Leif Geiger, Holger Giese, Albert Zündorf (Eds.)

FUJABA Days 2007

08th -09th October 2007
Kassel, Germany

Proceedings

Volume Editors

Dipl. Inf. Leif Geiger
University of Kassel
Research Group Software Engineering
Department of Computer Science and Electrical Engineering
Wilhelmshöher Allee 73, 34121 Kassel, Germany
leif.geiger@uni-kassel.de

Jun.-Prof. Dr. Holger Giese
University of Paderborn
Department of Computer Science
Warburger Straße 100, 33098 Paderborn, Germany
hg@uni-paderborn.de

Prof. Dr. Albert Zündorf
University of Kassel
Chair of Research Group Software Engineering
Department of Computer Science and Electrical Engineering
Wilhelmshöher Allee 73, 34121 Kassel, Germany
zuendorf@uni-kassel.de

Program Committee

Program Committee Chairs

Program Commitee Members

Editor´s preface

Fujaba is an Open Source UML CASE tool project started at the software engineering group of Paderborn University in 1997. It initially combined features of commercial “Executable UML” CASE tools with rule-based visual programming concepts adopted from its ancestor, the graph transformation tool PROGRES. In 2002, Fujaba was redesigned and became the Fujaba Tool Suite with a plug-in architecture allowing developers to add functionality easily while retaining full control over their contributions.

Fujaba followed the model-driven development philosophy right from its beginning in 1997. At the early days, Fujaba had a special focus on code generation from UML diagrams resulting in a visual programming language with a special emphasis on object structure manipulating rules. Today, at least seven rather independent tool versions are under development in Paderborn, Kassel, Darmstadt, Siegen, and Bayreuth for supporting (1) reengineering, (2) embedded real-time systems, (3) education, (4) specification of distributed control systems, (5) integration with the Eclipse platform, (6) MOF-based integration of system (re-) engineering tools, and (7) adaptable and integrated management of development processes. Several research groups have also chosen Fujaba as a platform for UML and MDA related research activities. In addition, quite a number of Fujaba users send us requests for extensions and improvements.

The 5th International Fujaba Days aim at bringing together Fujaba developers and Fujaba users from all over the world to present their ideas and projects and to discuss them with each other and with the Fujaba core development team. It is the second time the Fujaba Days take place in Kassel. The very first event was also held in Kassel in 2003. This year the Fujaba Days are collocated with the third International Workshop and Symposium on Applications of Graph Transformation with Industrial Relevance (AGTIVE 2007). The AGTIVE is held directly after the Fujaba Days (10th – 12th October). It is the biggest purely practice-oriented event of the Graph Transformation community. It serves as a forum for all those scientists of the graph transformation community that are involved in the development of graph transformation tools and the application of graph transformation techniques - usually in an industrial setting. This is especially true for most of the Fujaba users and developers. So we hope, that many discussions started at the Fujaba Days will be continued at the AGTIVE.

We received 10 papers, which were reviewed carefully by the program committee. 8 submissions were accepted for presentation. The papers are organized into sections on meta models, verification and validation, tool building and frameworks.

The workshop programme is decorated with an invited talk given by Arend Rensink, University of Twente, The Netherlands. Two Developer Sessions and one panel discussion on current and future work complete the workshop’s programme. The Developer Sessions provide a forum for Fujaba developers who can´t stop programming and are eager to learn about the most recent developments of their colleagues (and also like to demonstrate their own improvements).

The PC chairs would like to thank the PC members for their careful work in reviewing the papers and contributing to the quality of the Fujaba Days in this way. We hope that the workshop will be held in a lively atmosphere which encourages discussion and exchange of ideas.

Leif Geiger, Holger Giese & Albert Zündorf
Organizers

Table of Contents

Harmless Metamodel Extensions with Triple Graph Grammars

Jendrik Johannes
Technische Universität Dresden
Software Technology Group
01062 Dresden, Germany
jendrik.johannes@tu-dresden.de

Tobias Haupt
Technische Universität Dresden
Software Technology Group
01062 Dresden, Germany
s0500251@inf.tu-dresden.de

ABSTRACT

Designing Domain Specific Languages through metamodeling is an emerging engineering technique in software development. Language development, however, is also always tool development—which can be costly when languages are developed from scratch. Costs can be saved by developing new languages as extensions of existing ones—effectively by extending metamodels. Here, existing tools, developed for the existing language, can be reused to a certain degree. We argue that a certain group of metamodel extensions is *harmless*: keeping existing tools functional, while integrating new tools to handle additional functionalities. To realize a platform for this, two things are required: 1) a transformation engine for the metamodel extension and 2) a synchronization mechanism for tool integration. In this paper we show how Triple Graph Grammars can be used to define both and introduce an interpreter for these grammars that works in an environment based on the Eclipse Modeling Framework.

1. INTRODUCTION

In contrast to general-purpose modeling language development, Domain Specific Language (DSL) development has to be much more cost-efficient. High development costs are not acceptable since each DSL is designed for a specific problem domain with a limited set of application possibilities. A main issue for high development costs is tool building, which is crucial to render newly designed languages usable. Costs can be reduced by reusing existing tools where feasible instead of developing new ones from scratch. One way to achieve this is to reuse metamodels as well—extending existing ones rather than designing new ones.

Metamodel extension seems to be a promising direction for simplifying DSL design and implementation. Especially, if such extensions can be captured in a generic formalism for extending arbitrary metamodels in a similar fashion. Then tools based on the original metamodels and tools that know about the specific extension can be used in combination, without needs to develop new tooling. We have identified one such extension formalism—the Reuseware formalism [4][1]—which we believe of strong importance: adding reuse abstractions to a language. That is, adding notions of components, modules, aspects, or similar to leverage the reuse of artifacts written in that language.

[1]The formalism was defined for context-free grammars here. It was ported to metamodels, but publication is still pending.

The Reuseware extension of a metamodel is *harmless*: 1) all original language syntax and semantics are preserved and 2) the added features are syntactically closely related to original ones. For a harmless extension, reuse of the original modeling tools to design models is applicable: 1) the original language is supported anyway and 2) the new constructs can be mimicked by existing ones using naming conventions or an escape mechanism, like annotations or comments, if available.

Nevertheless, it is desired to translate such models into models conforming to the extended metamodel such that the extension specific (in this case the Reuseware specific) elements can be clearly identified. This allows further processing of the models (in this case by composition editors and engines). This dualism of tools working on two different representatives of the same model requires model synchronization.

In this paper we propose to utilize Triple Graph Grammars (TGG) [7, 5] for both, the metamodel extension and the synchronization of models. To realize this, we require a TGG environment which integrates into our tool chain. We are aiming for solutions based on the Eclipse Modeling Framework (EMF) [1]. Since there is no TGG engine available for EMF that supports model synchronization, we developed, driven by our use case, the TGG interpreter *Tornado*.

The remainder of this paper is structured as follows. In Section 2 the metamodel extension and model synchronization scenario within the Reuseware composition context is introduced. Following this, requirements for a TGG engine and their realizations in the Tornado engine are described in Section 3. Some related work is discussed in Section 4. Section 5 concludes and points at future directions of work.

2. METAMODEL EXTENSION AND MODEL SYNCHRONIZATION SCENARIO

We argue that through harmless metamodel extension a language can be enriched with new powerful features while preserving compatibility with original tooling. Such a harmless extension is the Reuseware formalism [4]. In this section we describe how Triple Graph Grammars are used 1) to perform the Reuseware extension on arbitrary metamodels and 2) to preserve tool compatibility.

2.1 Metamodel Extension

A metamodel extension can be expressed through a model transformation, which in turn can be formulated as a TGG.

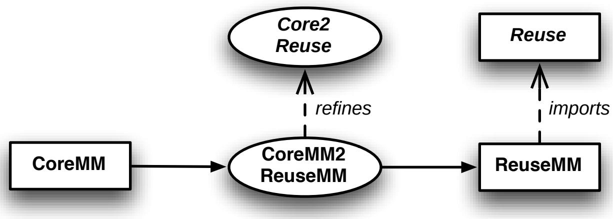


Figure 1: Reuseware metamodel extension formalism

Usinig this notion, the basic idea of the Reuseware metamodel extension is illustrated in Figure 1. A metamodel `CoreMM` of the original language (the *core metamodel*) is extended by the transformation `CoreMM2ReuseMM` to an extended metamodel `ReuseMM` (the *reuse metamodel*). The transformation is a specialization of a generic transformation (`Core2Reuse`) and the extended metamodel imports a metamodel defining the basic Reuseware concepts (`Reuse`).

For illustrative purposes, we will use a simplified version of the Reuseware metamodel extension formalism in this paper. The main idea is to introduce metaclasses for variation point definition[2] for a selected set of existing metaclasses. Variation points can be seen as placeholders for concrete instances of metaclasses. They can be used to define template-like incomplete models (so-called *model fragments*) that can be reused to compose complete models. A variation point is typed with the metaclass it stands for. That is, it can only be replaced by an instance of that metaclass during composition. This is reflected in the formalism, which can be formulated in abstract TGG rules. One rule is shown in Figure 2, where a metaclass (`Reusable`) is introduced as superclass of the original metaclass and the corresponding variation point metaclass. The latter additionally inherits features from the abstract concept of variation point. The other rules that complete the formalism are not shown.

As a concrete example, consider an UML-like metamodel defining the metaclasses `Package` and `Class` into which we would like to introduce variation points for classes. We specialize the TGG rule from Figure 2 to match the metaclass with the name `Class` by adding the constraint `name == "Class"` to the `coreEClass`. Application of the grammar, interpreted as a left-to-right transformation, gives us the extended metamodel, where `ClassVariationPoint`s are available as alternatives for `Class`es. Note that the specialization of the rule can be automated by a simple wizard tool, where the developer only states for which metaclasses variation points shall be introduced. This effectively hides the (sometimes complex) TGG rules from the developer.

2.2 Model Synchronization

The second, and more challenging, application of TGGs is the synchronization between *models* (instances of the core metamodel) and *model fragments* (instances of the reuse metamodel). The synchronization is required because models are used to impersonate model fragments which allows the reuse of existing editors. The real model fragment, used in Reuseware specific tools, then describes conceptual the same as the impersonator and both have to be synchronized. While we focus on editor integration in this section, it is also beneficial to enable synchronization between fragments and composed models. That is a powerful and desired feature in a Reuseware environment, since it allows to modify fragments directly in a composition result. It can be achieved by realizing the composition algorithm in terms of TGGs as well.

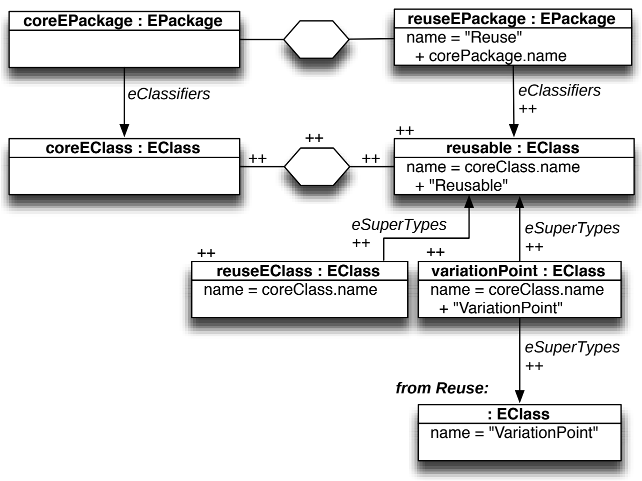


Figure 2: Operational TGG rule to introduce a variation point metaclass (in EMF-based metamodels)

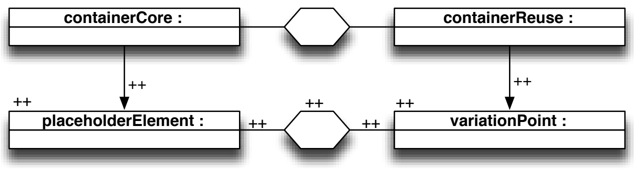
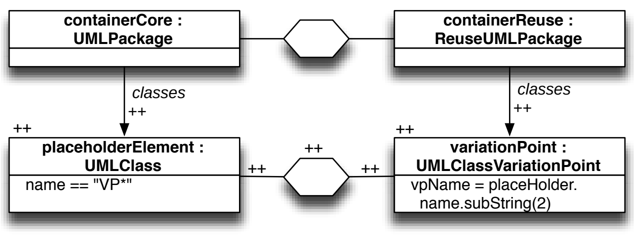


Figure 3: Generic synchronization rule for variation points



containerCore : UMLPackage
containerReuse : ReuseUMLPackage
classes
++
classes
++
++
++
++
placeholderElement : UMLClass
name == "VP*"
++
++
variationPoint : UMLClassVariationPoint
vpName = placeHolder.
name.subString(2)

Figure 4: Specialized synchronization rule for UML class variation points

[2]The full Reuseware formalism distinguishes different kinds of variation points and allows structuring and grouping of them as demonstrated in [3].

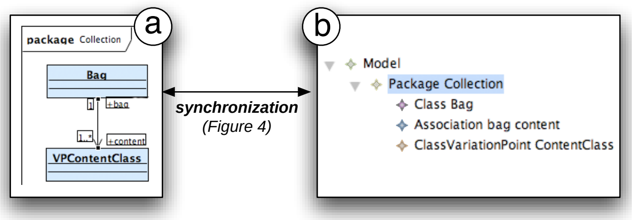


Figure 5: Synchronization of a UML model and a ReuseUML model fragment

As mentioned, we reuse existing editors and define model fragments by means of the core language to save the effort of developing new editors for every reuse metamodel. Variation points, not available as concepts in those editors, are expressed through name conventions, annotations, comments or similar means. TGG rules can then be applied to translate such marked fragments into real model fragments (i.e., instances of a reuse metamodel). An abstract TGG rule for this is illustrated in Figure 3.

The abstract rule can be specialized to be used in the UML scenario (cf. Figure 4). The rule defines that a naming convention `VP*` can be used to express variation points through classes. We can then use any EMF-based UML editor to model a UML model fragment (cf. Figure 5a). Application of the rule would translate it to a real model fragment (shown in an abstract tree notation in Figure 5b). Furthermore, if changes are made later on in the editor, the TGG engine will reflect those changes on the real fragment.

3. TORNADO TGG ENGINE

From the scenario presented in the last section, the following requirements for a TGG engine are derived:

- *Requirement:* EMF as a model repository and Ecore as a metamodeling language should be supported.
  Justification: EMF is an accepted tool platform for modeling and most (open-source) modeling editors are based on it, and are thus targets for tool integration
- *Requirement:* Triple Graph Grammar rules should be interpretable for incremental model synchronization.
  Justification: Different physical model elements can exist for the same conceptual element (e.g., model fragments and composed models) and have to be synced.
- *Requirement:* A rule abstraction mechanism (like rule inheritance) has to be provided to efficiently define generic rules and specialize them.
  Justification: The rules used in the metamodel extension and the model synchronization always have some common part independent of the concrete metamodel to extend. It is convenient to define this part once and specialize it for concrete metamodels.

The following sections describe how these requirements are fullfilled by the Tornado TGG engine and discuss problems that are still topics of research.

3.1 Processing of EMF–based Models

The kernel of the Tornado engine is a pattern matching and rule application algorithm which is capable of adding and synchronizing model elements by updating or reseting. It utilizes the reflection facility of EMF for both: the analysis and the updating of the model graph. In EMF, metaclasses have the type `EClass`. Attributes and references are stored in `EStructuralFeature`s possessed by `EClass`es. The identification of model element types is done by matching the `name` feature of their `EClass`. The matching of references and attributes is a name-matching over the names of the `EStructuralFeature`s of the corresponding `EClass`. Through this metamodel-independent implementation, models written in arbitrary EMF-based DSLs can be addressed on the left and right side of TGG rules.

3.2 Incremental Model Synchronization

To support incremental model synchronization on changes in the involved model graphs, the engine uses correspondence models that are persistent over all successive synchronisation tasks. These are also EMF-based models which consist mainly of `CorrespondenceNode`s and `CorrespondenceLink`s. The EMF model import feature—the possibility to reference elements in other models—is utilized to reference elements of the left- and right-hand-side models. EMF ensures that these links are kept when models are persisted.

Through the correspondence model, inconsistencies are recognized. If elements are deleted in one of the models this can be easily observed because a reference to the imported models breaks. Changes of elements (e.g., changes of attribute values) are in general difficult to track. Here another advantage of the EMF-based system can be utilized. If we activate the engine at runtime we can observe the model elements and react on changes immediately. This can be easily implemented since EMF comes with an observer mechanism that is inherited by all EMF-based applications.

In synchronization, the major problem is to derive the required and optimal transformation steps towards a consistent graph. As an example consider the change of attributes of an element. This can put constrains on the element that prevents the rule, used to create it, from matching any longer. The problem can be resolved by resetting the formally applied rule. Previously created elements have to be deleted and, consequently, all depending rule applications have to be reseted as well. An algorithm addressing this with certain optimizations is described in [5]. However, the reseting of rule applications is a severe action. Often, it leads to a deletion of elements that later have to be re-created by re-applications of rules leading to possible loss of information. We currently work on an improvement of the algorithm that tries to match other rules first on elements before deleting them irrevocably.

3.3 Rule Definition with Rule Inheritance

To realize the required abstraction mechanism for TGG rules, a rule inheritance mechanism has been realized in Tornado's TGG rule definition language. The mechanism supports multiple inheritance (as known from object-oriented languages) on the base of nodes and edges. That is, features of nodes can be refined and additional edges can be connected to nodes. Through multiple inheritance, nodes can be merged if their feature and connected edges conform. The mechanism allows for abstract (i.e., incomplete) rules which can not be applied without being refined. An example are nodes without type declaration as used in Figure 3. The other features of the rule definition language are designed based on the general TGG formalism.

We provide an editor for rule definition which has been designed and generated with the Graphical Modeling Framework[3]. In the Reuseware metamodel extension case it is primarily applied to define the abstract rules. Figure 6 shows the rule from Figure 3 defined for the Tornado engine utilizing the editor. The concrete specializations can be semi-automatically derived by DSL-developers using provided wizards.

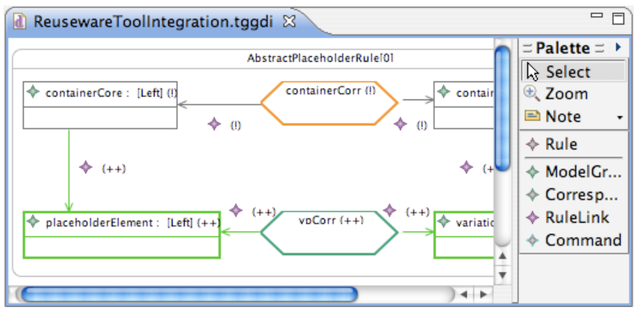


Figure 6: TGG rule editor

4. RELATED WORK

Fujaba provides the most advanced TGG engine at the moment [8]. Unfortunately, integrating it into an EMF environment is not trivial. A close integration, for instance, with the EMF observer mechanism to track changes is not realistic at the moment.

Another EMF-based interpreter is currently developed at the University of Paderborn [5, 6]. Unfortunately, up to now it only supports one-directional model transformations.

