Holger Giese, Bernhard Westfechtel (Eds.)

FUJABA Days 2006

28th -30th September 2006
Bayreuth, Germany

Proceedings

Volume Editors

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

Prof. Dr. Bernhard Westfechtel
University of Bayreuth
Chair of Applied Computer Science I
Universitätsstraße 30, 95440 Bayreuth, Germany
bernhard.westfechtel@uni-bayreuth.de

Program Committee

Program Committee Chairs

Holger Giese (University of Paderborn, Germany)
Bernhard Westfechtel (University of Bayreuth, Germany)

Program Commitee Members

Uwe Aßmann (TU Dresden, Germany)
Jürgen Börstler (University of Umea, Sweden)
Gregor Engels (University of Paderborn, Germany)
Pieter van Gorp (University of Antwerp, Belgium)
Sabine Glesner (University of Karlsruhe, Germany)
Luuk Groenewegen (Leiden University, Netherlands)
Reiko Heckel (University of Leicester, UK)
Jens Jahnke (University of Victoria, Canada)
Mark Minas (University of the Federal Armed Forces, Germany)
Manfred Nagl (RWTH Aachen, Germany)
Jörg Niere (University of Siegen, Germany)
Bernhard Rumpe (TU Braunschweig, Germany)
Andy Schürr (TU Darmstadt, Germany)
Wilhelm Schäfer (University of Paderborn, Germany)
Dániel Varró (Budapest University of Technology and Economics, Hungary)
Gerd Wagner (University of Cottbus, Germany)
Albert Zündorf (University of Kassel, Germany)

Editor´s preface

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

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

Several research groups have also chosen Fujaba as a platform for UML and MDA related research activities. In addition, quite a number of Fujaba users send us requests for extensions and improvements.

The 4th International Fujaba Days aim at bringing together Fujaba developers and Fujaba users from all over the world to present their ideas and projects and to discuss them with each other and with the Fujaba core development team. For the first time, they take place in Bayreuth, which is well-known for the music of Richard Wagner and the university.

We received 16 papers, which were reviewed carefully by the program committee. 10 submissions were accepted as long papers (up to 8 pages), 4 short contributions (up to 4 pages) complement the technical programme. The papers are organized into sections on story diagrams, constraints and rules, tool building, embedded systems, and graph transformations.

The workshop programme is decorated with an invited talk given by Hans Vangheluwe, Montreal, who makes a case for modeling and introduces the field of multi-paradigm modeling ("Model Everything! Exploring Multi-Paradigm Modelling.").

Two events are co-located with the Fujaba Days. The Fujaba Developer Days are held twice a year, one of the meetings being combined with the Fujaba Days. They provide a forum for Fujaba developers who can´t stop programming and are eager to learn about the most recent developments of their colleagues (and also like to demonstrate their own improvements). In addition, the 1st Triple Graph Grammar Workshop is co-located with the Fujaba Days as a self-standing event for presenting and discussing research into triple graph grammars.

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

Holger Giese & Bernhard Westfechtel
Program Committee Chairs

Table of Contents

Model Everything!

Exploring Multi-Paradigm Modelling

Hans Vangheluwe
McGill University, Victoria, Canada

Abstract

Engineering and Science invariably use models to describe structure as well as behaviour of systems. Models may have components described in different formalisms, and may span different levels of abstraction. In addition, model transformation is commonly used to transform models into domains/formalisms where certain questions can be easily answered. These various aspects are condensed into the term "multi-paradigm modelling".

Multi-paradigm concepts are naturally applicable to "domain-specific modelling". Using domain-specific modelling environments maximally constrains users, allowing them, by construction, to only build syntactically correct models. Furthermore, the domain-specific, often visual syntax used matches the users' mental model of the problem domain. The time required to construct domain/formalism-specific modelling, simulation and application synthesis environments can however be prohibitive. Thus, rather than using domain-specific environments, users resort to generic environments. Such generic environments are necessarily a compromise.

In this presentation, the foundations of (domain-specific) multi-paradigm modelling will be surveyed. It will be shown how all aspects of modelling can be explicitly (meta-)modeled enabling the efficient synthesis of domain-specific multi-paradigm modelling environments.

Various scientific challenges and open problems such as the management of model evolution, and the modularisation of transformation models will be presented. In the examples, our Computer Automated Multi-Paradigm Modelling (CAMPaM) tool AToM^3 (A Tool for Multi-formalism and Meta Modelling) will be used.

