Diagrams is syntax-driven and context-sensitive w.r.t. the component type definition. While modeling a Component Story Dia-

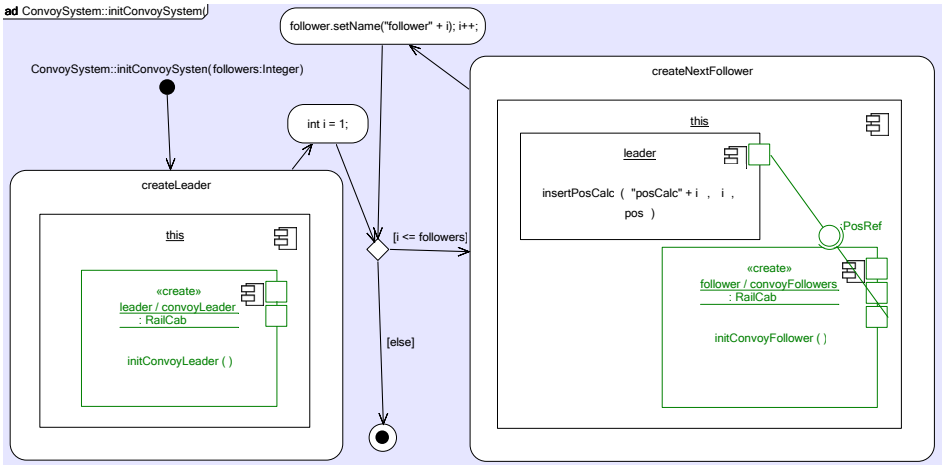

Figure 3: Component Story Diagram initConvoySystem [3]

gram for ConvoySystem, for example, as shown in Figure 3, the user can only assign parts to component variables which are defined in ConvoySystem, namely convoyLeader and convoyFollowers (cf. Figure 2). Both parts are interconnected by the assembly connector type PosRef attached to certain port types. Thus, you can only assign this connector type to the assembly link used in Story Pattern createNextFollower. Component variables can be associated with calls to Component Story Diagrams; the callee is selected from the Component Story Diagrams defined for the corresponding component type RailCab of the variable's part (cf. Table 1). Arguments representing a port variable, for example, have to be selected from the set of possible variables w.r.t. the signature of the callee. Further and in particular more complex Component Story Diagrams can be found in [3].

4. CODE GENERATION

To make use of existing code generation techniques, Component Story Diagrams are translated to conventional Story Diagrams based upon the MECHATRONIC UML metamodel. The translation is defined by TGG [5] rules, which thus describe the formal semantics of Component Story Diagrams. FUJABA's TGG plugins [7] are employed as concrete modeling and execution environment. A new code generation plugin takes the MECHATRONIC UML component type definition and Component Story Diagrams as input and transforms them using the translation rules to Story Diagrams classified by the metamodel.

Figure 4 shows a part of the Story Diagram which is generated from the Component Story Diagram of Figure 3.[2] The translation is for the most part straightforward. The control flow is basically unchanged. All variables in a Component Story Pattern are translated to objects of classes of the metamodel; for example, the component variable this is translated to the object parentComponentInstance:ComponentInstance. The variables' stereotypes are translated accordingly. Note the special case of the leader component variable translation. The translation honors the component type definition of Figure 2 which specifies that a convoy may only contain one leader RailCab. Adopting conventional Story Pattern semantics, an additional optional object leaderComponentInstanceToDelete is added to the generated Story Pattern which removes the old leader when a new one is created. Statement activities are taken over without modifications.

A more detailed view on the translation is sketched in [6], while the complete translation and an example of a translated Story Diagram is given in [3].

Executable code is generated from the translated Story Diagrams. Table 1 lists an overview of the used variables and links in Component Story Diagrams (second column) as well as in the translated Story Diagrams (third column) and of the generated Lines of Code (last column) for all evaluated Component Story Diagrams. This overview indicates the reduction of complexity by using Component Story Diagrams on MECHATRONIC UML component structures instead of modeling Story Diagrams based upon the MECHATRONIC UML metamodel. Note however that the generated LOC are not only related to the actual amounts of variables and links but to a greater extent to loops and iterations over $*$-associations.

The generated code can be executed at design time on component instance diagrams in Fujaba4Eclipse.

5. CONCLUSION AND FUTURE WORK

This paper introduced tool support for the modeling and the execution of sophisticated structural reconfigurations for component-based architectures by presenting an extended MECHATRONIC UML component type definition and a context-sensitive editor as well as a code generation plugin for Component Story Diagrams.

Currently there are two important open issues. First, Component Story Diagrams do not support negative or optional variables like conventional Story Diagrams. Since there are compositional dependencies between components and attached ports and between ports and linked connectors, the scope of a negative component variable, for example, includes also compositionally dependent variables. These have to be translated to a group of connected negative object vari-

[2]Actually, it is not a conventional Story Diagram but a slightly extended version which we chose due to implementation issues. The only difference in this example is the *transformation call* node. This node executes another Story Diagram after all specified changes like creating and destroying edges and nodes have been executed.