The Atlas Model Weaver [2] can handle EMF-based models and realizes a similar idea as TGGs. Instead of synchronization models, so-called *weaving models* express relations between elements, which do not necessarily express equality. Thus, there is no interpreter for the synchronization semantics we require. However, the underlying Atlas Transformation Language[4] provides a rule inheritance mechanism with similarities to our TGG rule inheritance.

Other EMF-based tools for model management tasks, which were considered as alternatives to TGGs, are developed in the GMT project[5] (Epsilon, VIATRA2, oAW, and others). While these are powerful tools, they all process models in a batch-like fashion, have little support for model update, and no build-in synchronization mechanism. Thus, they could have been used as a base for the implementation of the synchronization mechanism. Because of the desired close EMF integration we decided on a Java implementation.

[3]http://www.eclipse.org/gmf
[4]http://www.eclipse.org/m2m/atl
[5]http://www.eclipse.org/gmt

5. CONCLUSION AND FUTURE WORK

The result of our work is twofold: First, we showed how Triple Graph Grammars can be applied in a scenario of harmless metamodel extension. Second, we developed a TGG interpreter that meets our specific needs.

Our feeling is that the Reuseware formalism is only one prototype of what we called *harmless metamodel extensions*. It is an interesting direction to identify other useful extensions like this. Possibly, the generic Reuseware TGG rules can be further abstracted to a set of TGG rules reusable in any harmless metamodel extension. This, paired with tool support for semi-automatic TGG rule specification, could lead to powerful, yet easy to use, DSL development tools.

The development of the TGG engine driven by the concrete use case has revealed interesting challenges. At many points trade-off decisions had to be made concerning expressiveness versus usability. We are optimistic that ongoing development will result in an interpreter that is utilizable in the EMF world by TGG experts for several tasks. However, it seems that the direct usage of TGG rules for model synchronization in software modeling is a too complex and error-prone task for many developers. Systems, like the presented one, where TGG experts define a set of abstract rules which are then refined by developers in a semi-automatic way offer an interesting combination of the powerful TGG formalism with easy usability.

6. ACKNOWLEDGMENTS

This research has been co-funded by the European Commission within the 6th Framework Programme project Modelplex contract number 034081 (cf. `www.modelplex.org`).

7. REFERENCES

[1] F. Budinsky, E. Merks, and D. Steinberg. *Eclipse Modeling Framework 2.0.* Addison Wesley, Jan. 2007.
[2] M. Didonet Del Fabro, J. Bézivin, F. Jouault, E. Breton, and G. Gueltas. AMW: A Generic Model Weaver. In *Proc. of the 1res Journées sur l'Ingénierie Dirigée par les Modèles, France, Paris*, 2005.
[3] F. Heidenreich, J. Johannes, and S. Zschaler. Aspect-Orientation for Your Language of Choice. In *Proc. of the 11th Int'l Workshop on Aspect-Oriented Modeling (to appear), Nashville, TN*, September 2007.
[4] J. Henriksson, J. Johannes, S. Zschaler, and U. Aßmann. Reuseware – Adding Modularity to Your Language of Choice. *Proc. of TOOLS EUROPE 2007: Special Issue of the Journal of Object Technology (to appear)*, 2007.
[5] E. Kindler and R. Wagner. Triple Graph Grammars: Concepts, Extensions, Implementations, and Application Scenarios. Technical Report tr-ri-07-284, University of Paderborn, June 2007.
[6] C. Lohmann, J. Greenyer, J. Jiang, and T. Systä. Applying Triple Graph Grammars For Pattern-Based Workflow Model Transformations. *Proc. of TOOLS EUROPE 2007: Special Issue of the Journal of Object Technology (to appear)*, 2007.
[7] A. Schürr. Specification of graph translators with triple graph grammars. In *Proc. of the 20th International Workshop on Graph-Theoretic Concepts in Computer Science, Herrsching, Germany*, volume 903 LNCS. Springer, Berlin, 1994.
[8] R. Wagner. Developing Model Transformations with Fujaba. In H. Giese and B. Westfechtel, editors, *Proc. of the 4th International Fujaba Days 2006, Bayreuth, Germany*, volume tr-ri-06-275, pages 79–82. University of Paderborn, September 2006.

Breaking the Domination of the Internal Graph Model

Florian Heidenreich
Technische Universität Dresden
Software Technology Group
01062 Dresden, Germany
florian.heidenreich@tu-dresden.de

Ulf Wemmie
Technische Universität Dresden
Software Technology Group
01062 Dresden, Germany
s2918002@mail.inf.tu-dresden.de

ABSTRACT

Graph rewrite systems are often build to only transform graphs that are expressed using their internal graph modelling language. This prevents the use of the advanced techniques in graph rewriting on graphs or models that are not expressed in a way that the tool is able to understand. In this paper we present our approach to model migration for graph rewrite systems, that is, adaptation of graphs from external tools to the graph rewrite system's internal model. We exemplify our ideas by a prototypical implementation for Fujaba4Eclipse.

Categories and Subject Descriptors

D.2.2 [**Design Tools and Techniques**]: Computer-aided software engineering (CASE); D.2.2 [**Design Tools and Techniques**]: Object-oriented design methods

General Terms

Design, Languages

Keywords

Graph Rewriting, Model Migration

1. INTRODUCTION

One of the major drawbacks of current Graph Rewrite Systems (GRS) is the tight coupling of the system to a specific repository and a specific internal graph model. This situation often prevents use of the GRS in other context that was not foreseen by the developers of the specific GRS. To alleviate this situation, a general approach for adaptation of external models to an internal model of the GRS is crucial. Doing this manually is an error-prone and tedious task. Therefore we aim at providing a semi-automated generative solution for bridging from external models to internal models of the GRS (and back again). We use Fujaba4Eclipse [3] as GRS to exemplify our approach, since it also suffers from the limitations mentioned above.

The paper is structured as follows. At first we elaborate the domain of model migration and present the different steps that are needed to migrate an external model to the internal model of Fujaba. Section 3 highlights interesting points of the prototypical implementation while Section 4 presents related work. Section 5 summarizes the paper and discusses future work.

2. MODEL MIGRATION

As motivated in the introduction, for the exchange of models between different applications an adaptation between source and target is needed. During the adaptation process an *external model* is transformed to an *internal model*, i.e. the graph model of Fujaba4Eclipse. To access the model of the external application, access to its representation, i.e. in a specific repository, is needed. We call this part of the process *physical adaptation*. In many cases the external application and the GRS (Fujaba in our case) do not share the same metamodel. Therefore a second step is needed, where model elements of the external model are mapped to the target domain. This process of *domain adaptation* is based on the metamodel level.

Our approach for model migration within Fujaba4Eclipse is based on the notion of *adaptation chaining* where a separation of physical and domain adaptation allows for reuse of adapters. Furthermore, the development of the adapters can be done by different software developers where each of them has specific skills regarding repository or domain knowledge. The physical adaptation A_P is done programmatically on source code level, whereas the domain adaptation A_D is defined by Triple Graph Grammars [5, 7] with a graphical notation. The overall adaptation chain is depicted in Figure 1.

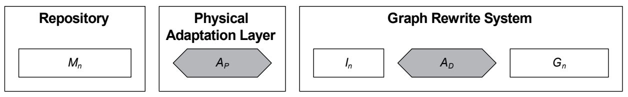


Figure 1: Adaptation chain of physical and domain adaptation.

2.1 Physical Adaptation

A physical adaptation A_P is an exogenous transformation [6] that transforms the source model[1] M_n of the external tool into an intermediate model I_n, where I_n acts as an interface between physical and domain adaptation. I_n still represents the concepts of the external model but is modelled using concepts of the graph rewrite system. To transform the model elements from the repository to the representation of I_n the physical adaptation layer from Figure 1 is refined into an adaptation chain consisting of *repository adaptation* A_{Rep} and *element adaptation* A_{Elem} (see Figure 2).

The bidirectional repository adapter provides a linking to the repository of the external application. It offers means to both access existing models and to create new models in the repository. The intermediate model I_n conforms to the metamodel M_{n+1} of the external model M_n but uses Fujaba classes and interfaces for its implementation of the element adapters. Thereby, persistence and the uniform processing of the intermediate model is assured. Since

[1]In the following we us the abbreviations M for the external model, I for the intermediate model, and G for the internal graph model.

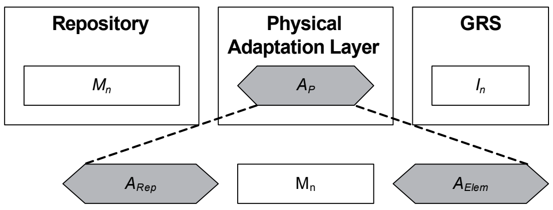


Figure 2: Physical adaptation refined to adaptation chain of repository and intermediate adaptation.

the number of metalevels is not fixed in general, one can imagine migration of models on multiple metalevels. Since Fujaba offers levels for metamodels, models and objects (besides its internal meta-metamodel) the specification of graph-rewrite rules is limited to the model level. This observation leads to a total of two metalevels that can be migrated within Fujaba, where the first level is subject for transformation through the GRS.

2.2 Domain Adaptation

Domain adaptation is an endogenous transformation [6] that provides a bidirectional mapping of elements from the metamodel I_{n+1} of the intermediate model I_n to the Fujaba metamodel. The general idea is to map elements from one domain to another domain, where the specific elements are playing *identical roles* in the specific domains. To exemplify this, the element `EClass` from the Eclipse Metamodeling Language Ecore [2] can be mapped to Fujaba's `UMLClass`, since both elements are playing the roles of the class concept in their domain.

2.3 Process Overview

While the last sections show the general ideas behind our approach to model migration, this section presents an overview to the overall process from the adaptation of the external application to the successful transformation of the adapted model in Fujaba4Eclipse.

First, a Fujaba conforming representation of the meta-metamodel[2] of the external application is needed. This representation can be modelled as Fujaba class diagram and afterwards generated to Java code. This is necessary because initially there are no means to import models of the external application to Fujaba. This initial step is depicted in Figure 3.

The modelled meta-metamodel acts as an interface between physical and domain adaptation. It is used to generate the physical representation of the external meta-metamodel as well as to the partial

[2]The indices refer to the standard metamodel levels of the MOF metalevel architecture.

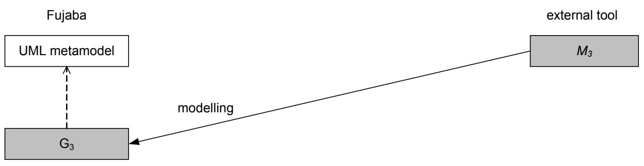


Figure 3: Modelling the meta-metamodel of the external application.

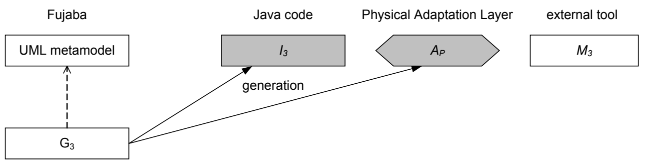


Figure 4: Generation of the code representation of the meta-metamodel and of the physical adapter.

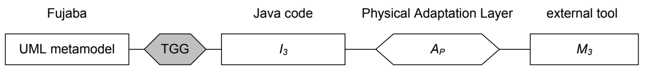


Figure 5: Specification of the domain adaptation.

generation of the physical adapter. Figure 4 shows the generation of the meta-metamodel to Java code and the generation of the physical adapter.

After that, domain adaptation can be specified. Therefore the UML metamodel of Fujaba and the generated physical representation of the external meta-metamodel are used. Triple Graph Grammars [5, 7] are utilised to graphically define the relationship between the different domains. These bidirectional rules are then transformed to Story Diagrams (Fujaba's notion of graph-rewrite rules) that are used to exchange models between the different metamodels (see Figure 5).

This complete adaptation chain now provides means for import and export of metamodels of the external application (see Figure 6). This occurs one metalevel below the metalevel on which the definition of the adapters was done. On import, the metamodel of the external application is transformed to the intermediate representation using the physical adapter. The intermediate representation is then transformed into a Fujaba UML class diagram using the rules that were specified using Triple Graph Grammars.

It is obvious, that an additional adapter is needed to process instances of the imported metamodel in Fujaba. But in contrast to the development of the previous adapter, the metamodel already exists as a UML class diagram and does not need to be modelled anymore. The next steps that are needed to generate an adaptation chain are straight-forward according to the previous procedure (see Figure 7). In the last step, a physical adapter to import instances of M_2 needs to be developed. This adapter allows for applying the GRS's rewriting capabilities on the imported models.

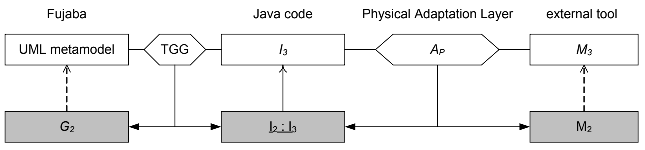


Figure 6: Migration of metamodels.

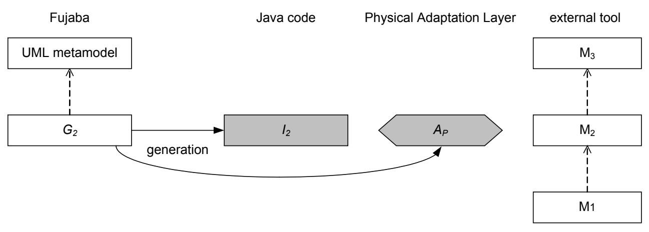


Figure 7: Initial situation for further adaptation.

3. IMPLEMENTATION

We have build a prototype implementation of the presented approach using *Fujaba4Eclipse* and the plug-ins *TGGEditor* (editor for Triple Graph Grammars) and *MoTE* (Model Transformation Engine). It uses the extension points offered by the three components. The prototype itself provides additional extension points, namely `PhysicalAdapter` and `DomainAdapter` for the adaptation of external models and `MigrationAdapter` for the configuration of the interaction of the physical and the domain adapter.

However, the generation of the adaptation chain is still in a very early stage and we try to investigate in further improvements with regards to the automation of the process.

4. RELATED WORK

The *Tool Adapter* proposed by Kindler and Wagner [5] is strongly related to our approach. The authors present an adaptation architecture for adaptation of proprietary metamodel implementations of external tools to the metamodel implementation of the standalone version of the Fujaba tool suite. As we do, they also use TGGs for domain adaptation but use hand-written adapters to adapt the external metamodel (while we follow a semi-generative approach). In contrast to our work, they do not use a dedicated intermediate metamodel on which the adapter is built upon, but transform the Fujaba-generated metamodel of the external application in an adapter to the external metamodel by replacing the bodies of accessor methods with delegation code to the API of the external application. This, on the one side, improves performance of the adaptation, but, on the other side, decreases flexibility in exchanging and reusing existing adapters.

Another approach that is related to our work is the concept of non-materialized model view specifications based on a extension of TGGs—the declarative view specification approach VTTG [4]. Instead of copying tool data and creating physical representations of the target model, the authors present an approach that offers virtual views of models that can be manipulated and are synchronised automatically.

The *Atlas Model Weaver* [10] offers a means for domain adaptation, where the adaptation is specified by a dedicated *weaving model*. This weaving model is used to generate model transformations from one domain to another domain. The approach is very similar to domain adaptation by Triple Graph Grammars but lacks its mathematical foundations [7].

The *Tiger EMF Transformation Project* [1, 11] is a framework for EMF transformations based on graph transformation. It supports the definition of graph-rewrite rules on arbitrary EMF-based metamodels and uses AGG [8, 9] as GRS. Internally, the imported EMF models are transformed to an AGG representation in a fully automated way. This high degree of automation is at the cost of flexibility regarding the supported external applications (which is Eclipse EMF in this case).

5. SUMMARY

In this paper we presented our work on model migration for graph rewrite systems, that is, mapping external models to internal models of the GRS and back again and presented a general architecture for this. Our approach is based on adaptation chaining, where multiple adapters are chained together and where each of them takes a specific role in the process of model migration. The decomposition of the adaptation problem allows for partial generation of the adapters—and more importantly—for reuse of adapters. We built a prototypical implementation based on Fujaba4Eclipse, which enables usage of Fujaba's advanced graph rewriting techniques on models from other applications in a systematic way.

In our future work we want to improve the generation of the adaptation chains through annotations in the modelled metamodels that would allow for an fully-automated mapping of the external model elements to the GRS's internal representation.

6. ACKNOWLEDGEMENTS

This research has been co-funded by the German Ministry of Education and Research (BMBF) within the project feasiPLe[3].

7. REFERENCES

[1] E. Biermann, K. Ehrig, C. Köhler, G. Kuhns, G. Taentzer, and E. Weiss. Graphical definition of in-place transformations in the eclipse modeling framework. In O. Nierstrasz, J. Whittle, D. Harel, and G. Reggio, editors, *MoDELS*, volume 4199 of *Lecture Notes in Computer Science*, pages 425–439. Springer, 2006.

[2] F. Budinsky, S. A. Brodsky, and E. Merks. *Eclipse Modeling Framework*. Pearson Education, 2003.

[3] Fujaba Project Team. Fujaba4Eclipse, Aug. 2007. URL http://www.fujaba.de.

[4] J. Jakob, A. Königs, and A. Schürr. Non-materialized model view specification with triple graph grammars. In *Proceedings of the 3rd International Conference on Graph Transformations (ICGT'06)*, volume 4178 of *LNCS*, pages 321–335. Springer, 2006.

[5] E. Kindler and R. Wagner. Triple Graph Grammars: Concepts, Extensions, Implementations, and Application Scenarios. Technical Report tr-ri-07-284, Software Engineering Group, Department of Computer Science, University of Paderborn, June 2007.

[6] T. Mens and P. Van Gorp. A taxonomy of model transformation. *Electronic Notes in Theoretical Computer Science*, 152:125–142, Mar. 2006.

[7] A. Schürr. Specification of graph translators with triple graph grammars. In *Workshop on Graph-Theoretic Concepts in Computer Science*, pages 151–163, 1994.

[8] G. Taentzer. AGG: A Graph Transformation Environment for System Modeling and Validation. In *Tool Exhibition at Formal Methods 2003*, 2003.

[9] The AGG Project Team. AGG: The Attributed Graph Grammar System, Aug. 2007. URL http://tfs.cs.tu-berlin.de/agg/.

[10] The AMW Project Team. Atlas Model Weaver, Aug. 2007. URL http://eclipse.org/gmt/amw/.

[11] Tiger EMF Transformation Project Team. Tiger EMF Transformation Project, Aug. 2007. URL http://tfs.cs.tu-berlin.de/emftrans/.

[3]cf. `http://www.feasiple.de`

Monitoring of Structural and Temporal Properties*

Holger Giese, Stefan Henkler, Martin Hirsch, Florian Klein, and Michael Spijkerman
Software Engineering Group, University of Paderborn,
33095 Paderborn, Germany
[hg|shenkler|mahirsch|fklein|spijk]@uni-paderborn.de

ABSTRACT

For the design of complex software systems, and self-adaptive systems in particular, appropriate techniques for the specification and verification of combined structural and temporal properties are required. This paper presents tool support in Fujaba which allows modeling and monitoring of Timed Story Scenario Diagrams that cover combined structural and temporal properties. A scheme to monitor the fulfillment of the visual scenario specifications is presented and the related generation of monitoring code is shown using a simplified example.