Beyond Story Patterns: Story Decision Diagrams *

Holger Giese and Florian Klein†
Software Engineering Group, Department of Computer Science,
University of Paderborn, D-33098 Paderborn
hg|fklein@uni-paderborn.de

ABSTRACT

Story Patterns are integral to Fujaba's approach to model-driven development, providing the operational semantics the UML is missing. They can also be employed as a formal specification language. They have been used to specify transition systems and safety properties which are then formally verified. However, Story Patterns face some restrictions, notably with respect to negation. When used for a formal specification, they also lack features of logics such as quantification. In this paper, we propose Story Decision Diagrams (SDD), an extension of Story Patterns capable of expressing complex properties while retaining or surpassing the intuitiveness of the original visual notation. The proposed extensions include quantization, implication, alternatives, negation of complex properties, and a concept for modularity.

1. INTRODUCTION

The UML provides an accessible, visual notation for specifying software that has developed into the de facto standard in software modeling. However, standard UML is lacking clearly defined semantics. In particular, there are no operational semantics that would allow to specify structural adaptations.

One extension that has been proposed to remedy this problem are Story Patterns [5]. They are at the heart of the Fujaba Tool Suite, enabling the code generation of complete programs. In Story Diagrams, they are used to model conditions and specify structural transformations of the system, thus providing the missing operational semantics.

As Story Patterns are based on the theory of graph transformation systems (cf. [8]), they also have formal semantics. Formal methods can thus be used based on a specification written with Story Patterns. E.g., it is possible to verify safety properties that are inductive invariants of the system [1], where both the model of the system behavior and the requirements are expressed by Story Patterns.

However, Story Patterns face some restrictions. Negation, i.e. forbidding the occurence of certain elements, is limited to single elements. They are also lacking the quantors and logical operators that would be required to be a full-scale replacement for the OCL [6], the UML's textual specification language for restricting (structural) properties. Finally, when encoding requirements, there is no way to reference previously defined properties.

In this paper, we therefore propose an extended notation based on Story Patterns. Story Decision Diagrams (SDD), are capable of expressing complex properties while retaining or, w.r.t. negation, improving on the intuitiveness of the original visual notation. The proposed extensions include quantization, implication, alternatives, negation of complex properties, and a concept for modularity.

In order to present the different aspects of the notation, we use illustrative examples inspired by the the RailCab R&D project, which is developing a system of autonomous shuttles travelling on a railway network. Our examples mostly deal with the structure of the system, which is modeled as a graph of small unidirectional track segments, each about as large as a shuttle, and the presence of coordination patterns ensuring the safe real-time coordination between the shuttles. Shuttles are located on one track and may have next relationships with other tracks to indicate where they are travelling. They perform the registration pattern with the controllers responsible for their current track section to publish their current position. Based on this pattern, shuttles that are close to each other can coordinate their movement using the convoy pattern. The class diagram in Fig. 1 provides an overview.

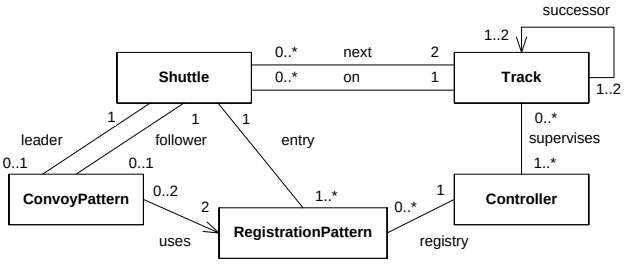


Figure 1: The elements of the shuttle system

The paper is organized as follows: We first provide a short overview of the state of the art, discussing Story Patterns and other approaches. We then introduce our extended notation in Section 3, followed by an overview of its complete syntax in Section 4 and its formal semantics in Section 5. We close with a short discussion of evaluation and future work.

†supported by the International Graduate School of Dynamic Intelligent Systems.
*This work was developed in the course of the Special Research Initiative 614 – Self-optimizing Concepts and Structures in Mechanical Engineering – University of Paderborn, and was published on its behalf and funded by the Deutsche Forschungsgemeinschaft.

2. STATE OF THE ART

Software is becoming more and more pervasive, and the problems tackled by it grow more and more sophisticated and critically important. Therefore, a growing number of modelers, who may actually work in related engineering disciplines, is faced with increasingly complex modeling problems. Examples include autonomous agents, structural adaptation at run-time or ad-hoc collaboration. However, standard UML provides little support for the specification of structural constraints or adaptions.

UML. For specifying more detailed structural properties, the UML only provides the OCL [6]. The textual representation forces the developers to translate their concrete ideas about the required structural properties from the familiar structural view in form of class and object diagrams into an often intricate textual syntax. When reading OCL, a complicated and error prone translation in the opposite direction is required.

