Fujaba Days 2011

Proceedings of
the 8th International Fujaba Days

University of Tartu, Estonia
May 11-13, 2011

Editor: Ulrich Norbisrath, Ruben Jubeh

UNIKASSEL
VERSITÄT

1 Introduction

Fujaba is an Open Source UML CASE tool project started at the software engineering group of Paderborn University in 1997. In 2002 Fujaba has been 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.

Multiple Application Domains 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 six rather independent tool versions are under development in Paderborn, Kassel, and Darmstadt for supporting (1) reengineering, (2) embedded real-time systems, (3) education, (4) specification of distributed control systems, (5) integration with the ECLIPSE platform, and (6) MOF-based integration of system (re-) engineering tools.

International Community According to our knowledge, quite a number of 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 requests for more functionality and extensions.

Therefore, the 8th International Fujaba Days aimed 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.

Organizing Committee

Ulrich Norbisrath, University of Tartu, Estonia

Ruben Jubeh, University of Kassel, Germany

Program Committee

Prof. Marlon Dumas (University of Tartu, Estonia)

Prof. Holger Giese (Hasso-Plattner-Institut Potsdam, Germany)

Prof. Jens Weber (University of Victoria, Canada)

Jendrik Johannes (Technische Universität Dresden, Germany)

Ruben Jubeh (University of Kassel, Germany)

Prof. Mark Minas (University of the Federal Armed Forces, Germany)

Ulrich Norbisrath (University of Tartu, Estonia)

Prof. Arend Rensink (University of Twente, Netherlands)

Christian Schneider (Yatta Solutions, Germany)

Prof. Andy Schürr (TU Darmstadt, Germany)

Prof. Wilhelm Schäfer (University of Paderborn, Germany)

Prof. Dr. Matthias Tichy (Universität of Augsburg, Germany)

Prof. Bernhard Westfechtel (University of Bayreuth, Germany)

Prof. Albert Zündorf (University of Kassel, Germany)

Content

A new Meta-Model for Story Diagrams

Christian Heinzemann*, Jan Rieke*,
Markus von Detten, Dietrich Travkin
Software Engineering Group,
Heinz Nixdorf Institute,
University of Paderborn, Germany
[c.heinzemann|jrieke|mvdetten|travkin]
@uni-paderborn.de

Marius Lauder†
Real-Time Systems Lab,
Technische Universität Darmstadt, Germany
marius.lauder@es.tu-darmstadt.de

ABSTRACT

Story-driven modeling (SDM) is a model-based specification approach combining UML activity diagrams and graph transformations. In recent years, the development in the SDM community led to many incompatible meta-models for story diagrams based on the same common concepts. The diversity of meta-models hindered the reuse of tools and limited synergy effects. In this paper, we introduce the new meta-model for story diagrams which was created in a joint effort of the SDM community. The new EMF-based model integrates the recent developments and paves the way for the interoperation of SDM tools with each other and with EMF-based tools.

1. INTRODUCTION

Story-driven modeling is a model-based specification approach which combines aspects from UML activity diagrams and graph transformations into an expressive and intuitive graph rewriting language, so-called *story diagrams* [3]. In the past years, story diagrams have received significant attention and have become the foundation of many different software engineering techniques and tools.

Ever since their inception, story diagrams have been used in widely different domains and for such different purposes as meta-model integration [1] or the specification of real-time systems [6]. Story diagrams can either be executed by generating appropriate code (e.g., [4]) or by interpretation [5]. These different approaches have led to a variety of extensions and specialized dialects of the original story diagram concept and were accompanied by a number of different tools for the specification, application and analysis of story diagrams. Unfortunately, due to this development, a number of different, incompatible meta-models for story diagrams have emerged which are all based on the same common concepts. Hence, reuse and the composition of tool chains is severely limited by these technical differences.

To cope with these problems, a new meta-model for story diagrams has been developed in a joint effort of the SDM community. The new meta-model integrates a number of useful concepts from the different dialects and provides an extensible framework for future developments. It is based on the Eclipse Modeling Framework (EMF) and thereby paves the way for the interoperation of SDM tools with EMF-based tools. In this paper, we present a slightly simplified version of the actual meta-model to allow for more concise explanations and the omission of technical details.

Before going into the details of the proposed meta-model, we briefly recall the concepts of story diagrams in Section 2. After introducing the meta-model in Section 3, we draw conclusions and sketch future work in Section 4.

2. STORY DIAGRAMS

Story diagrams allow to combine control flow with non-deterministic graph transformation rules. By means of graph grammars, they add a formal foundation to UML activity diagrams for the specification of behavior and, thus, enable their execution and analysis. A story diagram is a special activity diagram that specifies control flow by activity nodes and transitions (activity edges). In contrast to UML activity diagrams, activity nodes in story diagrams contain so-called *story patterns*.

A story pattern is a formal specification of a graph transformation and specifies an object structure (subgraph) that has to be matched in a model (host graph) as well as corresponding modifications of this structure. The structure is specified by special object diagrams in which the modifications, i.e., creation and deletion of elements as well as attribute value assignments, are designated accordingly. The object diagrams are typed over a set of classes.

3. THE NEW META-MODEL

In this section, we introduce the new meta-model package by package. Since the story patterns used for the specification of story diagrams are typed over a set of classes, a class model is required to specify story patterns. We model these classes by means of an Ecore model, thereby avoiding to define yet another meta-model for classes.

In Section 3.1, we give a short tour of the core elements. Then, we introduce the packages for story patterns and activities in Sections 3.2 and 3.3. Next, we discuss a simple example in Section 3.4. Finally, the new expressions and calls packages are presented in Sections 3.5 and 3.6, respectively.

*supported by the International Graduate School "Dynamic Intelligent Systems"

†supported by the 'Excellence Initiative' of the German Federal and State Governments and the Graduate School of Computational Engineering at TU Darmstadt

3.1 Packages and Core Elements

The package structure of the new meta-model is outlined in Figure 1. The modeling package contains the base classes and an annotation mechanism. It also includes subpackages for patterns, activities, expressions, and calls. The package patterns contains the meta-model classes for specifying story patterns. These classes have been separated from the package activities to enable the reuse of story patterns in other pattern languages, e.g., TGGs [8]. The package expressions contains a set of basic expressions while the package calls comprises the classes for modeling invocations of other story diagrams or operations. A detailed introduction to the packages is given in the subsequent sections.

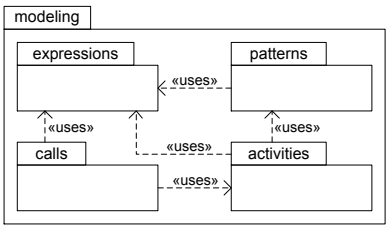

Figure 1: Package Structure

Figure 2 illustrates the core classes of our meta-model. The class ExtendableElement, being the super class of all meta-model classes, implements the annotation mechanism. Each element can be extended by subclasses of Extension. Additionally, we support to annotate our model elements (EModelElements) using EAnnotations.

The classes TypedElement, NamedElement, and CommentableElement are super classes of meta-model classes having a type, a name, or the ability to carry a comment, respectively. They are intended to be subclassed using multiple inheritance, if necessary.

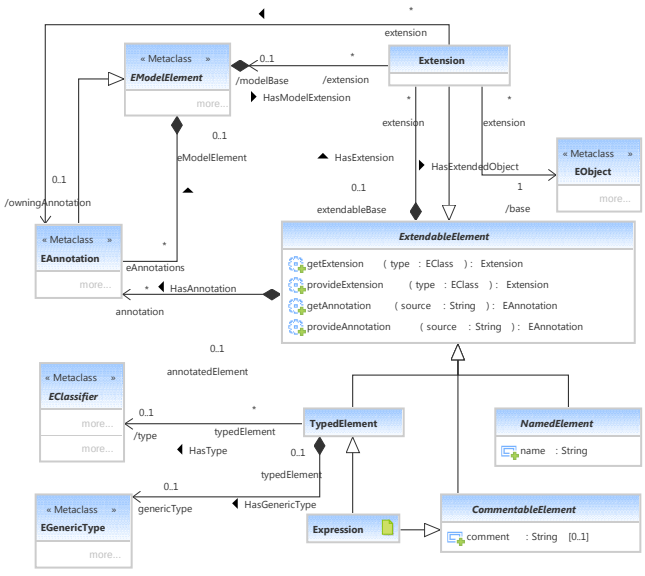

Figure 2: Core of the new Meta-Model

3.2 Story Patterns

The central role in story diagrams is played by *story patterns*, which are essentially in-place model transformation rules. The corresponding package patterns is depicted in Figure 3. Story patterns define object patterns and their modifications. Such structures are specified using ObjectVariables and LinkVariables. ObjectVariables are typed via the classifier attribute that points to an EClass of the underlying class model. LinkVariables are typed over the targetEnd attribute that points to an EReference. In case of bidirectional references, the derived attribute sourceEnd points to the according opposite reference.

Story patterns can be applied to models that contain objects and links which are instances of classes and references of the class model. First, the pattern is matched in the model. When there is a valid matching for the pattern (i.e., the matching is successful) modifications can be applied to the model. Otherwise, the matching fails and no modification is carried out.

The bindingOperator attribute of ObjectVariables and LinkVariables defines whether an element is to be created, deleted, or just matched. If the attribute is set to CHECK_ONLY or DESTROY, the according variables first have to be matched to objects and links in the model. As soon as all these variables have been matched, the model is modified: matched elements with the operator DESTROY are deleted and elements with the operator CREATE are produced. An AttributeAssignment alters an attribute value of an object represented by an ObjectVariable. This happens after the matching and the structural modification are completed.

In addition to the BindingOperator, variables have BindingSemantics. For a successful matching, MANDATORY variables have to exist in the model, while variables marked as NEGATIVE denote objects that must not exist. In contrast, OPTIONAL denotes objects that may exist. For example, a combination of OPTIONAL and CREATE is a compact way to express that an appropriate element will be created if it cannot be matched [7].

Since story diagrams consist of interconnected story patterns, they allow for the reuse of previously matched elements. An ObjectVariable is referenceable by its name. If the bindingState is UNBOUND, the pattern matching algorithm is forced to find a new object, even if the variable was already matched earlier in the story diagram. A BOUND ObjectVariable must have been matched previously. In case a BOUND ObjectVariable was not matched before, the story pattern execution is considered unsuccessful. A MAYBE_BOUND variable is a combination of both: if the variable has been bound before, it is reused; otherwise a new match will be determined.

A special case of an ObjectVariable is an ObjectSetVariable that matches an arbitrary number of objects of the same type. The number of matched objects can be restricted by ObjectSetSizeExpressions.

Path and ContainmentRelation are special link variables. The former is used to denote a connection via a sequence of links determined by a pathExpression. The latter denotes that an object is contained in a collection.

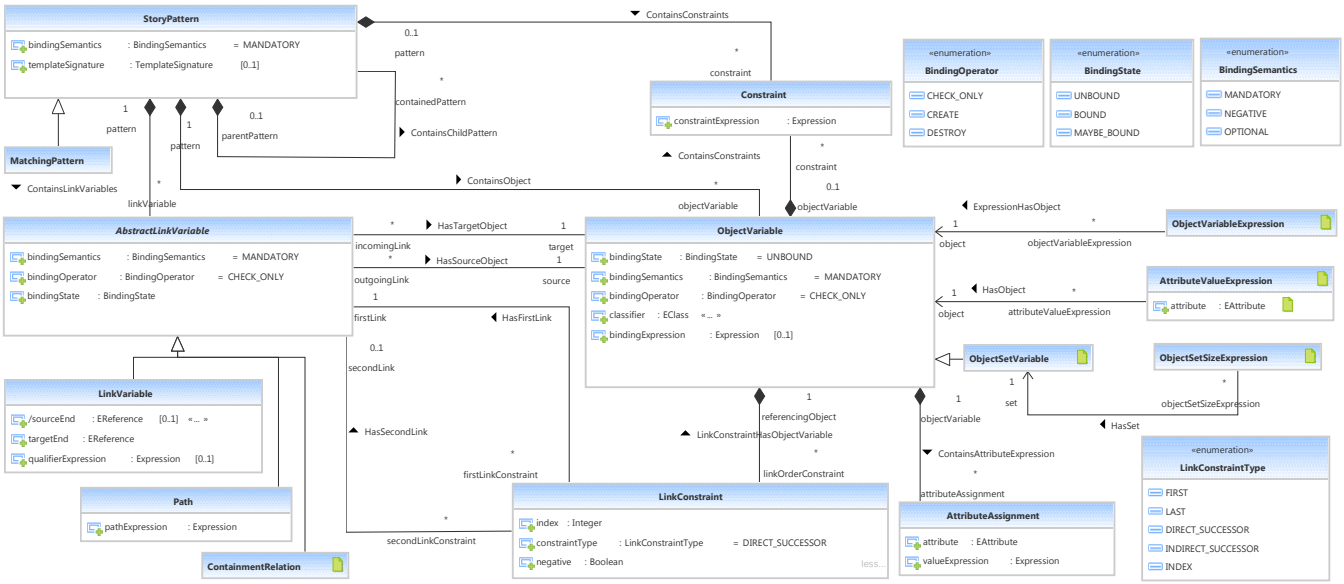

Figure 3: Story Patterns Meta-Model

The matching of a StoryPattern may be further refined using Constraints and LinkConstraints. A Constraint is a boolean expression that must evaluate to true for a matching to be successful. Its context is defined by its container, i.e., variable or pattern. For instance, within an ObjectVariable, you can directly access its attributes, while in a pattern, the ObjectVariable's name must be prefixed. The matching of a LinkVariable whose targetEnd is an ordered list can be constrained using a LinkConstraint. This way, it can be specified that the firstLink must either be the FIRST, LAST or a given INDEX in the list. Furthermore, given two links (firstLink and secondLink), the links' indices could be required to be DIRECT_SUCCESSORs or INDIRECT_SUCCESSORs in the list.

A MatchingPattern is a StoryPattern that is required to be non-modifying, i.e., it must only contain CHECK_ONLY variables and must not have AttributeAssignments. This allows creating side-effect-free story diagrams.

Finally, patterns are allowed to contain subpatterns. Whenever such a subpattern is found in a story pattern, it is matched as a whole. A subpattern may also be NEGATIVE or OPTIONAL. In the former case, the subpattern as a whole must not be found in the model, allowing more expressive negative application conditions. In the latter case, the subpattern is not required to be found. NEGATIVE subpatterns are matched before OPTIONAL subpatterns, but after matching the core (MANDATORY) pattern.

3.3 Activities

We developed a simplified meta-model for activity diagrams which is closely related to the corresponding UML specification. Our meta-model is depicted in Figure 4.

An activity diagram is represented by the Activity class. ActivityEdges connect ActivityNodes to specify the control flow. JunctionNodes are used to split and join the control flow. StructuredNodes are used to build a hierarchical activity diagram by embedding other activity nodes. StatementNodes offer the opportunity to textually specify algorithms with the help of expressions (see Section 3.5). Other activities can be called by means of ActivityCallNodes.

StoryNodes embed a story pattern using the storyPattern reference. To simplify analyses of graph transformations, we distinguish MatchingStoryNodes and ModifyingStoryNodes. While the former are only allowed to match a specified structure, the latter ones are also allowed to perform modifications. A MatchingStoryNode can for example be used to specify an Activity's precondition.

ActivityEdges can have guards, given by the guard attribute and the enumeration EdgeGuard. ActivityEdges with the guards SUCCESS and FAILURE distinguish the cases of (a) successfully executing the story pattern in the source activity node, i.e., completely match and modify the specified structure, and (b) missing to match the complete structure. NONE enforces to choose the ActivityEdge in either case.

Loops can be defined using an activity's forEach attribute. An ActivityEdge with an EACH_TIME guard is chosen for each match of the preceding forEach activity, while an END ActivityEdge is chosen if no such matching can be found anymore.

Boolean guard conditions (BOOL) are specified using the attribute guardExpression. ActivityEdges can also be chosen if an exception is thrown (EXCEPTION). In this case, the exception specified by the ExceptionVariable can be handled by following activity nodes. The activity node that is reached via the FINALLY edge is executed whether an exception is thrown or not.

A story diagram can be used to specify the behavior of an EOperation. We specify this with an OperationExtension which connects an EOperation to an Activity.

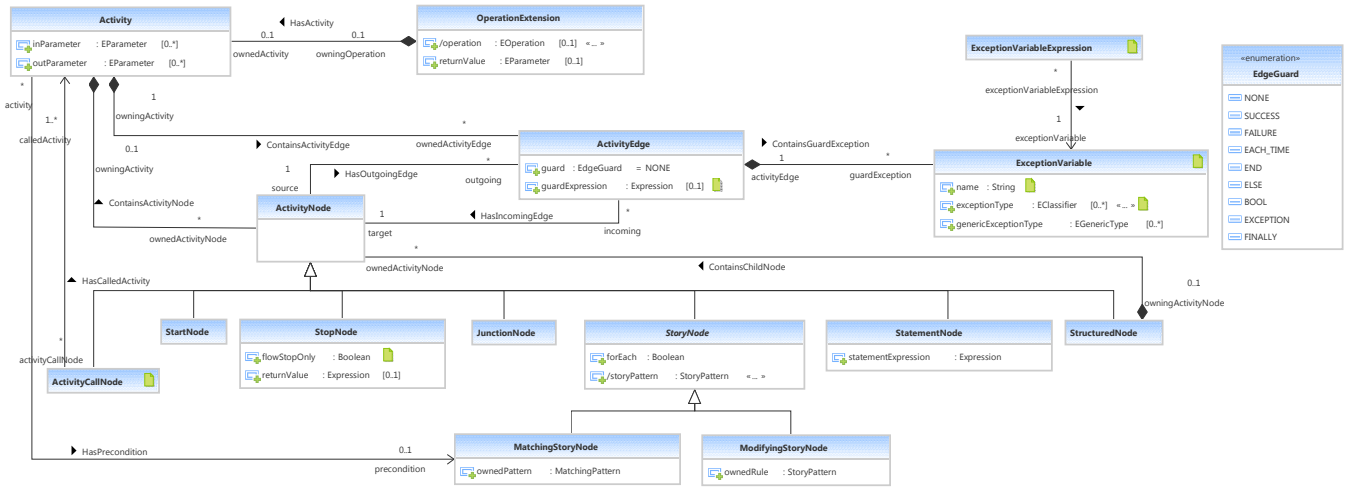

Figure 4: Activities Meta-Model

3.4 Example

Figure 5 shows the concrete syntax of an exemplary story diagram.

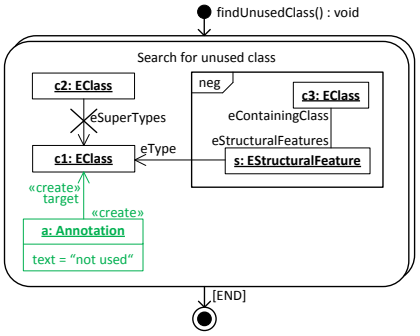

Figure 5: Usage of a Negative Subpattern

The "double" border of the StoryNode denotes a forEach node, i.e., it matches once for every possible matching in the model. The story pattern inside matches when there is a class `c1` which is not used by other classes.

The negative subpattern, denoted by the rectangle labeled with `neg`, is a negative application condition that has to be satisfied for a successful matching. In this case there must not be a structural feature of another class `c3` that references `c1`. Another constraint is given by the crossed-out link which specifies that `c1` must not be subclassed by any class `c2`. If a valid matching for this story pattern is found, the class `c1` is marked as "not used" by a newly created annotation `a`.

The story pattern is applied to each class that satisfies these constraints. Thus, after the execution of this story diagram, all classes that do not have another class using them are marked with a "not used" annotation.

3.5 Expressions

Although story diagrams are an expressive language, in some cases textual languages are better suited and more compact, e.g., for complex calculations or regular expressions.

Therefore, recent SDM tools allow to embed Java code in story diagrams. When generating code for a story diagram, the embedded Java code is included in the resulting code. As a consequence, the embedded code cannot be checked at modeling time (e.g., no type checking or model checking) and interpreting story diagrams that contain arbitrary Java code is hardly possible.

To improve this situation, the SDM community decided to explicitly model textual expressions in story diagrams. The expressions, on the one hand, still allow to embed textual languages like Java and OCL and, on the other hand, enable interpretation and type checking for most of them.

Our meta-model separates two cases: Either an arbitrary textual expression is represented as String in the class TextualExpression or the expression is modeled explicitly by building an abstract syntax model of the expression. In the former case, arbitrary code can be embedded for code generation, but comes with the cost of missing opportunity to analyze the expression. In the latter case, the expression model is more complex, but can be type-checked.

With our meta-model, we try to cover most common expressions in story diagrams and propose to explicitly model these to enable type checking at least for these cases. Examples for such expressions are matching constraints in story patterns or assignments of a certain value to an object's attribute.

Our story diagrams meta-model supports literals like `7`, `3.1`, `true`, or `"xy"` whose type is explicitly given (EDataType). Furthermore, we support logical expressions, arithmetic expressions, and comparing expressions. The expressions with an operator combine other expressions to build more complex expressions.

In addition, we allow for building expressions that represent an object variable in a story pattern, the value of one of its attributes, or the number of objects matched to an object set variable. Furthermore, method calls (MethodCallExpression, Figure 6), which are explained in the next section, can be modeled, too.

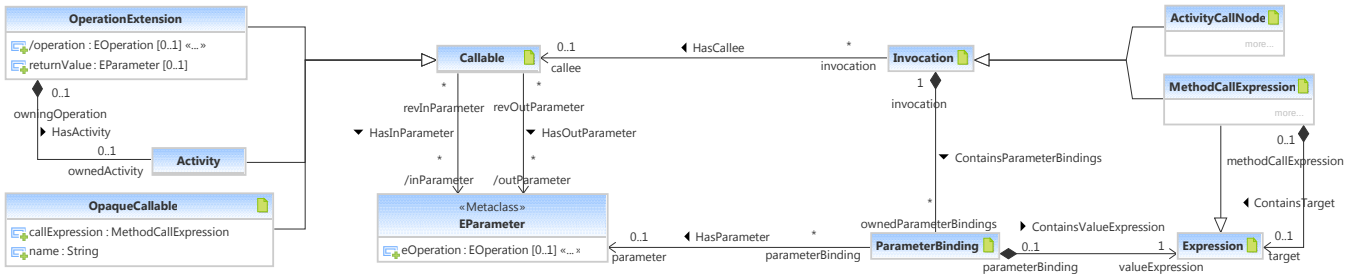

Figure 6: Calls package of the Meta-Model

3.6 Activity and Method Calls

The calls package of the new meta-model supports the invocation of so-called Callables directly from story diagrams (cf. Figure 6). Callables are Activities (i.e., other story diagrams), operations (represented by the wrapper class OperationExtension, which references an EOperation), and OpaqueCallables. EOperations are part of the model while OpaqueCallables are not represented in the model, but may, for example, be part of a library. A Callable can have in- and out-parameters as indicated by the two references from Callable to EParameter. While the number of parameters is unbounded in general, OperationExtensions and OpaqueCallables may only have one out-parameter. In contrast, Activities can have arbitrarily many out-parameters. The same object may be used as in-parameter and out-parameter, thereby emulating the in-out-parameters from other transformation languages like QVT.

Callables can be invoked by Invocations which can either be ActivityCallNodes or MethodCallExpressions. ActivityCallNodes are special ActivityNodes which can be used in story diagrams to represent the call of another story diagram. MethodCallExpressions represent the invocation of a method, i.e., either an EOperation or an OpaqueCallable. The target of a MethodCallExpression can be determined by an Expression which can, for example, be a variable or the result of another method invocation. Every Invocation must have a number of ParameterBindings that assign concrete arguments to the callee's parameters.

The calls package also provides a concept for the polymorphic dispatching of calls which is omitted here due to space restrictions. Details can be found in [2].

4. CONCLUSIONS AND FUTURE WORK

We presented the new common meta-model for story diagrams which was developed in a joint effort of the SDM community. It is the foundation for future projects as it provides a common basis for developments and facilitates the interoperation of SDM tools. In comparison to the different previous models, it especially simplifies static type checking due to the explicit modeling capability for expressions.

To facilitate the execution of story diagrams, it is necessary that the existing code generation and interpretation approaches are adapted to the new meta-model. This will be imperative for the development of SDM tools. In addition, all existing editors and tools will have to be adapted accordingly.

In this paper, we focused mostly on the abstract syntax of the story diagram meta-model and the semantics of some of the newly integrated features. While the concrete syntax of ActivityCallNodes has been defined in [2], a concrete syntax for other new elements still has to be defined in future works.

Acknowledgments

We would like to thank all other participants of the SDM unification task force for their ideas and contributions to the new meta-model: Steffen Becker, Stephan Hildebrandt, Ruben Jubeh, Elodie Legros, Carsten Reckord, Andreas Scharf, Christian Schneider, Gergely Varró, and Albert Zündorf.

5. REFERENCES

[1] C. Amelunxen, F. Klar, A. Königs, T. Rötschke, and A. Schürr. Metamodel-based tool integration with MOFLON. In *ICSE '08 Proceedings*, pages 807–810. ACM, 2008.

[2] S. Becker, M. von Detten, C. Heinzemann, and J. Rieke. Structuring Complex Story Diagrams by Polymorphic Calls. Technical Report tr-ri-11-323, University of Paderborn, Mar. 2011.

[3] T. Fischer, J. Niere, L. Torunski, and A. Zündorf. Story Diagrams: A New Graph Rewrite Language Based on the Unified Modeling Language and Java. In *TAGT '98 Selected Papers*, volume 1764 of *LNCS*, pages 296–309. Springer, 2000.

[4] L. Geiger, T. Buchmann, and A. Dotor. EMF Code Generation with Fujaba. In *Fujaba Days '07 Proceedings*, 2007.

[5] H. Giese, S. Hildebrandt, and A. Seibel. Improved Flexibility and Scalability by Interpreting Story Diagrams. In *GT-VMT '09 Proceedings*, volume 18 of *Electronic Communications of the EASST*, 2009.

[6] C. Priesterjahn, M. Tichy, S. Henkler, M. Hirsch, and W. Schäfer. Fujaba4Eclipse Real-Time Tool Suite. In *MBEES '07 Revised Selected Papers*, volume 6100 of *LNCS*, chapter 12, pages 309–315. Springer, 2009.

[7] S. Rose, M. Lauder, M. Schlereth, and A. Schürr. A Multidimensional Approach for Concurrent Model Driven Automation Engineering. In *Model-Driven Domain Analysis and Software Development: Architectures and Functions*, pages 90–113. IGI Publishing, 2011.

[8] A. Schürr. Specification of Graph Translators with Triple Graph Grammars. In G. Tinhofer, editor, *WG '94 Proceedings*, volume 903 of *LNCS*. Springer, 1994.

Interpreting Story Diagrams for the Static Detection of Software Patterns

Markus Fockel, Dietrich Travkin, Markus von Detten
Software Engineering Group, Heinz Nixdorf Institute,
University of Paderborn, Paderborn, Germany
[mfockel|travkin|mvdetten]@mail.uni-paderborn.de

ABSTRACT

Software maintenance tasks require knowledge about the software's design. Several tools help to identify implementations of software patterns, e.g. Design Patterns, in source code and thus help to reveal the underlying design. In case of the reverse engineering tool suite Reclipse [15], detection algorithms are generated from manually created, formal pattern specifications. Due to numerous variants that have to be considered, the pattern specification is error-prone. Because of this, the complex, step-wise generation process has to be traceable backwards to identify specification mistakes. To increase the traceability, we directly interpret the detection algorithm models (story diagrams) instead of executing code generated from these models. This way, a reverse engineer no longer has to relate generated code to the story diagrams to find mistakes in pattern specifications.

1. INTRODUCTION

Due to requests for new features and the discovery of defects, software has to be continuously adapted and maintained. For this purpose, developers have to understand the design of a given software. Software design patterns [5] are approved, widely used solutions for design problems. Knowledge about their usage in the software helps to understand how the original developers intended the software to be extended or adapted and, thus, helps to avoid design deviations or errors.

Incomplete documentation often complicates the task of identifying pattern implementations in source code. Several tools have been developed to automate this tedious task (Dong et al. give an overview [2]). Based on a formal specification of a pattern, usually represented by a set of conditions, these tools automatically detect pattern implementations in source code.

Nevertheless, due to numerous implementation variants[1] to be considered during pattern specification, the task of specifying a pattern is error-prone which sometimes results in missing pattern implementations (false negatives) or finding more than are actually present (false positives). To correct a specification, a reverse engineer has to identify the erroneous or missing conditions in the specification that lead to the unexpected detection results which, in turn, requires traceability of the detection process.

[1]For example, different loop implementations or the distinction between interfaces and classes in Java.