1. INTRODUCTION

It is expected that in future dynamic software architectures with structural adaptation at run-time will replace static architectures and models in order to realize more intelligent, efficient, and flexible software-intensive systems (cf. self - adaptive systems [16]).

While the envisioned dynamic architectures result in more flexibility, the design and validation of such flexible systems also become challenging. On the one hand we need description techniques to express requirements and commitments which cover the evolution of the structure over time (in fact we even need tightly integrated notations for the specification of properties covering structural and temporal aspects at the same time). On the other hand we also need analysis techniques that are capable of checking these requirements.

In this paper, the tool support for modeling *Timed Story Scenario Diagrams* (TSSD) and *Story Decision Diagrams* (SDD) as well as a monitoring scheme and the generation of monitoring code for their evaluation are presented.

The paper is organized as follows: A survey of the state of the art concerning modeling and analysis of properties which combine structural and temporal aspects is presented in Section 2. Then, the foundation of the presented work such as the TSSDs and SDDs as well as an application example are presented in Section 3. The monitoring scheme and code generation are outlined in Section 4. In Section 5, the evaluation of the approach is presented before the paper ends with a conclusion and an outlook on future work.

*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.

2. STATE OF THE ART

The UML only provides a textual specification language, the OCL [14], for structural properties, which requires the translation of structural properties from the familiar structural view in form of UML Class and Object Diagrams into an often intricate textual syntax. Several approaches try to overcome this problem by means of special visual techniques for constraints (e.g., constraint diagrams [10] or VisualOCL [3]). However, the resulting visually complex diagrams have only little relation to the original UML specification. Story Patterns (cf. [13]), in contrast, extend UML Object Diagrams and thus avoid this gap. The SDDs used in this paper further extend them to support quantification and negation.

Temporal logics, such as LTL or CTL, represent the standard for the specification of temporal properties. However, as reported in [5], temporal logics are hard to use to the degree that even experts have serious problems handling them. Scenario notations describing the interaction of predefined units such as UML sequence diagrams [15], Live Sequence Charts (LSC) [9], or Triggered Message Sequence Charts (TMSCs) [18] have been proposed as a more accessible means for specifying temporal properties. Visual Timed Event Scenarios (VTS) [1] are an approach which focuses on scenarios for pure events and thus abstracts from the units, an idea that is extended by TSSDs.

A visual approach to combine structural and temporal properties is introduced in [7]. The authors propose to embed graph patterns into LTL formulas. In [17] an extension of Story Diagrams [6] by annotating unary forward or past operators from LTL with additional explicitly encoded time constraints is presented. While the first approach is not applicable due to its use of LTL, the second provides at least the visual concept of an accepting automaton. However, automata often become rather cumbersome as they require the specification of the complete sequential order and not only a partial one.

As the state space of a GTS is typically infinite because new elements can be added to the system, the model checking of GTS poses a problem, even though there are approaches for verifying certain invariants of such infinite systems (cf. [2]). Complete formal verification is therefore only feasible for restricted properties or very small, finite systems.

A more realistic approach is usually to rely on automated testing, using a generated runtime monitor as an oracle. LTL Monitoring and Runtime Verification are two approaches based on the idea of generating such monitors from the specification of the system in LTL. While on-line monitoring with limited resources only seems possible for less expressive specifications, off-line checking of the observed behavior is feasible even when monitoring is very expensive.

3. FOUNDATIONS

The TSSDs employed in this paper combine the intuitive concept of matching structural patterns with decision diagrams, which foster a consecutive if-then-else decomposition of complex properties into comprehensible smaller ones, with a new notation for the temporal ordering inspired by the Visual Timed Event Scenario approach [1]. Therefore, both the structural and temporal aspects as well as their combination can be described in a comprehensible and intuitive manner. In [12, 11], it has been shown that all Property Specification Patterns proposed in [4, 5] can be easily encoded and derived in a compositional manner using TSSDs.

3.1 Application Example

We introduce a simple example: We want to test the autonomous behaviour of a shuttle. While driving on a circular railway track, it is supposed to stop at a maintenance point for inspection after a maximum working time constrained by technical premises. The following Figure 1 introduces the underlying Class Diagram.

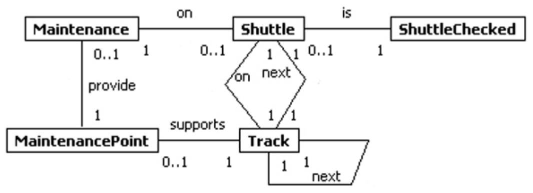


Figure 1: Shuttle class diagram

When reaching the maintenance point, the shuttle decides whether an inspection is necessary. The shuttle can request an inspection by generating a maintenance order, represented by a `Maintenance` object. The shuttle gets a confirmation in form of a `ShuttleChecked` object after maintenance has been performed.

3.2 Story Decision Diagrams and Timed Story Scenario Diagrams

Story Decision Diagrams (SDD) are an extension of Story Patterns that allow expressing more complex properties. The most significant enhancements they provide are quantifiers, implication, alternatives, negation of complex properties, and a concept for modularity (cf. [8]). Figure 2 shows

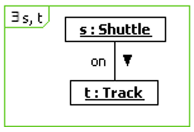


Figure 2: SDD node example

a simple SDDNode with an embedded Story Pattern. This SDD calls for at least one `Shuttle` object referencing a `Track` object in an instance graph.

Timed Story Scenario Diagrams (TSSDs) are based on the idea of specifying scenarios as sequences of *Situations*, each characterized by an embedded SDD. *Situations* are placed in temporal relation to each other through temporal connectors. A TSSD is fulfilled when a sequence of valid system states leading from an *Initial Node* to a *Termination Node* is observed (cf. [8]). *Time Bounds* can additionally place a lower and upper time bound on the interval between the observation of two *Situations*, whereas *Guards* forbid some *Situation* from occurring in that interval.

Figure 3 introduces a simple example. The TSSD consists

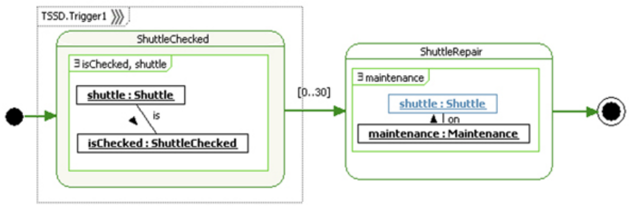


Figure 3: TSSD example

of two *Situations*. The first *Situation*, "ShuttleChecked", defines a system state where a shuttle has a reference to a `ShuttleChecked` object. It is surrounded by a *Trigger*, which indicates that the following scenario needs to be completed for every observation of such a shuttle.

The second *Situation* called "ShuttleRepair" describes a `Shuttle` with a `Maintenance` object, which indicates that the shuttle has requested the next inspection. Both *Situations* are connected by an *Eventually Connector*, which is constrained by a *Time Bound* [0..30]. The TSSD thus specifies that a shuttle needs to be inspected again no more than 30 time units after the last inspection.

Editors for creating SDDs and TSSDs have been added to Fujaba4Eclipse as plugins.

4. MONITORING

We now want to validate system consisting of the system model and the TSSD specification (see Figures 1 and 3), which we have modeled in Fujaba4Eclipse. The system code can be generated using the standard CodeGeneration2 plugin. Additional plugins are able to generate code for the TSSD specifications. The current implementation does not cover the complete TSSD language, but focuses on a core of syntactic elements consisting of *Situations*, *Eventually Connectors*, *Time Bounds*, *Triggers* and *Guards*, which are sufficient to emulate almost the complete syntax through appropriate mappings. The monitor code for a TSSD specification is generated against a TSSD monitoring framework which handles its invocation. The framework reacts to changes in the system by notifying each registered specification, triggering the next evaluation step. Changes are recognized by a central model manager registering as a property change listener with all objects. The manager also serves as a directory, which is necessary for enabling SDDs without a fixed context (*this*). Figure 4 provides an overview of the framework. The system code is generated from the system model defining its structure and, possibly, predefined behaviors which are defined as Story Diagrams. The generated classes are instrumented to register with the model manager and provide property change support. The scheduler manages a set of processes, each defining the behavior of an agent as a sequence of activities controlled by a script or a state machine. Activities describe state changes of the sys-

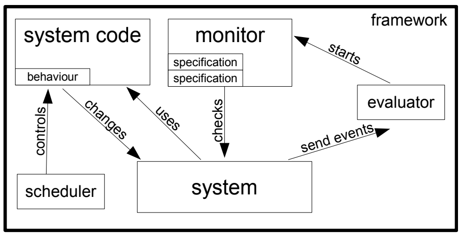


Figure 4: Overview framework

tem, the duration of the state change and the time a state change should be processed. While being very extendable, the TSSD monitoring framework currently only provides elementary methods and infrastructure for defining concrete test scenarios.

Fujaba4Eclipse allows grouping specifications into Constraint Sets, which in turn allows selecting which constraints should be analyzed by the monitor. An evaluator filters events according to the originating model elements and restricts evaluation to those specifications that might be affected by the current state change.

The system component represents the test framework and contains a main class that initializes the specifications, the model, the evaluator and the scheduler and starts the test system.

Figure 5 provides an overview of the test process: After

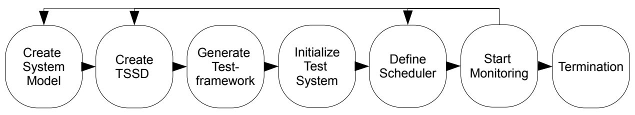


Figure 5: Test process

defining the system model and the TSSD Specification, the test framework can be generated. Then next step is to initialize the test system as described above. Afterwards, the system behavior needs to be implemented or scripted as processes. The test framework can then be executed. If the outcome is not satisfying, either the system behavior needs to be changed, or the specification is too restrictive, which makes refinements in the TSSD necessary.

In the following, we present an overview of the monitoring concept. Figure 6 provides a TSSD specifying the creation of a simple synthetic object structure. It has 4 *Situations*: *FindA* describes the identification of an `A` object. *FindB* and *FindC* enable the parallel search for objects `B` and `C` referencing the previously bound (indicated by the blue colour) object `A`. *TimeBounds* are constraining *FindB* and *FindC*. After identifying appropriate objects `B` and `C`, an object `D` referencing `B` and `C` must be found in *Situation FindD*. The instance path shows a system run. Every time a corresponding system state is identified for a *Situation*, a node in the calculation tree, called an *Observation*, is created. A sequence of *Observations* is a *Trace*. Each *Observation* is a correct extension regarding the TSSD, meaning that all required objects are bound, all necessary predecessor nodes are available and all *TimeBounds* hold in the *Trace*. Where several branches join, we need to find compatible *Traces* for each branch, recombining them into a single *Trace* fulfilling all preconditions. For that we copied the node *FindC* at time 3 to the parallel *Trace* 1. The copied *Observation* is

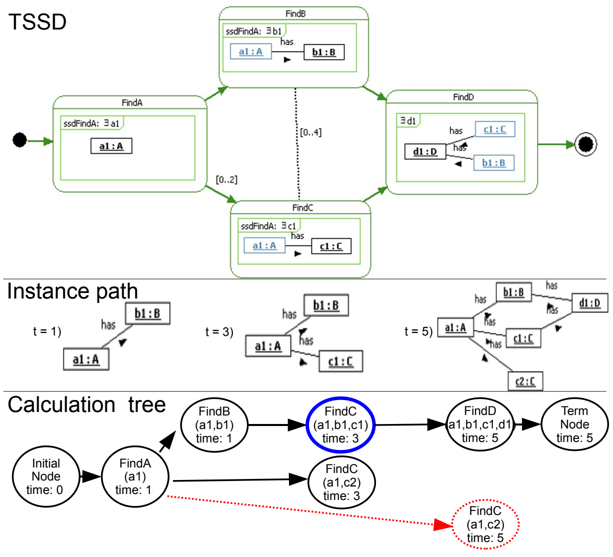


Figure 6: Monitoring concept

marked bold and blue. In time step 5 you could observe another *FindC Observation*, but this extension is prohibited because the *Time Bound* between *Situation FindA* and *FindC* allows at most 2 time units for observing *FindC*. Thus, the *Observation* is marked dashed and red. The goal is to observe a *Trace* with valid *Observations* ending in a *TerminationNode*. In this example we reach the *TerminationNode* in time step 5, making the system run valid according to the defined TSSD. This example gives only a shallow idea of the whole monitoring algorithm (cf. [19]).

The generated code implements the presented algorithm. It consists of both static information that represents the structure of the TSSD and code for managing the calculation tree. The TSSD structure is mapped to a static class structure. There are classes which represent the *Situations* and the *Eventually Connectors*. The pseudo states *Initial Node* and *Termination Node* are treated as *Situations* as well. The code to manage the calculation tree is generic and provided by abstract base classes. They exist for every core element and the overall diagram. An *Observation* is an instance of the corresponding *Situation* class. Using static factory methods, the extension is computed in the appropriate class. If all preconditions are valid and the embedded SDD returns a valid binding, the current class can be instantiated, extending the calculation tree.

5. EVALUATION

The efficiency of the validation depends on the number of existing *Observations* in the calculation tree. Each calculation step iterates through all nodes and tries to extend them, in turn checking the structural preconditions for each.

Unfortunately, the upper bound for the possible number of nodes is rather large. The dimension of the calculation tree depends both on system behavior and the size and structure of the specification. While the size of each *Trace* is limited by the number of *Situations* in the TSSD, many *Observations* can be made for the same *Situation*, leading to branching into a set of alternative *Traces* whose number can rise exponentially in the worst case.

To limit the size of the calculation tree, optimizations are applied that eliminate all calculation tree nodes which are unnecessary for further validity calculations. If a *Trace* is successfully extended to a *Termination Node* and no *Guards* or *Time Bounds* are defined the *Trace* is marked final. *Traces* containing *Time Bounds* defining an upper bound also necessarily become final, either when they are successfully extended or when the upper bound is reached. If a *Guard* is observed, the *Trace* cannot be extended any more and becomes final as well. An *Observation* which cannot be extended because its binding becomes invalid is obsolete. All markings made during validation of the calculation tree are propagated towards the *Observation* for the *Initial Node*. Only the first node of a final *Trace* needs to be retained until it becomes obsolete in order to prevent it from being observed again prematurely, all other final nodes are deleted.

The number of *Observations* significantly depends on the instantiated model elements. In the example, the number of new *Traces* is limited by the number of shuttles which are instantiated in the system as the specifications refer to the state of individual shuttles. In all evaluation examples the size of the calculation tree is importantly smaller than the worst case.

For a practical description of the evaluation, we run the shuttle with its defined behavior based on the system model introduced in Figure 1 to check if the behavior fulfills the specification shown in Figure 3. Figure 7 presents an in-

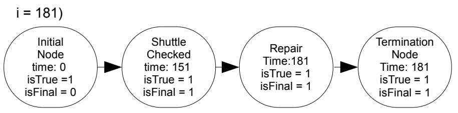

i = 181)
Initial Node time: 0 isTrue =1 isFinal = 0
Shuttle Checked time: 151 isTrue = 1 isFinal = 1
Repair Time:181 isTrue = 1 isFinal = 1
Termination Node Time: 181 isTrue = 1 isFinal = 1

Figure 7: Calculation tree at time = 181

stance of the calculation tree at the random time t=181 to give an example. At t=0 the *Observation* of the *InitialNode* was created at start of the evaluation. At t=151 the shuttle was repaired at the service station. The shuttle got a new `ShuttleChecked` object, and a corresponding *Observation* "ShuttleChecked" was created in the calculation tree. At t=181 the *Observation* "Repair" was made punctually. After 30 time units the shuttle started an inspection again, barely meeting the time constraint. The calculation tree could be directly extended to the *TerminationNode* because the connector leading to it has no *Time Bounds* or *Guards*. At this point, the TSSD therefore holds. However, this is not permanent as the `ShuttleChecked` that is created after the inspection will again trigger the scenario later. Older *Traces* are not visible in the calculation tree cause they were eliminated by the described optimizations, limiting the calculation tree to at most 4 *Observations*.

6. CONCLUSION AND FUTURE WORK

In this paper, we introduce an approach for monitoring TSSDs. We present a monitoring algorithm which evaluates a system specified by TSSDs and a test framework which executes the generated system and monitoring code. As future work, we plan to support all syntatic features of TSSDs in the code generation. We also need a scheme for strictly limiting the size of the calculation tree in a restricted environment. An advantage of our approach is that we do not need to insert monitoring code into the system code, which makes off-line monitoring possible once we develop a concept for storing the instance path.

REFERENCES

[1] A. Alfonso, V. Braberman, N. Kicillof, and A. Olivero. Visual Timed Event Scenarios. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 168–177, Washington, DC, USA, 2004. IEEE Computer Society.

[2] B. Becker, D. Beyer, H. Giese, F. Klein, and D. Schilling. Symbolic Invariant Verification for Systems with Dynamic Structural Adaptation. In *Proc. of the* 28th *International Conference on Software Engineering (ICSE), Shanghai, China.* ACM Press, 2006.

[3] P. Bottoni, M. Koch, F. Parisi-Presicce, and G. Taentzer. A visualization of OCL using collaborations. *Lecture Notes in Computer Science*, 2185:257–271, 2001.

[4] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Property Specification Patterns for Finite-state Verification. In *2nd Workshop on Formal Methods in Software Practice.* ACM Press, March 1998.

[5] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Patterns in property specifications for finite-state verification. In *ICSE '99: Proceedings of the 21st international conference on Software engineering*, pages 411–420, Los Alamitos, CA, USA, 1999. IEEE Computer Society Press.

[6] 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* 6th *International Workshop on Theory and Application of Graph Transformation (TAGT), Paderborn, Germany*, LNCS 1764, pages 296–309. Springer Verlag, November 1998.

[7] F. Gadducci, R. Heckel, and M. Koch. A fully abstract model for graph-interpreted temporal logic. In *Proc. of the Theory and Application of Graph Transformations*, volume 1764 of *Lecture Notes in Computer Science*, pages 310–322, 2000.

[8] H. Giese and F. Klein. Visual Specification of Structural and Temporal Properties. In H. Giese and B. Westfechtel, editors, *Proc. of the 4th International Fujaba Days 2006, Bayreuth, Germany*, volume tr-ri-06-275 of *Technical Report.* University of Paderborn, September 2006.

[9] D. Harel and R. Marelly. Playing with Time: On the Specification and Execution of Time-Enriched LSCs. In *Proc. 10th IEEE/ACM Int. Symp. on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS 2002)*, Fort Worth, Texas, USA, 2002. (invited paper).

[10] S. Kent and J. Howse. Mixing visual and textual constraint languages. In R. France and B. Rumpe, editors, *UML'99, Fort Collins, CO, USA, October 28-30. 1999, Proceedings*, volume 1723 of *LNCS*, pages 384–398. Springer, 1999.

[11] F. Klein and H. Giese. Integrated Visual Specification of Structural and Temporal Properties. Technical Report tr-ri-06-277, Computer Science Department, University of Paderborn, October 2006.

[12] F. Klein and H. Giese. Joint Structural and Temporal Property Specification using Timed Story Sequence Diagrams. In M. Dwyer and A. Lopes, editors, *Proc. of 10th International Conference on Fundamental Approaches to Software Engineering (FASE) 2007, held as part of ETAPS 2007, Lisboa, Portugal, March 24-April 1, 2007*, volume 4422 of *LNCS*, pages 185–199. Springer Verlag, March 2007.