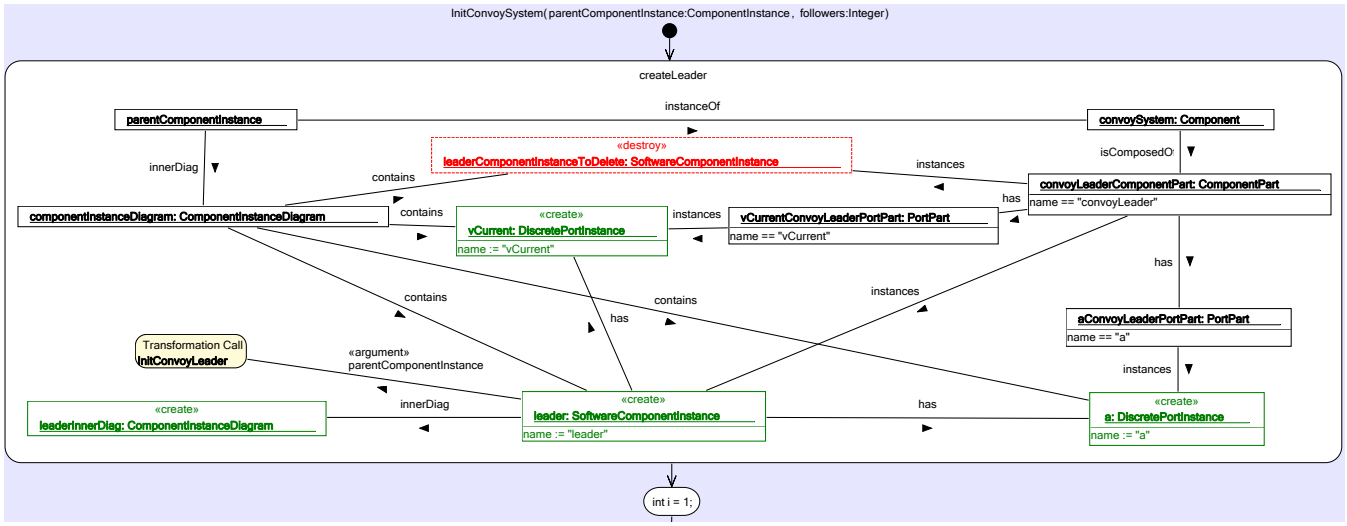

Figure 4: Excerpt of Story Diagram generated from Component Story Diagram initConvoySystem

Comp. Story Diag.	Variables and connector links	Object variables and links	LOC
ConvoySystem::initConvoySystem	12	89	760
ConvoySystem::insertFollower	8	69	605
ConvoySystem::removeFollower	9	71	618
RailCab::initConvoyFollower	11	89	695
RailCab::initConvoyLeader	15	137	968
RailCab::insertPosCalc	52	482	2806
RailCab::removePosCalc	42	389	2422
Overall	**149**	**1326**	**8874**

Table 1: Comparison of amounts of variables/links in Component Story Diagrams and in translated Story Diagrams and of generated Lines of Code [3]

ables in the resulting translated Story Diagram. Until now, conventional Story Diagrams forbid such negative variable groups, what also affects groups of optional variables. A concept has already been developed to solve this problem [3] but has not been implemented yet. Secondly, the presented Component Story Diagrams are selected and executed at design time by an user. To obtain a self-adaptive approach for mechatronic real-time systems, we are currently working on an integration of Component Story Diagrams into real-time statecharts.

Future works could include extensions like attributes as well as an inheritance concept for component types to provide Component Story Diagrams with attribute conditions and the matching of instances of derived component types.

6. REFERENCES

[1] S. Burmester, H. Giese, and M. Tichy. Model-Driven Development of Reconfigurable Mechatronic Systems with Mechatronic UML. In U. Assmann, A. Rensink, and M. Aksit, editors, *Model Driven Architecture: Foundations and Applications*, volume 3599 of *Lecture Notes in Computer Science*, pages 47–61. Springer Verlag, Aug. 2005.

[2] 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.

[3] J. Holtmann. Graphtransformationen für komponentenbasierte Softwarearchitekturen. Master's thesis, University of Paderborn, Germany, April 2008.

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

[5] A. Schürr. Specification of Graph Translators with Triple Graph Grammars. In *Proc. of the* 20^{th} *International Workshop on Graph-Theoretic Concepts in Computer Science*, Herrschin, Germany, June 1994. Spinger Verlag.

[6] M. Tichy, S. Henkler, J. Holtmann, and S. Oberthür. Component Story Diagrams: A Transformation Language for Component Structures. In *Postproc. of the 4th Workshop on Object-oriented Modeling of Embedded Real-Time Systems (OMER 4), Paderborn, Germany*, 2008.

[7] R. Wagner. Developing Model Transformations with Fujaba. In H. Giese and B. Westfechtel, editors, *Proc. of the* 4^{th} *International Fujaba Days 2006, Bayreuth, Germany*, volume tr-ri-06-275 of *Technical Report*, pages 79–82. University of Paderborn, September 2006.