In case of the reverse engineering tool suite Reclipse[2] [15], the detection process is quite complex. Reclipse automatically derives detection algorithms from pattern specifications, creates models of these algorithms in form of class and story diagrams [3], generates code out of these models, and executes this code to detect pattern implementations in given source code [10]. To trace the detection process, a reverse engineer has to observe the generated detection code's behavior, deduce the elements which are representing this behavior in the generated story diagrams, and identify the corresponding conditions in the pattern specifications. Hence, the reverse engineer has to bridge two semantic gaps: the one between code and story diagrams and the one between story diagrams and pattern specifications.

A pattern specification only describing a class declaration and a contained method declaration already results in about 1000 lines of generated code. As example take the following excerpt of code that would be generated based on such a pattern specification. The lines 1 to 3 contain declarations of two variables `clazz` and `method` to represent the declarations and an auxiliary variable for the iteration through all elements contained in a class. In lines 4 and 5 a part of the code to be analyzed is assumed to be the specified class declaration. The remaining lines describe the search for a method declaration contained in the class represented by the previously found class declaration. As a class can contain several method declarations, this is done in a loop.

```
  ...
1 ATypeDeclaration clazz = null;
2 Iterator fujaba__IterClazzToMethod = null;
3 AMethodDeclaration method = null;
  ...
4 JavaSDM.ensure(_TmpObject instanceof
                              ATypeDeclaration);
5 clazz = (ATypeDeclaration) _TmpObject;
  ...
6 fujaba__IterClazzToMethod = clazz
                    .iteratorOfBodyDeclarations();
7 while (fujaba__IterClazzToMethod.hasNext()) {
8   _TmpObject = fujaba__IterClazzToMethod.next();
9   JavaSDM.ensure( _TmpObject instanceof
                             AMethodDeclaration);
10  method = (AMethodDeclaration) _TmpObject;
    ...
  }
  ...
```

[2]http://www.fujaba.de/reclipse

The reverse engineer has to mentally bridge the semantic gap between this code and the corresponding story diagram and eventually the pattern specification. She has to find the conditions in the pattern specification that are represented by the variables in the generated code and then identify the error in the specification by debugging the code execution.

We're aiming to avoid the semantic gaps by directly interpreting the pattern specifications, thereby adding tracing functionality to Reclipse's pattern detection, similar to debuggers. As a first step, we remove the semantic gap between generated code and story diagrams by directly interpreting the story diagrams instead of generating code. As there is an existing interpreter for story diagrams [7] with a corresponding debugger being currently developed [8], this is a promising solution. Furthermore, by exploiting runtime information, interpreting story diagrams can be more efficient than executing code [7].

In this paper we present the actions we have taken to integrate the story diagram interpreter developed at the Hasso Plattner Institute in Potsdam [7] into Reclipse, the challenges we faced and an evaluation of the results.

2. THE PATTERN DETECTION PROCESS

Reclipse's current pattern detection process is depicted in Figure 1. First of all, the design patterns have to be defined manually as formal pattern specifications. Algorithms (in form of story diagrams) that describe the search for the specified patterns are automatically derived from the formal pattern specifications. These detection algorithm models are then used to generate code that is later called by the *inference algorithm*. The code in which to search for implementations of the specified patterns has to be transformed into an *abstract syntax graph* (ASG), which is done by Reclipse automatically. The inference algorithm receives the ASG and the generated detection algorithm code as input. It decides where in the ASG to search for a pattern and executes the respective detection algorithm code to do so. Finally, the output is an ASG in which the detected pattern implementations are marked by annotations. As Reclipse is based on the CASE tool Fujaba[3] [9], all models have been created or generated with that framework (signified in Figure 1 by the ellipses with the Fujaba inscription).

3. INTERPRETER INTEGRATION

In order to remove the semantic gap between story diagrams and generated detection code, we integrated the story diagram interpreter into the pattern detection process of Reclipse as illustrated in Figure 2. During that integration we faced several challenges.

First, the interpreter is based on a story diagram meta-model that is slightly different from the one used in Reclipse (provided by Fujaba). Hence, we had to translate the story diagrams from one dialect to another. Instead of generating Fujaba-conformant story diagram models from the pattern specifications, we now generate story diagram models that conform to the interpreter's story diagram meta-model (signified in Figure 2 by the different shape and number of elements in the detection algorithm models).

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

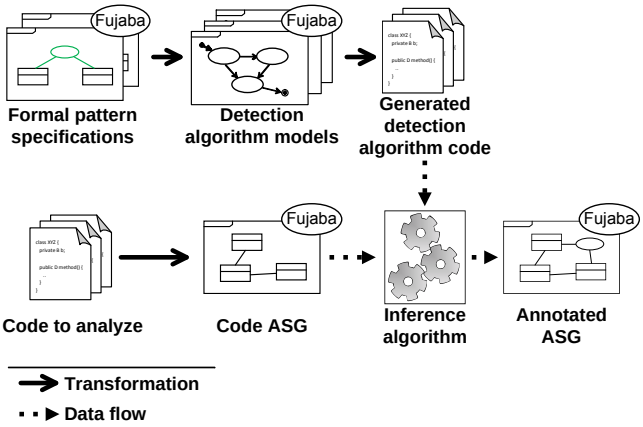

Figure 1: Original pattern detection process.

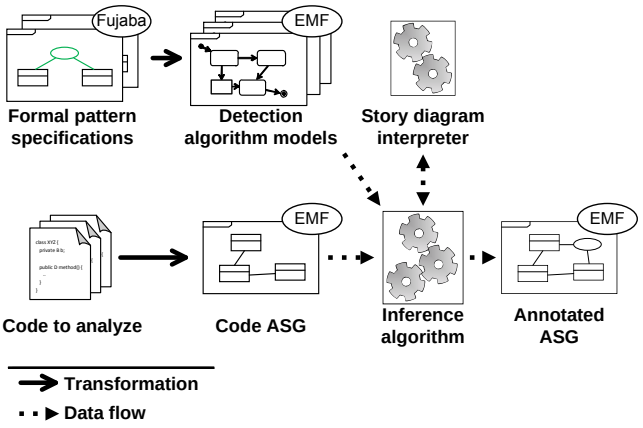

Figure 2: Adapted pattern detection process.

Second, Reclipse and the story diagram interpreter are implemented based on different frameworks. While Reclipse is based on the CASE tool Fujaba with its model format and API, the story diagram interpreter is based on the Eclipse Modeling Framework (EMF)[4] [14] and takes Ecore models as input. Thus, we also had to adapt or convert the class and story diagram models as well as the input ASG model from one technology to another (signified in Figure 2 by the ellipses with the EMF inscription).

Furthermore, instead of executing generated code we now have to run the interpreter on a detection algorithm model which enforces adaption of the inference algorithm.

3.1 Bridging the story diagram dialects

Reclipse and the story diagram interpreter use different story diagram meta-models which have different expressive power. Some things that can be modeled with Reclipse's meta-model cannot be modeled with the interpreter's meta-model. Other things are modeled differently. The Reclipse meta-model, for instance, contains an element called *statement activity* which can hold arbitrary Java code that is later integrated into the generated code. The interpreter obviously

[4]http://www.eclipse.org/modeling/emf/

does not generate any code, so its meta-model does not contain such an element.

Thus, to use the story diagram interpreter the Reclipse story diagram models had to be transformed into story diagram models conforming to the meta-model of the interpreter. This transformation had to take the meta-model differences into account. That means, some things are transformed into more complex "workaround" models and others cannot be transformed and thus may no longer be used in pattern specifications (unless the interpreter is extended).

For each element, we described a transformation rule from the Fujaba story diagram meta-model to the interpreter's story diagram meta-model, if possible. In the following, we describe the transformation of *story patterns* as an example. The full list of transformations can be found in a Master's thesis [4].

Story diagrams describe graph transformations. They closely follow UML activity diagrams and contain a number of story patterns connected by transitions that define the control flow. A story pattern contains a structure of objects that, if it is matched in a host graph (e.g. found in the ASG), is modified as defined by the story pattern (e.g. is annotated).

Figure 3 shows the meta-classes used to model story patterns in Fujaba (top) and the corresponding meta-classes of the story diagram meta-model used by the interpreter (bottom). In Fujaba, a story pattern (*UMLStoryPattern*) is contained in an activity (*UMLStoryActivity*). This activity can be marked as a *for-each* activity, which describes a loop in the control flow. A UMLStoryPattern contains a number of items (*UMLDiagramItem*) which are objects (*UMLObject*), links (*UMLLink*) and method calls (*UMLCollabStat*). Additionally, a story pattern can contain textual (Java) constraints (*UMLConstraint*) and *maybe* constraints that weaken the matching rule (i.e. allow to map more than one node in a story pattern to the same node in an ASG).

The story diagram meta-model of the interpreter in comparison combines the two classes *UMLStoryActivity* and *UMLStoryPattern* into one (*StoryActionNode*). It can contain constraints that are described by a hierarchy of *Expressions*. In the Fujaba model the Java constraints are integrated into the generated code whereas the interpreter evaluates the expression hierarchy. *Maybe* constraints are not supported by the interpreter. Objects and links are separately linked to a StoryActionNode. Methods calls are handled by another type of activity not shown in Figure 3.

This example shows that most elements can be translated quite easily, but some elements (e.g. *maybe* constraints) cannot be translated at all. We defined transformations for all elements that could be translated. These also contained elements that are part of the interpreter's meta-model, but not yet evaluated by the interpretation engine itself. So, once the engine is extended to evaluate these elements, they can be used for pattern specification again.

3.2 Bridging technical differences

Reclipse is based on Fujaba and the story diagram interpreter is developed with EMF. Fujaba and EMF are not

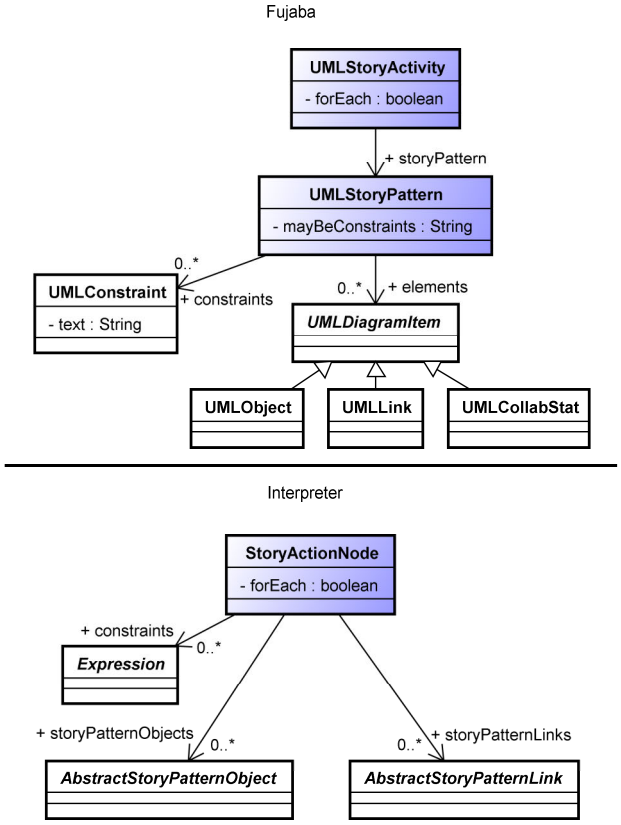

Figure 3: Story patterns in the two story diagram meta-models.

compatible. The saved models have different formats, the models are on different meta levels (UML vs. eMOF [11]) and the generated code follows different implementation conventions. To use the story diagram interpreter, we converted the Fujaba-based models generated by Reclipse (esp. story diagrams) and ASG models into EMF-based models.

The story diagram meta-model of the interpreter was created using EMF. Using the transformation described in Section 3.1, we could translate Fujaba story diagram models into EMF story diagram models. Based on this transformation, the Reclipse component that generates detection algorithm models from pattern specifications was replaced with a new component that creates detection algorithm models that are conformant to the interpreter's EMF-based meta-model. The new component was implemented manually. This way, it could easily be derived from the former (also manually implemented) component, so that it fits into the process and conforms to its interfaces.

Because of the interpreter's requirement to get EMF-based models as input, we also switched from the Fujaba-based ASG meta-model to an EMF-based ASG meta-model. The former, hardly maintainable component for parsing source code into an ASG was replaced by a manually implemented slim component that uses an existing Eclipse plug-in for parsing Java code (JDT[5]).

[5]http://www.eclipse.org/jdt/

3.3 Inference adaptations

In the original pattern detection process the inference algorithm selects an element of the ASG and starts the search for a pattern by executing the corresponding generated code. As we removed the code generation step, the inference algorithm had to be adapted so that it triggers the interpretation of a story diagram rather than the execution of code.

In addition, the inference algorithm had to be adapted to the use of EMF-based models instead of Fujaba-based models. Thus, we adapted existing interfaces and introduced new ones in the inference algorithm's implementation. This way, the reverse engineer now can choose whether she wants to use the former process based on code generation or the one based on the interpreter. The needed code adaptations and additions had to be done manually, because of the complexity of the existing (manually evolved) code base.

4. EVALUATION

We applied the adapted pattern detection process to evaluate our success. The traceability of the process improved, because the engineer no longer needs to bridge the gap between generated code and story diagrams. The interpreter provides a log of all interpretation steps. The coming story diagram debugger will further improve the traceability by visualizing the current state of execution and offering opportunities to observe and influence the execution. Although there are some limitations in the use of the story diagram interpreter, the first detection results are promising.

4.1 Limitations

Most story diagram elements could be translated from the one meta-model to the other. Except for *maybe* constraints, all non-translatable elements are provided in the interpreter's meta-model, but not yet supported by it's execution algorithm. For example, *paths* are not supported so far. A path between two ASG nodes describes that there is a directed, possibly indirect connection between the nodes. In the meta-model paths are represented by a special type of link between objects, but the execution algorithm does not separately handle them. In the story diagram translation we included these elements, so that they can be used as soon as the interpreter supports them.

4.2 Detection results

We evaluated the adapted detection process by detecting patterns in JUnit[6] 4.8.2 and comparing the detection results with those obtained with our previously applied detection process. For this purpose, we re-used an existing catalog of design pattern [5] specifications and auxiliary subpattern specifications.

The catalog had to be modified due to the limitations of the interpreter and its story diagram meta-model (cf. Section 4.1). Pattern specifications that could not be modeled for use with the interpreter were removed for both detection processes. Pattern specifications that had to be weakened (some conditions had to be removed because of the lack of expressiveness) for the interpreter use, were only modified for the run of the adapted detection process.

[6]http://www.junit.org

Pattern	SDI	CodeGen
AbstractStructureImplementation	114	78
AbstractType	17	16
ContainerWriteAccessMethod	379	12
DirectGeneralization	77	75
Field	194	191
Implementation	29	29
IndirectGeneralization	66	31
InterfaceType	13	10
MultiReference	276	7
OverriddenMethod	157	122
SingleReference	135	115
TemplateMethod	86	11
Visitor	1	1

Table 1: Pattern detection results.

Table 1 contains the pattern detection results. The "SDI" column lists the number of pattern implementation candidates detected by the adapted process using the story diagram interpreter. The column "CodeGen" lists the number of candidates detected with the original process using generated code.

As some patterns had to be removed from the catalog because of the interpreter limitations, the only "real" design pattern implementations found by either detection process were *Template Method* and *Visitor*. The latter was found equally often. Candidates for the *Template Method* pattern were found more often by the adapted process than by the original. The same holds for most other patterns (e.g. *ContainerWriteAccessMethod* and *MultiReference*). This is a result of the weakened pattern specifications. For example, the story diagram meta-model used by the interpreter does not support paths. So, they had to be removed, meaning that instead of searching for connected nodes, arbitrary, possibly unconnected nodes satisfying all other conditions are searched in the ASG. This results in more matchings.

Despite the fact that we have more false positives with our adapted detection process, the results are promising. Avoiding the code generation step significantly increases the traceability of the pattern detection. Debugging the pattern specifications and the detection process is easier and will be even more traceable with the interpreter's debugger [8]. The results obtained with the interpreter deviate from the original results (cf. Table 1) solely because of the weaknesses in the current interpreter implementation. Our story diagram translation already supports some story diagram elements that the interpreter does not yet consider. Thus, by improving the interpreter the pattern specifications will become more sophisticated and the detection results will equal the results achieved by the original process.

5. RELATED WORK

The overall goal of our work was to simplify the debugging of the detection process. The two main challenges with our approach were the transformation of the story diagram meta-models and the migration from Fujaba to EMF.

Geiger and Zündorf developed a tool to debug code generated by Fujaba and connect it at runtime to the correspond-

ing story diagrams [6]. As an alternative to using the story diagram interpreter, we could have used this approach, but that would have made the detection process more complex instead of simplifying it and we could not have benefited from the possible performance gain resulting from the use of information that is only available at runtime [7].

There are numerous approaches for model-to-model transformation which we could have used to translate the story diagrams from one meta-model to the other. Among them are *Triple Graph Grammars* (TGGs, [13]) and OMG's QVT (Query/View/Transformation, [12]). These approaches support model synchronization and bidirectional transformations. We decided against generating story diagrams conforming to one meta-model and then translating them to story diagrams conforming to another meta-model during each pattern detection. Instead, we decided to adapt the story diagram generation once and omit the creation of obsolete story diagram models. Since we already had a generator for Reclipse's story diagrams, we only had to replace the creation of story diagram elements such that they conform to the new meta-model. Furthermore, Java code represented by plain text in generated story diagrams significantly complicates the translation with TGGs and QVT.

Amelunxen et al. [1] developed an approach to tool integration using TGGs. This approach still requires manual code adaptations and as it uses TGGs it has the aforementioned disadvantages. Thus, we chose another solution.

6. CONCLUSIONS AND FUTURE WORK

To simplify the pattern detection process of the Reclipse tool suite and support the engineer in finding mistakes in his specifications, we integrated a story diagram interpreter and removed the code generation step.

The used interpreter still has some limitations. It does not yet support certain story diagram features that were supported by the formerly used story diagram meta-model. Adding these features is future work which is already started by the SDM unification task force that aims to unify the meta-models used by several teams from the Fujaba community.

Furthermore, the story diagram debugger [8] needs to be integrated. This debugger would simplify the search for pattern specification errors. The language used for pattern specifications is partly very similar to story patterns. So, if the debugger reveals an error in a story pattern, it will be easy to find the corresponding element in the pattern specification.

7. REFERENCES

[1] C. Amelunxen, F. Klar, A. Königs, T. Rötschke, and A. Schürr. Metamodel-based tool integration with MOFLON. In *ICSE '08: Proceedings of the* 30^{th} *International Conference on Software Engineering, Leipzig, Germany*, pages 807–810, 2008.

[2] J. Dong, Y. Zhao, and T. Peng. A Review of Design Pattern Mining Techniques. *International Journal of Software Engineering and Knowledge Engineering (IJSEKE)*, 19(6):823–855, Sept. 2009.

[3] T. Fischer, J. Niere, L. Torunski, and A. Zündorf. Story Diagrams: A new Graph Rewrite Language based on the Unified Modeling Language. In H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors, *Proc. of the* 6^{th} *International Workshop on Theory and Application of Graph Transformation (TAGT), Paderborn, Germany*, LNCS 1764, pages 296–309. Springer Verlag, November 1998.

[4] M. Fockel. Interpretation von Graphtransformationsregeln zur statischen Erkennung von Software-Mustern. Master's thesis, University of Paderborn, Oct. 2010. (In German).

[5] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, USA, 1995.

[6] L. Geiger and A. Zündorf. Design Level Debugging with Fujaba. In *International Workshop on Graph-Based Tools (GraBaTs), Barcelona, Spain*, 2002.

[7] H. Giese, S. Hildebrandt, and A. Seibel. Improved Flexibility and Scalability by Interpreting Story Diagrams. In T. Margaria, J. Padberg, and G. Taentzer, editors, *Proceedings of the Eighth International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT 2009)*, volume 18. Electronic Communications of the EASST, 2009.

[8] A. Krasnogolowy. Entwurf und Implementierung eines Debuggers für Story-Diagramme. Master's thesis, Hasso-Plattner-Institut für Softwaresystemtechnik GmbH, Potsdam, Germany, 2010. (In German).

[9] U. A. Nickel, J. Niere, and A. Zündorf. Tool demonstration: The FUJABA Environment. In *Proc. of the* 22^{nd} *International Conference on Software Engineering (ICSE), Limerick, Ireland*, 2000.

[10] J. Niere, W. Schäfer, J. P. Wadsack, L. Wendehals, and J. Welsh. Towards Pattern-Based Design Recovery. In *Proc. of the* 24^{th} *International Conference on Software Engineering (ICSE), Orlando, FL, USA*, pages 338–348. ACM Press, May 2002.

[11] Object Management Group. Meta Object Facility (MOF), Jan. 2006. OMG document formal/2006-01-01.pdf.

[12] Object Management Group. Query/View/Transformation (QVT), Apr. 2008. OMG document formal/08-04-03.pdf.

[13] A. Schürr. Specification of Graph Translators with Triple Graph Grammars. In G. Tinhofer, editor, 20^{th} *Int. Workshop on Graph-Theoretic Concepts in Computer Science, Heidelberg, Germany*, volume 903 of *Lecture Notes in Computer Science (LNCS)*, pages 151–163. Springer Verlag, 1994.

[14] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks. *EMF: Eclipse Modeling Framework*. The Eclipse Series. Addison-Wesley, 2^{nd} edition, Dec. 2008.

[15] M. von Detten, M. Meyer, and D. Travkin. Reverse Engineering with the Reclipse Tool Suite. In *Proceedings of the 32nd International Conference on Software Engineering (ICSE 2010), Cape Town, South Africa*, 2010.

A simple indoor navigation system with simulation environment for robotic vehicle scenarios

Ruben Jubeh, Simon-Lennert Raesch,
Albert Zündorf
University of Kassel, Software Engineering,
Department of Computer Science and Electrical Engineering
Wilhelmshöher Allee 73
34121 Kassel, Germany
[ruben | lra | zuendorf]@cs.uni-kassel.de
http://www.se.eecs.uni-kassel.de/se/

Figure 1: Robot vehicle

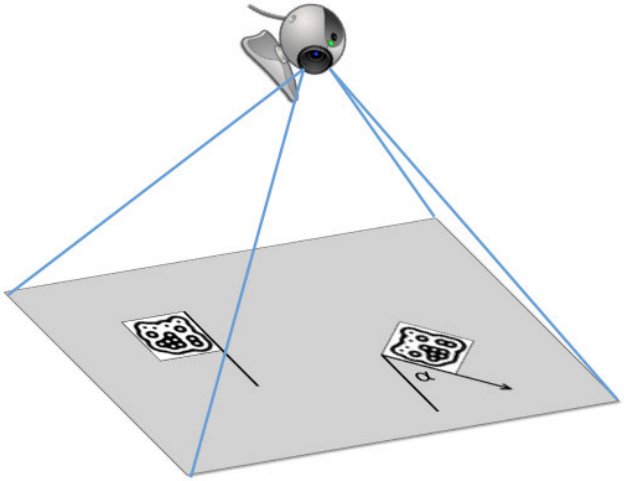

Figure 2: Webcam tracking

ABSTRACT

Following the initial idea of using Lego Mindstorm Robots with Fujaba, as the userbase in students grew larger, an indoor positioning system and simulation environment were required to allow for parallel work on components and projects. Limited resources in actual robots made it impossible for all students to actively work on the development of their projects and evaluating their progress. Sensors on the actual robots are difficult to configure and hard to put in an environmental context. A global positioning system shared among robots can make local sensors obsolete and still deliver more precise information than currently available sensors. A simulator for robots programmed with Fujaba and Java can be used by many developers and lets them evaluate their code in a simple way, yet close to real-world results.

1. INTRODUCTION

Fujaba has been used to model robotic applications via Lego Mindstorms for educational purposes since [3]. Using robotics in Software Engineering courses and projects is very attractive to students, as programs and algorithms can interact with real-world objects. This was shown already in [1] and [2], where a robot solved the towers-of-hanoi game. But using robotics also has its drawbacks: simple actions like moving the robot to certain places is an advanced task, since one needs to control the actuators (motors) and also verify through sensors, that the action succeeded. We experienced this with the towers-of-hanoi-robot, which had a certain unreliability using only light sensors for orientation. Generally, interpreting the real world state by various sensors is complex and often error-prone. Low level hardware interfacing software is difficult to develop and test. Mainly because the "Modeling with Robots"-Course at Kassel University was so popular that more students enrolled to that course than we had robotic kits, we decided to build a simulator around our framework. This simulator allows to develop and test the actual robot control code independently of the hardware. It provides a higher level of abstraction, so students don't have to deal with different sensors and actuators directly. They can rely on framework functionality, which behaves transparently in simulation mode.

In 2009, van Gorp et al. [4] presented multiple use-cases/scenarios the existing Fujaba Lego Mindstorms Library was used for. During the last year, we concentrated on modeling autonomous robotic vehicles. The robots are still remotely controlled by a PC via a bluetooth connection. The vehicles currently used are depicted in figure 1. Two big main wheels allow differential steering and turning on the spot. The yellow front fork is used to push things around, which

is a simplification for picking up or loading goods for transport. On top of the robot, there's an optical marker, which is used to track the robot optically. The robot doesn't use any directly attached sensors, which simplifies programming it a lot. A webcam is tracking all fiducial symbols in its view and broadcasts positions via the network. A detailed explanation of the technical realization can be found in section 3.1. By using this optical global-view tracking approach, each robot individually sees its own position and the position of other marked objects. This is the only sensory input source used in the programming model. This approach simplifies the simulator a lot, because we don't have to simulate any distance metering sensors like ultrasonic or laser/light sensors. Only position data has to be generated out of the movement simulation component. Details about the simulator can be found in section 3.2.

2. USE-CASES AND SCENARIOS

In the last two university terms, we organized a robots modeling project. Students were given a certain use-case scenario, for example the BoxMoveGame presented in section 2.2. They had to implement the autonomous robot vehicle control software and parts of the scenario as software for the simulator. This also included extensions to the Simulator GUI, as presented in section 3.2. At the beginning of each project the scenario was a textual description of what the robot vehicles had to achieve under given rules. Later on, the scenario was implemented in actual code setting up the robots etc. In case of a simulation, all the environment has to be simulated as well. So, a scenario offers basically two operation modes, controlling real robots in a real environment, or simulating robots within a virtual environment.

2.1 Cat and Mouse

This scenario consists of two robots, one playing the role of a cat (the hunter), the other playing a mouse (the prey). The cat periodically scans for the mouse and tries to catch it by driving to its position. The mouse moves randomly around, until there is a cheese object (simply a box with a fiducial marker) being placed in the field, which instantly attracts the mouse's attention. Because both control programs consist only of a single loop of evaluating positions, then blindly drive to a position, the cat will take a while to catch the mouse. What makes the scenario so attractive is that people can interact with both robots by placing the cheese somewhere, which enforces a reaction of the mouse and might help to escape or being trapped. Figure 6 shows cat and mouse together with an cheese object in the simulator.

2.2 BoxMoveGame

The *BoxMoveGame* is a contest game scenario, where two competing robots should move boxes from their home- to their target field. Figure 3 shows the competition area: Robots might drive on the dark areas, but not on white, fields are marked with green boxes. One Robot has to move its boxes from east to west, the other from north to south. The challenge lies in the central traffic light: Depending on the traffic light phase, it might be faster to use the outer ring to drive between fields than possibly wait in front of the traffic light.

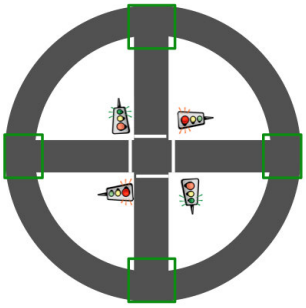

Figure 3: BoxMoveGame-Field

Students had to implement waypoint and path finding. Another challenging task was to deal with asynchronous and synchronous control of the vehicles motors; both alternatives have their pros and cons. It turned out that it was quite difficult to decide which way to choose because of unpredictable timing issues via bluetooth remote control. This scenario shares some runtime objects over the two contestants: the traffic light is run on a third machine as separate software component and coupled via network (RMI). So, this scenario usually involves three controlling instances, as shown in figure 5.

3. TECHNICAL REALIZATION

3.1 FIPS

FIPS stands for *Fujaba|Fiducial Indoor Positioning System*. It's based on the reacTIVision[1] framework, which uses image recognition techniques to detect predefined optical markers via webcam. These markers are called *fiducials*, an example can be seen in figure 4. Each symbol has an ID which can be detected along with the rotation and, of course, the absolute position. Especially having the exact rotation of the robot helps a lot for precisely navigating it, a feature that other in- or outdoor position systems can only achieve by utilizing a magnetic field sensor, which is unreliable even in indoor scenarios.