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

[14] Object Management Group. UML 2.0 Object Constraint Language (OCL) Specification, 2003. http://www.omg.org/docs/ptc/03-10-14.pdf.

[15] Object Management Group. *UML 2.0 Superstructure Specification*, 2003. Document ptc/03-08-02.

[16] P. Oreizy, M. M. Gorlick, R. N. Taylor, D. Heimbigner, G. Johnson, N. Medvidovic, A. Quilici, D. S. Rosenblum, and A. L. Wolf. An Architecture-Based Approach to Self-Adaptive Software. *IEEE Intelligent Systems*, 14(3):54–62, May/June 1999.

[17] T. Rtschke and A. Schrr. Temporal Graph Queries to Support Software Evolution. In *Graph Transformation: 5th International Conference, ICGT 2006, Rio Grande do Norte, Brazil, September 17-23, 2006*, pages 1–15, 2006.

[18] B. Sengupta and R. Cleaveland. Triggered Message Sequence Charts. In W. G. Griswold, editor, *Proceedings of the Tenth ACM SIGSOFT Symposium on the Foundations of Softare Engineering (FSE-10)*, Charleston, South Carolina, USA, November 2002. ACM Press.

[19] M. Spijkerman. Monitoring gemischt struktureller und temporaler Eigenschaften von UML Modellen. Diplomarbeit, Software Engineering Group, University of Paderborn/Germany, October 2007.

Model-Based Testing of Mechatronic Systems[*]

Holger Giese[†], Stefan Henkler, Martin Hirsch, and Claudia Priesterjahn
Software Engineering Group, University of Paderborn,
33095 Paderborn, Germany
[hg|shenkler|mahirsch|herklotz]@uni-paderborn.de

ABSTRACT

We report about a Fujaba plugin for the automatic test generation and execution for components modeled with Real-Time Statecharts. We automatically generate test suites by using counterexamples obtained from the existing model checking plugin. By using a model checker we are able to support different coverage criteria for the behavior with the generated test suites by invoking the model checker with specific formulas adapted from criteria-specific observer automata. The presented approach is efficient as we propose a compositional testing approach. Furthermore we support to execute the test suites for the generated code and measuring the coverage criterion.

1. INTRODUCTION

Embedded systems are successfully employed in many domains. Examples are automotive applications, automation or aerospace applications. To develop these systems, the complex distributed real time interaction within these systems has to be planned and its safe behavior has to be guaranteed, as non-compliance with crucial timing constraints can lead to fatal accidents.

The model-based development of embedded systems employing automatic code generation reduces the difficulties of developing complex embedded systems and thus gets more popular. By using model-based development approaches the formal verification of these models, e.g. in form of model checking, are not sufficient. The formal verification recognizes changes with respect to the specification but do not check if the noticeable behavior of the implemented or modeled system meets the requirements. Besides the problem of having an adequate specification the state explosion problem of the formal verification approaches limits these techniques to small systems or components. Furthermore, often not all parts of the system are generated automatically, manually the generated code is optimized or legacy components are integrated. Hence, dynamic analysis, in form of testing, becomes important.

In practice, testing is a widely used technique but also a high effort is needed for deriving tests. To overcome this problem automated test derivation in form of model-based testing is getting more popular in the last years (c.f. [3], [8], and [16]). As exhaustive testing is often impossible or at least to costly, a finite set of tests are applied which cover a certain criteria.

In this paper we present an extension of our modeling and analysis approach MECHATRONIC UML, which mainly considered formal verification [10], towards model-based testing. The extensions employ the model checking capabilities of our tool support by using the counter example provided by the model checker to derive tests (we use the UPPAAL model checker to generate test cases (cf. [14] and [11]) via the model checking plugin integrated in the Fujaba Real-Time Tool Suite [5]). The presented extension focuses on test cases to validate the real time behavior of Real-Time Statecharts with respect to a given coverage criteria.

The rest of the paper is organized as follows: In Section 2 we discuss the related work. Afterwards, we introduce MECHATRONIC UML and explain its support for modeling complex, distributed embedded systems. Based on the MECHATRONIC UML modeling approach, we sketch the existing formal verification capabilities. Then, the developed model-based testing approach is introduced and outlined using a simple example in Section 4. The paper closes with a conclusion and an outlook on planned future work.

2. RELATED WORK

Beyer et al [1] propose an approach to automatically generate traces from a given model by using a model checker as a state exploration tool. They also guarantee state coverage. We expand this approach by using templates adapted from Blom et al [2].

In [11] Hessel et al. introduces a new approach of emitting the fastest diagnostic trace of UPPAAL counter examples for conformance testing of black-box components. As the focus is on real-time systems, they also address the problem of the correct timing of test execution and limiting the number of test cases. To ensure time optimal testability, the semantics of UPPAAL is reduced to deterministic, input enabled and output urgent timed automata. However, the approach does not provide tool support.

In [15] Mücke and Huhn generate optimized test suites from a subclass of deterministic Statecharts with real-time constraints. In addition to the approach of Hessel et al this approach includes tool support. Further, a new coverage criterion is introduced to focus especially on the closest operands possible to test the limits of guards. The coverage criterion is called boundary coverage. Mücke and Huhn added Boolean coverage variables to the model to be aware

[†]Currently a visiting professor at the Hasso-Plattner-Institut, University of Potsdam.

[*]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.

of a certain coverage is achieved. The Boolean coverage variables enlarge the state space which is one of the main problems of this approach. The presented approach of Mücke and Huhn lacks in efficiency as they have no additional compositional consideration in contrast to our work.

3. MODEL-BASED VERIFICATION

The Mechatronic UML approach enables the development of complex mechatronic systems [7]. Components and patterns can be employed to model the architecture and real-time coordination behavior. An embedding of continuous blocks into hierarchical component structures permits to integrate controllers into the component model. The component and pattern definition are supported by the Fujaba Real-Time Tool Suite[1] while the blocks can be specified with CAE tool CAMeL.[2]

Discrete behavior of the components and patterns is specified by real-time statecharts [6] or their hybrid extension hybrid reconfiguration charts [4]. The provided concept enables the specification and modular verification of reconfiguration across multiple components [9].

The Mechatronic UML approach also supports model checking techniques for the real-time processing at the level of connected mechatronic systems. By supporting a compositional proceeding for modeling and verification of the real-time software [10], the approach avoids scalability problems to a great extent.

4. MODEL-BASED TESTING

The idea of model-based testing is to derive test cases from an explicit model describing the behavior of the unit under test while considering test cases as traces in the model. Thus test cases are specified by their input and expected output. During testing the implementation is invoked with the input. Afterwards the implementation's output is compared to the model's output to give a verdict about the correctness of the implementation's behavior. In case of reactive real time systems, the input and output are timed sequences capturing the interaction of the unit under test and the environment.

4.1 Overview

Figure 1 shows a basic overview of our approach in form of an activity diagram. First, we generate test cases that satisfy a certain coverage criterion. Afterwards the test cases are executed on the unit under test while monitoring the coverage. The test execution runs until the coverage criterion is satisfied, no test cases are left, or an error is found. Finally, we report the result that can be of the following form. In case the testing environment reported an error, the test case causing the error is returned. If no error was found, the output is "'OK"' and a statement about the completeness of the coverage.

4.2 Automatic Generation of Test Cases from Counterexamples

As proposed by Beyer et al. [1], we use a model checker to generate traces for our model. This is achieved by passing a constraint in the form of a temporal logic formula to the model checker that is known not to be satisfied by the model. The model checker returns an error trace leading to the part

[1]http://www.fujaba.de/

[2]http://www.ixtronics.de/English/indexE.htm

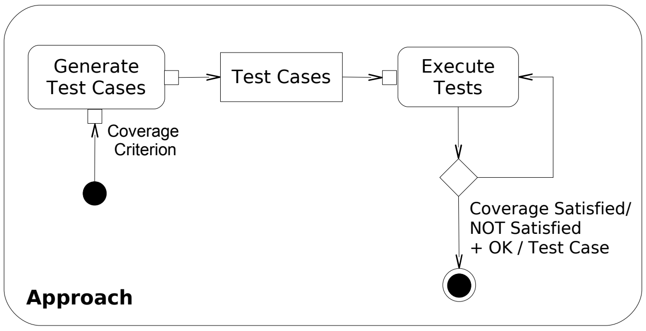


Figure 1: Employed model-based testing approach

of the model that violates the constraint. This trace is used to compute initial and final values for a test case.

Since there already exists a Fujaba plugin for the model checking of Mechatronic UML with the UPPAAL model checker by Hirsch [12], we use it to generate test cases. Furthermore we use a simple heuristic to optimize the test generation process (cf. Section 4.2.2).

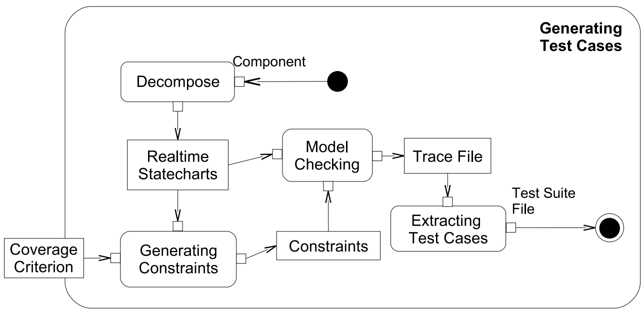


Figure 2: Overview of Test Generation

As shown in Figure 2, we start with a component that is decomposed into its Real-Time Statecharts. This allows us to apply component wise testing on complex models as proposed by van der Bijl et al. [17]. Afterwards the Real-Time Statecharts are used to generate constraints. Constraints are temporal logic formulae that are passed to the model checker as a property to be checked on the Real-Time Statechart. The Constraint Generator generates the constraints for a given coverage criterion on the passed Real-Time Statechart as described in Section 4.2.1. After the model checker generated a trace file, the test suite is extracted into a test suite file.

4.2.1 Test Coverage

To guarantee a certain test coverage, we pass special constraints to the model checker adapted from the principle of observer automata as proposed by Blom et al. [2]. Our approach is based on the reachability of elements that are to be covered. We use the model checker to generate a trace covering a certain element in the Real-Time Statechart.

A model checker can be used as a trace generation tool. By passing a negated temporal logic formula, we yield a trace to a certain state in our Real-Time Statechart.

To achieve that, we invoke the model checker with special constraints of the form $\mathbf{AG}\ \neg s$. The constraint is formed as follows. To check with the model checker whether state

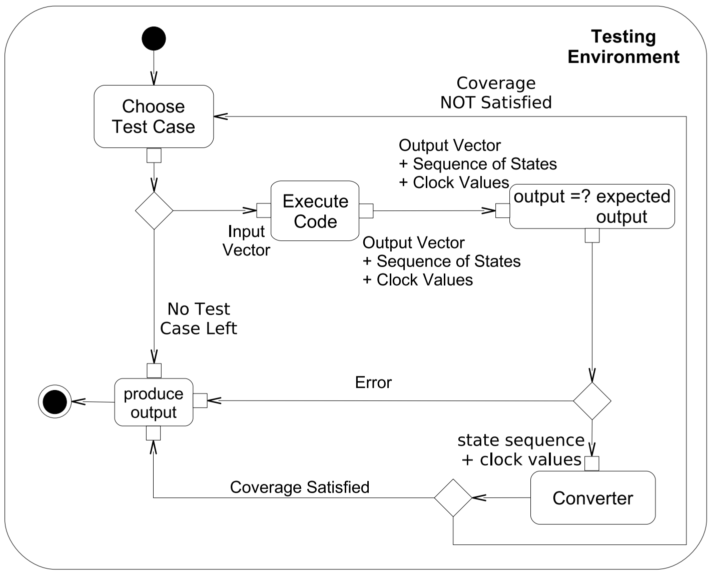

Testing Environment
Choose Test Case
Coverage NOT Satisfied
Output Vector + Sequence of States + Clock Values
Execute Code
output =? expected output
Input Vector
Output Vector + Sequence of States + Clock Values
No Test Case Left
Error
produce output
state sequence + clock values
Coverage Satisfied
Converter

Figure 3: Overview of Test Execution

s is reachable, we could use a formula of the form **EF** s. As this will of course succeed and will not produce the required counterexample, we have to check the opposite property which the model does not satisfy and for which thus a path leading to state s is returned. That is why we have to negate the formula considering the following equivalence: $\neg\mathbf{EF}\ s \equiv \mathbf{AG}\ \neg s$.

4.2.2 Heuristic

Since model checking is quite an expensive undertaking, we use a simple heuristic to reduce the requests to the model checker adapted from a heuristic proposed by Blom et al. [2].

In their approach, every constraint, that is associated to a target state that already has been covered by a formerly generated path, is discarded. In our approach we profit from the situation that in Fujaba each element is associated to a unique id. This id is based on a counter that increments with each new created element. That means, the younger an element the higher the id. That leads to the presumption that an element, that is situated later in the path usually has a higher id than all its predecessors, because it was created later. Thus we sort the constraints by ids of their target states into descending order. When we generated a path, then with high probability our target state is the latest in the path and we can discard as many constraints as possible because their target states have already been covered now.

4.3 Testing Environment

To execute the test cases we need an interface to the generated code - a testing environment. As depicted in Figure 3, the testing environment invokes the test cases on the generated code and compares its output to that specified in the test case. In order to monitor test coverage, we subsequently insert a structure into the generated code which collects the visited states and the corresponding clock values. This structure is being passed to the testing environment during communication with the generated code. The given information is sufficient to reconstruct the path passed during test execution. The possible reconstruction is necessary to monitor coverage criteria that differ from state coverage.

This principle of monitoring the test execution is adapted from observer automata. But it is much more economical, since we don't need to traverse the Cartesian product of the observer's and the model's state spaces of the product automaton. Instead we simply collect states in their order of traversing and keep track of those states that still have to be covered.

Legacy components are tested by the integration into components from which the structure is known. Since on legacy components we can only check the behavior that effects their environment, we use the known component to communicate with the legacy component. Thus we can check, if a certain input produces the expected output. As the structure of a legacy component is not known, a statement about test coverage is not possible.

4.4 Example

Figure 4 depicts a system model as modeled in Fujaba Real-Time[3]. To achieve state coverage, we have to build the related constraints. One possible constraint, that causes the model checker to produce a path to State6 with the corresponding timing constraint as a counterexample, is **A**[] not (State49 and $t0 < 30$). The model checker produces a trace in form of a sequence of states and transitions. The states of this trace are highlighted grey in the figure below for visualization.

Each state has a variable vector that contains the variable names associated to their current values. The initial values are just the initial state's variable values. To get the final values, we collect the variables that occur through the path and update a variable at each new occurrence at the successor state. As the states "‘State39"’, "‘State47"’ and "‘State48"’ are already covered by the produced path, their associated constraints need not to be passed to the model checker for test case generation.

[3]The name of the states has no specific meaning.

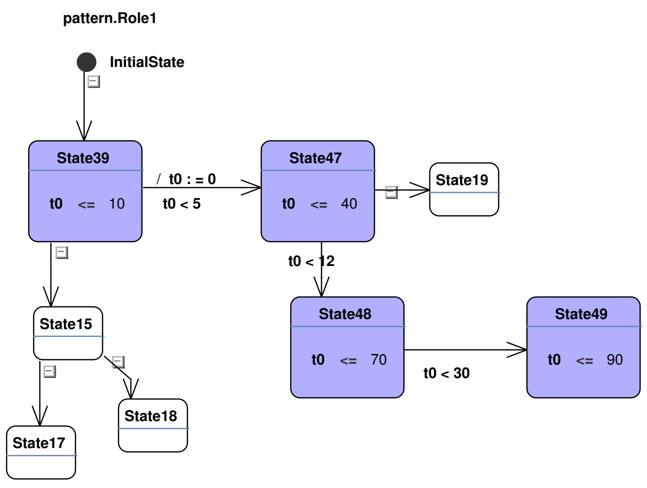


Figure 4: Example Trace

5. CONCLUSION AND FUTURE WORK

We presented a Fujaba plugin for automatic testing of Real-Time Statecharts. We are able to automatically generate test suites for state coverage and also execute and monitor tests. Besides we apply component-wise test generation and execution allowing efficient testing of complex systems. Since the presented plugin has the ability to work with arbitrary coverage criteria, we are planning to expand the test generation and test monitoring to more coverage criteria. As temporal specification languages have it's limitations in specifying coverage criteria (cf. [13]), the model needs to be instrumented to realize further coverage criteria. Future work is to specify the coverage criteria with observer automata. As a request to the model checker is expected to be quite expensive, we propose to further optimize test generation by sorting the constraints not by their id but by their actual depth in the path. This is to be achieved by performing breadth first search on the Real-Time Statechart.

REFERENCES

[1] D. Beyer, A. J. Chlipala, T. A. Henzinger, R. Jhala, and R. Majumdar. Generating tests from counterexamples. *icse*, 0:326–335, 2004.

[2] J. Blom, A. Hessel, B. Jonsson, and P. Petterson. Specifying and generating test cases using observer automata. In U. Assmann, A. Rensink, and M. Aksit, editors, *Proceedings of the 4th International Workshop on Formal Approaches to Testing of Software*, LNCS, pages 125–139, 2004.

[3] G. V. Bochmann and A. Petrenko. Protocol testing: review of methods and relevance for software testing. In *ISSTA '94: Proceedings of the 1994 ACM SIGSOFT international symposium on Software testing and analysis*, pages 109–124, New York, NY, USA, 1994. ACM Press.

[4] S. Burmester, H. Giese, and M. Hirsch. Syntax and semantics of hybrid components. Technical Report tr-ri-05-264, University of Paderborn, October 2005.

[5] S. Burmester, H. Giese, M. Hirsch, D. Schilling, and M. Tichy. The fujaba real-time tool suite: Model-driven development of safety-critical, real-time systems. In *Proc. of the 27th International Conference on Software Engineering (ICSE), St. Louis, Missouri, USA*, pages 670–671. ACM Press, May 2005.

[6] S. Burmester, H. Giese, and W. Schäfer. Model-driven architecture for hard real-time systems: From platform independent models to code. In *Proc. of the European Conference on Model Driven Architecture - Foundations and Applications (ECMDA-FA'05), Nürnberg, Germany*, Lecture Notes in Computer Science (LNCS), pages 25–40. Springer Verlag, November 2005.

[7] 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*, LNCS, pages 1–15, 2005.

[8] S. R. Dalal, A. Jain, N. Karunanithi, J. M. Leaton, C. M. Lott, G. C. Patton, and B. M. Horowitz. Model-based testing in practice. In *ICSE '99: Proceedings of the 21st international conference on Software engineering*, pages 285–294, Los Alamitos, CA, USA, 1999. IEEE Computer Society Press.

[9] H. Giese, S. Burmester, W. Schäfer, and O. 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, Nov. 2004.

[10] H. Giese, M. Tichy, S. Burmester, W. Schäfer, and S. 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, Sept. 2003.

[11] A. Hessel, K. Larsen, B. Nielsen, P. Pettersson, and A. Skou. Time-optimal real-time test case generation using uppaal. In *Formal Approaches to Software Testing, Third International Workshop on Formal Approaches to Testing of Software*, Lecture Notes in Computer Science, pages 114–130. Springer, 2004.