This mental mapping problem of textual OCL is already problematic in most standard software engineering environments, where OCL is therefore rarely employed. Important structural properties are often not documented and the related information is lost during the development, as no tool besides natural language seems to be able to capture them, at least not economically. The following snippet encodes a structural property that, though simple, already requires a significant parsing effort to understand:

```
context Track inv
  Track.allInstances()→forAll(t1, t2, t3 |
    (t1.successor→exists(tx | tx = t2)
       and t2.successor→exists(ty | ty = t3))
    implies (Controller.allInstances()→exists(c1 |
      c1.tracks→exists(ta, tb, tc |
        ta = t1 and tb = t2 and tc = t3))))
```

OCL 1: Consecutive tracks share a controller

Constraint diagrams. Constraint diagrams [3] visualize constraints as restrictions on sets using Euler circles, spiders and arrows. To compensate for the decrease in expressive power w.r.t. the OCL, constraint trees [4] combine them with the idea of parsing an OCL statement into a tree, replacing only selected constraints with constraint diagrams. The downside is that while quantification on sets is intuitive, structural constraints quickly result in intricate, visually complex diagrams with little relation to the original UML specification.

Story Patterns. Story Patterns are an extended type of UML object diagram (cf. [5]) that allow expressing properties and transformations, especially structural changes. A Story Pattern consists of two object diagrams representing a pre- and a postcondition, the left hand side (LHS) and the right hand side (RHS). At runtime, the LHS is matched against the instance graph, and the variables of the pattern are bound to specific nodes and edges. If a match is found, it is transformed in order to match the RHS by adding, modifying and deleting the appropriate nodes and edges using the Single Push Out strategy (SPO).

When specifying Story Patterns, the RHS is integrated into the LHS in order to obtain a compact representation. This is achieved by using the stereotypes ≪create≫ for marking exclusive elements of the RHS that need to be created and ≪delete≫ for denoting elements of the LHS which should be deleted as a side-effect of the rule.

Furthermore, it is possible to indicate forbidden elements in a Story Pattern by crossing them out. They can be employed to describe rules that only match when no match for any *one* of their forbidden elements is found, enabling more differentiated rules. However, it is not possible to express that a combination of elements should be absent, as the pattern application fails as soon as the first forbidden element is found. For the same reason, it is not possible to specify forbidden elements that are characterized by multiple associations, e.g., 'no pattern exists *between* shuttles A and B'. While there is a makeshift solution using optional elements ('A may or may not have a pattern, but if so, then not with B'), this solution is not robust, not semantically equivalent, and breaks down for more than two associations.

Story Patterns without side effects describe and allow testing for system properties. E.g., the Story Pattern in Fig. 2 matches if a shuttle's on and next associations point to adjacent tracks in the proper order. However, there is no way to make it explicit in the pattern that we would like this property to be a positive invariant of the system that is true for all shuttles. A common solution for GTS is the implicit convention that all patterns represent *negative* invariants of the system. However, the resulting restriction entails the use of unintuitive multiple negations, i.e. representing a required element as a forbidden element of a forbidden pattern.

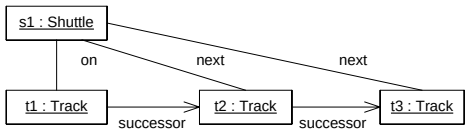


Figure 2: Story pattern: a simple positive invariant

3. STORY DECISION DIAGRAMS

Story Decision Diagrams (SDD) are an extension of Story Patterns that allow expressing more complex properties while retaining or surpassing the intuitiveness of the original visual notation. The extensions we introduce include quantors, implication, alternatives, negation of complex properties, and a concept for modularity.

Basic Principles. An SDD is a directed acyclic graph (DAG). Each node contains a Story Decision Diagram Pattern (SDDP) – basically a Story Pattern without side effects – that specifies some property. When evaluating the SDD, the nodes are processed starting from the root node. Which node is evaluated next depends on whether the current node matches or not. Each node in the DAG can essentially be seen as a local if-then-else decision for a binding. If a match is found, we follow the solid then connector; if no match is found, we follow the dashed else connector. New object and link bindings, i.e. successfully matched elements, are added to the current binding and propagated to subsequent elements. There are two special leaf nodes, (1) signifying *true* and (0) signifying *false*. When a binding reaches a leaf node, it evaluates to true or false, respectively. SDDs are thus similar to decision trees. However, like reduced binary decision diagrams (RBDD), SDDs are not trees, but allow sharing isomorphic subtrees and leaf nodes to reduce diagram size. Like in decision diagrams, consecutive conditions

correspond to logical conjunction, respectively implication.[1] Unlike standard decision diagrams, SDDs support alternatives by allowing multiple then or else connectors per node. It is then sufficient for one of the available paths to reach (1) in order to evaluate the whole branch as true.