Figure 4: Fiducial

In our case, the webcam is statically mounted at the room's ceiling, covering an area of approx. 4x3 m. Depending on the ambient light and tracking rate recognition, the camera is usually driven at a resolution of 1280x960 pixels and 10 frames per second, which leads to a theoretical optical resolution of <0.5mm per pixel. This is good enough to use fiducials at a size of 10x10 cm (small set, 24 distinguishable symbols available) resp. 14x14cm (default set, 214 symbols available). As shown in figure 5, the position information has to be shared across multiple control machines, which is done via UDP broadcast. Due to the image processing and network transfers, position updates have a certain latency and are initially limited by the webcam framerate. When a robot moves too fast, the fiducial tracking fails. These properties - resolution, update rate, update latency and maximum move speed - are also considered in the simulator, which will be described next.

[1] http://www.reactivision.org/

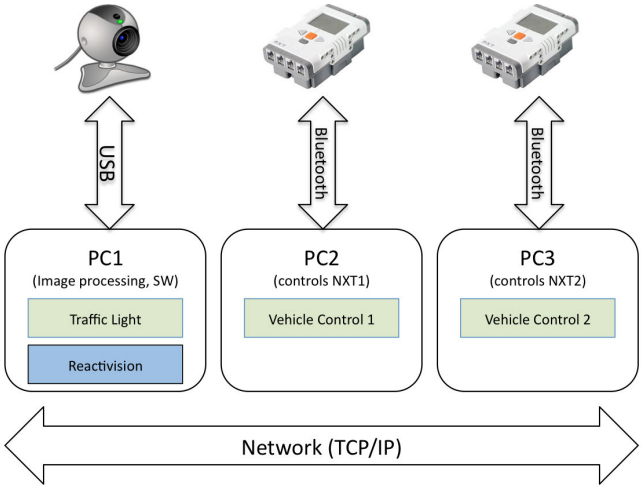

Figure 5: Architectual Overview

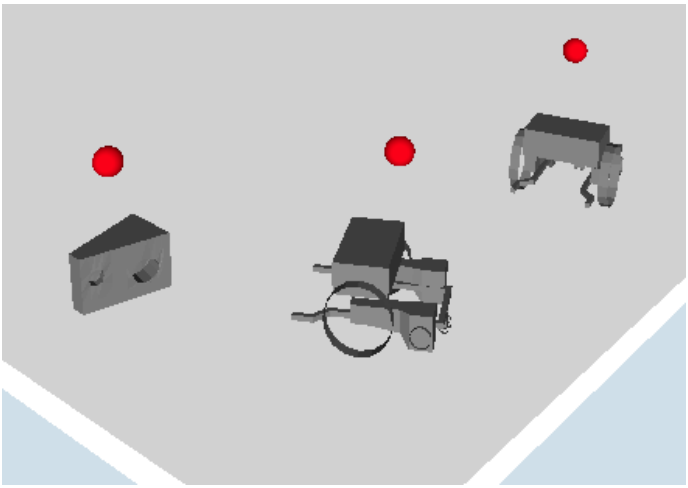

Figure 6: Simulator Screenshot

3.2 Simulator

The main motivation for the simulator is to allow transparent development of scenarios and control software without the need for actual robot hardware. It features a simple 3D graphical viewer written in Processing[2]. It doesn't simulate each driving motor separately, but attaches to the mindstorms libraries' `Navigator` interface and simulates linear movements and rotation in 2D space. Robot movements will be translated in x, y coordinates and a heading value. The navigator is the only used actuator in all scenarios presented here, which abstracts the underlying two wheel driving motors. Additionally, there's an integration of a simple 2D geometric/physics engine for collision detection between robots and other geometric shapes, which updates these coordinates as well. To close the simulation loop, the 2D positions of all objects (after a movement, rotation or collision) have to be fed back to the FIPS, which is the only sensory interface the robot control software uses. As mentioned in section 3.1, some fuzzing transformations identified in real-world tests are applied to the coordinates. This way, the robot can see itself and other tracked objects as soon as they are moving. Some scenarios, for example the robotic beginner use case *Follow a black line* [4], also require extensions to the simulator to allow light sensor feedback from a virtual ground when being simulated. Therefore, the simulator offers an open, extensible interface when a scenario is run in simulation mode.

4. CONCLUSIONS AND FUTURE WORK

The initial idea to use the fiducial framework came from Ulrich Norbisrath, Tartu University. A prototype setup was prepared within a few hours and requires nothing more than a webcam and some printed sheets of fiducials. This makes this system very applicable for our department, as we are software- but not hardware experts. The highly reduced complexity compared to other indoor positioning systems allow a quick and easy setup of the system on other sites. Students can even rebuild it at home. As we decided to abstract the 2D tracking and navigator interfaces for simulation, the framework is limited to use cases where robots use (primarily) these sensors and actuators. Adding more detail or other aspects to the simulation is not offered in the framework, but can be plugged in, like for the *FollowLine* use case. The framework still lacks a plug-and-play setup when connecting different robot control nodes. For example, the BoxMoveGame was the most complex scenario implemented so far and real world runs require a complex setup of connecting up to three machines, setting up fiducial IDs etc. Adding shared objects like the traffic light increases the complexity even more, as it requires a distributed software components layer. We're planning to integrate such a distributed component-based software layer for even more complex scenarios with cooperative robot vehicles in the future. Furthermore, the modeling methodology is insufficient. Using Fujaba statecharts in combination with story diagrams is not intuitive enough, so students still fall back to programming the robots using plain java source code after a while. We finally observe that the Fujaba Indoor Positioning System in combination with a simulator that can be used to realistically mimic robot behavior offer an easy to use yet highly precise tool to make it easy for students to start developing their own robot-code.

[2]http://www.processing.org/

5. REFERENCES

[1] I. Diethelm, L. Geiger, A. Zündorf. UML im Unterricht: Systematische objektorientierte Problemlösung mit Hilfe von Szenarien am Beispiel der Türme von Hanoi. *Erster Workshop der GI-Fachgruppe Didaktik der Informatik, Bommerholz, Germany*, Oct. 2002.

[2] I. Diethelm, L. Geiger, A. Zündorf. Fujaba goes Mindstorms. *Objektorientierte Modellierung zum Anfassen; in Informatik und Schule (INFOS) 2003, München, Germany*, Sept. 2003.

[3] R. Jubeh. Simple robotics with Fujaba. In *Fujaba Days*. Technische Universität Dresden, Sept. 2008.

[4] P. Van Gorp, R. Jubeh, B. Grusie, and A. Keller. Fujaba hits the Wall(-e) – Beta working paper 294, Eindhoven University of Technology. http://beta.ieis.tue.nl/node/1487, Nov. 2009.

SDM online

Interpreting story diagrams in JavaScript for NT2OD

Jörn Dreyer
Kassel University
Wilhelmshöher Allee 73
Kassel, Germany
jdr@cs.uni-kassel.de

Christoph Eickhoff
Kassel University
Wilhelmshöher Allee 73
Kassel, Germany
cei@cs.uni-kassel.de

Albert Zündorf
Kassel University
Wilhelmshöher Allee 73
Kassel, Germany
zuendorf@uni-kassel.de

ABSTRACT

There is currently no story diagram interpreter implementation that can be used in a web browser. This prevents web browser applications from executing story diagrams generated at runtime. Our new interpreter can be compiled to JavaScript and works on a data model that supports reflective acces in JavaScript. This allows the web application NT2OD to dynamically create story diagrams at runtime and use the new interpreter to execute them in a web browser.

Categories and Subject Descriptors

D.2.2 [**Software Engineering**]: Design Tools and Techniques

Keywords

NT2OD, NLP, object diagram, story diagram, interpreter, Fujaba, GWT, FUP, JUnit, test driven development

1. INTRODUCTION

With NT2OD Online [3] we have created a web application that generates simple object diagrams from scenario step descriptions. To further improve the object diagrams we have implemented a set of recommenders that examine the object diagram generated by NT2OD. The hints they produce are then visualized as a list of short story diagrams (see Figure 6). Applying the story diagrams to the object diagram would require a story diagram interpreter in JavaScript. Unfortunately, none is readily available.

In this paper we present an implementation for a story diagram interpreter that works on a simple data model suitable for web browsers:

- We provide a reflection mechanism that can be used with the Google Web Toolkit[1] (GWT) (section 4).
- We provide a Java implementation of a story diagram interpreter that can be compiled to JavaScript with GWT (section 5).
- We explain the use of the interpreter in NT2OD. Point your browser to `http://www.nt2od.org` to test the current implementation (section 6).

[1] `http://code.google.com/webtoolkit`

2. THE PROBLEM

To provide tool support for the early steps in the Fujaba Unified Process [5] (FUP) we created NT2OD [8]. NT2OD generates simple object diagrams from native text to visualize scenario step descriptions. The current results are promising, but the diagrams still lack technical details like attribute and class type information [3].

We need these technical details before we can use the simple object diagram to generate code for a JUnit [4] test. Take e.g. the sentence in Listing 1.

```
Alice and Bob are playing Ludo.
```

Listing 1: Example sentence

After identifying the grammatical relations with the Stanford parser [1] the current implementation of NT2OD produces the object diagram in Figure 1.

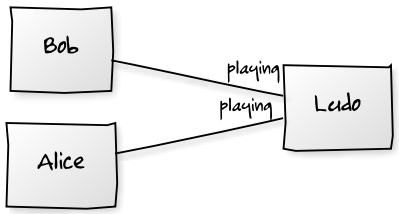

Figure 1: Object diagram for the example sentence

NT2OD identifies general concepts and links but there is no technical information in the simple object diagram. A seperate list of technical hints is vizualized, based on story diagrams that have been generated by a set of recommenders [2]. A simple web service parses URLs like `http://nt2od.org/fuml/diagram/story/[Alice]-playing-[Ludo],[Bob]-playing-[Ludo].png` and uses graphviz[2] to produce the di-

[2] `http://www.graphviz.org/`

agram images[3]. Although the data model for both types of diagrams is available in the browser we cannot apply the technical hints to the simple object diagram. A story diagram interpreter written in JavaScript is missing.

3. INTERPRETING STORY DIAGRAMS IN A BROWSER

Fortunately, we do not have to write JavaScript code but can rely on GWT to compile Java source code to JavaScript. This way we can reuse existing Java tool support for the development of browser applications. First, we have to add a reflection mechanism to the existing data model. In the second step, the interpreter will use the new reflection mechanism to match the features and attributes of classes. Finally, we will be able to use the story diagram interpreter in NT2OD to enhance the simple object diagram with technical hints.

4. ADDING REFLECTION TO THE DATA MODEL

The data model for object diagrams (Figure 2) and story diagrams (Figure 3) is quite simple, but it lacks a reflection mechanism than can still be used after compiling Java to JavaScript[4]. We decided to create a `HasReflectiveAccess` interface (Figure 4) that the interpreter can use to interact with the data model.

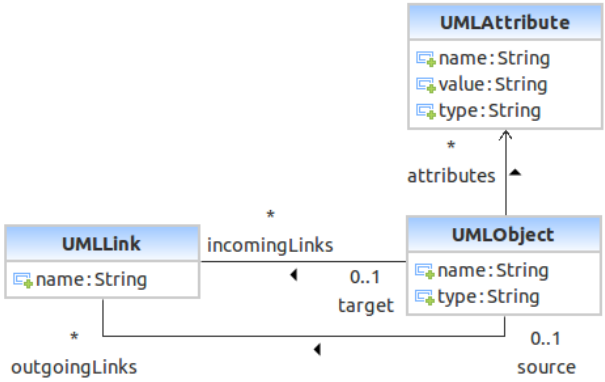

Figure 2: The object diagram data model

A JUnit test suite for both data models verifies the expected behaviour for field access and attribute assignments. The traditional test driven development allows us to implement the `HasReflectiveAccess` interface for each class without having to wait for the GWT compiler.

5. THE INTERPRETER

With the data model in place we can start working on the interpreter itself. Every year our students have to imple-

[3]The object diagrams of NT2OD were originally visualized by misusing the class diagrams produced by `http://yuml.me`. As their service suffered an extended downtime we used the opportunity to create our own implementation that supported object and story diagrams.

[4]see `https://code.google.com/webtoolkit/doc/latest/DevGuideCodingBasicsCompatibility.html`

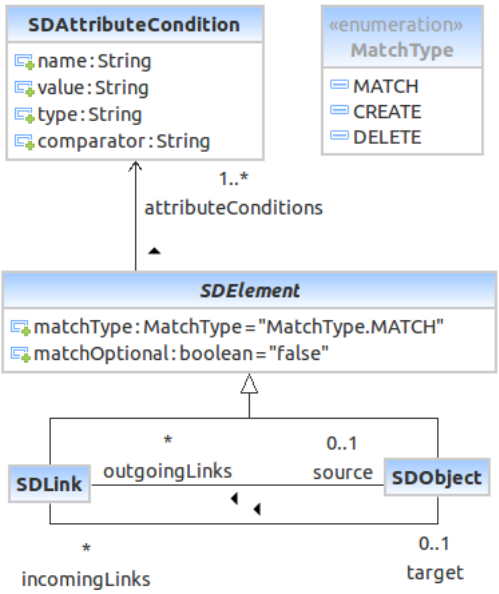

Figure 3: The story diagram data model

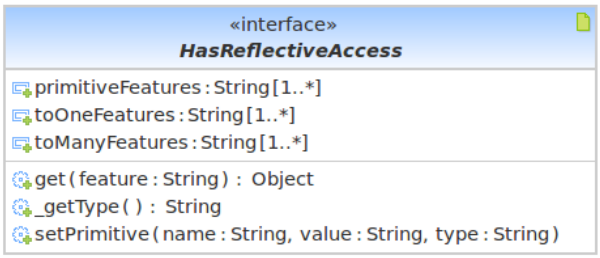

Figure 4: The `HasReflectiveAccess` interface

ment a story diagram interpreter[5]. Now it is time to do the exercise ourselves. As in the lecture, we are using a stack based approach for the matching algorithm in order to ease debugging compared to a recursive implementation.

Figure 5 shows the three classes that make up the implementation:

- The `Interpreter` that is initialized by setting a `StoryDiagram` and an `ObjectDiagram`. The `execute()` method is then used to match and apply the story diagram to the first or all occurences.
- Internally a stack of `Steps` is used in the matching process.
- Instances of the `Feature` class are used to remember source and target objects for matched features in the story diagram.

Using JUnit tests, we were able to, again, implement the

[5]see `http://seblog.cs.uni-kassel.de/fileadmin/se/courses/MDESS10/MDE04RuleMatching/MDE04RuleMatching.html`

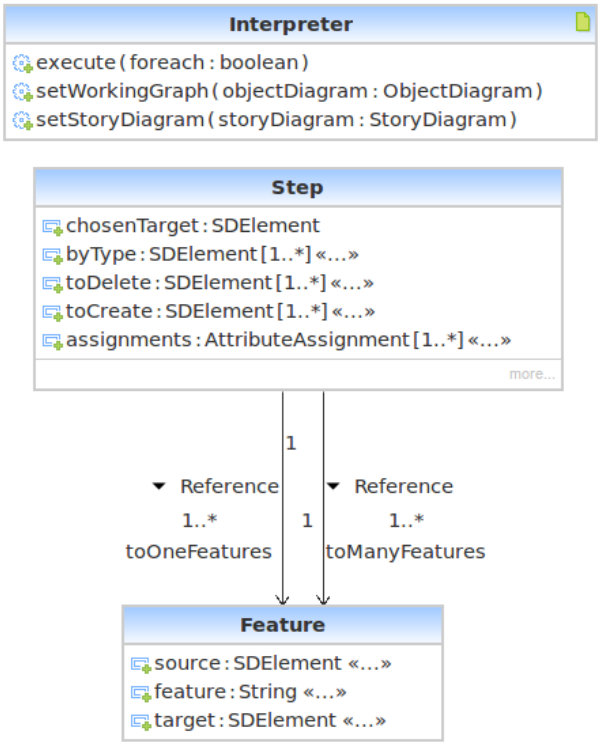

Figure 5: Important elements of the interpreter

complete matching algorithm without waiting for the GWT compiler.

At the time of writing this paper the code already passes the JUnit tests for matching the story diagram to the object diagram and the execution of create and delete expressions. Development is now focusing on negative and optional matches.

6. USING THE INTERPRETER IN NT2OD

The NT2OD workflow for creating a JUnit test for a textual scenario description consists of the following steps:

1. Parse the scenario description.
2. Render grammatical relations and simple object diagram.
3. Create a working copy of the object diagram by cloning it.
4. Collect meta information by asking recommenders for story diagrams they would apply to the object diagram.
5. Create a technical object diagram by executing the story diagrams on the object diagram copy.
6. Use the technical object diagrams to generate a JUnit test.

The mentioned recommenders are work in progress but we have already implemented recommenders that work on backends like YAGO2 [7] and `vornamen.com`. For the example sentence in Listing 1 the `vornamen.com` based recommender will create a story diagram as in Figure 6.

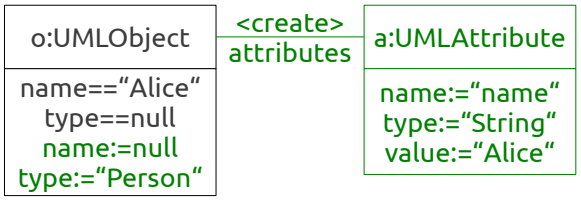

Figure 6: An example story diagram

NT2OD visualizes the list of story diagram under the "Technical hints" tab for the scenario step. In order not to irritate the user with the internal data model we are working on a graphical representation of story diagrams on the same meta-level as the object diagram.

Nevertheless, in step 5 of the NT2OD workflow the simple object diagram and story diagram are used to initialize and execute our new interpreter. The resulting technical object diagram now contains type information that can be used for code generation.

7. RELATED WORK

Traditional story diagram interpreters rely on the Java reflection mechanism to inspect the working graph and the story diagram. Unfortunately, JavaScript does not support reflection natively and GWT does not generate a replacement we could exploit. Therefore the interpreter implementation [6] working on an EMF based data model could not simply be cross compiled to JavaScript. Nevertheless, their work inspired us to start implementing our own interpreter and data model with a reflection mechanism that still works after cross compiling it to JavaScript.

8. CONCLUSION AND FURTHER WORK

In the context of NT2OD we are implementing tool support for the early steps of the Fujaba unified process as a web application. In this paper we add a reflection mechanism to the data model that can be cross compiled and used in JavaScript. We also implemented an interpreter that works on this data model and is used in NT2OD to add technical meta information to simple object diagrams. This is the first and important step towards a browser based version of Fujaba.

In the future we are planning to test several recommenders that are based on resources like existing ontologies and class diagrams. The user interface will also receive a diagram editor, maybe based on GWT-UML[6].

9. REFERENCES

[1] M.-C. de Marneffe, B. MacCartney, and C. D. Manning. Generating Typed Dependency Parses from Phrase Structure Parses. In *Proceedings of the IEEE / ACL 2006 Workshop on Spoken Language Technology*. The Stanford Natural Language Processing Group, 2006.

[6] `http://http://code.google.com/p/gwtuml/`

[2] J. Dreyer, C. Eickhoff, and A. Zündorf. Semantic breadcrumbs - Applying sematic hints from the web to NT2OD. In S. de Cesare, editor, *ODISE'11: Ontology-Driven Information Systems Engineering*, London, GB, Apr. 2011.

[3] J. Dreyer, S. Müller, B. Grusie, and A. Zündorf. NT2OD Online - Bringing Natural Text 2 Object Diagram to the web. In S. de Cesare, editor, *ODiSE'10: Ontology-Driven Software Engineering Proceedings*, Reno/Tahoe, Nevada, USA, Oct. 2010.

[4] M. Fowler. A UML testing framework. *Software Development*, 7:41–46, April 1999.

[5] L. Geiger. Automatische JUnit Testgenerierung aus UML-Szenarien mit Fujaba. Master's thesis, Braunschweig, Germany, 2004. Diploma Thesis.

[6] H. Giese, S. Hildebrandt, and A. Seibel. Improved Flexibility and Scalability by Interpreting Story Diagrams. *Electronic Communications of the EASST*, 18, 2009.

[7] J. Hoffart, F. M. Suchanek, K. Berberich, and G. Weikum. YAGO2: a spatially and temporally enhanced knowledge base from Wikipedia. Research Report MPI-I-2010-5-007, Max-Planck-Institut für Informatik, Stuhlsatzenhausweg 85, 66123 Saarbrücken, Germany, November 2010.

[8] A. Zündorf and J. Dreyer. NT2OD - From natural text to object diagram. In P. V. Gorp, editor, *Fujaba Days 2009: proceedings of the 7th international Fujaba days*, pages 56–58. Technische Universiteit Eindhoven, November 2009.

Yet another TGG Engine?

Interpreted Triple Graph Grammars with Fujaba

Nina Geiger, Bernhard Grusie, Andreas Koch, Albert Zündorf
Kassel University, Software Engineering,
Department of Computer Science and Electrical Engineering,
Wilhelmshöher Allee 73,
34121 Kassel, Germany
[nina.geiger | bgr | andreas.koch | zuendorf]@cs.uni-kassel.de
http://www.se.eecs.uni-kassel.de/

ABSTRACT

Invented by Andy Schürr in 1994, Triple Graph Grammars have become a decent way for bidirectional mapping and transformation. However, there is still the impression that there are more Triple Graph Grammar engines and approaches out there than applications really using them. Despite this knowledge, we have decided to build yet another TGG implementation. The implementation described in this paper is based on the work of Robert Wagner [6], but has some remarkable differences. These differences lie in the execution semantics and reuse of objects of the target document, as well as in the less restricted conditions for source and target models. We chose to have an interpreted rule execution because we wanted our design to be as flexible and incremental as possible.

1. INTRODUCTION

The goal of the FAST European project [10] was to create tools to build complex client side gadgets. These gadgets run inside a web browser and are defined as small applications combining different screens and having a specified screen flow. The client side gadgets rely on semantic backend services. Thus we had to deal with different ontologies which had to be transformed into each other. Having our background in software engineering we decided to treat the ontologies as meta models and do the transformation using Triple Graph Grammars.

1.1 Triple Graph Grammars

In 1971 T.W.Pratt invented pair grammars as declarative approach for model transformations [7]. These pair grammars were mostly designed to transform strings into graph languages and vice versa. In 1994 Andy Schürr extended these pair grammars to Triple Graph Grammars (TGGs) [8]. Triple Graphs are defined as pairs of graphs which are linked by a so called correspondence graph or traceability graph. The TGGs always consist of a set of rules that are defined on the meta model of the Triple Graph. This means the rules contain meta model elements of the source and target graphs which will be transformed into each other. Additionally the rules contain elements of the correspondence graph which are used for the mapping of source and target elements. The execution of TGGs is performed by specific graph engines which will also determine the hierarchy of rule execution. TGGs can be used for the transformation and synchronisation of graph based models. The transformation can be executed from source to target model (forward) and vice versa (backward). The synchronisation uses the correspondence graph to determine the mapping between both of the models. The approach presented in this paper defines a TGG engine with interpreted rule execution as well as rule definition using existing editors.

1.2 Why another TGG engine?

The problem we had to solve was the transformation of different ontologies into each other. Doing this using Triple Graph Grammars was obvious for us. Since the whole development process was based on Fujaba and we use Fujaba for the generation of web enabled code, we wanted to stay in this domain. Analyzing existing TGG engines, the work of Robert Wagner [2] was inspected in more depth. The usage of the MoTE and MoRTEn Fujaba plugins would have been the first step, but we soon recognized some disadvantages. First of all, these engine and plugins were implemented for stand-alone Fujaba and we would have had to port the editor and engine to Fujaba for Eclipse. Nevertheless, it still was the only Fujaba compliant engine, since the others out there use different meta models. After discussing pros and cons of different approaches and solutions we decided to have a completely new implementation based on the ideas of Robert Wagner. Adding more value to the new one, we additionally decided to break with the generation of forward, backward and mapping rules as storydiagrams. Instead we wanted to have a completely interpreted approach where rules can be edited at runtime. To ease the edition of rules, we stayed with the storydiagram editor of Fujaba and only changed the semantics to use this as TGG rule editor. The approach presented in this paper shows the first version of our new TGG interpreter which was implemented by Bernhard Grusie during his master thesis [3].

The paper is structured as follows: Section 2 presents an overview of the TGG interpreter and the associated rule editor. Section 2.1 describes the execution semantics of our approach and the graph-pattern matching in more depth. Section 3 shows similarities and differences of the approach presented and other TGG engines. Section 4 summarizes our work and points out future work to be carried out, problems which have to be solved and critical points which will have to be reviewed.

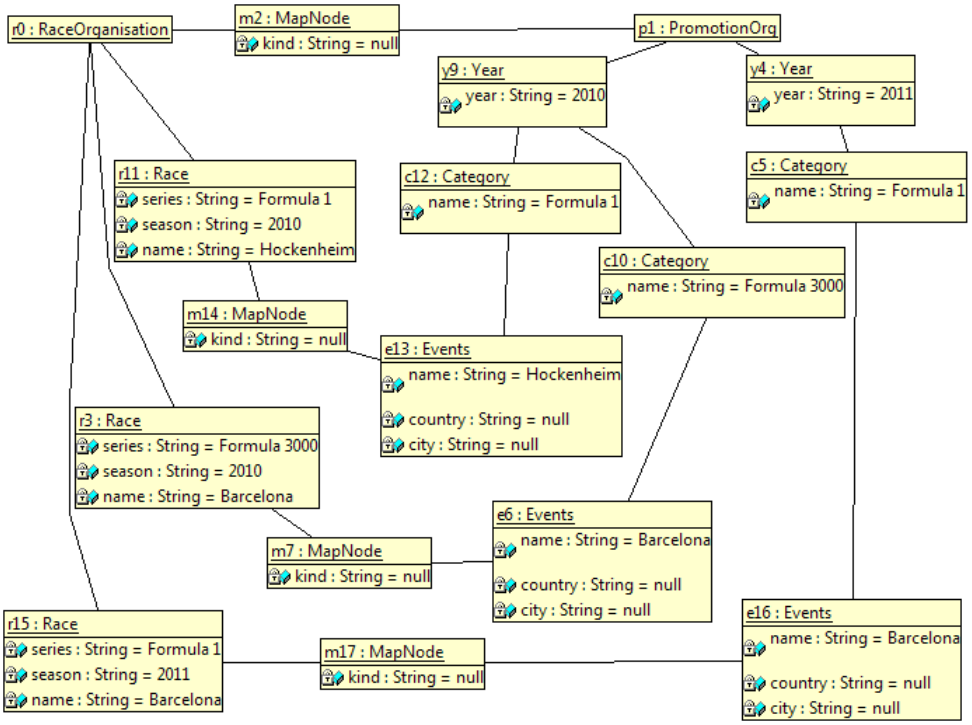

Figure 1: Mapped example documents

2. INTERPRETED TRIPLE GRAPH GRAMMARS WITH FUJABA

The Triple interactive Graph grammar Interpreter (TiG) [3] is used to interactively specify and run Triple Graph Grammars on any data model. The rules specified with our editor do not have to be compiled to source code. The elements of source and target models do not have to fulfill many preconditions, except having a parameterless constructor. This way, we are able to use the TiG for the transformation and mapping of domain specific ontologies as used in the FAST project. These ontology models have to be run inside the web browser and thus will be derived by the Object class.

Using our approach rule execution can be monitored by exploring the runtime object structure with eDobs [1]. While the rules can be changed on the fly and the eDobs view is dynamically updated, the rule editing process can be done step by step. The TiG approach uses standard Fujaba story diagrams [11] as rule editor. This has several advantages. First of all, we did not need to implement or adapt any Triple Graph rule editor. Secondly, the rule execution flow can be determined by the story diagram control flow. This means, we are able to describe rule hierarchies with the commonly known patterns of storydiagrams. However, there exist some remarks for the rule creation. To specify a new set of rules for the TiG engine, the user has to create a storydiagram called `trafoSpec()` in the associated Fujaba project. The engine will automatically search the rules inside the body of this storydiagram and read out the content at runtime. The syntax of the TGG rules is based on the one of Robert Wagner's editor. Objects having no special signing (black ones) will be treated as context of the rule. These objects have to be present in the runtime structure for the rule to be able to be executed. Objects having `create` or `delete` stereotypes will either be created or deleted during the execution of the rule. The same holds for links. Attributes can be assigned or checked the same ways it is done in storydiagrams. The first story activity of the `trafoSpec()` diagram always has to contain the axiomatic rule. This means it is only valid if all contained objects and links have the `create` stereotype assigned. The correspondence graph of the TiG approach consists of objects of class `MapNode`. This class has `source` and `target` associations to class `Object`. This way, correspondence nodes can be linked to the appropriate objects in the data structure. The reverse direction of these links is implemented using hash tables within the rule interpreter. Thus, the targeted objects do not need to implement any predefined interface but our TiG interpreter is applicable to general object structures. Rules can be specified using the standard storydiagram control flows. However, cycles inside the control flow will result in invalid rule diagrams. The actual execution of the rules uses the eDobs and the Java Debug Interface (JDI) [5] to retrieve the runtime object structure from the heap. After retrieving the runtime data TiG executes the specified rules and changes the data structure accordingly. These changes are shown in the eDobs view, instantly. An example eDobs structure is shown in Figure 1.