[12] M. Hirsch. Effizientes Model Checking von UML-RT Modellen und Realtime Statecharts mit UPPAAL, June 2004.

[13] H. S. Hong, I. Lee, O. Sokolsky, and H. Ural. A temporal logic based theory of test coverage and generation. In *TACAS '02: Proceedings of the 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 327–341, London, UK, 2002. Springer-Verlag.

[14] K. G. Larsen, P. Pettersson, and W. Yi. UPPAAL in a Nutshell. *Int. Journal on Software Tools for Technology Transfer*, 1(1–2):134–152, Oct. 1997.

[15] T. Mcke and M. Huhn. Generation of optimized testsuites for uml statecharts with time. In R. Groz and R. M. Hierons, editors, *In 16th International Conference on Testing of Communicating Systems, (TestCom 2004)*, Lecture Notes in Computer Science (LNCS), pages 128–143. Springer Verlag, 2004.

[16] J. Offutt and A. Abdurazik. Generating tests from UML specifications. In R. France and B. Rumpe, editors, *UML'99 - The Unified Modeling Language. Beyond the Standard. Second International Conference, Fort Collins, CO, USA, October 28-30. 1999, Proceedings*, volume 1723, pages 416–429. Springer, 1999.

[17] M. van der Bijl, A. Rensink, and J. Tretmans. Compositional testing with ioco. *Proc. 3rd Intl. Workshop on Formal Approaches to Testing of Software*, 2003.

The Alternate Editing Mode for Fujaba

Bernhard Grusie
SE, Kassel University
Wilhelmshöher Allee 73
34121 Kassel
bernhard.grusie@student.uni-kassel.de

Christian Schneider
SE, Kassel University
Wilhelmshöher Allee 73
34121 Kassel
christian.schneider@uni-kassel.de

Albert Zündorf
SE, Kassel University
Wilhelmshöher Allee 73
34121 Kassel
albert.zuendorf@uni-kassel.de

1. INTRODUCTION

Once upon a time, Thorsten Fischer came into the office of Albert Zündorf in order to present his user interface concept for Fujaba, cf. [1, 2, 4]. Albert didn't like it. "It's state of the art", Thorsten replied. Thus, fate took its turn and the PE dialogs were created. History continued. There were many complaints but no escape. Until in 2004 Christian Schneider decided to change it. We started to develope an *Alternate Editing Mode* for Fujaba. The idea was to avoid dialog windows at all and to enable mouse gestures, inline editing, and keyboard oriented editing as much as possible. After a first student failed to finish Christian's work, in 2006 Bernhard Grusie picked up the challenge and developed the Alternate Editing Mode into a usable state. This paper presents the concepts and features of the Alternate Editing Mode plugin for Fujaba.

2. FEATURES

The Alternate Editing Mode is a plugin for Fujaba that adds a button to the central tool bar allowing to switch between the old Fujaba editing mode and the Alternate Editing Mode, cf. Figure 1. If a diagram is opened with alternate editing enabled, the new user interface is switched on for this diagram. Up to now, alternate editing is available for class diagrams and rule diagrams.

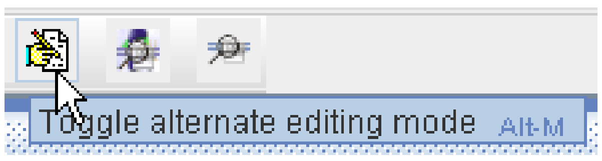


Figure 1: The Alternate Editing Mode Button

2.1 Gestures

After all, UML diagrams consist of lines and boxes. Thus, it should be easy to create these artifacts. The Alternate Editing Mode allows to open a rectangle by dragging with the mouse on the background of a diagram. Depending on the diagram type this either creates a class, cf. Figure 2, or an activity or (within a story activity) an object. For activity diagrams, a left drag creates a story activity while a right drag creates a popup menu showing the available activity types.

Lines may be created by dragging one box on another box. Again, a left drag creates some default connection while a right drag opens a popup menu with availble choices. For classes, dragging may thus create associations or inheritance links or directed associations or aggregations or compositions. For activities dragging creates transitions and for objects dragging creates links. On the creation of links, the Alternate Editing Mode checks available associations and by default chooses one of them or presents them on right dragging.

As an additional comfort, one may also drag the ends of existing associations from one class to another. Similarly, inheritance relations may be changed. The same holds for transitions[1] and links within rule diagrams.

Whithin class diagrams it is also possible to drag attribute or method declarations from one class to another. Shift dragging of attributes or methods creates a copy, however, currently this works only for attributes. Here we rely on the new Copy&Paste Module for Fujaba, cf. [3].

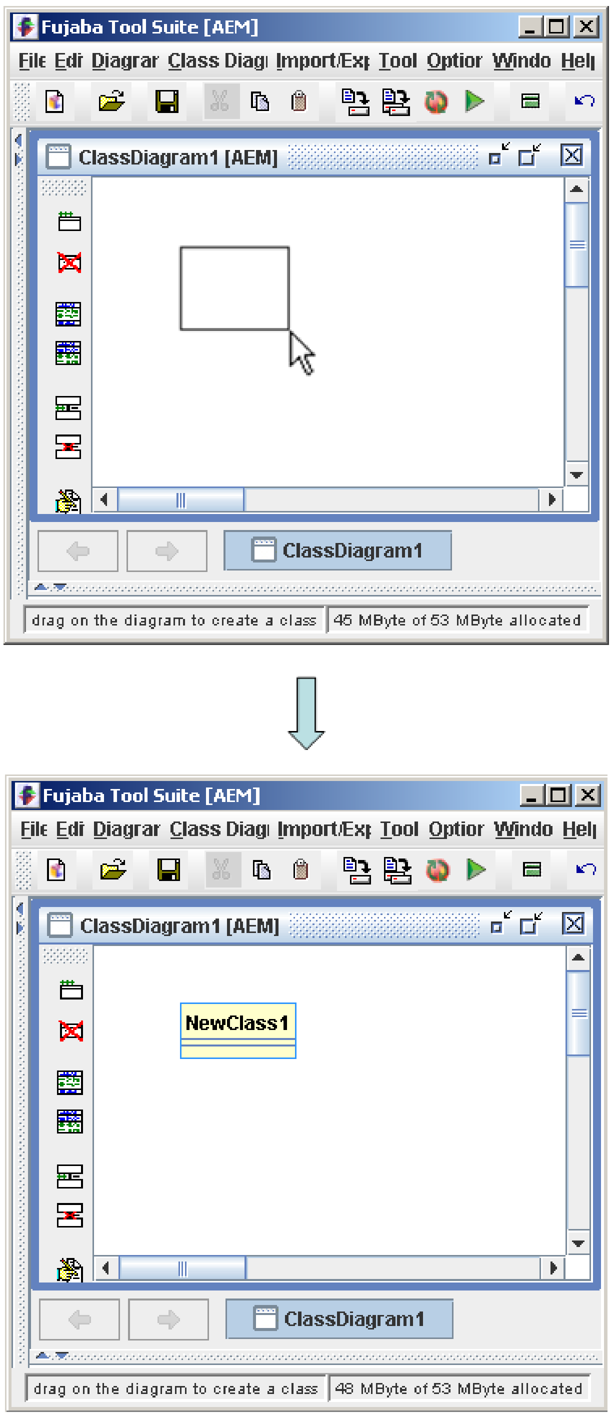
2.2 Text Input Field

Editing with gestures creates default names and types for the new elements. In order to enable dialog free changing of names and types and in order to enable dialog free editing of other kinds of inscriptions like attributes, methods, transition guards, attribute conditions, or collaboration messages, the Alternate Editing Mode provides a context and input depended text field.

The text input field opens below the currently selected diagram elements as soon as one starts to type some input. As an example, Figure 3 shows the text input field for a class.

On closing of the input field, the Alternate Editing Mode parses the text input and decides depending on the selected diagram element what item of the model is going to change. In case of a class, a simple word is interpreted as a new class name. If the text contains round brackets, the Alternate Editing Mode assumes that a new method declaration has been entered. In case of a colon, an attribute declaration is added. If an attribute declaration is selected, a normal word would change the attribute name. A word after a colon is interpreted as the type of the selected attribute. Note, the Alternate Editing Mode provides completion for types on control blank.

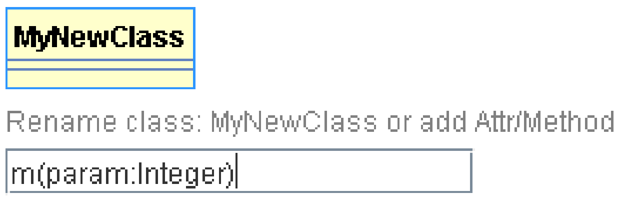
[1]To be honest, dragging transition ends is not yet working



drag on the diagram to create a class | 45 MByte of 53 MByte allocated

drag on the diagram to create a class | 48 MByte of 53 MByte allocated

Figure 2: Gesture creating a class



Rename class: MyNewClass or add Attr/Method

m(param:Integer)

Figure 3: Text Input Field for a class

Similarly, the input field for an association allows to change the association name. The input field for a role allows to change the role name or the cardinality may be changed e.g. to "0..*".

On any element, "<<x>>" may be used to add steretype "x" and ">>x<<" may be used to remove it.

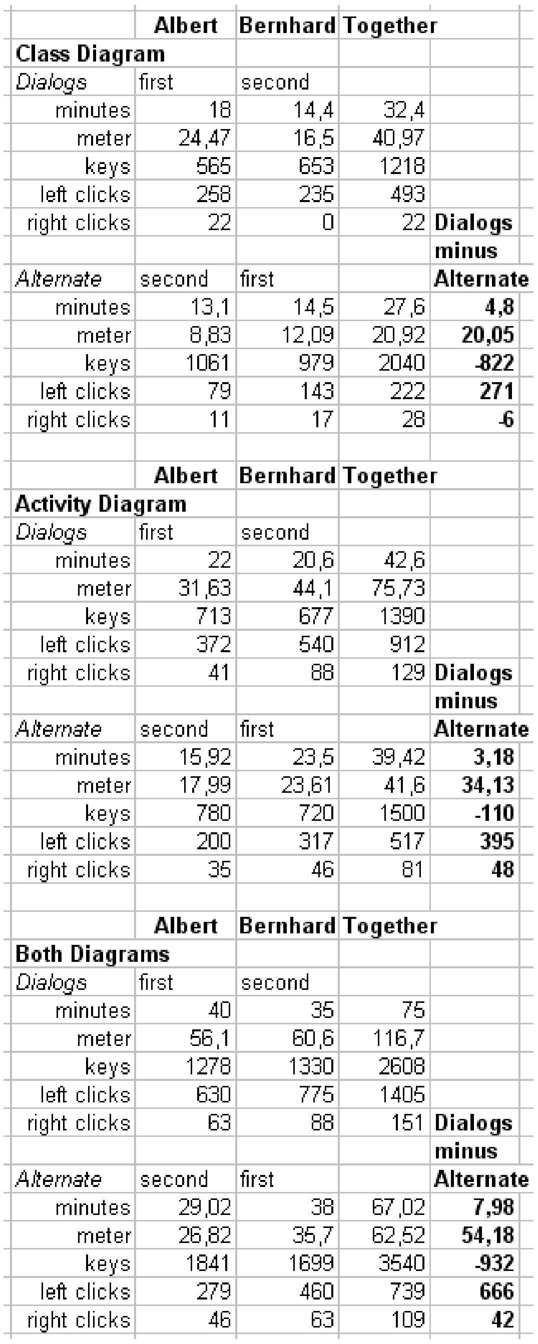
The most sophisticated parsing of input lines happens for activity diagram elements. On objects, with a plain word one may change the object name or with a colon one may change the object type. In addition, an attribute constraint may be entered. For objects and links, ++ and −− add modifiers while == removes the modifiers. ## toggles the bound property of objects. And so on. Basically, in activity diagrams almost all properties that may be changed using the former dialogs may be altered using special characters at the input line. Note, the status line at the bottom of the Fujaba window shows a short help for special characters available for the selected diagram element.

3. EXPERIMENTAL COMPARISON

As a first little comparision of the old dialog mode with the alternate editing mode we have run a simple experiment. Albert and Bernhard have both edited the class diagram and the activity diagram shown in the appendix one time using the dialog mode and one time using the Alternate Editing Mode. Both guys are familar with both editing modes. Well, as developers of the Alternate Editing Mode they may be a little biased. Since each of the two candidates have to enter the diagrams twice, we expect a certain learning curve. I.e. they should be faster the second time. In order to deal with this learning curve, Albert did the dialog mode first and the Alternate Editing Mode second and Bernhard the other way round. We used mouseometer as a tool for measuring the time, the key strokes, the overal mouse movement, and the mouse clicks during editing. Table 4 shows the results of this experiment.

First of all, both candidates show the expected learning curve as they both needed about 40 minutes for the first round and about 30 minutes for the second round. Overall, the learning curve was more significant to the time consumption than the editing mode used. And due to the small size of the experiment, we do not consider the 12% time reduction as statistically significant. However, concerning key strokes, mouse meters, and mouse clicks the learning curve does not show up. Both candidates used more key strokes in alternate editing and more mouse activity in the dialog mode. This meets our expectation.

The bottom of the right most column shows the differences between the sum of the measurements of both candidates for the different modes. In the sum of both candidates, the Alternate Editing Mode saved about 8 minutes, 54 meters of mouse movement, and 700 mouse clicks. This is payed for by 930 additional key strokes, 888 of these extra key strokes already in the class diagram. The latter number is unexpected high. A closer examination revealed that in alternate editing one has to enter additional syntax like parenthesis and colons which sums up to about 240 key strokes as sum for both candidates. Note, to enter a parenthesis or a colon on a german keyboard you need the shift key which is counted as a seperate key stroke by the mouseometer. In addition, in alternate editing we have typed in the ≪JavaBeans≫ stereotype while in the dialog mode this is just a click. This accounts for another 432 of the extra key strokes. 80 of the remaining extra key strokes are probably accounted for starting the type completion using ctrl blank (2 to 3 key strokes each time).



	Albert	**Bernhard**	**Together**	
Class Diagram				
Dialogs	first	second		
minutes	18	14,4	32,4	
meter	24,47	16,5	40,97	
keys	565	653	1218	
left clicks	258	235	493	
right clicks	22	0	22	**Dialogs**
				minus
Alternate	second	first		**Alternate**
minutes	13,1	14,5	27,6	**4,8**
meter	8,83	12,09	20,92	**20,05**
keys	1061	979	2040	**-822**
left clicks	79	143	222	**271**
right clicks	11	17	28	**-6**
	Albert	**Bernhard**	**Together**	
Activity Diagram				
Dialogs	first	second		
minutes	22	20,6	42,6	
meter	31,63	44,1	75,73	
keys	713	677	1390	
left clicks	372	540	912	
right clicks	41	88	129	**Dialogs**
				minus
Alternate	second	first		**Alternate**
minutes	15,92	23,5	39,42	**3,18**
meter	17,99	23,61	41,6	**34,13**
keys	780	720	1500	**-110**
left clicks	200	317	517	**395**
right clicks	35	46	81	**48**
	Albert	**Bernhard**	**Together**	
Both Diagrams				
Dialogs	first	second		
minutes	40	35	75	
meter	56,1	60,6	116,7	
keys	1278	1330	2608	
left clicks	630	775	1405	
right clicks	63	88	151	**Dialogs**
				minus
Alternate	second	first		**Alternate**
minutes	29,02	38	67,02	**7,98**
meter	26,82	35,7	62,52	**54,18**
keys	1841	1699	3540	**-932**
left clicks	279	460	739	**666**
right clicks	46	63	109	**42**

Figure 4: Experiment Data

Overall, the new editing mode saves about 12% time within this small experiment. It seems that the additional key strokes are outweight by the saved mouse activities. Qualitatively, the two candidates have the impression that using the Alternate Editing Mode for class diagrams is much more convenient than the dialog mode. The main advantage is that one does less frequently switch between mouse and keyboard. In activity diagrams, the object dialogs are somewhat easier to use than the Alternate Editing Mode. In dialog mode, you start the dialog with alt O, you enter the name, you tab to the type field and you enter some letters until the desired type shows up. Return closes the dialog. In alternate editing, you drag a rectangle to create the object, you type the object name, a colon, some letters of the type name, ctrl blank for completion and return to submit. Thus, in alternate editing you have a change from mouse to keyboard between dragging the object and entering its name. In addition, starting the completion mode in alternate editing needs an extra crl blank. Thus, as a result of this experiment, we plan to improve alternate editing on objects.

4. CONCLUSIONS

By now, the Alternate Editing Mode is almost stable. There is still a list of missing features and known bugs. There are a number of users at Kassel University that use the Alternate Editing Mode for serveral month now. The reported impressions are very supporting. In addition, the experiment reported in this papers gives some hints on advantages of the Alternate Editing Mode. Editing with the Alternate Editing Mode is more convinient than the old dialogs due to lesser changes between mouse and keyboard. Concerning functionality, the alternate editing mode is sufficiently complete such that the old dialogs are only seldom used. In seldom cases the Alternate Editing Mode behaves unexpected and performs undesired model changes. However, the gain in editing comfort outweighs still exisiting problems by far. Thus, the usage of the Alternate Editing Mode is now recommended for experienced Fujaba users.

The Alternate Editing Mode plugin for Fujaba is available via Fujaba's plugin download mechanism from University of Kassel.

5. REFERENCES

[1] T. Fischer, J. Niere, and L. Torunski. Konzeption und Realisierung einer integrierten Entwicklungsumgebung fr UML, Java und Story-Driven-Modeling (Diploma Thesis, german), 1998.

[2] T. Fischer, J. Niere, L. Torunski, and A. Zündorf. Story Diagrams A new Graph Grammar Language based in the Unified Modeling Language. In *TAGT '98 - 6th International Workshop on Theory and Application of Graph Transformation.* Technical Report tr-ri-98-201, University of Paderborn, 1999.

[3] L. Geiger and C. Schneider. The Copy Paste Modul for Fujaba. submitted to: International FujabaDays 2007, Kassel, 2003.

[4] The Fujaba Toolsuite. http://www.fujaba.de/, 2006.