Using these technologies enables us to have an interactive way of triple graph rule specification. It is possible to change the rules inside the Fujaba story diagram and run them directly to see how the specified rules change the object structure. This way, we can have instant feedback of our rule execution and we may change the rule again if their effects do not reflect our intentions or expectations.

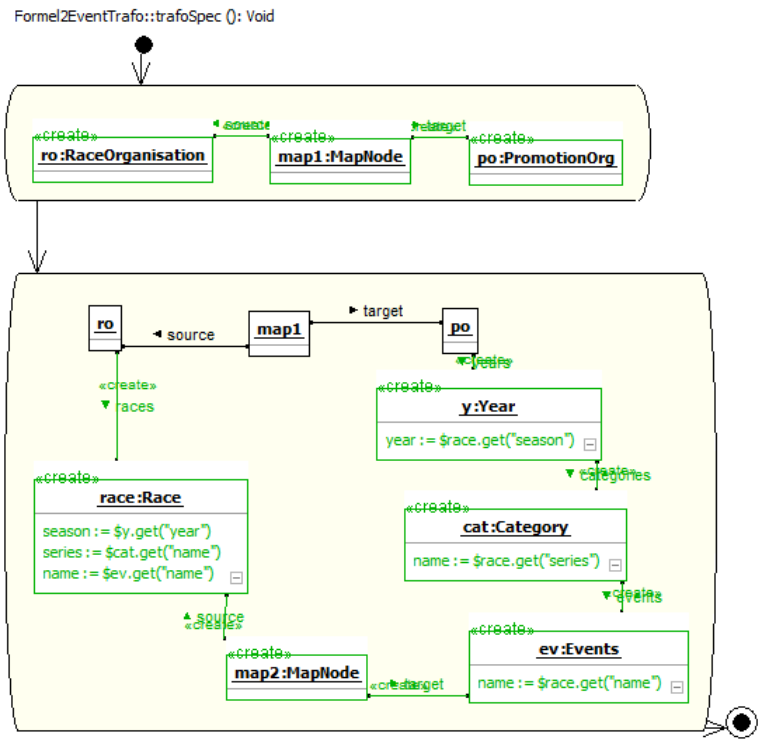

Figure 2: Example TGG transformation diagram

2.1 Pattern matching by running example

As running example for this paper we use the transformation of object structures describing motor races into object structures of a marketing enterprise organizing promotion events. In Figure 1, object `r0` (upper left corner) represents the root of the motor race data structure. In addition, Figure 1 shows on the left three `Race` objects `r11`, `r3`, and `r15`. Each `Race` object stores the name of its `series`, the `year` of its occurrence and the `name` of the race circuit. The root of the marketing enterprise object structure is represented by the `PromotionOrg` object `p1` (upper right corner of Figure 1). The marketing enterprise object structure uses explicit `Year` objects referencing `Category` objects for different kinds of sport events occurring in that year. For example, `Year y9` references two `Category` objects `c12` and `c10` for Formula 1 and Formula 3000 races, respectively. Each `Category` object references all the `Events` of that sport in that year. In our example, there is only one `Event` per `Year` and `Category`.

Adding a new `Race` to the `RaceOrganisation` object structure, a mapping to the `PromotionOrg` object structure should create a new `Events` object and sort it into the existing `Year` and `Category` object hierarchy. Only, if the new `Race` uses a `Category` and `Year` combination that not yet exists, new `Category` and or `Year` objects are created. Due to our knowledge, the reuse of old object hierarchies is a problem for traditional TGG approaches. Therefore, we have developed a new TGG semantics that tries to reuse existing object structures whenever possible.

Figure 2 shows the TGG diagram used for our example mapping. As stated above, the traditional Fujaba storydiagram editor is used for rule definition. Since one activity always defines exactly one rule we are able to use the storydiagram transitions to express a hierarchy on the rules. We do not use events or conditions on the transitions and we consider cycles as invalid diagrams, yet. The execution order of rules is determined by the control flow of the storydiagrams in the same way the Code Generator does. The `MapNodes` of the correspondence graph do not contain any hierarchical information and are not linked with each other. The hierarchy of TGG rules in TiG is derived from the storydiagrams only. The rule execution order is always computed in a deterministic way, leading to exactly one rule being able to be executed.

TiG is able to perform forward and backward transformations of object structures. Depending on the chosen option the rules are interpreted in different ways. For a forward transformation, the TGG interpreter searches for a match of the context elements of a rule and for all elements of the source object structure. A usual TGG approach would now create a copy of the new elements of the target structure and of the new `MapNodes` and insert it into the host graph. In our approach there exists a difference between elements of the target model being connected to `MapNodes` by the rule and elements of the target model without direct connection to `MapNodes`. Those elements with connection to a `MapNode` will always be created during rule execution and linked to the corresponding element in the source structure. The elements of the target model without connection to `MapNodes` will be reused and connected with appropriate objects in the target structure whenever possible. We achieve this behavior by continuing the pattern matching process after finding the match for the context and for the source structure. We additionally try to find matches in the target object structure for as many target rule objects as possible. If

we find an existing object in the target structure that fulfills all available attribute and link conditions, we reuse that existing object rather than creating a new object for the target structure. Only target objects that are already used by another `MapNode` will not be reused. This way it is possible to reuse objects having additional links to other objects. This may lead to additional information being valid in the target object structure that was not explicitly intended by the TGG rule. Often this additional information contains annotations or layout information that do not violate any object structure and is worth having. Rules can be specified using attribute conditions, which normally leads to exact object reuse and prevents the reuse of objects with wrong information.

In our example, the second rule of Figure 2 relates a `Race` object to an `Events` object `ev` that is attached to a `Category` object `cat` that in turn is attached to a `Year` object `y` that is attached to a context object `po`, the root of the `PromotionOrg` structure. If we apply this TGG rule to a new `Race` object that has not yet been mapped, the rule interpreter first matches the new `Race` object, its parent context object `ro` of type `RaceOrganisation`, and the `MapNode` `map1` that maps `ro` to the `PromotionOrg` object `po`. This forms a valid match for the forward execution of our rule. Next, the TGG interpreter checks whether the matched race object has already a `MapNode` attached to it. If such a `MapNode` exists, it determines the mapping for the `Events` object `ev` of our rule. In our example, the new `Race` object of our host graph has no `MapNode` attached to it and thus the TGG interpreter knows that it has to create a new mapping. For this new mapping, the TGG interpreter tries to reuse existing host graph objects whenever possible. Thus, the TGG interpreter continues the matching process by trying to find matches for the `Year` object `y` and the `Category` object `cat`. Therefore, the TGG interpreter looks up `years` links leaving the match of the `PromotionOrg` object `po`. Each `Year` object that is found in the host graph is checked for the attribute condition depicted in Figure 2. Note, we use Velocity [9] templates as attribute condition languages. This allows our interpreter to do some simple string manipulation, computation, and comparisons on attribute values. If the `year` attribute matches the `season` attribute of our `Race`, the `Year` object is reused, otherwise a new `Year` object is created. After handling the `Year` object, the interpreter continues to search for a match of the `Category` object `cat`. If we have reused an old `Year` object, there might also exist an old `Category` object attached to that `Year` object that matches our needs. In case of a new `Year` object, we will fail to find an attached `Category` object. If we fail to find a matching old `Category` object, the TGG interpreter will create a new one.

In our rule, the `Events` object `ev` is connected to a new `MapNode` via a `target` link. Thus, the `ev` object is handled differently. If the original `Race` has already got a `MapNode` attached to it, this `MapNode` provides us with the `Events` object to be reused for the mapping. If there is no old `MapNode`, as in our example, the TGG rule will create a new `MapNode` and a new exclusive `Events` object. Thus, in our example, the TGG interpreter does not try to find existing `Events` objects attached to the match of the `Category` object `cat`. Instead, the TGG interpreter creates a new `Events` object and a new `MapNode` object and the three links attached to these objects and it executes the depicted attribute assignment.

In summary, our TGG interpreter distinguishes between elements of the target model attached to `MapNodes` and elements of the target model without `MapNodes`. Elements of the target model without `MapNodes` are reused when possible. This is achieved by continuing the pattern matching process for the elements of the target model without `MapNodes`. In principle, this approach has the disadvantage that the degree of reuse somehow depends on our pattern matching strategy. In case of a complex target object structure, the TGG interpreter starts from the context elements in order to find target objects for reuse. If there are alternative paths how a certain object may be reached, the TGG interpreter will use the "most efficient" one for its search. If this "efficient" path fails on some early objects, that object will be created and continuing search from that new node will fail from that on. Thus, in complicated cases the TGG interpreter might fail to reuse existing objects that it might have found if it would have used another search strategy. We tried to address this problem in our TGG interpreter by using a kind of breadth-first strategy for the search plan construction. This means, if there are multiple paths leading to a target object and if one of this paths fails, the TGG interpreter first tries to follow alternative paths before it eventually creates a new node. Still, we are not sure whether this guarantees maximal reuse in all cases. However, until now, these problems did not occur in our applications translated using the TiG approach.

A special problem for TGG approaches is the handling of deleted source elements and of changes to source elements that invalid existing mappings. For example the change of the `season` attribute of a `Race` object requires that the corresponding `Events` object is moved to another `Year` object. In our example, this also requires the move to another `Category` object. In principle, the old `Year` and `Category` objects might be deleted, if they are no longer used. Since our TGG semantics advocates for the reuse of existing objects whenever possible, our TGG interpreter will just establish the new target structure without deleting target elements of the no longer valid old match. We keep the potentially unused elements because later transformations might reuse them. Thereby, we keep manual annotations on those old objects. If a source element is actually deleted, our TGG interpreter notices the orphan `MapNode` and deletes the corresponding target node, too. However, additional target elements created by the old rule execution but not attached to a `MapNode` will be kept for later reuse.

3. RELATED WORK

As stated in 1.2 we based our interpreter on the work done by Robert Wagner [6]. While this work needs the rules to be compiled to source code before executing the rules, our approach allows for runtime rule editing and does not need any code generation. The Triple Graph implementation explained in [4] is also based on the work of Robert Wagner. Similar to our approach it has support for incremental model transformations and reuse of target model structures. However, while this approach uses connection form every source or target object to correspondence nodes, the reuse mechanism is slightly different from the one presented in our approach. In TiG it is possible to have elements in the source or target object structure which do not have any connection to correspondence nodes. This enables us to use e.g. one `Year` object for multiple `Category` objects. Another difference to the approach presented in [4] is the creation of rule

execution orders. While the rule hierarchy of the Potsdam implementation is built upon links in the correspondence nodes, the TiG determines the rule order from the storydiagram control flow.

4. SUMMARY AND FUTURE WORK

Our TGG interpreter imposes a minimal impact on the participating object structures. The MapNodes use hash tables to store the reverse direction of their source and target links. Therefore, the application classes do not need to implement any special interface.

Using an interpreter, our TGG approach is very flexible and especially tuned for incremental mappings triggered by change events. If e.g. the name of a Category object is changed, this Category object could be used as a starting point for the mapping of a backward execution of our second TGG rule. Using multiple start points for the pattern matching process is somewhat complicated for compiler based approaches. In addition, the interpreter approach facilitated the implementation of our partial object reuse strategy for the target object structures. While it is surely possible to do this with a compiler based approach, too, we had no instant idea how this could easily be done. For the interpreter this was easy to achieve.

Our TGG approach introduces a new TGG semantics. Elements of the target model attached to `MapNodes` are handled as in classical TGG semantics. Elements of the target model without `MapNodes` reuse existing host graph objects in the target structure as much as possible. Classical TGG approaches can reach this effect to a certain degree using optional nodes. However, optional nodes usually have certain restrictions (two optional nodes may not be connected by links) that restrict their general applicability. Thus, our new TGG semantics gives the user additional flexibility.

Due to our experiences, our new TGG semantics is especially useful for incremental re-transformations. If rules are executed again, after some changes to the source model, our TGG interpreter manages to identify target objects that do not longer match. In addition it frequently manages to identify alternative target objects that now match. Thus, the target structure is reconfigured using existing objects instead of the creation of new objects. Thereby, additional manual changes to reused target objects (and e.g. layout information) survive.

We also have made promising experiences with the evolution of transformation rules in case of existing mappings. In our example projects, we have developed the transformation rules interactively. After applying a certain version of the rules to an example object structure, we modified and extended some rules and then we just re-ran the transformation. In most cases the TGG interpreter correctly removed old mappings that were no longer valid and replaced them by new appropriate mappings. During this remapping, the reuse of target objects resulted in minimal changes to the target structure keeping manual annotations and e.g. the eDobs layout of the object structure almost intact.

Our current implementation has still some bugs under work. In addition, the current TGG interpreter relies on eDobs for the analysis of and access to the host graph. While this has the advantage of an uniform access to applications running remote or in the TGG interpreter VM or in a debugger VM, this creates a certain runtime overhead. Generally, we have the problem that the TGG interpreter uses the Fujaba editors to access the TGG rules. For a standalone application, it would be nice to have a small runtime library allowing to represent the TGG rules without the need to add all the Fujaba jars to your application. Finally, we did not yet implement the incremental triggering of TGG rules by change events. This would enable a really incremental approach, transforming only changed elements rather than analyzing the whole source object structure. Once these technical problems are solved, we plan to look into extended rule features like negative elements. Further work lies in the analysis of rule execution order from storydiagram control flow. We plan to have additional features like conditions at transitions between the activities, here.

5. REFERENCES

[1] L. Geiger and A. Zündorf. eDOBS - Graphical Debugging for Eclipse. Natal, Brasil, September 2006.

[2] H. Giese and R. Wagner. Incremental model synchronization with triple graph grammars. In O. Nierstrasz, J. Whittle, D. Harel, and G. Reggio, editors, *Proc. of the 9th International Conference on Model Driven Engineering Languages and Systems (MoDELS), Genova, Italy*, volume 4199 of *Lecture Notes in Computer Science (LNCS)*, pages 543–557. Springer Verlag, 10 2006.

[3] B. Grusie. Ein objektorientierter, interaktiver triple graph grammatik interpreter. Master's thesis, Universität Kassel, Fachgebiet Software Engineering, 2010.

[4] S. Hildebrandt. Effiziente modellsynchronisation mit triple-graph-grammatiken durch wiederverwendung von transformationsergebnissen, 2007.

[5] JDI - Java Debug Interface Specification. http://download.oracle.com/javase/6/docs/jdk/api/jpda/jdi/index.html, 2011.

[6] E. Kindler and R. Wagner. Triple graph grammars: Concepts, extensions, implementations, and application scenarios. Technical report, Software Engineering Group, Department of Computer Science, University of Paderborn, 2007.

[7] T. W. Pratt. Pair grammars, graph languages and string-to-graph translations. *J. Comput. Syst. Sci.*, 5(6):560–595, 1971.

[8] A. Schürr. Specification of graph translators with triple graph grammars. In *in Proc. of the 20th Int. Workshop on Graph-Theoretic Concepts in Computer Science (WG '94), Herrsching (D.* Springer, 1995).

[9] Apache Velocity User Guide. http://velocity.apache.org/engine/releases/velocity-1.5/user-guide.html, 2011.

[10] FAST. http://fast.morfeo-project.eu/, 2011.

[11] A. Zündorf. Rigorous object oriented software development. Habilitation Thesis, University of Paderborn, 2001.

A Master Level Course on Modeling Self-Adaptive Systems with Graph Transformations

Matthias Tichy
Organic Computing, Department of Computer Science
University of Augsburg, Augsburg, Germany
tichy@informatik.uni-augsburg.de

ABSTRACT

A growing emphasis in software and systems development is being laid on the integration of self-x characteristics like self-healing, self-adaption, self-optimization. More often than not those characteristics can be modeled in terms of graphs and graph transformations. At the Organic Computing group at the University of Augsburg we addressed the topic of modeling self-adaptive systems in a semester long course at the master level. It was designed to contain extensive practical applications of software modeling tools to go along with the lectures. We employed GROOVE and Fujaba as tools for the practical assignments w.r.t. to graph transformations. We report about the course topics, the practical assignments and lessons learned.

Categories and Subject Descriptors

D.2.2 [**Software Engineering**]: Design Tools and Techniques

General Terms

Design

Keywords

Self-X, model-driven development, graph transformations, teaching, Fujaba

1. INTRODUCTION

The current trend for the increasing embedding of software in technical systems is accompanied by requirements that these technical systems should be working as good as possible by optimizing itself and adapting to changes in the environment or the users needs.

This requirement adds additional complexity to the already complex distributed software in todays embedded systems. Model-driven development approaches are employed to counter this growing complexity by abstracting from details of the hardware platform or the underlying middleware as far as possible. Model-driven approaches are often built on domain specific languages which are tailored to the problems and needs of a specific domain.

Kramer and Magee [10] suggest to adapt and to use the three layer architecture of Gat [5] for self-adaptive systems consisting of the following layers: (1) *goal management*, (2) *change management*, and (3) *component control*. The component control layer does contain the architectural configuration of the self-adaptive systems, i.e. the components and their connections that are active in a certain configuration. Besides the execution of the components, this layer is responsible for the execution of reconfiguration plans. These plans, which consist of actions like adding, removing, and replacing of components and connectors, are stored in the change management layer and are executed in response to events. The plans are computed in the goal management layer and correspond to goals.

The graph transformation formalism is a natural fit for the specification of structural self-adaption as envisioned by Kramer and Magee. A number of different approaches based on graph transformations have been developed in the past [17, 11, 19, 8] including our own [18] for the modeling of self-adaptive systems.

In the master level course "Modeling self-adaptive systems" at the University of Augsburg in the winter term 2010/2011, we took the challenge to give an overview about model-driven approaches for the development of self-adaptive systems. The lectures contain material about all parts of the three layer architecture. So, the students did learn about components and architectures, the specification of behavior for the components, specification of reconfiguration actions with graph transformations, automated planning for the computation of reconfiguration plans, and requirements for self-adaptive systems. We did focus on the aspect of modeling reconfigurations using graph transformations. We did employ GROOVE and Fujaba as tools. The course was attended by nine students as the master program at the University of Augsburg as well as the Organic Computing chair is rather new.

Sections 2 and 3 contain a presentation of the topics for the lectures and the practical assignments, respectively. We conclude with a summary of lessons learned and an outlook on future work in Section 4.

2. LECTURE TOPICS

There is currently no standard approach for the development of self-adaptive systems even less a standard text book which can be used in lectures because this research area is new and addresses a wide variety of problems. Additionally, we intended to provide the students with lecture topics that they can also use in non self-adaptive systems.

Therefore, we decided to build the lecture on the following three themes being conscious of the fact that these

themes only cover a small part of the current research on self-adaption: (1) model-driven development, (2) architectural/structural approaches to self-adaption, and (3) practical experience with modeling tools.

The lecture was divided into the following content blocks:

Introduction In the introduction, we reviewed several examples of technical systems which include a heavy amount of software as e.g. current airplanes. We finally did take a look at the autonomous vehicles of the RailCab-project at the University of Paderborn. The software of these autonomous vehicles exhibits many characteristics of self-x systems like self-healing, self-optimizing, self-adaption. We employed the RailCab-project as a running example for the lectures.

Definitions This content block begins with an introduction to modeling and model-driven development. The main part of this lecture deals with the different terms which are used in the research community like self-adaption, organic computing. As there are currently no standard definitions for this term, several different definitions were given and discussed with the students.

Architecture One of the main themes of the lectures is self-adaption by architectural reconfiguration. Therefore, we reviewed in this content block definitions of architecture, configuration and components as well as architecture description languages. Our focus was laid on the architectural patterns for self-adaptive systems. This includes open and closed control loops as well as patterns as the aforementioned three layer architecture and the MAPE-K architecture [13].

Graph transformations We focus in the lecture on the structural adaptation as one kind of self-adaptation. We introduced graph transformations as basic formalism for the specification of single structural changes. As specific graph transformation formalism we introduced GROOVE [14], Story Diagrams [3] and Component Story Diagrams [18]. The last one is a variant of Story Diagrams which specifically targets architectural reconfiguration in self-adaptive systems.

Automated Planning According to the three layer architecture, plans are an ordered set of single actions which fulfill a certain goal. Automated planning (cf. [7]) is the discipline which targets the computation of such plans. The Planning Domain Definition Language (PDDL) [4] is a textual modeling language for planning problems. In this content block of the lecture we introduced the PDDL as well as different planning algorithms. Graph transformations can be easily mapped to actions in the PDDL with the exception of node creation and deletion. Therefore, automated planning approaches can be integrated with graph transformations.

Automata / Statecharts In this content block, we introduced automata, timed automata and Statecharts and their varying semantics for the model-driven development of the state-based behavior of components in self-adaptive systems.

Requirement Languages Finally, we did take a short look on requirements languages for self-adaptive systems which focus on the inherent uncertainty of self-adaptive systems. Specifically, we did take a look at RELAX [20].

The lecture can be extended by content blocks about quantitative analysis for self-adaptive systems like stochastic petri nets or probabilistic automata for availability and reliability analysis.

3. PRACTICAL ASSIGNMENTS

The lecture topics are accompanied by practical assignments for groups of two students. We decided to use a tool-centered approach for the practical assignments as we believe that students are better motivated when using tools and trying to get their solutions working than doing pen and paper assignments – if the tools work.

After a presentation of the running example, we describe those practical assignments which deal with graph transformations. We keep to simple examples due to space constraints.

3.1 Running Example

We use a self-healing distributed system as a running example throughout the practical assignments. The example is inspired from distributed embedded systems as e.g. in automotive systems. Currently, neither self-adaption nor self-healing is employed in current automotive systems but envisioned in current research approaches [1, 12, 9].

The distributed system consists of a hardware layer and a software layer. The hardware layer is built of nodes, called electronic control unit (ECU), and busses which connect an arbitrary number of nodes. The software layer consists of a set of component types which need to be instantiated on the ECUs in order for the system to be operating. Each component type may be available in different variants. Component types additionally specify required connections between their instances. The corresponding connections of the component instances are called links.

Finally, component variants and connections require certain types of resources (e.g. RAM) from an ECU or a bus, respectively. EUCs and busses provide those resources. We refrained from using real hardware and kept the scenario to simple simulations. This allowed us to concentrate on the modeling part and not to mess with the additional complexity of embedded hardware and software.

We employed other examples as e.g. the famous elevator and ferryman examples in the practical assignments as well. They are much simpler than the self-healing scenario and therefore provide an easier introduction to the different modeling formalism and corresponding tools.

3.2 Modeling the structure

The first assignment consisted of modeling the structure of the self-healing system. Figure 1 shows the resulting class diagram.

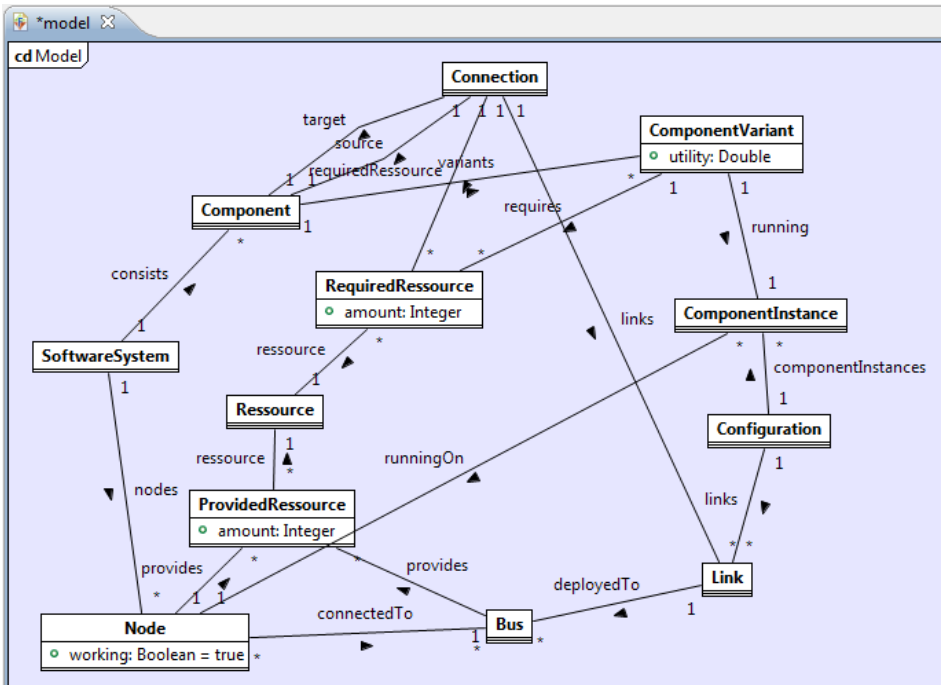

Figure 1: Class diagram of the self-healing example.

A second part of the structural modeling deals with a simulation environment. The simulation environment is event-driven, i.e. it contains a sorted set of events. The events are sorted with respect to the time the events should be executed. Examples of events are failures of component instance, busses and nodes as well as self-healing actions.

3.3 Modeling self-healing actions

Self-healing actions in the running examples are starting and stopping of component instances, repairing of nodes and busses as well as connection the component instances. The practical assignments were initially restricted to stopping and starting of component instances.

We initially used GROOVE as the tool for the specification and execution of self-healing actions as it enables an easy and fast way to introduce graph transformations to new people. We switched to Fujaba after the students were familiar with graph transformation in order to employ its code generation facilities and tight integration with Java.

Figure 2 shows a story diagram which models the creation of a new instance of a component. The story pattern selects a running node and instantiates an arbitrary component variant. Additionally, a failure event with a trigger time is already created and added to the event queue.

In addition to the self-healing actions, the simulation environment contains sensors which periodically measure system properties like the system's availability and output them to files for post-processing (e.g. plotting).

3.4 Modeling a planning problem

The three layer architecture groups single actions into plans which are executed in order to reach goals. In the context of our self-healing system, goals could be that every component is instantiated somewhere in the system and each component instance is correctly connected.

```
(:action startComponent
  :parameters (?n -Node ?inst - ComponentInstance)
  :vars (?variant - ComponentVariant ?c - Component)
  :condition (and
    (not (instanceWorking ?inst))
    (variants ?c ?variant)
    (instances ?variant ?inst)
    (working ?n)
  )
  :effect (and
    (instanceWorking ?inst)
    (running ?inst ?variant)
    (runningOn ?n ?inst)
  )
)
```

Algorithm 1: action startComponentInstance

We employed the planning software SGPlan in order to compute repair plans for such self-healing systems. Algorithm 1 shows the specification of an action which instantiates a non-working component. This action nicely resembles the story diagram of Figure 2 without the event handling. The event handling is not a part of the planning actions as they are a part of the simulation environment.

Note that the PDDL does not support the creation and deletion of objects. Therefore, we had to simulate that by the predicate instanceWorking.

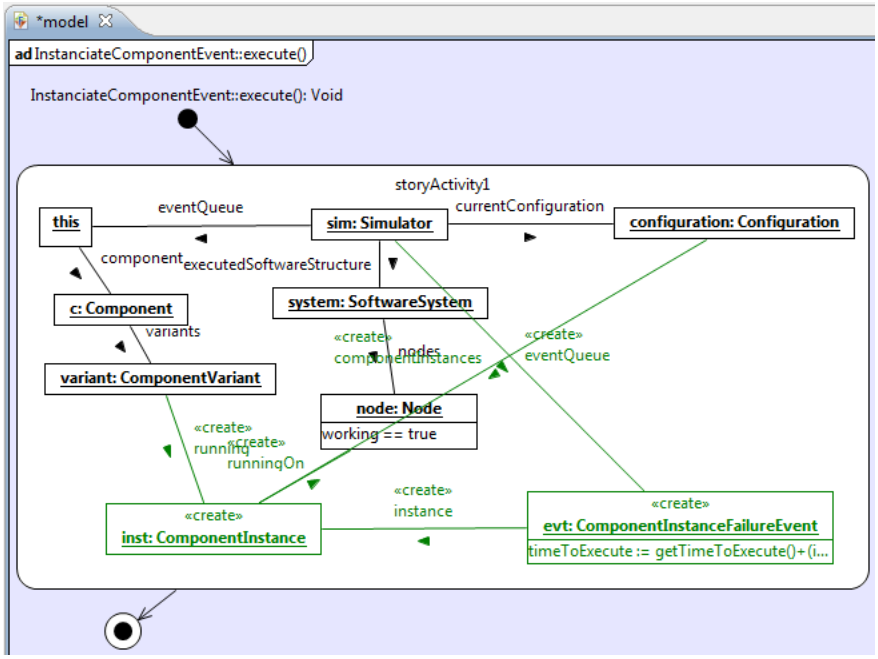

Figure 2: Event for instantiation of a component.

The following is a repair plan which is returned by SGPlan which not only instantiates component CI1 but also moves other component instances to other nodes (e.g. CI2 from node E2 to node E4) in order to free up resources for the instantiation of CI1 on node E2.

```
0.001: (UNCONNECTCOMPONENTINSTANCESBUS CI2 CI3 B2) [1]
1.002: (STOPCOMPONENT E2 CI2) [1]
2.003: (STARTCOMPONENT E2 CI1) [1]
3.004: (STOPCOMPONENT E4 CI3) [1]
4.005: (STARTCOMPONENT E3 CI3) [1]
5.006: (STARTCOMPONENT E3 CI2) [1]
6.007: (CONNECTCOMPONENTINSTANCESVIABUS CI1 CI2 B1) [1]
7.008: (CONNECTCOMPONENTINSTANCESVIAECU CI2 CI3) [1]
```

3.5 Integration with automated planners

Finally, the Fujaba models are integrated with the plans which are returned by the SGPlan. For this, the Fujaba models had to be closely aligned to the PDDL specification. Then, after execution of the planner the resulting plan can be executed by calling the story diagrams with the correct arguments.

Due to time reasons, we did not consider the self-healing scenario in these practical assignments., but only the simpler ferryman problem.

4. CONCLUSIONS

We presented a course on modeling self-adaptive systems with a focus on graph transformations in this paper. The course is built around the three layer architecture for self-adaptive systems and introduces techniques for several parts of this architecture. We did focus on graph transformations as formalism for the specification of actions. The course was accompanied by practical assignments which heavily employed existing tools like Fujaba.

4.1 Lessons Learned

We and the students learned a lot about self-adaptive systems as well as the pro and cons of requiring working with concrete software tools. Overall the practical work with tools increases the motivation of the students as they do not only write text and draw boxes and lines on paper but will test, refine, and improve their solutions until they work. This is difficult using only pen and paper. However, the employed tools have to be more polished than the typical research prototype.

Concerning Fujaba, we decided to give the students an Augsburg version of Fujaba4Eclipse which we also used to develop our sample solutions. According to our knowledge, in Paderborn, Kassel, Tartu and elsewhere different individual versions of Fujaba4Eclipse are given to the students, too. We did make an Ubuntu virtual machine available which contained this Fujaba version as well as GROOVE and SGPlan.

In our opinion, it would be beneficial to join forces and develop and maintain a single version of Fujaba for Education similar to the old Fujaba Life [15] which would be available at a central location.

This Fujaba for Education needs better, central and up-to-date documentation than we have today. We provided screencasts to demonstrate the standard activities like modeling of class and story diagrams as well as code generation and the integration with eDOBS similar to screencasts in Kassel and Tartu. A central location for those screencasts in the documentation (which must align with a single educational version of Fujaba) may also help.

Concerning Fujaba, the students were impressed with the idea of graphical modeling of graph transformations and subsequent code generation that can be actually used in normal

Java programs

Besides the question for better documentation, the students did ask for automatic or better layouting. Another topic was the support of design level debugging [6] in our version of Fujaba which would greatly help in testing and debugging the modeled specification. Finally, a better support for error messages especially during code generation was on the students' wish list. The error messages (e.g. by the sequencer) should be more specific about the error's location or even feed errors back as annotations or markups in the diagram.

We currently work on an automatic generation of PDDL specifications from Fujaba models similar to [2]. PDDL actions more or less resemble story patterns but are more powerful since universal and existential quantifiers are supported in single actions. This was heavily used by the students when implementing the PDDL actions. For a better integration, we lobby for the inclusion of universal and existential quantifiers in the new common SDM-Ecore model as proposed by Stallmann [16] which is similar to the GROOVE syntax.

5. ACKNOWLEDGMENTS

Matthias Tichy is currently on leave from the Software Engineering Group at the University of Paderborn.

6. REFERENCES

[1] B. Becker, H. Giese, S. Neumann, M. Schenck, and A. Treffer. Model-based extension of autosar for architectural online reconfiguration. In S. Ghosh, editor, *MoDELS Workshops*, volume 6002 of *Lecture Notes in Computer Science*, pages 83–97. Springer, 2009.

[2] S. Edelkamp and A. Rensink. Graph transformation and ai planning. In S. Edelkamp and J. Frank, editors, *Knowledge Engineering Competition (ICKEPS)*, Canberra, Australia, September 2007. Australian National University.

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

[4] M. Fox and D. Long. Pddl2.1: An extension to pddl for expressing temporal planning domains. *J. Artif. Intell. Res. (JAIR)*, 20:61–124, 2003.

[5] E. Gat. Three-layer architectures. *Artificial Intelligence and Mobile Robots*, 1997.

[6] L. Geiger and A. Zündorf. Design level debugging with fujaba. In *International Workshop on Graph-Based Tools (GraBaTs), Barcelona, Spain*, 2002.

[7] M. Ghallab, D. S. Nau, and P. Traverso. *Automated Planning: Theory and Practice*. Morgan Kaufmann, May 2004.

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

[9] B. Klöpper, J. Meyer, M. Tichy, and S. Honiden. Planning with utilities and state trajectories constraints for self-healing in automotive systems. In *Proc. of the Fourth IEEE International Conference on Self-Adaptive and Self-Organizing Systems Budapest, Hungary, September 27-October 1, 2010*, 2010.

[10] J. Kramer and J. Magee. Self-managed systems: an architectural challenge. In *2007 Future of Software Engineering (May 23 - 25, 2007). International Conference on Software Engineering*, pages 259–268. IEEE Computer Society, Washington, DC, USA, 2007.

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

[12] F. Nafz, H. Seebach, J. Holtmann, J. Meyer, M. Tichy, W. Reif, and W. Schäfer. Designing self-healing in automotive systems. In *Proc. of the 7th International Conference on Autonomic and Trusted Computing (ATC 2010), Xi'an, China, 26-29 October, 2010*. Springer Verlag, 2010.

[13] M. Parashar, editor. *Autonomic computing: concepts, infrastructure, and applications*. CRC Press, 2007.

[14] A. Rensink. The GROOVE simulator: A tool for state space generation. In J. Pfalz, M. Nagl, and B. Böhlen, editors, *Applications of Graph Transformations with Industrial Relevance (AGTIVE)*, volume 3062 of *Lecture Notes in Computer Science*, pages 479–485. Springer Verlag, 2004.

[15] C. Schulte, J. Magenheim, J. Niere, and W. Schäfer. Thinking in objects and their collaboration: Introducing object-oriented technology. *Computer Science Education*, 13(4), December 2003.

[16] F. Stallmann. *A Model-Driven Approach to Multi-Agent System Design*. PhD thesis, University of Paderborn, Germany, 2009.

[17] G. Taentzer, M. Goedicke, and T. Meyer. Dynamic change management by distributed graph transformation: Towards configurable distributed systems. In *TAGT'98: Selected papers from the 6th International Workshop on Theory and Application of Graph Transformations*, pages 179–193, London, UK, 2000. Springer-Verlag.

[18] M. Tichy, S. Henkler, J. Holtmann, and S. Oberthür. Component story diagrams: A transformation language for component structures in mechatronic systems. In *Postproc. of the 4th Workshop on Object-oriented Modeling of Embedded Real-Time Systems (OMER 4), Paderborn, Germany*. HNI Verlagsschriftenreihe, 2008.

[19] M. Wermelinger and J. L. Fiadeiro. A graph transformation approach to software architecture reconfiguration. *Sci. Comput. Program.*, 44(2):133–155, 2002.

[20] J. Whittle, P. Sawyer, N. Bencomo, B. H. C. Cheng, and J.-M. Bruel. Relax: Incorporating uncertainty into the specification of self-adaptive systems. In *RE 2009, 17th IEEE International Requirements Engineering Conference, Atlanta, Georgia, USA, August 31 - September 4, 2009*, pages 79–88. IEEE Computer Society, 2009.

Using Fujaba in Systems Modeling A Teaching Experience Report

Artjom Lind, Ulrich Norbisrath
Institute of Computer Science, University of Tartu
J. Liivi 2, Tartu, Estonia
{FirstName.LastName}@ut.ee

Ruben Jubeh
University of Kassel
Kassel, Germany
ruben.jubeh@uni-kassel.de

ABSTRACT

In this paper, we will describe our experience in using Fujaba in the Systems Modeling courses series at the University of Tartu. The main focus is on the three aspects of modeling Class Diagrams, Story Diagrams, and Story Boards in Fujaba. We will outline how we taught the access to these skills and how the students mastered them. We taught this course with the *strictly objects-first* approach, first in theory, than practically with Fujaba. We will show that Fujaba supports *strictly objects-first* development conceptionally very good, but is not accepted as a tool to help in doing the shift of perspective by the students. Especially advanced students deny the use of story driven modeling for their own development projects. One major hurdle are here several serious usability bugs in Fujaba4Eclipse, which we will point out. These make the reasoning for the teacher actually employing Fujaba in their class even harder. We complement this report with a couple of small tips and hints, which might be useful for other teachers employing Fujaba in their classes or teaching *strictly objects-first* and Story Driven Modeling.

1. INTRODUCTION

We have now taught systems modeling for four times at the computer science department at the University of Tartu. Three times, we employed Fujaba for showing *Story Driven Modeling* (SDM) in this course. In the last version (fall term 2010), we used Fujaba in more than two thirds of the course. We covered the following parts: SDM with Fujaba, design patterns, state machines, and petri nets. A specially customized version of Fujaba4Eclipse from the University of Kassel was used as practical support for the main part of this course (excluding the state machines and petri net part). This is the first time at the University of Tartu that Fujaba was used for the major part of a compulsory course. In the two courses before, Ruben Jubeh from University of Kassel was teaching an intense week on story driven modelling and design patterns as part of the course. We will report here about the new course design and our experiences teaching the course. The Fujaba related part of the course was taught by one lecturer and one teaching assistant. Our students are mainly master students, being often employed by a software development company. This means that most of the students have significant software development experience, probably resulting in some of the problems later explained.

It was a special requirement for this course to teach SDM and strictly objects-first [1, 3]. As we have taught the systems modelling course already several times we discovered that the students had major problems giving concrete examples and formulating scenarios. This was our main motivator to teach story driven modelling and strictly objects-first in such intensity this time. As Fujaba is still the only program supporting SDM and generating fully executable code out of models, it was for us a natural choice to use it here.

To give the course a concrete background, we provided three cases: (1) The Study Right University, (2) The Card Game MauMau, (3) The board game Mancala. The Study Right University is a slightly more politically correct derivation of the Rettet Ada case described in [2]. It was used in the lectures to explain and motivate the strictly objects-first approach. MauMau is a very simple card game, which can be easily extended with lots of rules making it a good candidate for using changing requirements. Several always extended versions of MauMau were used for several homework tasks. Mancala is a very old strategy game, played with two persons a couple of pits and stones or marbles. In a course project, the students were supposed to develop a multiplayer version of Mancala using SDM and the strictly objects-first approach. In another iteration the course project was to be refactored to reflect some major design patterns. These cases all have in common that they have a very low complexity and easily can be implemented by a small group of students in the time frame of a 6 credit point course. Also the number of attributes which have to be used in the involved objects are very limited so that most of the system behavior can be expressed with changing object associations.

2. GENERAL COURSE INFORMATION

In this section we describe the general course information as well as our expectations and preparations we made before the course started. Here we explain the most important decisions and changes we made based on our previous experience in teaching Fujaba.

	Lectures	Rate
Total	8	100%
Average	7.23	90.5%
Maximal	8	100%
Minimal	1	12.5%

Table 1: Lecture attendance

Lectures Attended	Attendance Rate	Students**	Students Rate	Avg. Points*	Grade
8	100%	40	63.5%	87.6	B
7	87.5%	13	20.6%	82.5	B
6	75%	4	6.3%	71.8	C
<= 5	<= 62.5%	6	9.5%	59.8	E

Table 2: Students performance in relation to attendance rate
*** including the possible bonus points**
**** students count in respect to attendance rate**

The course had 63 listeners in total and was scheduled once a week for 4 academic hours (45 minutes) with one short brake in between. There were no separate lab sessions for this course. We decided to give 30-50 minutes practical tasks in the lectures right after explaining new material. Therefore, the students would be able to apply the acquired knowledge (by working on their own laptops), immediately verify the solutions, and get feedback. This approach proved its efficiency: the students who were regularly attending lectures had less trouble solving home tasks and less questions while doing the course project. Table 2 and 3 illustrate the student grades in respect to attendance and attendance in respect to their grades. Giving immediate verification and feedback for each single student in personal is not feasible in a 63 persons course. Therefore we, grouped the students into 15 teams (3 to 5 students per team, most teams consisted of 4 students).

There were 8 lectures in total where we covered the strictly objects-first approach and continuously increased the level of abstraction. From the previous courses we knew that most of the students at this point are familiar with class-based programming. This means they have no difficulties in thinking abstract, but are usually stuck with thinking in

Avg. Lectures Attended	Avg. Attendance Rate	Students**	Students Rate	Avg. Points*	Grade
7.9	98.4%	23	36.5%	97.9	A
7.3	91.7%	18	28.6%	84.9	B
6.6	82.5%	10	15.9%	75.6	C
6.4	80.2%	12	19.0%	57.0	<= D

Table 3: Attendance rate in respect to final grades
*** including the possible bonus points**
**** students count in respect to grade**

concrete terms. We budgeted additional lectures explaining more object-based thinking and solving practical tasks on "concrete" and "abstract" examples. The way we set the focus in our previous courses the students only remember Fujaba as a class diagram modeling tool. The strength in supporting SDM and the possibility to easily generate Tests were usually quickly forgotten. However, these are the concepts setting Fujaba apart from third party tools. Therefore, we increased the focus on these features this time. The previous Fujaba courses were much more focused on design patterns. Therefore, the acquired knowledge was class diagram related. This time we designed the lectures to cover all the aspects of Fujaba in respect to SDM. The corresponding task flow was the following:

(1) Explore the task and provide user scenarios
(2) Break scenarios into user stories
(3) Identify objects and associations, draw object diagrams
(4) Generalize objects to classes, draw class diagrams
(5) Generalize user stories to use cases
(6) Identify methods from the actions in the user stories
(7) Prepare test cases using user stories
(8) Implement methods, test the methods with pre-made test cases

Each of the previous tasks was first of all explained as theory in lectures, followed by direct practical training in small exercises in the same lectures. For the homework, there was a set of exercises similar to those covered in the lectures, but with a changed context. Finally, we gave a larger scale course project exercise to examine the student's understanding and applicability of the Story Driven Development process. We demonstrated Fujaba solving the demo tasks in the class and trained the students to use it in the practical sessions and homework. From the previous experience we knew that students had much trouble using Fujaba. We tried to figure out the core of the complaints by not taking into consideration the technical issues. It turned out that most of the incorrect usage originated from breaking the rules of the strictly objects-first approach. For the final project we were not that strict about the tools the students used due to lots of complaints about usability problems in Fujaba. However we were strict about the application of the strictly objects-first approach. With the final course project we wanted to answer these secondary questions: (1) How many students find it easier to follow object-first approach using Fujaba? (2) What aspects of objects-first approach were mostly implemented in Fujaba? (3) How many students are able to follow the object-first approach using alternative tools? (4) What aspects of objects-first approach were mostly implemented using alternative tools?

3. STUDENT PERFORMANCE

In the previous section we described the changes we made in order to make the subject more understandable to the students and improve the overall teaching experience. In this section, we describe how the students mastered the subject.

From the very beginning, it turned out to be extremely hard

Figure 1: The LEGO Object Game showing an initial situation from the Study Right University

for most of the students to get a grip on the objects-first approach and especially the reason why to use Object Diagrams. Giving examples and being concrete seems to be an extremely challenging task for our computer science students. Therefore, we spent another lecture explaining the change of perspective and the way how the objects are actually handled in a system like for example the Java Virtual Machine (JVM). The object game turned out to be good help to explain not only the change of perspective but also the concept of links between the objects, attributes, referencing and de-referencing objects. We built our object game out of LEGO bricks, see Figure 1.

Before proceeding to application design in Fujaba, several lectures were spent explaining and trying the Object-first method, which included understanding User-Scenarios, User-Stories, and Object-Diagrams. Corresponding to the *strictly objects first*-methodology, the students design the tests before actually proceeding with the implementation. First, we gave tasks to write user stories and draw corresponding object diagrams. Initially, the students claimed these tasks to be senseless. However later, we associated object-diagrams to class-diagrams by letting students identify classes on the object-diagrams. At this point we made the students use Fujaba to design the class diagrams. Most of the students mastered in spite of usability complaints the Fujaba class diagram editor in a few practical sessions. The most often complaints were:

> - Problems with or invisibility of the Fujaba panel. Also the fact that some actions are best triggered via the panel and not via the menu.
> - Resizing the elements brakes the layout.
> - Automatic layout makes it nearly impossible to read the diagram.
> - Very unpredictable way to handle bends.

After introducing Fujaba Story Boards and explaining the concepts of Story Driven Development the students were able to associate user stories and object diagrams to Fujaba. At this point the students were able to design the test cases (based on User Stories and Object Diagrams) using Fujaba Story Boards. However, it turned out that only few of them used the Story Boards correctly. The preconditions and postconditions were present as well as the call of the action method, but there was a lot of confusion about the "create", "delete", and "optional" modifiers of associations and objects. In spite of multiple explanation, it was unclear that there is no need to recreate all the objects and associations from the preconditions block also in the postconditions block, only changed associations and objects have to be mentioned. Also confusing for the students was the fact that they had to give the create modifier to an object, which is a singleton or initializes an attribute, but not to others.

There is near to no documentation for working with Fujaba available – neither online nor offline. The information available is often erroneous or outdated and only applies to very specific Fujaba versions. Most insightful are lectures from Albert Zündorf[1] which are screencasts of his lectures, but they are in German and not easy to navigate in terms of specific problems. While teaching at University of Tartu (UT), Ruben Jubeh was adapting this technique in courses taught at UT to topic/task specific screencasts. We extended his method with a screencast version, where a teaching assistant is actually performing modeling in Fujaba and the lecturer moves freely in the class while explaining and interacting with the students in the class and the teaching assistant. The screencasts created this way the students labeled as the most helpful screencasts. They were a very popular solution for students to use documentation and they quickly started demanding more. We created the following screencasts during the course:

> Adding user libraries in eclipse; Use user libraries in eclipse; Singleton in plain Java; Singleton in Fujaba; Activity diagram modeling in lecture for Study Right with Assignments; Getting started with activity diagrams; Becoming faster: The Alternate Editing Mode; Changing the memory of the Fujaba virtual machine; Run tests and eDOBS; Storyboards, Junit tests, and more build path fixes; Setting special parameters in eclipse to collapse the Fujaba menu and have some more memory in the Java stack; Creating a new Fujaba project; Simple Hanoi (story diagrams) 4x; Distributed Modeling; Visitorpattern (and delegating file); Catching an exception

In order to verify how the students mastered the presented material they were given a homework each second week. Initially we wanted to give them homework every week, but such a workload cannot be handled with one lecturer and one teaching assistant. The home tasks were following the increasingly-more-abstract style applied in the lectures. Before letting the students actually use Fujaba, they had to master the concepts of Scenario, User-Story, Object-Diagram, and Class-Diagram – especially how to identify objects, associations, classes and inheritance, and how to define operations on the objects on paper or in simple drawing tools like *dia* or *UMLet*. Detailed feedback was given for these exercises. We often used the MauMau game case for the homework exercises. It was selected because it allowed

[1] `http://seblog.cs.uni-kassel.de/category/pastterms/ss10-pastterms/pm/`

students to consider only really needed aspects of the game following the strictly object-first method. Though the tasks were very simple and concrete, several students regarded these tasks as too easy and rather started to write about abstract concepts like general GUI, network layer, or system failure requirements instead of looking at a specific part of the game logic. This again proves, how much our students are dependent on the traditional classes-first method.

Regarding the homework we made the following observations. Only a few teams completed the tasks 100% correctly in Fujaba, regardless the usability issues. They explored bugs in Fujaba and tried to workaround bugs individually without the help from our side. They successfully dealt with strongly growing crt files, memory issues, and GUI problems. These were the students, who already in advance had proven to be among our strongest students. The following mail exchange illustrates how students helped out with finding a workaround to a bug:

> **student**: *Fujaba slowdown As our .ctr file grows and grows (around 200M now), the time it takes to load Fujaba increases. Currently it can take almost 10 min to load. Other actions such as code generation seem to be slowing as well. Is there anyway to reduce the garbage (I can only assume its garbage) in the .ctr file?*
> **lecturer**: *As far as I know, there is something to compact the undo-information, I contacted my colleagues in Kassel. I hope I have something for you soon.*
> **lecturer** (a little later): *Chris, Ruben, and me have kind of solved the compactification problem. If anybody else has this problem (Fujaba file >100MB), please contact me.*

Most of the students mastered the objects-first method, however still had troubles using Fujaba. They especially complained about usability issues, illustrated in the following mail exchange:

> **student**: *What to do when Fujaba resists to save? I just got this message:*
> `An internal error occurred during: "Saving Fujaba Model File [project]".`
> `line could not be converted to change: ...`
> `Error deserializing enclosing transaction.`
> *That's quite not funny...*
> **lecturer**: *you can take the line out of the Fujaba file.*

Few students did not master the strictly objects-first method and failed to use Fujaba mainly because of this reason, but they claimed the usability as a first reason of their failure. For example the students wanted to use explicit array indexing, but the use of objects is more preferable. The following mail exchange illustrates such an attempt:

> **student**: *Is there a "Fujaba" way of extracting the index of an object in an ordered collection? Given a Collection of Item, in a collaboration statement I can do:*
> `int i = Collection.indexOf(Item);`
> *I just want to know if there is a more graphical way.*

Count*	Points	Fujaba Usage**
9	0	Not Used Fujaba
15	1	Correct design of Class diagrams and Storyboards for tests
4	2	*Intermediate*
4	3	Incompletely designed or not functional Story diagrams
0	4	*Intermediate*
29	5	Complete design and functional Story diagrams. Full automatic code generation. All tests cases are realized with Fujaba Storyboards.

Table 4: Students count in respect to the Fujaba usage Degree
*** two students missing (one from the beginning, one from the middle of the course)**
**** Every next level includes the statements from the previous one**

> **lecturer**: *Not in this explicit way - if you want to be more graphical - I would reconsider how you want to use this index - not sure where you need it. So if you select the object you find it and then can reuse it. Can't you just use the object instead of the index?*

As an interesting side-note, also some students did not master the object-first method, but succeeded with Fujaba applying the class-first method.

Conceptionally, Fujaba performed reasonably for the selected complexity of the tasks carried out. However, the constant complaints from the students about usability, lost data, and very slow performance wear strongly on the teacher's reputation. This has a negative impact on the motivation of the students as well as the teacher's. Bending under the heavy complaints of the students, we allowed in the course project to use external tools. To raise the interest in implementing with Fujaba artificially, we introduced bonus points for using Fujaba in the course project.

In the course project, only a few teams used Fujaba for 100% code generation. The main reason not to use it were the experiences from the course, especially the Fujaba bugs. Table 4 illustrates the Fujaba usage in the course project.

4. TASK-BASED ANALYSIS

In this section, we evaluate the Fujaba usage in respect for solving specific tasks. It was very obvious that the most successful task would be the designing of class diagrams. This is mainly due to the fact that the students were familiar to the classes-first method based on their previous development experience. Therefore, they could solve the class diagram tasks without actually visiting lectures and knowing the concepts of the strictly objects-first approach. However, this means that the Fujaba class diagram editor is in spite of lots of complaints intuitive enough to handle without much preparation. The complaints about the Fujaba class diagram editor were mostly about bugs and not the

user interface. Concerning functional complaints, we heard most about the complexity and amount of GUI-operations to do simple things like renaming the class or adding attributes. Most of these issues disappeared after showing the advanced editing mode. The class diagram task was also the most popular Fujaba task in the course project, even if the usage of Fujaba was optional for the course project. Even though it gave only one bonus point, most teams fulfilled this tasks. We gave the point if at least the diagram picture was present making it the cheapest one from the entire project. It turned out that actually most of the students who got only one point here did also generate a code skeleton out of the class diagram. Code generation was not needed to get another point. As the students still used it, we conclude that they actually liked the code generation feature of Fujaba.

Another often fulfilled optional task was the use of the Fujaba Story Boards to write tests. As we mentioned in the previous section, students had first to write the User Stories and next proceed to Object Diagrams and test cases. In the home exercises most of the students performed well with the Story Boards, however they complained about a lot of time spent. Therefore, we let them choose how to design the test cases in the course project – manually or with Story Boards in Fujaba. We were expecting most of the students to fall to the manual approach. In contrast, there was not much difference in the numbers for each approach. That means, half of the students found it easier or more attractive with the promised bonus point to design test cases with Fujaba Story Boards than to draw Object diagrams and to code tests manually.

The most difficult point was the usage of Fujaba Story Diagrams. Only three teams completed the tasks 100% correctly in Fujaba, getting all the bonus points. However, four teams created Story Diagrams for activities, which actually did not transform into executable code. We once honored this with a bonus point, as some of the activities were actually correct. Overall this means that at least a third of the students actually wanted to use Fujaba for modeling the behavior of their designed systems. Personal inquiry to the respective students why they did not manage to provide a running version were all answered with the fact that they did not have enough time to run the necessary tests and taking the intense look at the produced source code of their Story Diagrams.

5. CONCLUSION

Fujaba is still the only CASE tool out there supporting SDM and being able to generate fully executable code out of purely graphical models. In theory, together with the strictly objects-first approach and the use of eDOBS, a purely graphical, object oriented development process should be possible. However, we can conclude that in reality this ideal is still far away. Fujaba is way too buggy to use it in a complementary course. Some students always feel forced to use something they do not like, do not understand and therefore do not want to understand. The gap between traditional software development, especially the processes being practiced in the industry, and the Fujaba development approach, is way too big. Several basic concepts clashed in this course: textual versus graphical, Object-orientation versus procedural programming, strictly objects-first versus static type systems. Furthermore, it is difficult to change someone's perspective who already has in-depth experiences of classic concepts to some very basic approaches like the "LEGO Object Game".

Sometimes we have the feeling that the used Fujaba4Eclipse Tool is a living monster, very difficult to tame. The user interface is unintuitive and crude. Bugs are difficult to reproduce, no helpful error messages appear or cannot be found easily. There is often only a single way to achieve a certain task, and the single steps to that are secretly hidden. Debugging the generated source code is hard as the developers are confronted with some textual artifact they do not understand. Especially when you never looked "inside" Fujaba, its metamodel and technical concepts, it is hard to understand what happens under the hood and why. We suspect that you have to be a Fujaba developer to use it successfully and efficiently.

During the course preparation, the Kassel University team claimed that Fujaba is actually mature enough to do a 60+ attendants course. We have the feeling that having an experienced developer team in the background helps a lot to deal with the daily problems of the students. Active Fujaba developers know what kind of bug a student experienced even when getting a vague error description. The team can fix vital bugs instantly and do a new software rollout. Fortunately, our course was scheduled directly after Kassel's programming methodologies course finished, so we have the most recent and stable Fujaba4Eclipse version available.

For use of Fujaba in the given teaching context, it definitely has to improve usability and stability. However, always being an academic tool, this might never happen. Current development efforts concentrate on the central metamodel and story diagram interpreters. We hope that the commercial spinoff, UMLLab, will provide a stable, free to use CASE tool with an open interface, so that the promising SDM concepts still can be used in a professional context. The recently presented UMLLab-to-Fujaba-Adapter seems to be the most promising approach.

6. REFERENCES

[1] Diethelm, I. Strictly models and objects first – Unterrichtskonzept für objektorientierte Modellierung. In *Informatik und Schule – Didaktik der Informatik in Theorie und Praxis – INFOS 2007 – 12. GI-Fachtagung 19.–21. September 2007, Siegen* (Bonn, September 2007), S. Schubert, Ed., no. P 112 in GI-Edition – Lecture Notes in Informatics – Proceedings, Gesellschaft für Informatik, Köllen Druck + Verlag GmbH, pp. 45–56. `http://subs.emis.de/LNI/Proceedings/Proceedings112/gi-proc-112-004.pdf` – geprüft: 19. Juni 2009.

[2] Diethelm, I., Geiger, L., and Zündorf, A. Rettet prinzessin ada: Am leichtesten objektorientiert. In *INFOS* (2005), S. Friedrich, Ed., vol. 60 of *LNI*, GI, pp. 161–172.