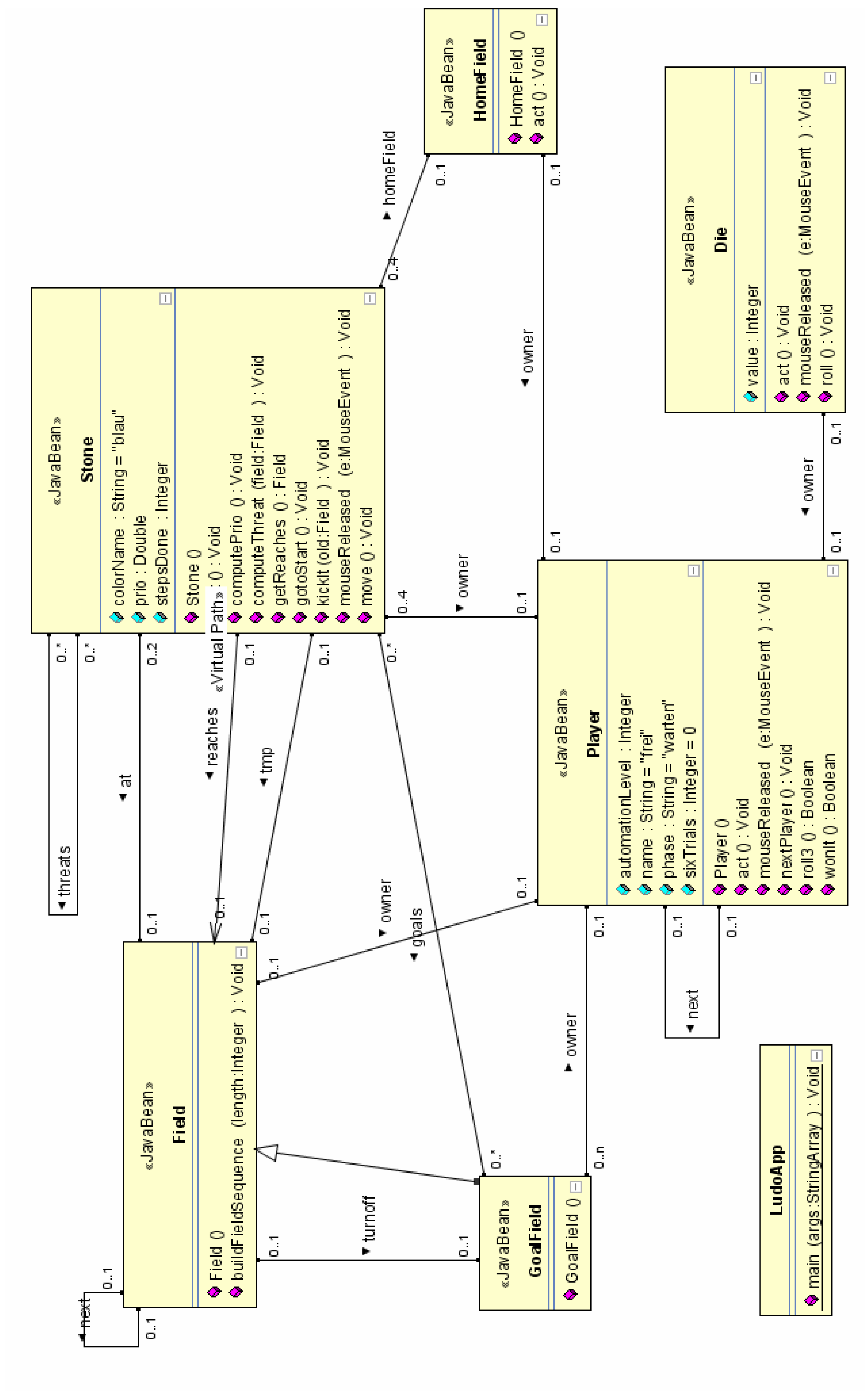


Figure 5: Example Class Diagram

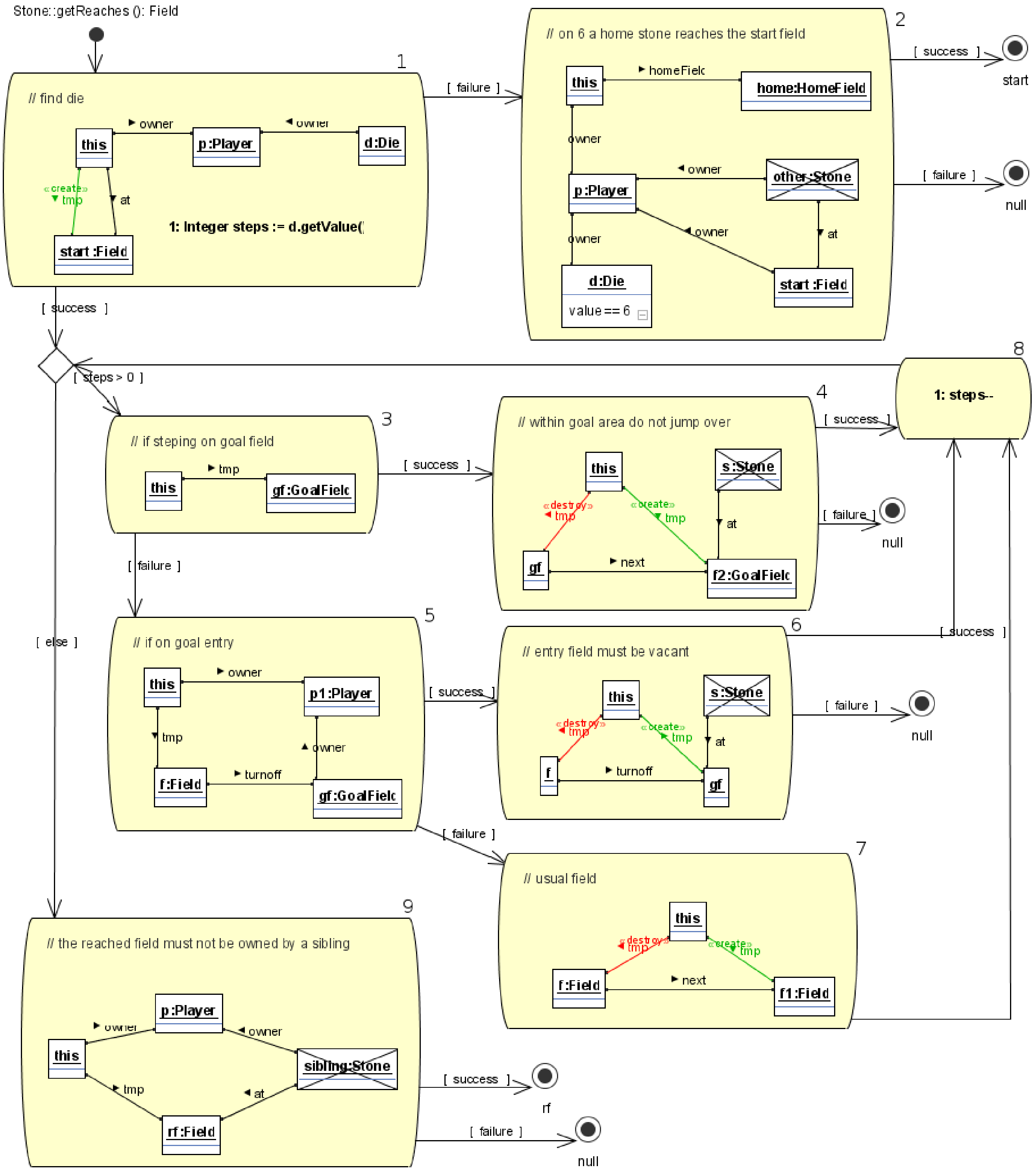


Figure 6: Example Activity Diagram

Copy & Paste concept and realization in Fujaba

Leif Geiger
SE, Kassel University
Wilhelmshöher Allee 73
34121 Kassel
leif.geiger@uni-kassel.de

Christian Schneider
SE, Kassel University
Wilhelmshöher Allee 73
34121 Kassel
christian.schneider@uni-kassel.de

ABSTRACT

This paper describes a generic approach to implementing a copy & paste framework. The framework should be flexible and easy to maintain. Especially new meta model elements (e.g. such as added by plugins) should be easy to adapt to work with copy & paste, as well. Our approach uses role categories to build the set of objects and links to copy and to restore external references when pasting. We have implemented the framework using CoObRA 2 [3, 4] as serialization framework and Java annotations to sort the roles into a role category. We have integrated the framework into Fujaba[1] and implemented copy & paste for UML class and activity diagrams as well as SDM diagrams.

1. INTRODUCTION

We faced the task to implement copy & paste for Fujaba and other research applications. Given a nice persistency library the actual task seems quite easy. Copy some objects with all their properties and probably move them into a new context. But there are several problems that make this task complicated: It was unclear how one would determine the set of objects to be copied. It had to be decided which links to uncopied objects should be restored. And the effect of moving objects into a new context needs to be defined.

Usually a set of objects is chosen to be copied, e.g. by selection markers set by the user in a graphical application. Though this initial set of objects does not suffice for the actual copy mechanism. When the user selects a UML class to be copied he would expect the methods in that class to be copied as well. But typically these are separate objects in the application model. On the other hand the user would not expect to get a copy of the whole diagram the class resides in, even though that diagram object is probably referenced by the class to be copied. Furthermore it is expected to maintain the result type of the copied method, without getting a copy of the referenced class. These copied links to uncopied objects are called *external references* throughout this paper.

Obviously the actual set of object that should be copied depends in the initial set of objects (user selection) and some specific properties of the meta model. Figure 1 shows the mentioned scenario of objects.

The second challenge when copying objects is to determine the context of a copy (or cut) operation and afterwards the context of a paste operation. Additionally one might be interested in the difference between these contexts to apply proper relocation of the pasted objects.

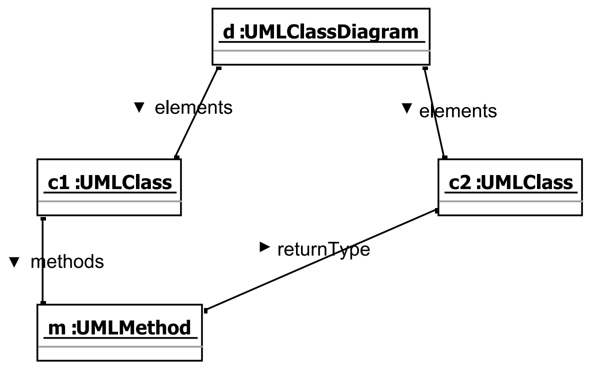


Figure 1: Object structure excerpt of a UML class diagram in Fujaba

In our simple example of copying a UML class the copy context could be the class diagram the class is selected in. When pasting the context could be the currently visible class diagram to have the copied elements shown in the current view. On the other hand, the user might want to paste the class into a different package. In this case the diagram should not get replaced by the package but the copy context must take the current package of that class into account.

So the contexts to copy from and to paste into depend on the copy and paste actions the user might perform. The context also seems to relate to some meta model semantics.

2. ROLE CATEGORIES

Given an initial set of objects a search algorithm to find the set of object to be copied can be easily sketched: For each object in the set the neighbor objects, reachable via a link, are added to the set, if that link's role is known to be a *containing* role, regarding the copy semantics. This is repeated until no more objects matching this condition are found. The crucial point is to determine the set of *containing* roles. Which is the mentioned property of the meta model.

After computing the set of copied objects the set of copied links must be determined. This usually contains all links

[1]The Fujaba Toolsuite, http://www.fujaba.de/

from a copied object to another copied object plus some *external* links (links from a copied object to an object that is not copied). If an external link is copied, again, depends on a list of roles to be defined in the meta model.

To summarize, we need to specify two boolean flags for the association roles:

- One flag to specify if object referenced via this role are copied with the referencing object.
- One flag to specify if references via this role are copied even if the referenced object is not copied.

Obviously the second flag makes sense only if the first one is false. So this results in three categories for the association roles.

In Fujaba 4 we tried to apply heuristics for choosing the categories for the roles and thus the set of objects and the external references to be copied, based on the cardinalities of associations. While defining lots and lots of exceptions to these rules we realized that cardinalities were not suited as a decision base but the category must be specified or derived from other role properties.

The context of a copy operation is defined via a subset of the set of external references. In our example scenario this would be the reference to the class diagram if a UML class is copied. Thus for the context definition we need to introduce a fourth category of association roles. In conclusion we get these categories for the roles:

- **Containment**: Objects referenced from the set of copied objects via this role are added to the set of copied objects.
- **Copied reference**: References from the set of copied objects via this role are added to the set of copied references.
- **Copied context reference**: This category is a superset of the *copied references* and has the same meaning, but additionally the referenced objects are marked as context objects.
- **None**: External references via this role can be ignored by the copy mechanism.

3. META MODELS

The analysis of the Fujaba UML meta model, the UML 2.0 meta model and some other research meta models we found a strong match between association role adornments and the role categories for the copy mechanism:

- Usually one would expect to get neighbors copied with an object if they are linked via composition associations, so composition maps to the *containment* category introduced before.
- Links which are instances of an aggregation should be copied even if they are external references, thus aggregations define a part of the *copied references.*
- There are additional links that must be copied as external references which are instances of associations without any adornment, these define the other part of the *copied references.*
- The opposite direction of composition and aggregation usually link to context objects (in UML 2.0 the parent role may be to-many), hence we have a correlation to the *copied context references.*

For our initial example these adornments are annotated in Figure 2. The *elements* association from UMLClassDiagram to UMLClass is an aggregation, thus the links got a clear diamond. These links are copied if the diagram is copied and mark the diagram as context if the UML class is copied - matching the expectations. The *methods* association, in turn, is a composition, depicted as filled diamond on the links in Figure 2. So methods are included in the set of copied object if the UML class is copied, while the class is marked as context if a method is copied. Finally one direction of the *returnType* association must be annotated to be stored. This is indicated by the dashed arrow head.

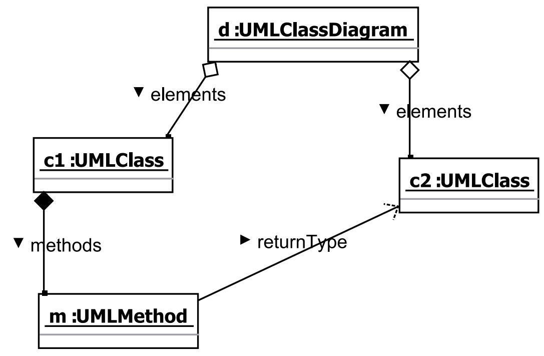


Figure 2: Object diagram with annotated role adornments

The other major subtask for copy & paste is to come up with a method to identify objects which are referenced by copied objects but are not copied themselves. A common approach to have object references persisted is to create globally unique identifiers for the objects and persist those. But this does not take the specific needs for context changes into account: An object copied from one context and pasted into another context might get different neighbors depending on the second context. So instead of generating an ID the Identifiers have to become 'meaningful'.

Thus an appropriate ID for a UML class could be it's full qualified name. If the package of that class is the context, on the other hand, the ID could be the unqualified name of the class. While pasting a UML method object into another Fujaba project the return type of that method would be resolved by the name of that type, not relying on some unique persistence ID. The concept of namespaces for e.g. object diagrams can be mapped to these context IDs, as well.

4. IMPLEMENTATION

The Java Feature Abstraction[2] and CoObRA 2 [3, 4] were extended to provide basic support for copy & paste. The Feature Abstraction library can read adornments from the meta model and provides visitors for traversing the model to find neighbors and references for the set of copied objects and links. CoObRA on the other hand provides a mechanism for generating persistency data of visited objects with custom context identifiers. It provides an interface for obtaining tool specific identifiers and resolving them again.

4.1 Fujaba

We decided to specify the role categories directly in the source code to make it easy for e.g. plugin developers to adapt their code for copy & paste. Thus we employed Java annotations as a means to denote the adornments. We already used a `Property` annotation in some of our classes which is used to mark fields and their accessors as a property. A `Property` has a name, a partner (the reverse link, if it exists) and a kind (attribute, to-one, to-many). We added an `adornment` attribute to that annotation. The possible values for this attribute are defined by an enumeration named `ReferenceHandler.Adornment`:

Unknown: The adornment of the annotated property is not known. Our implementation will show a warning if it finds such an annotation while copying.

None: Properties marked with the `NONE` adornment are not copied.

Composition: The annotated property is the containing property of a composition, this maps to the *containment* category from Section 2. The partner property must be marked as `PARENT`.

Aggregation: The annotated property is the containing property of an aggregation. It's instances are *copied references*. The partner property must be marked as `PARENT`.

Usage: The property is a external reference which still has to be copied. Instances are *copied references* as well.

Parent: The property is the link to the parent of an aggregation or a composition. The links are *copied context references* (same as copied references but denote that the target is part of the context.

We have changed the Fujaba code generation in order to generate such annotations for models specified in Fujaba. The `Composition`, `Aggregation` and `Parent` adornments are generated from the association adornments in the model, `None` is the default value for normal association and `Usage` has to be specified explicitly using stereotypes. The retention policy of the `Property` annotation is set to runtime which means that this annotation can be accessed at runtime using reflection.

Our copy mechanism now uses the Feature Abstraction library to read these annotations. In case of the Fujaba meta model this in turn uses Java Reflection to read the annotations. With this information it can then build the set of objects and links to copy. The set of possible context objects is calculated as well. The set of objects and links to copy is then serialized to text using CoObRA and stored in the clipboard. To get an ID for externally referenced objects the method `getContextIdentifier()` is called on those objects. The set of contexts is passed as parameter. The method is declared in `ASGElement`, the abstract superclass of every Fujaba meta model element, and implemented by its concrete subclasses. The `getContextIdentifier()` method returns an Identifier for the object which is unique for the passed context.

When the paste action is executed the textual representation in the clipboard is deserialized. To restore the external references we have introduced a singleton called `NamespaceManager`. The `NamespaceManager` has a chain of responsibility of `NamespaceHandlers`. If an external ID has to be resolved the `findByContextIdentifier()` method of the `NamespaceHandler` in the chain are called until one returns a result. The context to paste is passed as parameter as well as the ID, the class of the object to find and if the object to find itself was a context.

We have added the annotations needed for copy & paste and implemented the methods needed to serialize and deserialize external references for Fujaba's class diagrams. This implementation has already been tested by the community and can be classified as stable.

4.2 Plugin Meta models

Using the class diagram example from above, we will show how plugin developer can make their own meta models capable for copy & paste. First, one needs to add the annotations: If the meta model is specified in Fujaba one can easily change its associations to compositions or aggregations or add stereotypes where needed. After generating code with CodeGen2[1] the needed annotations are generated into the source code. Alternatively, one may write the needed annotations into the code manually.

For the class diagram part we imported the Fujaba meta model in Fujaba. We used the reflection based importer deployed with the *Refactorings* plugin since it is able to import existing annotations. The annotated class diagram of our short example is shown in Figure 3. Afterwards we generated code from this model. Since the Fujaba code differs a lot from the generated code, we could not replace the original with the generated. So we wrote an eclipse plugin which only copies the annotation of one source folder to the code in another source folder. Using this plugin we copied the correct annotations into the original Fujaba source code. The code for the field `resultType` of class `FMethod` is shown in Listing 1.

Listing 1: Annotations of field `FMethod.resultType`

```
public final static String
   RESULT_TYPE_PROPERTY = "resultType";

@Property(name=RESULT_TYPE_PROPERTY,
  partner=FType.REV_RESULT_TYPE_PROPERTY,
  kind=ReferenceHandler.ReferenceKind.TO_ONE,
```

[2]http://www.se.eecs.uni-kassel.de/se/?features

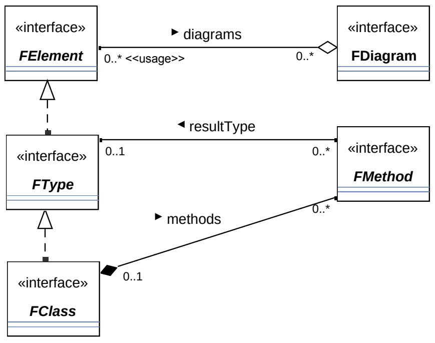


Figure 3: Class diagram of class diagram example

```
  adornment=ReferenceHandler.Adornment.CONTEXT)
public void setResultType(FType value);


@Property(name=RESULT_TYPE_PROPERTY)
public FType getResultType();
```

Next we needed to implement the `getContextIdentifier()` method for all classes that were target of an external reference which should by copied. In our example this is e.g. the `FClass`. A class can typically be identified by its full qualified name independent of the context, so our implementation for `UMLClass` looks like shown in Listing 2.