[3] Diethelm, I., Geiger, L., and Zündorf, A. Teaching modeling with objects first. In *WCCE 2005, 8th World Conference on Computers in Education* (2005).

Visualization of Pattern Detection Results in Reclipse

Marie Christin Platenius, Markus von Detten, Dietrich Travkin
Software Engineering Group, Heinz Nixdorf Institute,
University of Paderborn, Paderborn, Germany
[mcp|mvdetten|travkin]@mail.uni-paderborn.de

ABSTRACT

Reverse engineering tools can simplify the recovery of a software system's design by detecting design pattern implementations. This helps to understand a software system and thereby supports the process of maintaining or extending a software. Because the manual specification of patterns has to maintain the balance between precision and generality, detection results may contain incorrectly detected pattern implementations. Usually, a detected candidate cannot be displayed in detail so that interpreting the detection results is difficult. In this paper, we present an approach for a comprehensive and comprehensible visualization of detection results in the reverse engineering tool suite Reclipse.

1. INTRODUCTION

Reverse engineering is the task of analyzing a software system in order to understand its design. A helpful part in the recovery of the design is the detection of design patterns. Design patterns represent general, reusable, and commonly accepted solutions to frequently occurring problems in object-oriented software design [4]. Knowledge about the presence of pattern implementations in a software helps to understand the software system by revealing the original developers' design intentions. It thereby supports the process of maintaining or extending a software. Reverse engineering tools can automatically detect pattern implementations and thereby simplify the reverse engineering process. In the last years, the detection of design pattern implementations in source code has been the subject of many scientific publications (Dong et al. give an overview [3]).

To automate the detection, a formal specification of the patterns is needed. However, patterns can be implemented in several ways and there can be many different variants of a pattern which are difficult to capture in a single formal specification. This leads to the fact that the detection results, the so-called *pattern candidates*, can contain false positives. This problem can be mitigated to a certain degree, by specifying a mandatory pattern core which has to be present and several additional conditions whose detection increases the confidence in the correctness of a detected pattern implementation. As a consequence, the detection results must be inspected and for each candidate it has to be decided if it is a true or false positive.

The static pattern detection of the reverse engineering tool suite Reclipse[1] [7, 8] detects pattern implementations in source code. The pattern detection results are currently displayed as a simple list of pattern candidates in which the involved classes are listed. An automatically calculated percental rating value indicates how much a pattern candidate conforms to its specification. A low rating value means that the candidate does only contain the mandatory core and few or none of the specified additional conditions. Therefore, this could indicate that the candidate is a false positive.

As patterns are specified by the user, the specifications can contain (possibly subtle) flaws. Such an incorrect specification can result in erroneous detection results. Examples are false positives, or misleading rating values that are too high or too low. Currently, the user cannot distinguish these cases by looking at the list of rated detection results and she cannot see how the rating is calculated. Thus, more information about the detected candidates is needed to make use of the results.

In this paper, we present how the detection results can be visualized in a comprehensive and comprehensible way. The goal of the visualization is to provide a more detailed image of the detected pattern candidates to the user and to make the candidates' rating more transparent.

The remainder of this paper is organized as follows: First, we give a general overview of the pattern detection process, the pattern specification and the current presentation of results in Reclipse. In Section 3, requirements for a visualization of detection results are proposed and in Section 4 our visualization approach is presented. Section 5 deals with related work. We finish with conclusions and ideas for future work.

2. PATTERN DETECTION IN RECLIPSE

The static pattern detection in Reclipse uses a graph matching approach: The system, i.e. the source code under analysis, is represented by an abstract syntax graph (ASG) in which an inference algorithm detects subgraphs that comply to the structure of pre-specified graph patterns [5]. This results in a list of so-called *pattern candidates*. The pattern specifications consist of a number of conditions which have to be satisfied for a successful match. A percental rating value is computed for each candidate. The rating value determines the ratio of a candidate's satisfied conditions to all conditions in the corresponding pattern specification. Thereby, the rating quantifies the completeness of a candidate and, thus, indicates if the candidate is a real pattern implementation or a false positive.

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

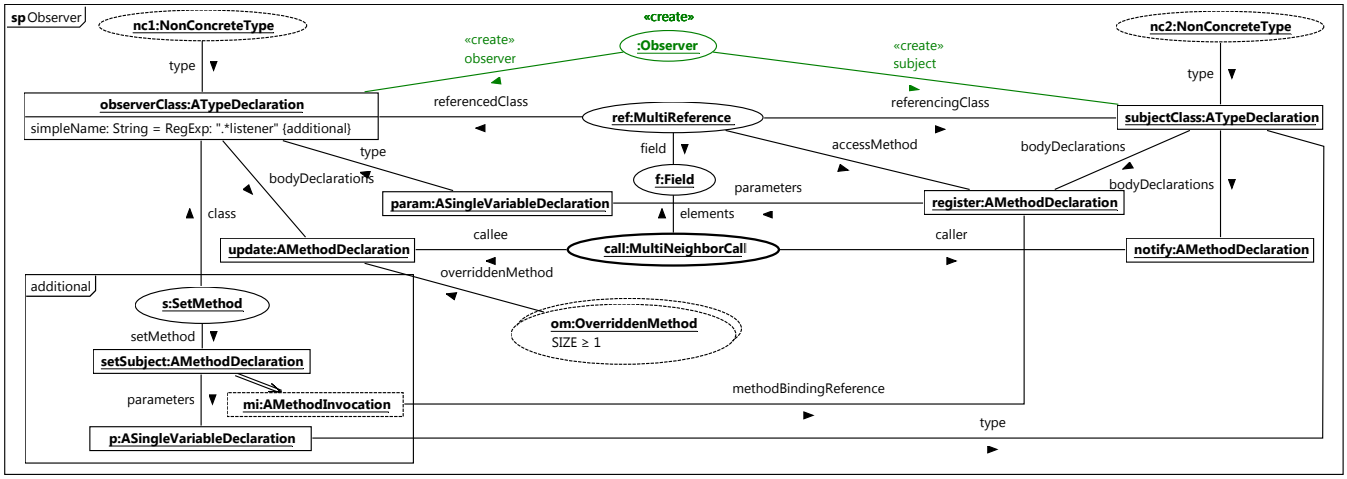

Figure 1: *Observer* structural pattern

In the following, the pattern specification and the presentation of the pattern detection results are explained in detail.

2.1 Pattern Specification

In Reclipse, a pattern's structure is specified with a pattern specification language based on graph grammars, the so-called *structural patterns*. Throughout this paper, we use the *Observer* pattern as an example. The Observer pattern's intent is to "define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically" [4]. Figure 1 depicts our specification of the Observer pattern. The rectangular objects represent elements of the system's abstract syntax graph, e.g. classes and methods. The objects are variables that are matched to real objects in the given ASG during pattern detection. The ellipses are so-called *Annotations* and represent subpatterns that are specified in other diagrams. Each element is a condition of the pattern specification.

When the depicted structure is detected in an ASG during the patten detection process, the `Observer` annotation that is marked with `create` is created. It tags the structure as candidate for the Observer pattern and marks objects that play key roles in the pattern (here the observer class and the subject class).

The Observer pattern's structure contains the classes `subjectClass` and `observerClass`. The observer class has a method `update`. The subject class has the methods `register` and `notify`. The `register` method takes an object of the type `observerClass` as parameter. The subpattern `MultiReference` expresses that a subject references arbitrarily many observers. The subpattern `MultiNeighborCall` specifies that the subject's `notify` method contains multiple calls of the observer's `update` method.

The dashed lines (e.g. of the `NonConcreteType` annotations) indicate objects that are not mandatory for the detection of the Observer pattern. They form additional conditions. In the same way, the subgraph in the rectangle to the lower left marked with `additional` (a so-called *additional fragment*) is not mandatory for the detection of an Observer pattern implementation. Detected additional elements increase the number of satisfied conditions and thus the candidate's rating value. The `observerClass` element includes an additional condition on its `simpleName` attribute that defines a condition for the name of the type bound to the observer role. The name has to match the specified regular expression which, in this case, declares that the string should end with "listener".

The `OverriddenMethod` annotation `om` is drawn with a second border and thereby marks the node to represent a set of objects in the ASG. This means that an arbitrarily large number of objects can be mapped to this element. The expression "SIZE $\geq$ 1" indicates that there has to be at least one element in this set.

More details on the specification language used in Reclipse can be found in other publications [5, 7, 10].

2.2 Pattern Detection Results View

Figure 2 shows the current results view of Reclipse. It presents an excerpt of the detection results of a static pattern detection on JHotDraw 5.1 [10]. Besides others, we found some candidates for the Observer pattern. The view lists one candidate with its annotated elements (i.e. `observerClass` and `subjectClass`) and additionally shows the rating value and the detected subpatterns with their ratings. In this example, Reclipse detected an Observer pattern candidate with a class named `StandardDrawing` that plays the role of the subject class and a class `DrawingChangeListener` that represents the observer class. All subpatterns (`Field`, `InterfaceType`, `MultiNeighborCall`, `MultiReference` and `OverriddenMethod`) were detected with a rating value of one hundred percent. However, the Observer candidate as a whole only received a rating of 79.31%. That means that some of the conditions in the pattern specification are not satisfied. Unfortunately, the user is not able to see where exactly the candidate deviates from the specification, i.e. which conditions of the corresponding pattern are not satisfied. Furthermore, it is not shown which other objects besides observer and subject class were matched. For

Annotation	Rating	Annotated Elements
Observer	79,31%	observer=CH.ifa.draw.framework.DrawingChangeListener, subject=CH.ifa.draw.standard.StandardDrawing
detected subpatterns ...		
Field	100,00%	fragment=fListeners, type=CH.ifa.draw.standard.Vector, declaration=no Name, owningClass=CH.ifa.draw.standard.StandardDrawing
InterfaceType	100,00%	type=CH.ifa.draw.framework.DrawingChangeListener
MultiNeighborCall	100,00%	calleeClass=CH.ifa.draw.framework.DrawingChangeListener, callee=drawingInvalidated, caller=figureInvalidated, callerClass=CH.ifa.draw.standard.StandardDrawing, elements=Field
MultiReference	100,00%	referencedClass=CH.ifa.draw.framework.DrawingChangeListener, referencingClass=CH.ifa.draw.standard.StandardDrawing, field=Field, accessMethod=addDrawingChangeListener
OverriddenMethod	100,00%	subClass=CH.ifa.draw.standard.StandardDrawingView, overriddenMethod=drawingInvalidated, overridingMethod=drawingInvalidated, superClass=CH.ifa.draw.framework.DrawingChangeListener

Figure 2: Detection results from a static pattern detection of JHotDraw 5.1

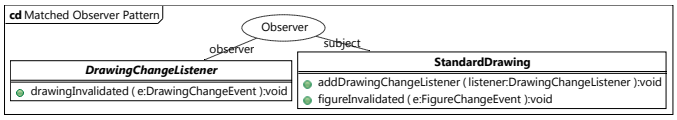

Figure 3: The class view of the candidate

example, the user cannot see which concrete methods were matched for the register, the notify, or the update method roles from the specification.

Essentially, the user has no overview of the (un-)satisfied conditions. As a result, it is not comprehensible why the actual rating values are as they are and the detection results can barely be interpreted.

3. VISUALIZATION REQUIREMENTS

Visualization is important for pattern detection tools because it helps to envision the detection results so that the user can easily understand the system [3]. Backofen identified several requirements for an adequate visualization of detection results of a static pattern detection [1]:

R1 To attain clarity and comprehensibility, a compromise between a detailed visualization and a compact, well-arranged view has to be found.

R2 The presentation of the detection results should show which conditions of a pattern specification are satisfied and which are not.

R3 To provide a better understanding of the pattern detection results, it should be easy to relate the matched pattern candidate to the pattern's specification.

R4 All specification elements that were matched to the pattern candidate should be visualized accordingly to provide the user with detailed information about a candidate.

R5 The concrete values of an object's attributes should be presented to inform the user about the concrete reason why a condition is (not) satisfied.

4. PATTERN MATCHING VIEWS

As an addition to the current presentation of detection results as a list of candidates, we developed a graphical visualization, the *pattern matching views*. In the following the new visualization approach is presented.

The Reclipse tool suite, which is based on Fujaba, is a collection of plug-ins for Eclipse. The pattern matching views were also realized in an Eclipse plug-in. There are three different views: The *class view*, the *pattern view* and the *abstract syntax view*. The views can be displayed for each detected pattern candidate and satisfy the requirements identified in Section 3.

In the following, the three views are described in detail. As an example, the Observer candidate from Figure 2 is used.

4.1 Class View

In Reclipse, we mostly deal with patterns at the design level which are primarily concerned with classes. Accordingly, their natural syntax is a class diagram. Because of this, the *class view* shows the pattern candidate in a UML class diagram. Class diagrams are a language that most users are familiar with, so they can see immediately which classes play the key roles in the candidate. In addition, this illustration is very compact and thereby provides a convenient overview to the user (cf. requirement R1).

Figure 3 shows the class view for the Observer candidate. The `DrawingChangeListener` and the `StandardDrawing` classes from the example in Figure 2 are presented with their roles in the Observer pattern.

4.2 Pattern View

The pattern view shows the pattern specification of a pattern candidate, enhanced by information about which conditions are satisfied by the selected candidate, and which are not (cf. requirement R2). Satisfied conditions of the pattern are shown in black. Conditions that are not satisfied are marked as `unsatisfied` and are visualized in gray.

In Figure 4, the pattern view for the Observer candidate is shown. In this example, the objects in the additional fragment on the lower left are conditions that are not satisfied. That means, the candidate has no set method in the Observer class that takes an object of the subject class as parameter and calls the subject's register method. Also, the attribute expression for the name of the observer class is not satisfied: the class' name does not end with "listener". The `NonConcreteType` annotation on the right side is not matched either, which means that the type which represents the subject is neither abstract nor an interface.

4.3 Abstract Syntax View

The abstract syntax view shows the subgraph of the ASG that was matched for the candidate. The advantage is that this is similar to the syntax of the pattern specification, which means that the user is able to easily compare the candidate to the specification (cf. requirement R3).

In Figure 5, the Observer candidate is visualized in the abstract syntax view. Here, all matched objects are presented (cf. requirement R4). For example, the observer class

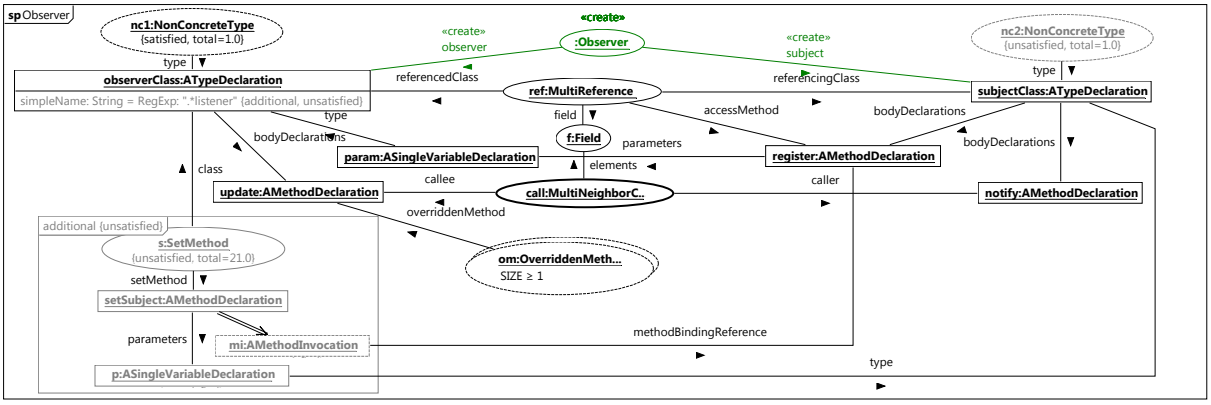

Figure 4: The pattern view of the candidate

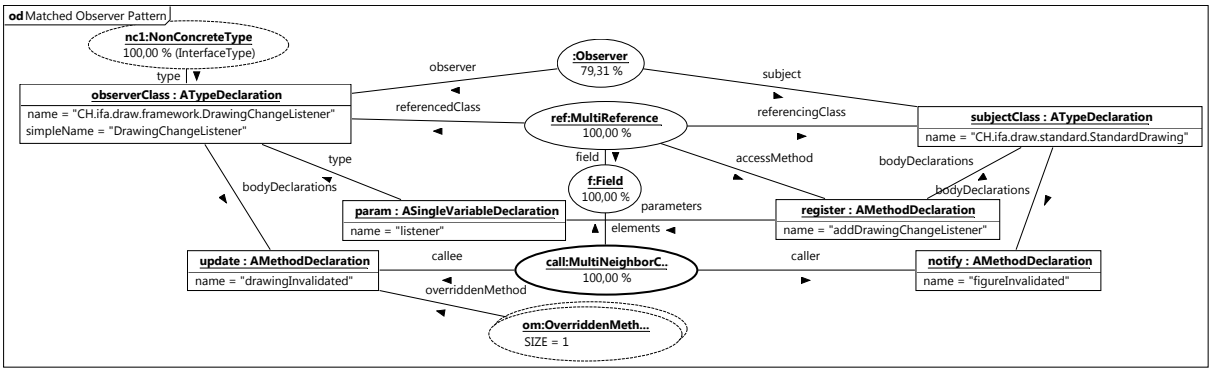

Figure 5: Abstract syntax view of the candidate

is named `DrawingChangeListener` and the subject class is named `StandardDrawing`. Also the names of all other matched objects are visualized (cf. requirement R5). The annotations show that the matched subpatterns are all rated with one hundred percent. Additionally, the size of the `Overridden-Method` annotation is presented. In this example, the update method is overridden only once as indicated by the expressions `SIZE = 1`.

Figure 4 shows that the attribute expression of the observer class' name is not satisfied. The reason for this is revealed in the abstract syntax view: The name of the class is `Draw-ingChangeListener`. This does not match the regular expression ".*listener" from Figure 4. This hints at a flaw in the specification. The expression could be corrected to ".*(l|L)istener" to improve this condition.

4.4 Additional Features

To support the user in comparing the pattern candidate and the specification, the three views provide a consistent selection. If an element in one of the views is selected, the corresponding elements in the other views are highlighted as well.

To simplify the comparison, the layout of the elements shown in the pattern view and in the abstract syntax view is based on the layout of the pattern specification (cf. requirement R3). The user is able to customize the layout of all three views by dragging the elements to new positions.

To enable a clear, well-arranged view of the pattern candidate in abstract syntax, only attributes that have a corresponding condition in the pattern specification are shown (cf. requirement R1).

If the selected pattern candidate includes annotations that represent subpatterns, the user is able to directly open the matching views for the subpattern out of the currently opened views. For example, from the visualized Observer candidate in the abstract syntax view, the user can jump to the detected candidate of the MultiReference pattern to see details about the relation between the observer class and the subject class.

Furthermore, if the pattern specification contains sets of objects or annotations, the user can expand the contained elements for inspection by selecting an action from the context menu. In the pattern view for the Observer candidate, for example, the user can display all methods that are bound to the OverriddenMethod annotation, i.e. all methods that override the observer's update method.

5. RELATED WORK

There are many approaches which deal with the detection of patterns. In their survey paper, Dong et al. present several pattern detection approaches that also provide visualization support [3]. Most of those tools present their results as UML class diagrams, in which the pattern roles are marked. One of the approaches proposes a UML profile containing new stereotypes, tagged values and conditions and thereby ex-

tends UML diagrams for visualizing pattern-related issues [2]. Another visualization technique proposed by Dong et al. is a class hierarchy in addition to class diagrams. There, the first level nodes under the root are the classes participating in the pattern while the roles that a class plays are defined as their children [3].

Wiebe et al. use a pattern detection approach similar to the analysis Reclipse uses [12]. After executing a graph matching algorithm, the detected pattern candidates are evaluated and presented. However, the candidates are visualized exclusively as UML class diagram.

Schauer and Keller present an approach where the pattern candidate is juxtaposed with the description from literature [6]. But the informal description is not equivalent to the used formal pattern specification. Thus this approach is not sufficient because it does not provide appropriate information about the discrepancies between pattern specification and candidate.

In summary, none of these pattern detection tools satisfies all of the requirements described in Section 3.

6. CONCLUSIONS AND FUTURE WORK

With the pattern matching views, Reclipse provides a visualization of pattern candidates that illustrates the detected candidates in a comprehensive and comprehensible way. Our visualization results in a more transparent rating and thereby supports the user by simplifying the decision if a candidate is a false positive or a real pattern implementation. Furthermore, the user now can compare pattern candidates to the specification. In our Observer example, we received a more detailed view of the classes in JHotDraw which are responsible for updating a drawing because the matching views displayed the methods that play important roles in this mechanism. Furthermore, we were able to correct our Observer specification, because we noticed the flawed attribute condition for the observer class name.

However, the visualization approach still provides space for enhancements. For instance, only attributes that have a corresponding condition in the pattern specification are shown, which is useful, but in some cases not sufficient. The user should be given the additional possibility to view the values of attributes which are not involved in the pattern specification to get a more detailed view of the candidate. An additional idea for the visualization of a candidate is a source code view. Another interesting feature could be the comparison between several candidates of the same pattern.

Moreover, the detection results are non-persistent at the moment. The ability to save the results would allow the user to review them later and to compare different results from multiple analysis runs. This would further support the flaw detection in pattern specifications.

Furthermore, Reclipse also provides a dynamic analysis that analyzes a pattern candidate's runtime behavior. The dynamic pattern detection can be used to reject or verify pattern candidates from the static analysis based on their behavior [9, 11]. The results of the dynamic pattern detection could be used to enhance the pattern matching views by additional information. Thereby the user could gain an even more comprehensive illustration of the detected design pattern implementations in the analyzed software.

7. ACKNOWLEDGMENTS

We would like to thank Andre Backofen for his conceptual work [1] on the approach and his help in implementing the pattern matching views in Reclipse.

8. REFERENCES

[1] A. Backofen. Visualisierung von Musterfunden bei der statischen Software-Muster-Erkennung. Bachelor's thesis, University of Paderborn, Nov. 2009.

[2] J. Dong, S. Yang, and K. Zhang. Visualizing design patterns in their applications and compositions. *IEEE Transactions on Software Engineering*, pages 433–453, 2007.

[3] J. Dong, Y. Zhao, and T. Peng. A Review of Design Pattern Mining Techniques. *International Journal of Software Engineering and Knowledge Engineering (IJSEKE)*, 2009.

[4] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, USA, 1995.

[5] J. Niere, W. Schäfer, J. P. Wadsack, L. Wendehals, and J. Welsh. Towards pattern-based design recovery. In *Proc. of the* 24^{th} *International Conference on Software Engineering (ICSE), Orlando, FL, USA*, pages 338–348. ACM Press, May 2002.

[6] R. Schauer and R. Keller. Pattern visualization for Software Comprehension. In *Proceedings of the 6th International Workshop on Program Comprehension*, pages 4–12. IEEE, 2002.

[7] M. von Detten, M. Meyer, and D. Travkin. Reclipse – a reverse engineering tool suite. Technical Report tr-ri-10-312, University of Paderborn, Paderborn, Germany, 2010.

[8] M. von Detten, M. Meyer, and D. Travkin. Reverse Engineering with the Reclipse Tool Suite. In *Proceedings of the 32nd International Conference on Software Engineering (ICSE 2010), Cape Town, South Africa, May 2-8, 2010*, volume 2, pages 299–300. ACM Press, May 2010. Informal Research Demonstration.

[9] M. von Detten and M. C. Platenius. Improving Dynamic Design Pattern Detection in Reclipse with Set Objects. In *Proceedings of the 7th International Fujaba Days*, pages 15–19. Eindhoven University of Technology, 2009.

[10] M. von Detten and D. Travkin. An Evaluation of the Reclipse Tool Suite based on the Static Analysis of JHotDraw. Technical Report tr-ri-10-322, Universitiy of Paderborn, 2010. Vers. 1.0.

[11] L. Wendehals. *Struktur- und verhaltensbasierte Entwurfsmustererkennung*. PhD thesis, University of Paderborn, September 2007. In German.

[12] E. Wiebe, S. Keul, S. Staiger, and G. Vogel. Entwurfsmuster-erkennung mit bauhaus. In *Proceedings of the 10th Workshop Software Reengineering*, volume 126 of *LNI*, pages 181–185. GI, 2008.

Providing Timing Computations for FUJABA*

Tobias Eckardt, Christian Heinzemann
Software Engineering Group,
Heinz Nixdorf Institute
University of Paderborn
Warburger Str. 100
D-33098 Paderborn, Germany
tobie|c.heinzemann@uni-paderborn.de

ABSTRACT

Model-based software engineering aims at specifying the system under construction by abstract models that can be used for formal verification of the system behavior. In the case of real-time systems, such verification requires special algorithms dealing with time computations. These computations can be performed efficiently by using zone graphs [1, 3]. Current implementations, however, cannot be used in FUJABA. Therefore, we introduce a TCP/IP-based client/server architecture wrapping an existing implementation in a server such that it can be used by arbitrary clients. In our evaluation, we show that the TCP/IP overhead is negligible compared to the total run-time.

1. INTRODUCTION

Model-based software engineering aims at specifying the system under construction by abstract models that can be used for formal verification of the system behavior. This approach can also be used in the domain of real-time systems in order to build safe real-time systems by using appropriate models and verification techniques addressing the real-time characteristics [8]. The MECHATRONIC UML approach is one technique for model-based development of real-time systems [9].

A suitable formalism to model the behavior of real-time systems is given by timed automata [1] that have been extended to real-time statecharts [7] in the MECHATRONIC UML. Timed automata have successfully been used in Uppaal as a formal model for the verification of real-time behaviors [3]. Uppaal, however, cannot be used for all analysis techniques being applied in MECHATRONIC UML like refinement checking [10] or a behavioral synthesis [6]. Thus, such algorithms have to be implemented separately which requires the implementation of time computations in the case of real-time systems.

Time computations, as they are needed for our analysis techniques, can be efficiently performed by using so-called *zone graphs* [1] that are also used in Uppaal [3]. For Uppaal, there exists a C++ library, the Uppaal DBM library [4], implementing the necessary functionality for computing zone graphs. Additionally, a Ruby binding of this library exists. Both implementations have in common that they cannot be used in FUJABA directly.

We try to overcome this problem by providing a TCP/IP-based client/server architecture that allows to use the existing Uppaal DBM library by clients being implemented in arbitrary programming languages supporting TCP/IP. On the one hand, our architecture consists of a server, written in Ruby, that directly uses the ruby binding of the Uppaal DBM library. On the other hand, we provide a reference Java interface and a TCP/IP-based implementation of this interface managing the communication with the server. That interface can be used directly in FUJABA to implement the timing computations needed for our analysis techniques.

An alternative to our TCP/IP based client/server architecture would be, obviously, to write specific adapters to the Uppaal DBM library for each programming language and compile it for each operating system. In case of Java, a JNI (Java Native Interface) binding to the C++ library would be possible. Probably, such binding would be more efficient than our approach, but it restricts the usage of the library to one specific language and requires to re-compile the library for all required operating systems. The latter was simply not possible in our case due to missing third-party libraries.

The contribution of this paper is a TCP/IP-based client/server architecture providing efficient clock zone computations to all programming languages supporting TCP/IP.

The paper is structured as follows. First, we introduce the foundations of the paper (Section 2). Afterwards, the general architecture of our approach is described in Section 3. Then, we discuss the client and the server in detail in Sections 4 and 5, respectively. Finally, we present our evaluation results concerning the TCP/IP overhead in Section 6 before we conclude the paper in Section 7.

2. FOUNDATIONS

For the illustration of the possibilities using clock zones and clock zone operations we employ a timed automaton as a behavioral model with timing constraints as it can be specified in Uppaal (Figure 1).

Informally, a timed automaton consists of finite sets of *locations*, *transitions* and real-valued *clocks*. Starting in the *initial location*, it may either rest in a location or switch between locations using transitions and corresponding event occurrences. *Events* are modeled using a synchronous channel concept, where events can either be thrown using the special symbol "!" or received using the special symbol "?".

The example automaton in Figure 1 describes the behavior of a simple lightswitch. By pressing the switch once the light is switched to dim; by pressing the switch twice within 10 ms the light is switched to bright. If pressing the switch a second time does not happen within 10 ms, the light is switched of again. If the light is currently switched to bright, it can also be switched off

*This work was developed in the course of the Collaborative Research Center 614 – Self-optimizing Concepts and Structures in Mechanical Engineering – University of Paderborn, and was published on its behalf and funded by the Deutsche Forschungsgemeinschaft.
This work was developed in the project "ENTIME: Entwurfstechnik Intelligente Mechatronik" (Design Methods for Intelligent Systems). The project ENTIME is funded by the state of North Rhine-Westphalia (NRW), Germany and the EUROPEAN UNION, European Regional Development Fund, "Investing in your future".

by the press operation. If this is not performed, it switches to off automatically inbetween 59.5 s and 60 s.

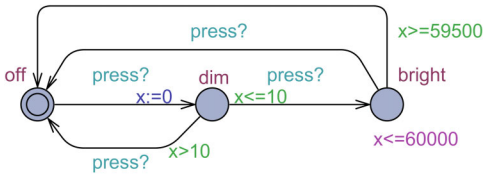

Figure 1: Example of a Timed Automaton describing a lightswitch

The timing of the behavior is specified using *time guards*, *clock resets* and *location invariants*. Initially, all clocks' values are set to zero. From then on, time can only pass, i.e. all clocks' values increase by the same value, while the automaton rests in a location, not while a transition is executed. Clocks can be reset using clock resets and the execution of a transition can be constrained to an integer-bound interval of clock values using clock constraints.

In the example, the clock x is used to measure the time that the system rests in dim. The time guard x<=10 at transition dim to bright further specifies, that this transition can only be executed if the value of x is between 0 and 10. If it is greater and the press-signal occurs, the transition from dim to off will be executed corresponding to its time guard x>10.

Finally, *location invariants* may be used to describe progress conditions. A location invariant describes an upper bound for the clock values in a certain location. In the example, the location invariant of location bright (in combination with the empty transition from bright to off) is used to specify that the light switches to off again automatically after 1 minute (60000 ms) at the latest, if the switch was not pressed before.

2.1 Analysis of Timing Specifications

While the untimed behavior of a state based specification can simply be performed by examining the states and transitions between states, this becomes more complex for timed specifications. Here, clock values, clock resets and clock constraints have to be taken into account. A suitable formalism for analyzing sets of clock values is the clock zone formalism [2, 1, 3] that is briefly described in the following.

A clock zone is syntactically described by a boolean conjunctive formula where the atomic propositions are inequialities with clock references and integer values describing clock value lower and upper bounds. Semantically, it describes an infinite, integer-bound set of clock values that can be visualized as a convex set in a k-dimensional euclidean space for k clocks being contained in the zone [1]. An example of a clock zone describing all values of clock x between 0 and 10 and all values of clock y higher than 20 is $x < 10 \wedge y > 20$.

If a clock zone is combined with a system state, for example a timed automaton location, this clearly defines a distinguishable timed state of the system, where the timing part is respresented as a set of clock values. For the calculation of transitions between states, whose execution is restricted to a distinct time interval only, for the consideration that time may elapse in some states and for the case that clocks may be resetted, operations on clock zones are provided. Four of these operations, which are the most important ones, are explained in the following.

The *time elapse* operation ($^{\uparrow}$), also called up-operation, describes the elapse of an arbitrary amount of time for a clock zone. It is realized by removing all upper bounds of a clock zone. The time elapse operation applied, for example, on the above given zone, denoted $(x < 10 \wedge y > 20)^{\uparrow}$ results in $y > 20$.

The *clock reset* operation describes the appliance of clock resets, that is setting the value of a set of clocks to zero. Applying this operation on the clock y and the example zone, denoted $(x < 10 \wedge y > 20)[\{y\} := 0]$, results in the zone $(x < 10 \wedge y = 0)$.

The *intersection* of clock zones ($\wedge$), also called and-operation, describes the set of clock values that are in both of the intersected zones. Intersecting the example zone with the zone $x > 5$, denoted $(x > 5) \wedge (x < 10 \wedge y > 20)$ results in $(x > 5 \wedge x < 10 \wedge y > 20)$.

The *subtraction* operation on clock zones subtracts one clock zone from another. This means that the subtrahend set of clock values is removed from the minuend set of clock values. An example is $(x < 10 \wedge y > 20) - (x <= 5)$ which results in $(x > 5 \wedge x < 10 \wedge y > 20)$.

In case of substractions on zones, the convexity of the set of clock values is no longer guaranteed. In this case, a time interval can be removed from the zone. The result is a non-convex set of clock values, called a *federation* [3], that can be represented by a finite number of convex sets (zones).

To give an example for the application of clock zones and corresponding operations, we show how a timed analysis model of the example timed automaton (Figure 1) can be created in the following. This timed analysis model (Figure 2), also called the *zone automaton* or *zone graph* [1], can, for example, be used to perform a reachability analysis over the timed system states.

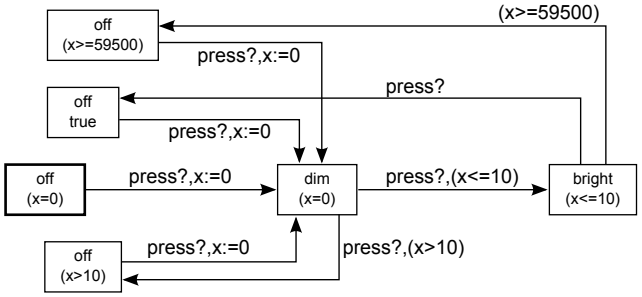

Figure 2: Zone Automaton of the Timed Automaton of the Lightswitch Example According to [1]

The zone automaton is created by starting in the first location, and the zone where all clocks are set to zero, in this case $(x = 0)$. For each outgoing transition, a successor zone location is now created by (1) applying the time elapse operation on the original zone, (2) applying an intersection with the location invariant of the source location, (3) applying an intersection with the time guard of the transition, (4) applying the clock resets of the transition and, finally, (5) applying an intersection with the location invariant of the target location. The resulting clock zone describes those clock values that are possible at the moment where the next target location is entered. In the example, the transition from (off,x=0) with the clock reset x:=0 leads to (dim,x=0) as the clock x must be zero when entering dim. On the other hand the transition from (dim,x=0) with the time guard x<=10 leads to bright with the zone x<=10 as the exact value of clock x is not known when entering bright, only that it is somewhere between 0 an 10.

After computing a successor zone in a zone automaton, a so-called normalization can be applied [3]. The normalization computes a canonical form of the zone and guarantees that the corresponding zone automaton of a timed automaton is always finite.

Other application examples, appart from model checking timed automata, are checking a refinement of timed automata as described in [6] or applying a reachability analysis on timed graph transformation systems as described in [11].

3. GENERAL ARCHITECTURE

The Uppaal DBM[1] library (UDBM, [4]) is a C++ library that was originally designed for the Uppaal model checker [3]. It implements operations on clock zones and federations (cf. Section 2.1) using DBMs [5] as an internal data structure for efficient memory management. We integrated this library into FUJABA using a client/server approach. The general architecture is shown in Figure 3.

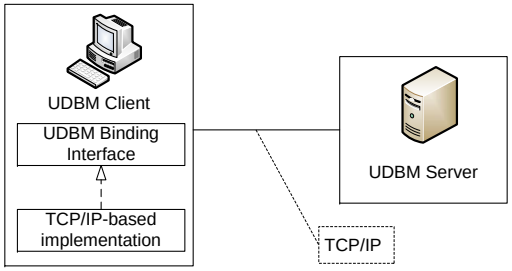

Figure 3: Architecture of the UDBM Integration

The UDBM Server executes the DBM operations using the Uppaal DBM library implementation. The UDBM client consist of an abstract UDBM Binding interface which can be used by application programs and a TCP/IP-based implementation of the interface managing the communication with the server in order to execute the operations requested by the application programs. Detailed information on our server and client implementation can be found in the subsequent Sections 4 and 5.

The client/server architecture allows to implement more than one client (even in different languages) for the same server as well as implementing more than one realization of the DBM computations without changing the client interface. Additionally, our architecture allows to execute client and server on different machines using different operating systems.

4. UDBM SERVER

The UDBM server is implemented in Ruby[2] and uses the precompiled Ruby binding of the Uppaal DBM library. The server manages the communication with the client and delegates DBM operation requests to the UDBM library. By default, the server opens a socket on port 8326 on localhost for client communication, but it is possible to pass a different port and hostname to the server on start up as a parameter.

The server implements the statemachine shown in Figure 4 that specifies the protocol to interact with it. The events before the "/" have to be passed as strings to the server, the events after the "/" are sent as strings back to the client. Strings in italics denote ruby code that is passed and directly executed by the server as described below.

The server starts in state idle. First, a so-called *context* has to be created by passing the command createContextReq to the server. A context is required for the execution of the DBM operation as it defines the names of the clocks to be used. The server answers

[1] http://www.cs.aau.dk/~adavid/UDBM/
[2] http://www.ruby-lang.org

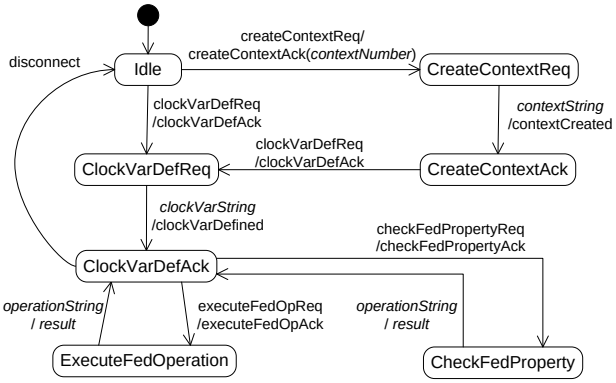

Figure 4: Statemachine of the UDBM Server

with an acknowledgement and a unique number for the next context to be created. Then, the client can submit an operation creating a context. This operation is submitted as ruby code which is then directly interpreted by the Ruby interpreter. In our example, a context for one clock x has to be created using the ruby code `c = Context.create('c0',:x)` for the context number 0.

The server answers with contextCreated to acknowledge the creation of the context before switching to CreateContextAck. The creation of a new context can be omitted if the context did not change compared to the last operation being executed. That allows to reuse contexts from prior operations thereby reducing the memory consumption of the server.

The second step is defining the clock variables. In order to use clock variables for the specification of clock constraints, the clocks being defined in the context have to be bound to variables. The client submits a clockVarDefReq to the server which switches into ClockVarDefReq. Again, the definition of the variables is encoded into ruby code submitted as a string. For our clock x, the submitted ruby code would be `x=c0.x;` As before, the server acknowledges the operation and switches to ClockVarDefAck.

The state ClockVarDefAck has two outgoing transitions representing the two possible classes of DBM operations. First, a property of a DBM can be checked. Such an operation always evaluates to either *true* or *false*. Second, an operation can be executed on a DBM such as intersection with another DBM. Such an operation always evaluates to a DBM. The client can select the desired operation by submitted either checkFedPropertyReq or executeFedOpReq to the server. Then, the actual operation to be executed has to be passed as ruby code as before and is directly interpreted. The result, either a Boolean or a DBM, is returned to the client. For instance, the operation `((x>=0)).and!(x<=10)` will evaluate to the DBM `((x>=0)&(x<=10))`.

After all operations have been executed, the client can send disconnect to the server causing it to switch to idle.

Then, a new context having a different number of clocks compared to the prior context can be created. That allows to support changing DBM sizes during the run of an algorithm on the client side.

5. JAVA UDBM CLIENT

In addition to the server, we have implemented a Java side client for the UDBM server. As introduced in Section 3, the client consists of an abstract interface for modeling DBMs as shown in Figure 5 as well as a TCP/IP-based implementation managing the communication with the server. Both are implemented as Eclipse plugins

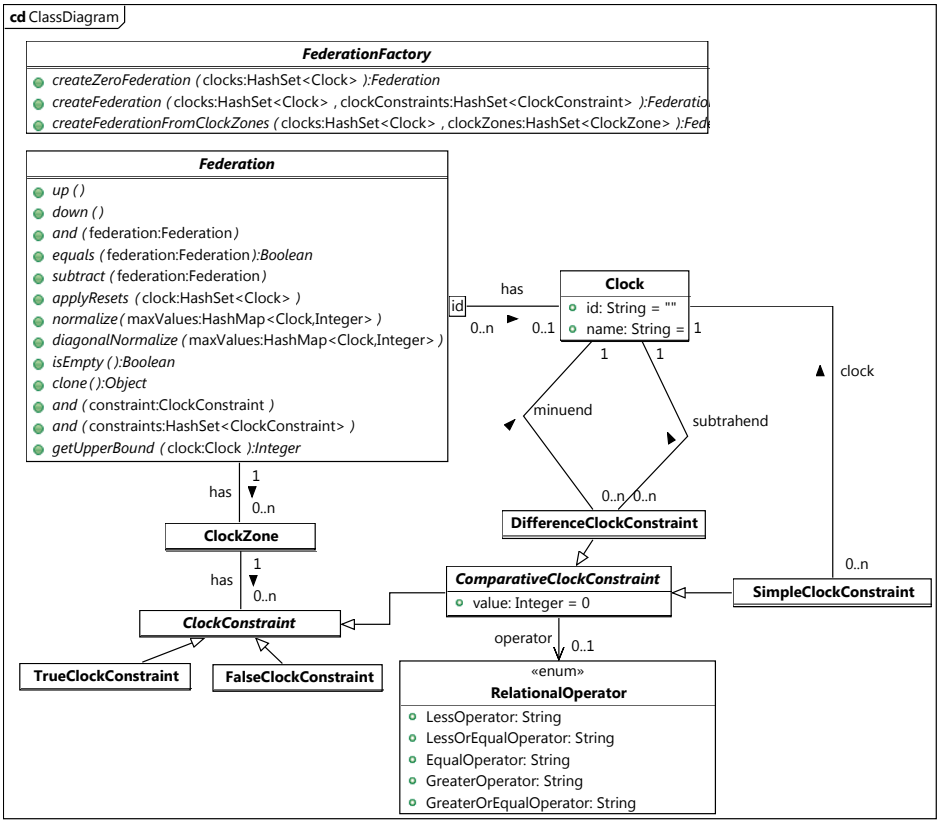

Figure 5: Class Diagram of the UDBM Java Client

and can be used in any Eclipse based tool such as FUJABA.

The bottom part of the client model allows the definition of clock constraints as defined in Section 2.1. In the two simplest cases, a clock constraint is either true or false represented by the classes TrueClockConstraint or FalseClockConstraint. In a ComparativeClockConstraint, a comparison with an integer is supported. Therefore, these clock constraints have a value and an operator. In a SimpleClockConstraint, the value of one clock is compared to the integer while in a difference clock constraint the difference of two clocks is compared to this value. In our example, $x <= 10$ of $x == 0$ are instances of a SimpleClockConstraint using the clock x. The classes can be used to model all valid clock constraints.

The left hand side of the model (ClockZone and Federation) is used to model clock zones and federations. For the sake of consistency, each zone must be contained in a federation even if there is only one zone in the federation. In our example in Figure 2, each represented zone can be represented in one zone and thus, each federation consists of one zone, only.

The class Federation also defines the interface to the operations which can be performed on a federation. The operation and, e.g., allows to intersect a federation with additional clock constraints or another federation. The executed operation is then transformed into a query to the server and the provided result and parsed back into a federation.

Clocks are assigned to federations because all zones in one federation must be specified over the same set of clocks. In order to improve memory efficiency, clocks can be used for different federations. In our example, all federations share the same clock object x.

The Java interface allows to add and remove clock instances from federations. The addition and removal of clocks can be easily done on the object level. Clocks being added to a federation are initialized with the value 0.

In some application scenarios, e.g. the reachability analysis introduced in [11], fast equivalence checks on DBMs are required. Therefore, the client interface implements a hash algorithm on federations fulfilling the general hash function contract.

$$f_1 \equiv f_2 \Rightarrow hash(f_1) = hash(f_2)$$

That means whenever the federations are equal, their hash values are equal as well. Thus, the equality check invoking the server only has to be executed in case of equal hash values.

6. EVALUATION

We evaluated the performance of our server and the TCP/IP connection using a socket via localhost utilizing the rechability analysis and the example presented in [11]. There, nine samples for run-times of a reachability analysis were presented. We choose to use this example, although it produces some odd numbers of DBM operations, because we wanted to have a realistic sample of DBM operations. During the reachability analysis, the size of the DBM varies such that multiple contexts have to be created (cf. Section 4). The results are summarized in Table 1. In the table, one DBM operation refers to the execution of one operationString in the protocol of Figure 4.

The runtime results have been obtained by first measuring the runtime on the Java side in order to obtain a runtime result including the TCP overhead. Second, we measured the runtime inside the ruby server to obtain a runtime result without the TCP overhead. Finally, the TCP overhead has been obtained by arithmetics. The

Table 1: Evaluation results

# of DBM Operations	Run-time of DBM Operations in s			Server memory in MB
	Server incl. TCP	Server excl. TCP	TCP	
108	0,3	0,3	0,0	15
342	1,0	1,0	0,0	15
737	3,4	3,1	0,3	16
1329	8,5	8,0	0,6	25
2154	17,1	16,4	0,7	25
3248	33,4	32,4	1,0	31
4647	61,1	59,1	2,0	35
6387	109,9	106,4	3,4	45
8504	190,1	185,3	4,8	63

run-time results in Table 1 are the sum of all executed DBM operations. The results show that the runtime increases slightly faster than the number of executed DBM operations. This is due to the fact that the maximum size of the DBMs increases from row to row. Thus, the additional runtime results from the fact that operations on larger DBMs consume more computation time. The overhead introduced by the TCP/IP connection to the server is approximately 3% of the overall runtime which we consider as quite low.

The memory consumption of the server includes the memory consumption of the ruby interpreter running the server script and the ruby binding of the Uppaal DBM library. Due to the reuse of contexts, the memory consumption increases only slowly for a large number of DBM operations. In cases where the DBM dimension does not change during runtime, the increase in memory consumption will be 0.

7. CONCLUSION AND FUTURE WORK

In this paper, we introduced a client/server architecture for integrating time computations into FUJABA. The Java client allows to model clock as well as constraints on these clocks that can be represented by clock zones. The server uses the Uppaal DBM library to perform the actual time computations. Our presented architecture is flexible as the server can be used by application programs written in any programming language supporting TCP/IP communication. Additionally, our client interface is independent of the actual server implementation. The overhead introduced by the TCP/IP communication is negligible according to our evaluation results.

Our implementation allows for an easy integration of time computations in any real-time analysis algorithm.

In our future work, we will try to apply further optimizations to our implementation. One of these optimizations is the support of concurrency in the server by allowing and processing multiple connections in parallel. Additionally, different server implementations could be evaluated for obtaining the most efficient realization of time computations.

8. REFERENCES

[1] R. Alur. Timed Automata. In N. Halbwachs and D. A. Peled, editors, *Proceedings of the 11th International Conference on Computer Aided Verification (CAV '99), July 6-10, 1999, Trento, Italy*, volume 1633 of *Lecture Notes in Computer Science (LNCS)*, pages 8–22. Springer Verlag, 1999.

[2] R. Alur, C. Courcoubetis, N. Halbwachs, D. L. Dill, and H. Wong-Toi. Minimization of timed transition systems. In *Proceedings of the Third International Conference on Concurrency Theory (CONCUR '92)*, Lecture Notes in Computer Science (LNCS), pages 340–354, London, UK, 1992. Springer-Verlag.

[3] J. Bengtsson and W. Yi. Timed Automata: Semantics, Algorithms and Tools. In J. Desel, W. Reisig, and G. Rozenberg, editors, *Lectures on Concurrency and Petri Nets*, volume 3098 of *Lecture Notes in Computer Science*, pages 87–124. Springer, 2003.

[4] A. David. *UPPAAL DBM Library Programmer's Reference*, Oct. 2006. http://www.cs.aau.dk/~adavid/UDBM/manual-061023.pdf.

[5] D. L. Dill. Timing assumptions and verification of finite-state concurrent systems. In J. Sifakis, editor, *Automatic Verification Methods for Finite State Systems, International Workshop, Grenoble, France, June 12-14, 1989, Proceedings*, volume 407 of *Lecture Notes in Computer Science (LNCS)*, pages 197–212. Springer Berlin / Heidelberg, Feb. 1990.

[6] T. Eckardt and S. Henkler. Component behavior synthesis for critical systems,. In *Architecting Critical Systems*, volume 6150 of *Lecture Notes in Computer Science*, pages 52–71. Springer Berlin / Heidelberg, 2010.

[7] H. Giese and S. Burmester. Real-time statechart semantics. Technical Report tr-ri-03-239, Lehrstuhl für Softwaretechnik, Universität Paderborn, Paderborn, Germany, June 2003.

[8] H. Giese, S. Henkler, M. Hirsch, V. Roubin, and M. Tichy. Modeling techniques for software-intensive systems. In D. P. F. Tiako, editor, *Designing Software-Intensive Systems: Methods and Principles*, pages 21–58. Langston University, OK, 2008.

[9] H. Giese, M. Tichy, S. Burmester, W. Schäfer, and S. Flake. Towards the compositional verification of real-time uml designs. In *Proc. of the European Software Engineering Conference (ESEC), Helsinki, Finland*, pages 38–47. ACM Press, September 2003.

[10] C. Heinzemann, S. Henkler, and A. Zündorf. Specification and refinement checking of dynamic systems. In P. V. Gorp, editor, *Proceedings of the 7th International Fujaba Days*, pages 6–10, Eindhoven University of Technology, The Netherlands, November 2009.

[11] C. Heinzemann, J. Suck, and T. Eckardt. Reachability analysis on timed graph transformation systems. In *Proceedings of the Eighth International Workshop on Graph Based Tools (GraBaTs 2010)*, volume 31 of *Electronic Communications of the EASST*, 2010.

UML Toolchain

Using Fujaba and UML Lab in a toolchain

Andreas Koch, Albert Zündorf
Kassel University, Software Engineering,
Department of Computer Science and Electrical Engineering,
Wilhelmshöher Allee 73,
34121 Kassel, Germany
[andreas.koch | zuendorf]@cs.uni-kassel.de
http://www.se.eecs.uni-kassel.de/

ABSTRACT

Every CASE-Tool has its strengths and weaknesses. Of course, Fujaba is not apart from this rule. Thus, why not combine the strengths of Fujaba with another application in a toolchain.

This paper introduces a toolchain covering Fujaba and UML Lab. Traditionally a toolchain is based on the use of an im-/export functionality to transfer data between the different tools by persisting this data with a common file format. As the requirements of the introduced toolchain cannot be fulfilled by this mechanism, a synchronization of models based on the Fujaba respectively UML Lab metamodel is implemented. One requirement is to ensure that changing anything in the model of one tool has an immediate impact on the related model in the other tool. Therefor the model synchronization handles every change separately by analyzing change event objects resulting from each model modification and ensures a (preferably) immediate handling of changes.

1. INTRODUCTION

Each software application is developed in terms of a specific purpose. In the context of this purpose the application can (or at least it should) provide any necessary functionality. However, the requirements of an user often exceed these functionalities or the user prefers a similar approach offered by another application. Either way, one application alone cannot satisfy the requirements of this user. Therefore the creation of a toolchain containing all necessary applications to complete the desired task is recommendable.

The concept of a toolchain can be interpreted differently. It can describe an unidirectional order of tools (see figure 1(a)) as well as a bidirectionally traversable chain (see figure 1(b)) or any other set of tools. Depending on the structure creating a new toolchain is affected by different challenges, but a main task always refers to the data exchange between the included tools. This problem can be separated into two connected sub tasks. The first task addresses the general way how data, created in one tool, can be transferred into the next tool (in the chain) to proceed working. The second task deals with handling subsequent changes in an inbound document. This means to define how, if at all, the modification of data in one tool can be applied to existing data in another tool.

A traditional approach to provide the data exchange is the use of an im-/export functionality: at first, all necessary data is persisted by exporting it with tool A. Afterwards tool B imports this data and the user can continue to work. This approach lacks, amongst other things, in the necessity of a (manual) intervention each time a data exchange is performed. Additionally, with a common and standardized data format, there can be difficulties persisting any tool specific data (e.g. positioning of GUI elements) or preserving this data after a future export by another tool. This lost data is usually not relevant in an unidirectional toolchain, but impedes the usage of a bidirectional traversable toolchain. There, every step back would result in the, at worst manual, restoration of all lost data.

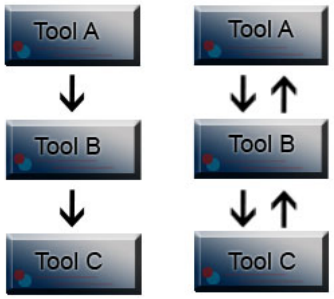

Figure 1: Structure of different toolchains

The introduced *UML Toolchain* with the CASE-Tools Fujaba, to be more precise Fujaba4eclipse, and UML Lab is targeted on a tight integration of their functionalities and bidirectional traversable. Therefore it is necessary that changes, for example on a model in Fujaba, are immediate applied to the equivalent model in UML Lab. Using an im-/export functionality does not fulfill these requirements in several points. Besides the manual interaction to im-/export any data, this approach does not rely on altering existing data, but storing and restoring (means deleting and recreating) of all available data. To address these problems an automatical synchronization of equivalent parts of the metamodels in both tools has been implemented. Additionally, this synchronization depends on the usage of event objects triggered by each modification of a model. This means: every change is handled separately.

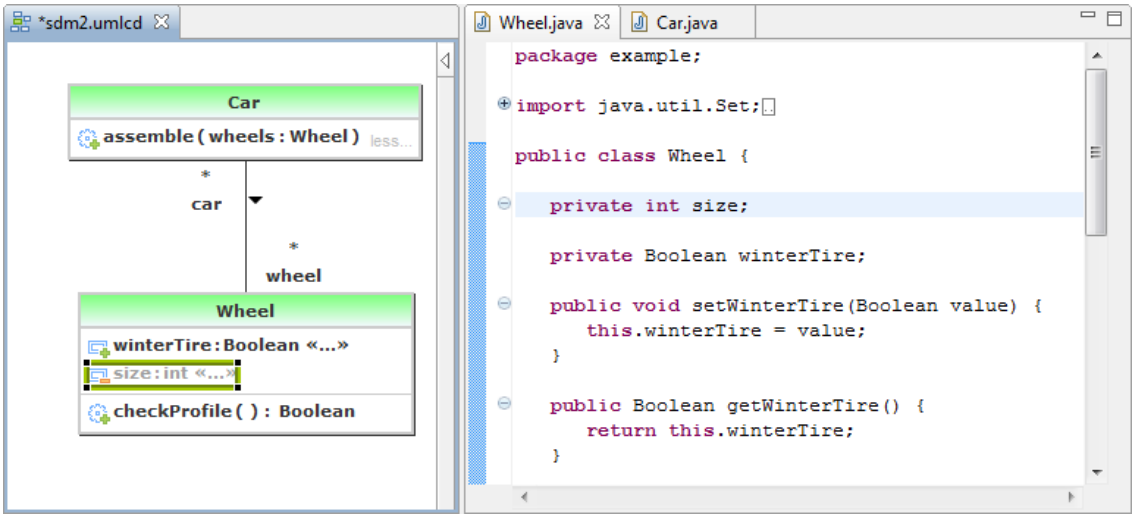

Figure 2: screenshot of UML Lab showing source code and the resulting class diagram

2. UML LAB

UML Lab is a modeling IDE developed by the Yatta Solutions GmbH[7]. As it is based on the eclipse platform [2], the implementation of the UML specification 2.x[6] for the eclipse platform is used as metamodel. The influence of Fujaba during the development of UML Lab is reflected in the effort to combine code generation and reverse engineering. Based on the concept "From UML to Java and back again" a language independent synchronization of source code and model was developed and integrated into UML Lab. This Round-Trip-Engineering^{NG}[1] uses textual templates for generation as well as reverse engineering to enable the parallel work in source code and diagram as shown in figure 2.

3. DEFINING THE TOOLCHAIN

To define the core requirements of the introduced toolchain two different groups of users have to be analyzed.

The first group consists of users, that are familiar with Fujaba. They usually work with UML Lab only, if it provides a noticeable advantage. The Round-Trip-Engineering^{NG}, accessible through the toolchain, enables the synchronization of generated source code with the Fujaba class diagram and therefore falls into this category. Additionally, as UML Lab uses the implementation of a current version of the UML metamodel, the toolchain provides access to new features of the UML specification without the necessity to change the Fujaba metamodel.

The second group of users are familiar with UML Lab. These consider using Fujaba only where UML Lab does not provide the appropriate tools. Especially the concept of *Story Driven Modeling*(SDM)[3] [8] in Fujaba has to be mentioned in this context. This includes the creation/editing of story diagrams, which can (now) be based on Fujaba and UML Lab models. An additional use case is to integrate source code for these modeled story diagrams into UML Labs generated source code. An explicit request apart from this is, that legacy projects from Fujaba shall be usable (and editable) in UML Lab.

To satisfy both groups needs, the effort to use the other tool has to be kept as low as possible. In addition, as one wants to use his tool-of-choice, a class diagram has to be editable with both tools.