Listing 2: `getContextIdentifier()` of class `UMLClass`

```
public String getContextIdentifier
  (Collection<? extends FElement> context)
{
   return getFullClassName();
}
```

The last thing to do is to implement `NamespaceHandler`s which are able to resolve the stored external references. As example we show the `NamespaceHandler` for a project context when a class is passed in Listing 3. Here, if the passed ID is a reference to an `FClass` (line 9-10), the class in the context project with the same full qualified name is returned (line 12-14).

Listing 3: `NamespaceHandler()` for projects

```
 1 class ProjectNamespaceHandler
 2    implements NamespaceHandler
 3 { ...
 4    public FElement findByContextIdentifier
 5       (FElement context, String identifier,
 6         ClassHandler type, boolean isContext)
 7       throws ElementNotFoundByContextIdentifier
 8    { ...
 9       if (module.getClassHandler(FClass.class.
10           getName()).isAssignableFrom(type))
11       {
12          return project.getFromFactories
13            (FClass.class).getFromProducts
14            (identifier);
15       }
16    }
17 }
```

Finally the `NamespaceHandler` has to be registered at the `NamespaceManager` singleton to make copy paste work.

5. RELATED WORK

Existing frameworks for copy & paste focus on providing an interface for the operations, while the actual implementation is left on behalf of the model developer. In the Eclipse Modeling Framework[3] the documentation suggests to copy objects regarding to the MOF containment hierarchy, but the effects of external references (references to objects which are not copied) remain unclear.

Porres and Alanen do present a more generic aproach in a technical report [2]. They utilize the containment hierarchy of MOF meta models to determine the set of instances to be copied in a deep copy operation. Though decisions about external references are not handled explicitly. The only rule proposed in by Porres and Alanen is to omit references with a backward to-one cardinality. Like in the EMF Framework Models are expected to be selfcontained. This means no cross-references between models are respected.

6. CONCLUSIONS

The approach to use association adornment and additional *usage* annotations in the meta model to determine the set of copied objects and set of copied external references enabled us to integrate support for copy & paste in Fujaba's class and SDM diagrams quickly. The NamespaceManager handles resolution of the context identifiers, which are individually created by all referenced elements. In contrast to the former copy & paste implementation in Fujaba 4, based on heuristics on the role cardinalities, the new implementation appeared reliable and stable in the tests.

Due to the easy extensibility we hope to make use of the new functionality in package diagrams, state charts and plugin meta models soon, to get comprehensive copy & paste support throughout Fujaba 5.

7. REFERENCES

[1] L. Geiger, C. Schneider, and C. Record. Template- and modelbased code generation for MDA-tools. In *3rd International Fujaba Days*, Paderborn, Germany, 2005.

[2] I. Porres and M. Alanen. A generic deep copy algorithm for mof-based models. Technical report 486, ISBN: 952-12-1073-7, Turku Centre for Computer Science, Software Construction Laboratory, 2002.

[3] C. Schneider. CASE Tool Unterstützung für die Delta-basierte Replikation und Versionierung komplexer Objektstrukturen (Diploma Thesis, german), 2003.

[4] C. Schneider, A. Zündorf, and J. Niere. CoObRA - a small step for development tools to collaborative environments. In *Workshop on Directions in Software Engineering Environments; 26th international conference on software engineering.* ICSE 2004, Scotland, 2004.

[3]EMF, http://www.eclipse.org/modeling/emf/

EMF Code Generation with Fujaba

Leif Geiger
Universität Kassel
Wilhelmshöher Allee 73
34121 Kassel
leif.geiger@uni-kassel.de

Thomas Buchmann
Universität Bayreuth
Universitätsstr. 30
95447 Bayreuth
thomas.buchmann@uni-bayreuth.de

Alexander Dotor
Universität Bayreuth
Universitätsstr. 30
95447 Bayreuth
alexander.dotor@uni-bayreuth.de

ABSTRACT

Fujaba is a powerful tool for model driven development. But when it comes down to the development of graphical user interfaces, developers are still in need of massive manual coding. On the other side, GMF provides a way of generating graphical user interfaces, but it is tightly coupled to an underlying EMF model. In this paper we show a way how to extend Fujaba's code generation to preserve most of Fujaba's modeling and code generation benefits and map the Fujaba model onto a EMF specification.

1. INTRODUCTION

In its current state Fujaba generates standard Java code which can be easily integrated into most Java applications. The eclipse project offers a platform for editors and similar tools. With GEF and GMF [6] eclipse offers frameworks which simplify the construction of visual editors. To use these frameworks the underlying meta model has to be compatible to the EMF code style. It would be nice to be able to combine these two powerful tools: Fujaba and eclipse / GMF. Then the meta model and the transformations made on it can be done with Fujaba and the construction of the visual editor, menus, toolbars etc. can be built using eclipse / GMF. This work now introduces an EMF code generator for Fujaba.

2. COMPARING THE FUJABA META MODEL, ECORE AND EMOF

To be able to generate EMF compatible code from models specified in Fujaba it is necessary to define a mapping from the concepts used in Fujaba onto those used in EMF. In other words it is necessary to map the Fujaba metamodel onto Ecore. By comparing both meta models it is possible to identify which features of Fujaba can be mapped onto Ecore at all and which features will be lost during the generation process.

We compare the Fujaba metamodel as it is implemented in Fujaba 5 with the Ecore model as it is defined in EMF 2.3.1 [4]. As the EMF introductions says "There are small, mostly naming differences between Ecore and EMOF" [5], we compare both meta models with EMOF 2.0, too [11]. Table 1 shows the result of the comparison. The first column names the compared feature while the following columns contain either *yes* or *no* depending on whether the feature is available in the model or not accordingly.

Feature	Fujaba 5	Ecore 2.3.1	EMOF 2.0
Classes			
abstract	yes	yes	yes
interface	yes	yes	no
reference	yes	yes	yes
Attributes			
read-only	no	yes	yes
final	yes	no	no
static	yes	no	no
transient	yes	yes	no
default value	yes	yes	yes
Methods			
public	yes	yes	yes
protected	yes	no	no
private	yes	no	no
static	yes	no	no
exceptions	yes	yes	yes
Associations			
1:1	yes	yes	yes
1:n	yes	yes	yes
n:n	yes	yes	yes
unidirectional	yes	yes	yes
ordered	yes	yes	yes
int. qualified	yes	yes	no
ext. qualified	yes	no	no
Behavior			
Behav. model	yes	no	no

Table 1: Comparison of Fujaba metamodel and Ecore

Fujaba provides the most features followed by Ecore and EMOF at last which means that Fujaba is far more expressive than Ecore or EMOF. Table 1 highlights in bold letters the features of the Fujaba meta model that are lost when it is mapped onto Ecore. The following constraints must be made to be compatible to Ecore:

1. no concept of *static*, i.e. static final constants.

2. public methods only.

3. no externally qualified associations.

4. no behavioral modeling.

While the first three constrains are acceptable the 4th one is not as it renders one of the key features of Fujaba useless. So the code generation for EMF must preserve the behavioral model by generating the java code for it separately.

The comparison to EMOF shows that its features are a subset of Ecore. This means a mapping from Fujaba to EMOF must fulfill all constraints for Ecore and three more: no interfaces, no transient attributes and no qualified associations at all (highlighted in italics in table 1). Additionally we want to point out that EMOF is a true subset of Ecore and the statement "There are small, mostly naming differences between Ecore and EMOF" [5] does not hold anymore.

3. EMF CODE GENERATION

For generating EMF compatible java code from Fujaba models there exist two different approaches:

1. Generating EMF compatible java code directly from Fujaba models

2. Generating a Ecore model file and use the EMF code generator to generate the java code

The first approach has the benefit that the Fujaba code generation does not depend on other code generators to generate executable code and that the code executing the graph transformations can be easily integrated into the generated classes. As drawback the code generation has to be changed every time the EMF code generation changes. That was the reason (and because it meant much less work) why we decided to implement the second approach. That keeps the maintainance effort low even if the EMF specification changes but on the other hand we need to inject the code for the graph transformations into the code generated by the EMF code generator. Fortunately, the EMF code generator offers a way that makes this very easy.

Figure 1 depicts the various artifacts used during the code generation process as well as their dependencies. The code generation for EMF can be separated in two distinct parts concerning the structural (color on diagram: yellow) and the behavioral part (color in diagram: orange) of the Fujaba model. The first part maps the structural elements of the model (i.e. classes and associations) onto equivalent elements of the Ecore model. The second part generates immediately java code for the behavioral part of the model (i.e. story diagrams). To generate a fully executable java model the EMF code generation must be used to generate java code for the structural part of the model. The EMF code generation is able to merge the already generated code for behavioral part with the newly generated structural code.

We used Fujaba's template based code generation CodeGen2 [8] to generate the needed artifacts. CodeGen2 supports so called `CodeStyles`. That means that the developer can mark

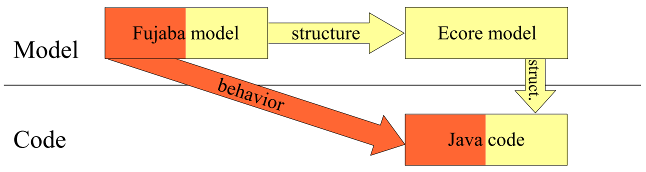


Figure 1: Different levels of code generation

model elements with `CodeStyle` tags. It is now possible to use a different set of templates for e.g. a EMF `CodeStyle` tag. We wrote such templates that are able to generate an XMI file representing the Ecore model from the model elements marked as EMF. Additionally we changed the java code generation so that it generates EMF implementation classes containing only the code for the graph transformation rules for such model element. So, the generated java code only contains the methods and still lacks the implementation of attributes and associations. This is added by the EMF code generator once it is invoked on the generated XMI file.

The generation of the XMI file is straight forward: Fujaba classes are mapped to EMF classes, methods to EMF operations etc. The generation of the java files requires some more work since the EMF mechanism for creation and deletion of objects and links have to be used.

Object creation in EMF can not be done using the `new` operator, one has to use factories. There is one factory responsible for each package which is able to create objects for all classes of this package. This factory can be found using EMF naming conventions (package name followed by `Factory`. Such object creation is done by the following (simplified) template snippet:

```
#set( $factory = "$utility.upFirstChar
      ($package.Name)Factory" )
$name = "${factory}.eINSTANCE.create${type}()";
```

Object removal is still done calling the `removeYou` method which removes all links of the object to make it collectable for the garbage collection. All other object operations like bound checks or assignments can be done as in the java code generation.

For link queries and manipulations one has to consider that some access methods for EMF associations differ from those normally generated by Fujaba. For to-1 associations nothing changes. So the old templates can be used. But for to-many associations instead of a `iteratorOf<role name>` method which returns a `Iterator`, EMF has a `get<role name>` method which returns the set. All queries and mutations have to be executed on this set. So creating a link results in an `add` operation, deleting in a `remove` operation etc. This can be easily implemented with the templates. Since all to-many associations in EMF are ordered what means they are implemented by lists, one may always specify an integer as range when querying a link. But if that index is not valid an `IndexOutOfBoundsException` is thrown which the Fujaba generated code can not handle. So we added an index check before every check or search operation with a specified range. Such an index check is generated by the template shown below:

```
fujabaIndex = $range;
JavaSDM.ensure ((fujabaIndex >= 0) &&
  (fujabaIndex < ${name}.get$roleName ().size()));
```

Another operation that might be thrown when manipulating EMF object structures is the `ConcurrentModificationException`. This exception is thrown when a set is changed while iterating through it. This might happen in Fujaba in an forEach activity when a link which is used for searching is then deleted or a new one of the same type is added. The standard generated code uses set classes which do not throw this exception. For EMF code this is not possible. This is why we decided to use a pre-select semantics (cf. [15, 14]) for forEach searches here. We copy the set and iterate through the copy. Changes are performed on the original set. So changes will not affect which objects are visited during the search. This is a slight difference to the semantics normally used in Fujaba but we consider it as a minor drawback.

The last missing element of story diagrams are path expressions. Such expressions are generated as Strings into the code and interpreted at runtime. So we do not have to change the code generation here but adapt the path interpreter. There is an implementation of the path interpreter which uses the so called feature abstraction developed at the university of Kassel for link searches. The feature abstraction defines an abstract way to access classes, their methods and their fields. There exists implementations for the java reflection layer, the java debug interface and JMI. We wrote an implementation for EMF. This implementation can be used by the path interpreter and so path expressions do also work for EMF.

EMF offers methods to serialize object structures to XMI. Anyhow using CoObRA [12, 13], the framework for object persistency in Fujaba, has several benefits. In addition to persistency for object structures CoObRA offers generic support for undo-redo operations and multiuser environments including merging of object structures. Fortunately, CoObRA also uses the feature abstraction to access the model. So the implemented EMF feature abstraction can be used here, too. To be able to have full CoObRA support for the generated EMF models, it was neccessary to implement an adapter from EMF property changes to JavaBeans property changes which are used in CoObRA.

4. EXAMPLE

In this section we present a small example to show how the EMF code generation works. The example is derived from the project management domain and is a simple model derived from dynamic task nets [9] and is a simplification of our dynamic task net editor [2]. The model is depicted as UML class diagram in figure 2 and it consists of three classes:

1. `DynamicTaskNet`, which instances represent dynamic task nets

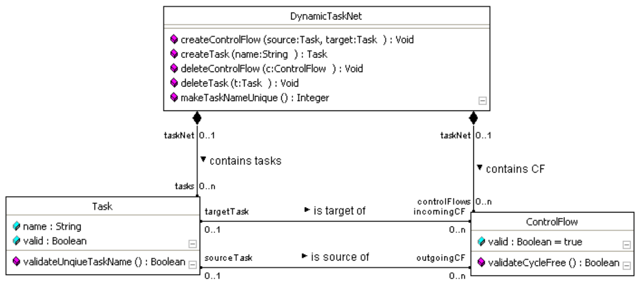

DynamicTaskNet
createControlFlow (source:Task, target:Task) : Void
createTask (name:String) : Task
deleteControlFlow (c:ControlFlow) : Void
deleteTask (t:Task) : Void
makeTaskNameUnique () : Integer
taskNet 0..1
contains tasks
tasks 0..n
taskNet 0..1
contains CF
controlFlows 0..n
incomingCF
Task
name : String
valid : Boolean
validateUnqiueTaskName () : Boolean
targetTask
is target of
0..1
0..n
sourceTask
is source of
outgoingCF
0..1
0..n
ControlFlow
valid : Boolean = true
validateCycleFree () : Boolean

Figure 2: Class diagram of the example model

2. `Task`, which instances represents specific tasks within the net

3. `ControlFlow`, which represents specific relations between tasks, e.g. sooner-later-dependencies

The code generation maps the structural part of the model onto the Ecore metamodel by generating an ecore file. Figure 3 shows a section of the ecore file for our example model: the description of class `Task`. An *eClassifier* with name “Task” is defined and within it several *eStructuralFeatures* (attributes) and *eOperations* (methods). Note the two different types of *eStructuralFeatures*: “EAttribute” and “EReference”. The first is used for primitive data types, the second for associations including cardinalities and containment relationship.

The behavioral part is mapped directly onto executable java code. For each class in Fujaba with class name *name* a corresponding class *name*Impl is generated which contains all code generated from story diagrams. In case of the sample class `Task` the generated *TaskImpl* contains code for two methods: *validateUniqueTaskName()* and *removeYou()*. When the EMF code generation is applied onto the ecore file it merges the existing Impl-files with the generated ones. *TaskImpl* contains then additional 17 methods for accessing and changing its attributes and links.

5. CONCLUSION

We have shown that the Ecore model is a subset of the Fujaba language and thus it is possible to generate EMF compatible code from Fujaba models. We have developed such a code generator as a plugin for Fujaba. That code generation not only translates the static part of the model to Ecore but is also able to generate code for story diagrams. Thus the developer can now specify behavior for EMF models with Fujaba. Here the full expressiveness of story diagrams can be used. We have evaluated our tool in a first example case study at the University of Bayreuth and a student project at the University of Kassel. In those projects we were able to specify complex models including behavior with Fujaba and generate EMF code from those models. We used the GMF framework [6] to build a graphical user interface on top of this code. Using these techniques the students in Kassel built a petri net editor and the team in Bayreuth a dynamic task net editor [2]. Note, that thanks to the model based specifications in Fujaba and in GMF, those two projects could be realized with very few lines of hand written code.

```
<eClassifiers xsi:type="ecore:EClass" name="Task" abstract="false" interface="false"
    eSuperTypes="">
   <eStructuralFeatures xsi:type="ecore:EAttribute" name="name"
       eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EString" transient="false">
   </eStructuralFeatures>
   <eStructuralFeatures xsi:type="ecore:EAttribute" name="valid"
       eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EBoolean" transient="false">
   </eStructuralFeatures>
   <eOperations name="validateUnqiueTaskName"
       eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EBoolean" eExceptions="">
   </eOperations>

   <eStructuralFeatures xsi:type="ecore:EReference" name="taskNet" ordered="false" lowerBound="0"
       upperBound="1"eType="#//DynamicTaskNet" transient="false" containment="false"
       eOpposite="#//DynamicTaskNet/tasks">
   </eStructuralFeatures>
   <eStructuralFeatures xsi:type="ecore:EReference" name="outgoingCF" ordered="false" lowerBound="0"
       upperBound="-1"eType="#//ControlFlow" transient="false" containment="false"
       eOpposite="#//ControlFlow/sourceTask">
   </eStructuralFeatures>
   <eStructuralFeatures xsi:type="ecore:EReference" name="incomingCF" ordered="false" lowerBound="0"
       upperBound="-1" eType="#//ControlFlow" transient="false" containment="false"
       eOpposite="#//ControlFlow/targetTask">
   </eStructuralFeatures>
   <eOperations name="removeYou"/>
</eClassifiers>
```

Figure 3: Ecore representation of the class *Task*

6. RELATED WORK

The *TIGER* project [7] is based on AGG, an approach to graph transformations based on category theory. It is dedicated to the automatic generation of GEF editors using graph transformations, in contrast to our work, which uses GMF for editor generation. Therefore we may benefit from further developments for GMF, i.e. design tools for graphical elements and sophisticated wizards for the generation process. Furthermore our editors are extensible like any other GMF-editor. The Tiger EMF Transformation Project [1] adds support for graph transformations on EMF models to TIGER. Here actions on the EMF model can be modeled using graph transformations. But one actions always is one graph transformation rule. More elaborate features as control flow are not yet supported.

The DiaMeta tool [10] is an freehand editor generator based on graph grammars. The grammars are specified against MOF 2.0 meta models specified in MOFLON [3]. Since MOFLON is the MOF 2.0 plugin for the Fujaba Tool Suite, the normal Fujaba graph transformations can also be specified for MOFLON meta models. But the code generated by MOFLON is not compatible to EMF, so the generated applications do not easily integrate into eclipse and can not benefit from all the tools and libraries already available for EMF / eclipse.

7. FUTURE WORK

Following issues will be addressed in the future to improve the Fujaba code generation for EMF: First the mapping of classes with *reference* stereotype onto ecore by using EMFs *EDataType*. Second we are looking for a way to map constants onto ecore. And third we still have to implement support for the new qualified associations of EMF 3.3.

8. REFERENCES

[1] E. Biermann, K. Ehrig, C. Köhler, G. Kuhns, G. Taentzer, and E. Weiss. Graphical Definition of In-Place Transformations in the Eclipse Modeling Framework. In *MoDELS'06*, 2006.

[2] T. Buchmann, A. Dotor, and B. Westfechtel. Model driven development of graphical tools: Fujaba meets gmf. In *Proceedings of the 2nd International Conference on Software and Data Technologies (ICSOFT 2007)*, pages 425–430. INSTICC, jul 2007.

[3] T. R. A. S. C. Amelunxen, A. Königs. MOFLON: A Standard-Compliant Metamodeling Framework with Graph Transformations. In *in: A. Rensink, J. Warmer (eds.), Model Driven Architecture - Foundations and Applications: Second European Conference, Heidelberg: Springer Verlag, 2006; Lecture Notes in Computer Science (LNCS), Vol. 4066, Springer Verlag, 361–375*, 2006.

[4] Eclipse Foundation. *The Eclipse Modeling Framework (EMF) Overview*, 2005. `http://dev.eclipse.org/-viewcvs/indextools.cgi/org.eclipse.emf/doc/-org.eclipse.emf.doc/references/overview/-EMF.html` – last visited: 03/08/2007.

[5] Eclipse Foundation. *The Eclipse Modeling Framework (EMF) Overview*, 2005. `http://dev.eclipse.org/-viewcvs/indextools.cgi/org.eclipse.emf/doc/-org.eclipse.emf.doc/references/overview/-EMF.html` – last visited: 03/08/2007.

[6] Eclipse Foundation. *Graphical Modeling Framework*, 2007. `http://www.eclipse.org/gmf/` – last visited: 30/08/2007.

[7] K. Ehrig, C. Ermel, S. Hänsgen, and G. Taentzer. Generation of visual editors as eclipse plug-ins. In *ASE '05: Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pages 134–143, New York, NY, USA, 2005. ACM Press.

[8] L. Geiger, C. Schneider, and C. Record. Template- and modelbased code generation for MDA-tools. In *3rd International Fujaba Days*, Paderborn, Germany, 2005.

[9] P. Heimann, C.-A. Krapp, and B. Westfechtel. Graph-based software process management. *International Journal of Software Engineering and Knowledge Management*, 7(4):431–455, 1997.

[10] M. Minas. Generating meta-model-based freehand editors. In *Electronic Communications of the EASST, Proc. of 3rd International Workshop on Graph Based Tools (GraBaTs'06), Natal (Brazil), September 21-22, 2006, Satellite event of the 3rd International Conference on Graph Transformation*, 2006.

[11] Object Management Group. *Meta Object Facility (MOF) Core Specification 2.0*, 2006. `http://www.omg.org/`.

[12] C. Schneider. CASE Tool Unterstützung für die Delta-basierte Replikation und Versionierung komplexer Objektstrukturen (Diploma Thesis, german), 2003.

[13] C. Schneider, A. Zündorf, and J. Niere. CoObRA - a small step for development tools to collaborative environments. In *Workshop on Directions in Software Engineering Environments; 26th international conference on software engineering.* ICSE 2004, Scotland, 2004.

[14] M. Tichy, M. Meyer, and H. Giese. On Semantic Issues in Story Diagrams. In *Fujaba Days 2006*, 2006.

[15] A. Zündorf. Rigorous Object Oriented Software Development, 2001.

WhiteSocks - A simple GUI Framework for Fujaba

Ira Diethelm[1], Ruben Jubeh[1], Andreas Koch[1], Albert Zündorf[1]

[1]Universität Kassel, Wilhelmshöher Allee 73, 34121 Kassel
whitesocks@andreaskoch.net | (ira.diethelm | ruben.jubeh | albert.zuendorf)@uni-kassel.de
http://www.se.eecs.uni-kassel.de/se/

ABSTRACT

While Fujaba is appropriate for modeling the application logic, building a GUI has not yet been specifically supported by the Fujaba tool set. This paper presents the WhiteSocks project library for Fujaba. WhiteSocks is a Fujaba project that provides a small framework and a set of standard means for the construction of simple graphical user interfaces. A user project may import and exploit the WhiteSocks project library in order to come up with a graphical user interface e.g. for a simple board game. This paper presents some core features of the WhiteSocks project library.

1. INTRODUCTION

Teaching object oriented modelling with Fujaba has always the problem that, once the application logic has been build by a student, she or he would like to "see" it working, i.e. they want to have some user interface. However, programming user interfaces with frameworks like Swing or Gef is too challenging for freshmen. Thus, we wanted to have a simple GUI framework (for simple applications) integrated into the Fujaba tool suite.

Luckily, Michael Kölling presented the Greenfoot environment within a talk at the Humboldt University on 20.10.2006. The general idea of the Greenfoot environment is to have a singleton root object *world* that owns a number of so-called *actors*. The world opens a window and creates a representation of the actors within this window. Most easily, this representation is some icon at some position, cf. Figure 2. In order to make the actors active, the world iterates through its actors continuously and invokes act() and display() methods on each of them. By overriding the act() method, an actor may perform specific activities and change their state. Then, the display() method may update the representation of the actor within the world window. For example, the icon may be exchanged, moved or rotated. In addition, an actor may react on mouse clicks or on key strokes in order to allow user interaction.

With these basic mechanisms, it is easy to create simple user interfaces offering the possibilities of sprite graphics as they have been used in many early computer games. Such simple user interfaces enable programming freshmen to build their first visual applications. This is very motivating for students and improves their interest in learning programming and modelling. The Greenfoot environment has already been used with great success for teaching freshmen in Java programming.

Thus, we had the idea to come up with a similar environment, the Whitesocks framework. Whitesocks takes over most of the concepts of the Greenfoot environment and adapts them to work with the Fujaba environment.

2. ARCHITECTURE

Similar to Greenfoot, WhiteSocks provides a singleton class $WSWorld$ that owns a number of WSActor objects, cf. Figure 1. WSWorld opens a window/JFrame and then it creates a representation of each of its WSActors within this window. Basically, each actor is represented by an icon subscribed with a textual label. The default icon for an actor is a white sock. However, if the icon attribute of the WSActor object points to an icon file, this icon is used instead. The x and y positions of the WSActor are provided by the corresponding attributes of class WSPersistent. Similarly, the label attribute of class WSActor provides a string to be shown below the icon.

Each WSActor object provides an act() method. Within a nonterminating loop, WSWorld iterates through all its WSActor objects and calls method act() on each of them. Then the representation of the objects is updated. After a short pause (in range of 1...100ms) this is reiterated. By overwriting method act(), subclass objects may perform continuous activities and react on changes within their environment. For example, an actor could change its position a few pixel towards a certain direction on each call to its act method thus creating a smooth movement. In addition, method mouseReleased() may be overridden, in order to react on mouse clicks on a WSActor subclass object.

3. MULTI-USER-SUPPORT

Multi-user-support is realized by utilizing the Coobra2 library. The Coobra2 library provides a mechanism to replicate an object structure on multiple (Java virtual) machines and to keep these copies consistent by forwarding changes between these copies. On each copy, a local Coobra2 component subscribes itself as listener to all objects / actors. If a change occurs, this change is forwarded to a Coobra2 server that in turn forwards the change to all other object structure copies. This mechanism allows to run a Whitesocks applica-

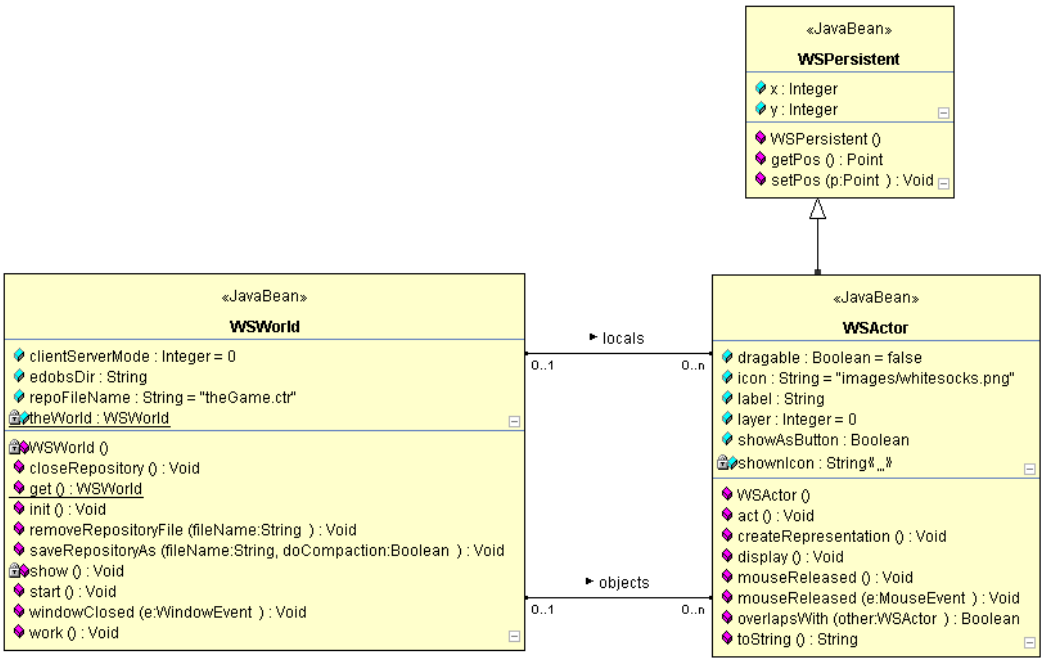


Figure 1: Basic WhiteSocks Architecture

tion on multiple machines while the actors on the different machines are synchronized by the Coobra2 system.

4. EXAMPLES

4.1 Ludo

The simple means described above already suffice to build simple graphical user interfaces of e.g. many popular (board) games. Figure 2 shows a screen dump of a ludo game created with WhiteSocks.

Basically, the ludo game consists of Field objects rendered with a blue ball icon, home fields rendered as houses, goal fields rendered as rocks, players rendered as persons, and stones rendered as man icons. The lower part of the window shows some special command objects. For the Ludo game, all objects have been created within EDobs, cf. [2], and then stored using Coobra2, cf. [1]. All fields, houses and players have been positioned manually, using EDobs, [2]. (By default, WSActor objects take over the pos information stored by EDobs.) The position of stones is computed either by the act method of class Stone (if it is on some field) or by the player (if stones line up at home.)

Class Die overwrites the mouseReleased method of WSActor to call method roll on itself. Thus, the user may click on the dice to roll it. The act method will update the icon, accordingly. Stones overide mousRelesed to invoke their move method. Thus, the user may click on one of his stones to move it. Then the move method will also hand over the die to the next player.

Using the (standard WhiteSocks) commands at the bottom of the figure, the user may start or connect to a server for multi player support over the network. This is again based on Coobra2 features.

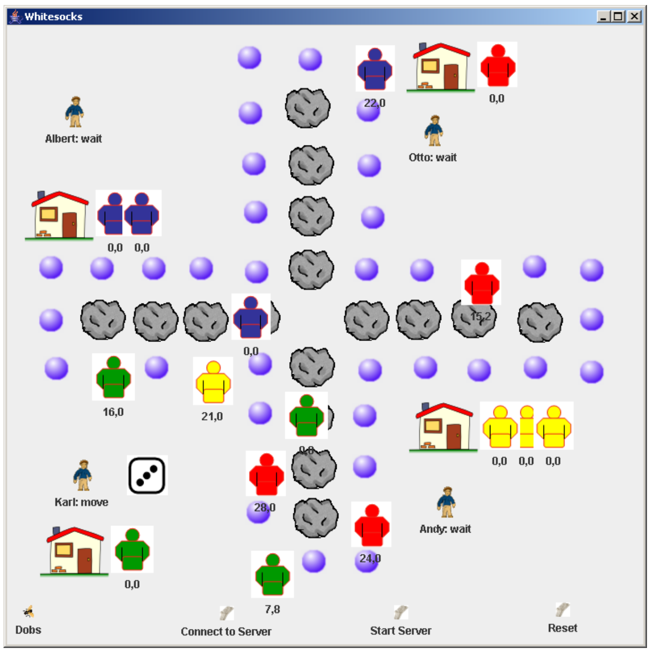


Figure 2: A Ludo game build with Fujaba and WhiteSocks

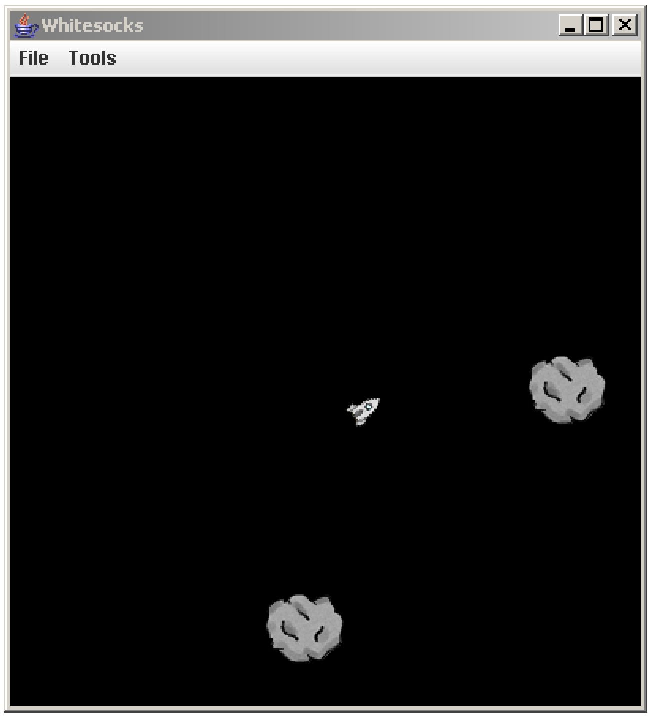


Figure 3: Asteroids game build with Fujaba and WhiteSocks

4.2 Asteroids

Asteroids is a simplified version of the well known Arcade Game *Asteroids*, originally a Atari game. Asteroids is implemented in Greenfoot as well, and has been choosen as Example to compare the features and usability of both environments.

The user can control the Rocket with keyboard commands, e.g. accelerate, decelerate, change direction and fire. The asteriods are moving randomly though the space. When a asteroid is hit by the rocket's fire a few times, it explodes and splits into two smaller asteroids.

Asteroids is realized by deriving the game's objects from the $WSActor$ class. Figure 4 shows the Design of the Asteroids game: each object type in space is modeled as subclass of $WSActor$. The class Rocket is derived from $WSMovingActor$, so mouse and keyboard events are delivered without any delay.

5. LIMITATIONS

Currently, there is very limited support for standard GUI widgets like Buttons, Textfields, Textarea etc. At the current state, WhiteSocks concentrates on easy and intuitive support for building board-like games. Building other kinds of graphical applications is supported, but requires knowledge and manual integration of the GUI widgets (Swing) into the WhiteSocks application.

Furthermore, in our examples we do not properly seperate model, view, and controller. Frequently, logic classes just inherit from WSActor and overwrite some methods for GUI adaption. For example, the application model classes

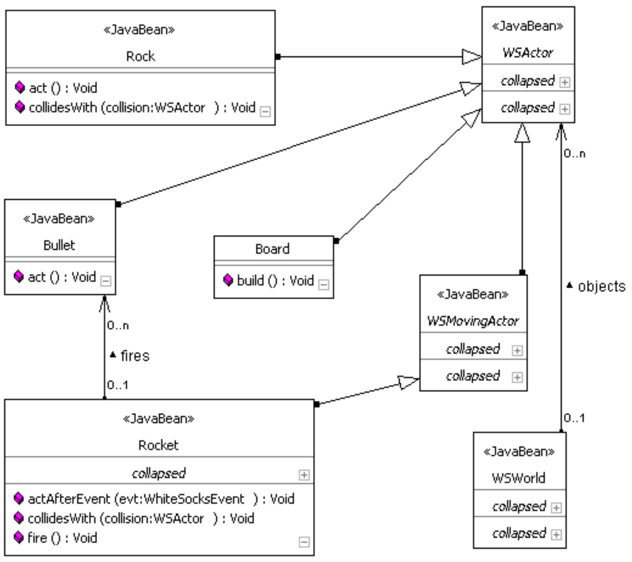


Figure 4: Asteroids Class Diagram

directly provide their icon and position for the $display()$ method. This becomes a problem, is one object shall be shown in different views. For example, we might want to display a ranking of players by showing the player actors at different hights at some ladder. In addition, a player icon shall be shown at near to the home field of that player. This is not supported by the current display mechanism, since each actor has only one icon attribute and only one set of x an y position attributes. To avoid this problem, a logic object should not inherit from WSActor but have one or more specific actor object for its representation in different views/positions. While the framework allows this, our examples give wrong hints to our users. On the other hand, merging logic and representation facititates things for beginners.

6. CURRENT AND FUTURE WORK

As stated, WhiteSocks is based on the ideas of the Greenfoot environment. Actually, we would have loved to use Greenfoot as a GUI library for models build with Fujaba. In order to be able to exploit all the existing Greenfoot applications and in order to avoid reinventing the wheel, we have tried this but ran into multiple problems caused by incompatibilities of code styles of Fujaba and Greenfoot. In addition, the BlueJ compiler behind Greenfoot and the build mechanism of Fujaba tent to embrace each other until they end in a deadlock.

Thus, we finally decided to come up with our own clone. Using WhiteSocks for building simple user interfaces for applications build with Fujaba works quite nicely, now. One basically has to turn certain classes of his application into subclasses of WSActor in order to add the corresponding objects to the screen. Then, one overrides the display method in order to adapt the presentation of each object and the act method in order to achieve reactive behavior.

Current work tries to add more GUI widgets to WhiteSocks.

Thus, we would like to provide multi line text fields for longer texts and editable text fields for user input (e.g. for a chat box). And input fields for numbers. And Drag&Drop support. And various kinds of lists and tables to structure certain information. For educational purposes, we also try to add some turtle graphics features (as provided by Greenfoot). This might also be useful for charts, e.g. for plotting functions.

In addition, we have to improve the integration of WhiteSocks into the development process supported by the Fujaba Environment. Currently, one adds the WhiteSocksLib project to his Fujaba environment and then his application project may import the WhiteSocks project. Then, the user might inherit from WSActor to make his objects visible and she or he has to inherit from WSWorld to provide a standard main method. Finally, the user either has to code some initializer for the creation of an initial object structure or the user integrates EDobs into his application in order to be able to add objects and do debug his object structure. This needs to become easier to handle.

As soon as Whitesocks has become mature enough, we want to use it in programming and modelling courses at our University and at some high schools in order to increase the student's motiviation to stick with learning.

7. REFERENCES

[1] CoObRA 2.
http://www.se.eecs.uni-kassel.de/se/?coobra, 2006.

[2] The EDobs Dynamic Object Browser.
http://www.coobra.cs.uni-kassel.de/index.php?edobs, 2006.

[3] The Greenfoot Environment.
http://www.greenfoot.org/, 2006.