4. IMPLEMENTING THE TOOLCHAIN

To discuss the details of the implementation, an example usage of the toolchain is given:

1. Reverse Engineering of existing source code (UML Lab)
2. Extend one class with an additional method (Fujaba or UML Lab)
3. Create a story diagram for this method (Fujaba)
4. Generate code (UML Lab (with Fujaba))

After one or more existing classes are reverse engineered with UML Lab (step 1), the resulting class diagram can be edited with both tools - depending on one's preferences. In step 3 switching to Fujaba is necessary to create the story diagram for a formerly created method. The last step, the code generation, needs more explanation. As both tools provide an own code generation, each tool can solely generate executable code. However, UML Lab does not generate code for story diagrams and the code generation of Fujaba is not automatically linked to the previously reverse engineered classes. Accordingly, the combined generation with both tools is recommended: first of all the source code for story diagrams is generated with Fujaba; afterwards it is passed to UML Lab and integrated into its generated source code. Therefore switching back to UML Lab is required again.

As steps like these are executed regularly and in an undefined order, one has to switch between Fujaba and UML Lab frequently. Two requirements result from this conclusion. First of all, as much as possible should be done without any explicit user interaction. This includes the automatical synchronization of model changes or the combined code generation. With the automatic model synchronization one has not to take care about the currently active tool, because both models (Fujaba and UML Lab) are always consistent with each other. Secondly, features of the other tool have to

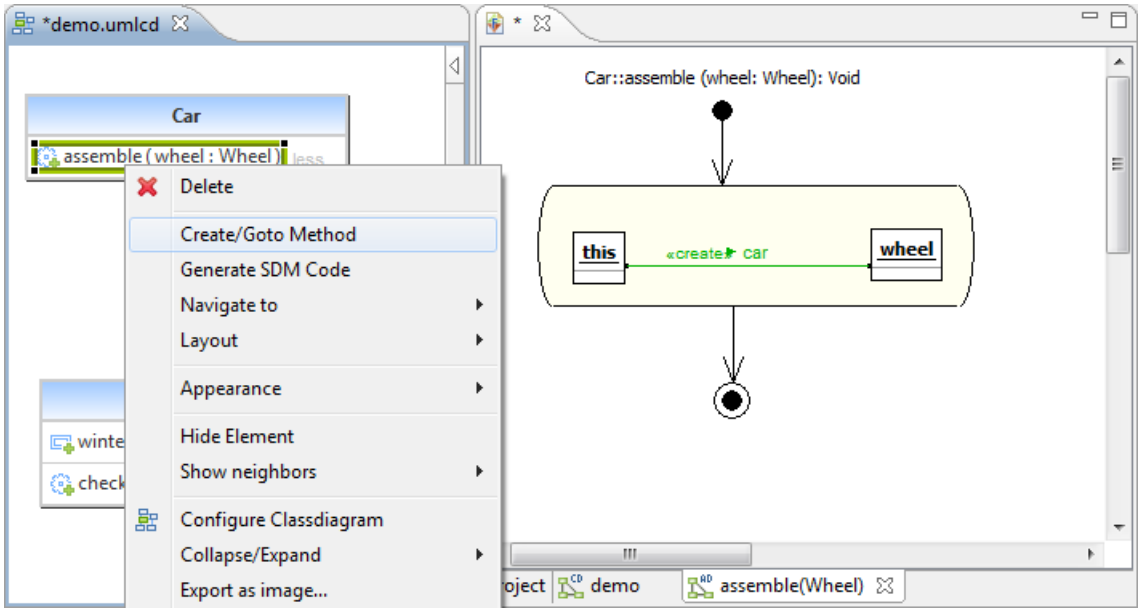

Figure 3: Usage of the story diagram editor in the UML Lab class diagram editor

be accessible directly. A simple example: to create and edit story diagrams while using the class diagram editor of UML Lab, its context menu is extended with a new menu entry to open the story diagram editor and switch to it automatically (figure 3).

All those features are provided externally and added by using plugin mechanisms; on the one hand to avoid dependencies between the tools, on the other hand because they shall not (Fujaba) or cannot (UML Lab) be modified. For this reasons an adapter is used to connect Fujaba and UML Lab. This adapter takes care of the model synchronization and the extension of the tools; for example the additional entries in context menus.

5. MODEL SYNCHRONIZATION

The model synchronization is not discussed in detail in this paper, but we will give a brief overview of its general idea and structure. For the implementation details see [5].

To perform a valid synchronization of two models their metamodels need to be examined and similar parts identified. By analyzing the metamodels of Fujaba and UML Lab one common part concerning class diagrams can be found. Accordingly the synchronization is restricted to this common elements. These elements are afterwards used to find suitable mappings; for example *FMethod* (Fujaba metamodel) can be mapped to *Operation* (UML 2.x metamodel). Based on this mappings different handlers are implemented, each responsible for one field of one metamodel element pair; for example one handler synchronizes the return type of *FMethod* with *Operation* and vice versa.

Imagine the following scenario: there are two already synchronized models in Fujaba and UML Lab that contain a method (*assemble()*) associated with a story diagram. This method is extended by a new parameter (*wheel:Wheel*) using the UML Lab class diagram editor. After switching back to its story diagram, this new parameter should be immediately added and usable. The result is shown in figure 3 with the class diagram editor on the left and the story diagram editor on the right side.

As both metamodels provide an implementation of the observer/listener pattern, these are used to synchronize every change separately; for example events are fired after the creation of the new parameter, the change of its name or adding it to a method. These events are analyzed and delegated to the responsible handlers. The only information that cannot be extracted from these events is the target object the change should be applied on. For this purpose the adapter manages all known object mappings and makes them available to the handlers. These mappings are generated every time a new object is created as well as after loading and scanning of associated models. Finally the handler combines the informations from the event with the mapped object to synchronize the change.

Avoiding inconsistent models was a main challenge during the creation of the synchronization mechanism; especially asymmetric structures were problematic. For example *Property* needs to be mapped to *FAttr* or *FRole* depending on attribute values of a *Property* object. Therefore every time a *Property* object is changed the mapped target object has to be evaluated correctly. As a result some events have to be collected and not synchronized until a consistent result can be ensured.

6. SUMMARY AND FUTURE WORK

UML Toolchain can be used to work in parallel with Fujaba and UML Lab. This means one can create/edit class and story diagrams, generate code or reverse engineer existing code without a manual switch between the tools. Therefore every model change is automatically synchronized i.e. immediately applied to the related model in the other tool.

For the future work two topics can be examined separately. The first deals with the ideas to expand the toolchain; not by integration another tool, but by extending the synchronized part of the metamodels. This includes for ex-

ample the synchronization of story diagram when they are integrated into UML Lab. Additionally, the usability can be improved in some points for example to administrate the synchronized models. This includes a temporary disconnect of models or a general overview to show all synchronized (loaded or not loaded) models.

The second issue is aimed onto the model synchronization. As the example uses a programmatically implementation, the framework provides interfaces to easily attach other mechanisms. For example the integration of the triple interactive graph grammar (TiG) interpreter[4] is planned in the near future.

7. REFERENCES

[1] M. Bork, L. Geiger, C. Schneider, and A. Zündorf. Towards roundtrip engineering - a template-based reverse engineering approach. In I. Schieferdecker and A. Hartman, editors, *ECMDA-FA*, volume 5095 of *Lecture Notes in Computer Science*, pages 33–47. Springer, 2008. http://dblp.uni-trier.de/db/conf/ecmdafa/ecmdafa2008.htmlBorkGSZ08.

[2] Eclipse platform. http://www.eclipse.org/eclipse/.

[3] T. Fischer, J. Niere, L. Torunski, and A. Zündorf. Story diagrams: A new graph rewrite language based on the unified modeling language. In *Proc. of the* 6^{th} *International Workshop on Theory and Application of Graph Transformation*. Paderborn, Germany, 1998.

[4] B. Grusie. Ein objektorientierter, interaktiver Triple Graph Grammatik Interpreter. Master's thesis, University of Kassel, Kassel, Germany, 2010.

[5] A. Koch. Echtzeit Synchronisierung von UML-Modellen unterschiedlicher technischer Basis am Beispiel von UML Lab und Fujaba. Master's thesis, University of Kassel, Kassel, Germany, 2010.

[6] Implementation of the UML 2.x metamodel for the Eclipse platform. http://wiki.eclipse.org/MDT-UML2.

[7] Yatta Solutions GmbH. http://www.uml-lab.com/en/uml-lab/.

[8] A. Zündorf. Rigorous object oriented software development. Habilitation Thesis, University of Paderborn, 2001.

Difference Visualization for Models (DVM)

Visualizing model changes directly within diagrams

Andreas Scharf, Albert Zündorf
Kassel University, Software Engineering,
Department of Computer Science and Electrical Engineering,
Wilhelmshöher Allee 73,
34121 Kassel, Germany
[andreas.scharf | zuendorf]@cs.uni-kassel.de
http://www.se.eecs.uni-kassel.de/

ABSTRACT

Today's software development and maintenance is time consuming, cost-intensive and particularly an iterative process. Since models and diagrams are the main artifact in the development process of numerous research institutes and software companies, it is necessary to show and merge differences as the model evolves. While there are plenty of difference tools available for textual artifacts (like source code) this does not hold for diagrams.

This paper presents an approach to show and merge deltas of different model versions directly within the corresponding diagram editor. This is done by integrating the *Difference Visualization for Models (DVM)* framework into existing editors with as little effort as possible.

1. INTRODUCTION

Modern software consists of several million lines of code which are changed frequently by software development teams. Producing software does not mean to simply write the code once and never touch it again. In fact it is an iterative process: Bugs have to be fixed and the team has to take care for changing requirements which increases the code size. Every change (or *delta*) can be interpreted as a new version of the source code. This in turn leads to different versions of the same document which are mostly managed with version control systems like CVS or subversion.

During the development phase of a software project there often is the need of showing deltas between different versions of the same document. In the majority of cases these documents are interpreted as flat and unstructured text - content and logical configuration are not considered. The user then gets information about added, deleted and modified lines and is able to merge both versions of the document. The tool support for visualizing such changes for unstructured text documents like source code is excellent. However for more and more developers the source code is not the primary artifact anymore.

To design structure and behavior of large software projects the Unified Modeling Language (UML) [3] is used by developers. Particularly class diagrams are utilized to describe (parts of) systems. Like source code, diagrams are subject to frequent modifications which also raises the requirement of visualizing and merging these changes. Unlike source code, diagrams are structured files. Hence the traditional algorithms and tools to compute, visualize and merge deltas can not be applied in a meaningful way.

Existing approaches for visualizing model based changes are either difficult to use or hard to implement. The EMF Compare [5] for example provides a generic tree editor which is difficult to use because the UI is completely different from your usually used editor. The approach presented in [2] on the other hand integrates such a mechanism into Fujaba. This allows you to visualize differences directly in the class-diagram editor by the cost of duplicating the meta model to annotate the model elements with delta information.

We present an approach to visualize model based deltas directly within their corresponding diagram editors. Because you are already familiar with these editors you don't have to invest time to learn a new tool. Instead overlays techniques are used to enhance the existing GUI to visualize the deltas and give you the possibility to merge the different versions. The *Difference Visualization for Models (DVM)* framework may be integrated into existing Eclipse diagram editors build with the Graphical Editing Framework (GEF). This is done by providing new classes on the one hand and by exposing some Eclipse extension-points on the other hand. EMF Compare is used to calculate the delta between two model versions but the DVM framework is open for your model compare framework. To evaluate the DVM framework it has been integrated into the modeling IDE UML Lab [6].

2. MOTIVATION

Software artifacts like source code or models are changed frequently. If the software contains bugs and you know the point in time where everything worked well you might consider to have a look at the changes that were applied to the code between these two times. Another scenario is using a version control system: Before you update or commit your changes you often review the changes between your working copy and the remote files.

Thus, the correct computation and particularly the visualization of deltas is a central aspect in a team for an error-free and efficient work process. For you as the developer it is crucial to understand which changes occurred, if they are meaningful or not and if they correlate with your work in any way. Besides the visualization it is important to also provide merge functions to combine your local and remote artifacts.

The described features are well supported concerning unstructured text files like source code. If you use the synchronizing view in eclipse for example to review changes in shared software artifacts like source code, algorithms like the

Longest Common Subsequence (LCS) [1] are used to compute the delta between different versions of files. You then get a line based comparison of the changes mostly displayed in a simple text editor. While this approach is sufficient for unstructured artifacts like source code this does not hold for the model based development approach because of two reasons:

- Computing the delta of models by using algorithms like the LCS is not possible. Even if you consider a XMI representation of your model there are several disadvantages using a text based approach. As an example moving a model element yields removed and added lines in the XMI document and you loose the semantic of the move operation.
- Because models are modified by using special diagram editors displaying the differences in a simple text editor is not an option. Furthermore it is eligible to really use your existing and well known editor rather than using a dedicated tool because this saves time to learn a new tool.

Even though we did not focus on computing the delta of models we want to point out some approaches to better understand the requirements that a proper framework should meet. The technique presented in [4] is based on recording every single change that occurs in a model and simply save the change stream. This way it is possible to compute the delta in an elegant way. Using heuristics to match elements of different model versions and then compute the delta is another way to solve this problem even if finding good heuristics is not a trivial task. EMF Compare uses several weighted heuristics to compute the delta and serves as the diff-algorithm in our solution.

Mostly you are in the situation that you already have a diagram editor and only want to integrate the described functions with as little effort as possible. This requires a potential framework to be as generic and flexible as possible regarding the computation, visualization and merge functions. To our knowledge there exists no suitable solution which meets all the mentioned requirements. The approach presented in [2] is the closest match because it integrates a delta visualization mechanism into the classdiagram editor of Fujaba but does not provide any merge functionality and is really limited to this editor.

3. THE DVM FRAMEWORK

As already mentioned it is a good idea to visualize model based deltas directly within their corresponding diagram editors. In the eclipse environment the preferred technology to create diagram editors is GEF. The DVM framework allows you to add the difference visualization and merge functionality into existing GEF based editors with as little work as possible.

To be able to recognize the deltas within the diagram at first sight and distinguish them from other diagram elements we decided to use an overlay technique. This has some advantages:

- Overlays can be distinguished from your diagram elements easily.
- The UI for visualizing deltas does not depend on the UI used by the diagram editor which means that the overlays can be used generically. Changing the editors UI does not affect the delta visualization.
- To integrate the delta UI elements with your existing editor you don't need to change your figures.
- The design of the delta UI elements is very similar to the presentation used by conventional text based editors. Thus you intuitively know what is meant by a given delta UI element.

In the following sections we point out some details of the DVM framework beginning with the UI representation of differences, followed by the merge mechanism and then discussing some implementation details.

3.1 Overlay technique

Figure 1 shows an example used to visualize a delta concerning a changed attribute of a class within a classdiagram.

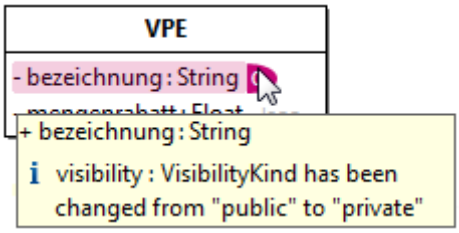

Figure 1: Tooltips provide detailed information

Beside the visualization for any type of node (e.g. classes, attributes, operation etc. in classdiagrams) it also important to support diagram edges. Figure 2 shows how the DVM framework visualizes a deleted association. In addition to the edge itself, all elements that belong to this edge are are overlayed with a delta annotation.

As you can see from figure 2 it is not clear at the first sight what exactly has been changed. To provide more information about the change and support merging of deltas the overlays are extended with special markers. Hovering over these markers populates a tooltip which displays detailed information about the delta. Figure 1 shows an example of a changed visibility of the attribute `bezeichnung:String`.

3.2 Context dependent delta visualization

The kind of the delta of course has great influence on the overlay generated by the DVM framework. Table 1 gives an overview of the possible delta types along with the letters used for the special markers and the default colors.

Type	Abbr.	Description
ADD	A	Element added
DELETE	D	Element removed
MODIFY	C	Element modified
MOVE	M	Element moved

Table 1: Default colors of delta types

The DVM framework takes care for choosing the correct overlay in case of added or removed elements. This operation is context dependent. An example: Let's assume that you have a class `Person` in your local model which does not exist

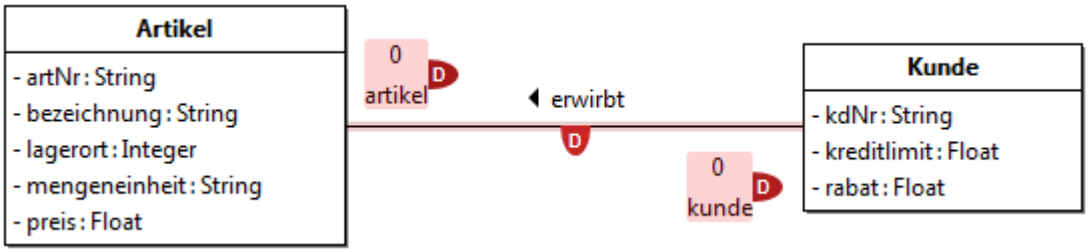

Figure 2: Overlay to visualize a delta for an association

in the remote version. Further your local version of the model is newer than the one on the server then this means that the class `Person` has been added and thus the DVM framework generates an overlay of type `ADD`. If the remote version is newer than your local one then the framework generates an overlay of type `DELETE`. In the former case this can be called *forward-diff* and in the latter *backward-diff*.

3.3 Merging deltas

We already talked about the special marker which extends every overlay. Besides the tooltip which gives detailed information about the delta, the marker is clickable allowing you to trigger merge actions. Figure 3 shows a marker menu exemplary for a delta of type `MODIFY`. This enables you to either accept oder discard this change.

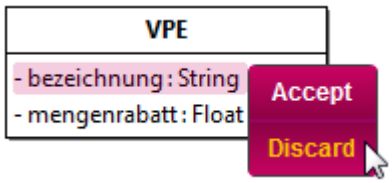

Figure 3: The merge menu

The displayed text in this menu also depends on the delta kind. If you have a delta of kind `ADD` for example then you have the choices *Keep* and *Remove* instead of those shown in figure 3.

3.4 Diff Algorithms

As already stated we did not focus on implementing a new diff algorithm to compute the delta between different versions of a model. Instead we provide a `DiffAlgorithm` and a `Merger` interface. The former provides access to the underlying diff algorithm while the latter is able to execute merge operations. The DVM engine uses the `DiffAlgorithm` interface to compute the delta between two versions of a model and transforms the result of this operation into an internal `DiffViewerModel`. This model is then used for further operations like generating the overlays or triggering merge actions. Using an additional model rather than simply use the EMF Compare model has the advantage of decoupling the DVM framework from special diff and merge implementations.

The default `DiffAlgorithm` is the `EMFCompareDiffAlgorihthm` which is able to compare ecore based models. The `EMFCompareMerger` is used to merge the deltas

3.5 How to use DVM

We now want to give a brief overview of how the DVM framework can be integrated into existing GEF diagram editors. To display the generic overlays you have to annotate your model with the delta information provided by the DVM `DiffViewerModel`. This is done by implementing the `DiffViewerAnnotator` interface and executing the following steps:

1. Merge the delta into your local model if the delta kind is `DELETE`. This is necessary to display local deleted elements.
2. Find the element related to the delta in your local model and register it in the DVM framework
3. Undo any changes made in the first step if the delta kind is `DELETE`. Note that care must be taken to not remove any visual representation of your local element because otherwise the deleted element can not be displayed.

Steps 1 and 3 need a bit more explanation: If you want to display a locally deleted element (which means that you do not have the element in your domain model) you first have to get this element into your domain model and create the corresponding visual representation. This also means that the DVM framework needs to modify your local domain model even if you just want to display the differences. We decided to undo any of these changes made in the domain model but keep the visual representation in Step 3. Another way could be to not undo the changes, keep track of the modified elements and undo the changes for discarded deltas and do nothing for accepted ones. This really is the most difficult step while integrating the DVM framework into your editor. We did not find a generic way to do this because the mechanism to undo changes is specific to your application.

After implementing the `DiffViewerAnnotator` you have to expose this class to the DVM framework using a special eclipse extension point provided by the DVM framework. In the last step two new GEF EditPolicies, `DiffVisualizationNodeEditPolicy` and `DiffVisualizationEdgeEditPolicy`, are used to visualize the generic overlays and provide support for merge actions. EditPolicies in GEF are the controllers in the MVC design pattern which means that they can get information for both model and the corresponding visual representation. Depending on whether your EditPart is a node or an edge you register the former or the latter EditPolicy on your EditPart. If now a new delta is registered in step 2, both EditPolicies are able to find the correct GEF figure, determine it's size and position and finally create the

generic overlay. To visualize all deltas in a new diagram you have to implement the DVM `DiagramCreator` interface. Existing diagrams can be found by implementing the `DiagramFinder` interface.

As you can see, integrating the DVM framework is relatively easy. To evaluate the framework it has been successfully integrated into the classdiagram editor of the modeling IDE UML Lab. Figure 4 contains a screenshot of the classdiagram editor of UML Lab extended by DVM delta overlays.

4. RELATED WORK

To our knowledge the number of comparable work is very low and there is no framework like the DVM. However we want to present two approaches to compare models and visualize deltas.

We already mentioned the EMF Compare framework. This framework consists of a couple of eclipse plugins and thus is intended to be used inside the eclipse environment. EMF Compare provides algorithms to calculate the delta between two versions of an ecore model. Therefore as long as your model is an ecore model you can use the generic diff algorithm of EMF Compare. Besides the diff functionality EMF Compare provides a generic tree editor to visualize and merge the differences. This generic tree editor both has advantages and disadvantages: On the one hand you can use your ecore model without modification. On the other hand it means that you have to learn new tool instead of using your own diagram editor.

The solutions presented in [2] integrates the visualization of deltas into the classdiagram editor of Fujaba. This is done by importing a special XMI file which contains the model elements along with the delta information. To annotate the model elements a duplicate of the meta model which is extended by the delta model elements is necessary. Furthermore special components are required to finally display the changes within the classdiagram editor. The workflow to compute and visualize the delta between two versions of a model is complex in this approach because you have to provide the mentioned XMI file by yourself with third party tools. Additionally the delta visualization support is limited to the classdiagram editor and it is not possible to merge any differences.

5. SUMMARY AND FUTURE WORK

In this paper we have presented an approach to visualize differences between models directly within the corresponding diagram editors. The DVM framework can be integrated in GEF based diagram editors in the eclipse environment and supports both visualizing and merging of deltas. We evaluated the DVM framework by integrating it into the classdiagram editor of the modeling IDE UML Lab and saw that the approach works as intended.

The framework lacks some features. For example a three-ways-diff is not possible by the time writing this paper. The DVM framework therefore cannot visualize any conflicts as they involve three versions of a model: The local, the remote and the base. As this is necessary while working in a team which uses some kind of version control system we plan to implement this feature as soon as possible.

Furthermore we have some ideas where the DVM framework could be used as well. One example is the visualization of the delta triggered by a refactoring operation. If the refactoring is complex it may not be obvious what exactly has changed in the model. Thus visualizing these changes is a useful feature.

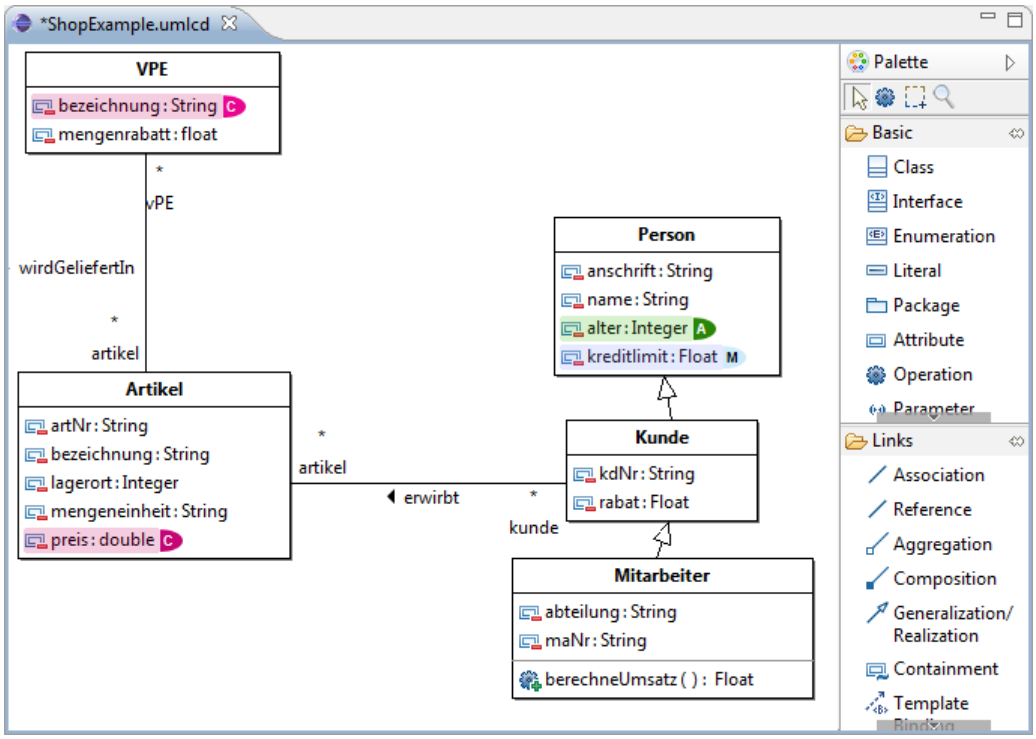

Figure 4: DVM framework integrated in UML Lab

6. REFERENCES

[1] L. Bergroth, H. Hakonen, and T. Raita. A survey of longest common subsequence algorithms. In *Proc. Seventh International Symposium on String Processing and Information Retrieval SPIRE 2000*, pages 39–48, Sept. 27–29, 2000.

[2] S. Lück. Ein differenzanzeige- plugin für klassendiagramme in fujaba. Master's thesis, Universität Siegen, 2004.

[3] Object Management Group. Unified modeling language.

[4] C. Schneider. CASE Tool Unterstützung für die Delta-basierte Replikation und Versionierung komplexer Objektstrukturen. Master's thesis, Technische Universität Braunschweig, 2003.

[5] N. Skrypuch. EMF Compare. http://www.eclipse.org/modeling/emf/?project=compare.

[6] Yatta Solutions GmbH. UML Lab. http://www.uml-lab.com/, 2009.

Schedule

11.05.2011: Begin at 9:00

- 09:00 Welcome – Ulrich Norbisrath
- 09:15 A new Meta-Model for Story Diagrams – Markus Von Detten, Jan Rieke, Christian Heinzemann, Dietrich Travkin and Marius Lauder
- 09:45 Interpreting Story Diagrams for the Static Detection of Software Patterns – Markus Fockel, Dietrich Travkin and Markus Von Detten
- 10:15-10:45 Coffee break
- 10:45 3. Garage48 and Mooncascade – Where former Fujabadevelopers can end up – Asko Seeba
- 11:30 4. A simple indoor navigation system with simulation environment for robotic vehicle scenarios – Ruben Jubeh, Albert Zuendorf and Simon-Lennert Raesch
- 12:00-13:30 Lunch break
- 13:30 SDM online – Jörn Dreyer, Christoph Eickhoff and Albert Zündorf
- 14:00 Yet another TGG Engine? -- Nina Geiger, Bernhard Grusie, Albert Zündorf and Andreas Koch
- 14:30-15:00 Coffee break
- 15:00 A Master Level Course on Modeling Self-Adaptive Systems with Graph Transformations – Matthias Tichy
- 15:30 Using Fujaba in Systems Modeling – A Teaching Experience Report – Artjom Lind, Ulrich Norbisrath, and Ruben Jubeh
- 16:00-16:30 Coffee break
- 16:30-18:00 Informal sessions on Teaching Fujaba and New Metamodel (and more)
- 19:00 Social event (sauna + drinks + snacks in highest building in Tartu)

12.05.2011: Begin at 10:00

- 10:00 "Keynote" Agile modeling and coding with UML Lab – Andreas Scharf
- 11:00 Coffee break
- 11:30 Visualization of Pattern Detection Results in Reclipse – Marie Christin Platenius, Markus Von Detten and Dietrich Travkin
- 12:00 Providing Timing Computations for Fujaba – Tobias Eckardt and Christian Heinzemann
- 12:30-14:00 Lunch break
- 14:00 UML Toolchain – Andreas Koch and Albert Zündorf
- 14:30 Difference Visualization for Models (DVM) – Andreas Scharf and Albert Zündorf
- 15:00-15:30 Coffee break
- 15:30-open-ended practice sessions
- approx. 19:00 Pizza+drinks at the university

13.05.2011: Begin at 9:00

- 09:00-11:30 Practice sessions
- 11:30 Goodbye
- 12:00-13:00 Final lunch
- Departure