In the SDD in Fig. 3, the root node matches when a given Controller is the supervisor of the Track a given Shuttle is on. Now, if the root node matches, then (left child) the Controller and the Shuttle have to run a RegistrationPattern, else (right child) they must *not* be running such a pattern.[2]

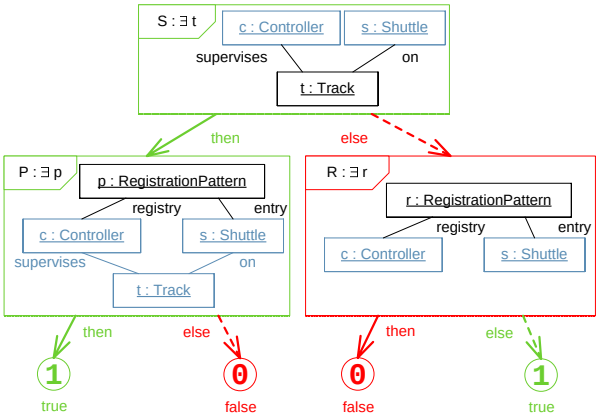


Figure 3: SDD Fragment illustrating basic syntax

Observe that there are only positive elements in the patterns - besides their limitations, negative elements often prove problematic when interpreting Story Patterns. The negation in the right branch is expressed by modeling the forbidden situation as a positive match and switching the then and else connectors, i.e. a match leads to failure and no match leads to success. By appropriately chaining the corresponding nodes, complex negative conditions can be expressed.

In absence of negation, most leaf nodes can be omitted. If not specified otherwise, a node is interpreted as a positive requirement: matching (then) results in success, i.e. (1), not matching (else) results in failure, i.e. (0). All connector and leaf labels are optional.

Though we extensively use color to make diagrams more readable, color is *never* semantically relevant, i.e. the coloring is deduced automatically from the structure. Unbound elements in SDDPs are black, bound elements blue (grey). (1) and connectors leading to (1) are green (light grey), (0) and connectors leading to (0) are red (dark grey). The remaining then connectors are green, the remaining else connectors are red. Node frames share the color of their then connector as a visual cue to make undesired properties stand out. Figure 4 illustrates this principle, marking the existence of an accident (two Shuttles on the same Track) as an undesired instance situation.

[1]There are two equivalent interpretations of the statement if a then b else c - $(a \wedge b) \vee (\neg a \wedge c)$, using conjunction, and $(a \Rightarrow b) \wedge (\neg a \Rightarrow c)$, using implication.

[2]The SDD thus corresponds to either $((\exists t|S[c,s,t]) \Rightarrow \exists t'|(S[c,s,t'] \wedge \exists p|P[c,p,s,t'])) \wedge ((\nexists t|S[c,s,t]) \Rightarrow \nexists r|R[c,r,s])$ or, using the equivalent but more compact interpretation based on conjunction, $(\exists t|(S[c,s,t] \wedge \exists p|P[c,p,s,t])) \vee ((\nexists t|S[c,s,t]) \wedge \nexists r|R[c,r,s])$.

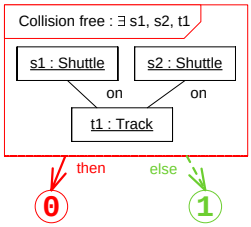


Figure 4: No two shuttles on the same track

Quantification. As a significant enhancement, we allow quantification over the unbound variables of an SDD node. Accordingly, we differentiate between existential nodes, which require at least one binding for which a successful path exists, and universal nodes, which require such a path for every binding they propagate.

Existential nodes may be *existentially quantified* nodes that either bind *explicitly* named variables var_i to objects and links ($[\exists\, var_i^+ : P]$) or only bind *anonymous* variables to links ($[\exists\, \_ : P]$), and *guard* nodes $[\bullet : P]$ that do not bind any new variables, but merely act as a filter on the current bindings depending on whether they match the SDDP P. Quantified nodes can thus generate new alternative bindings that need to be evaluated, whereas guard nodes can only constrain the set of existing bindings. If there is no else connector, a connector to (0) is implied.

The only universal node type is the *universally quantified* node $[\forall\, var_i^+ : P \Rightarrow ...]$ that requires that for each binding matching SDDP P, a successful path exists in the SDD. If no binding matching P exists in the first place, the expected semantics of $\forall$ require that the expression evaluate to true – therefore, an else connector to (1) is always implied.

Figure 5 encodes the requirement that for any three consecutive Tracks ($\forall$), there must be a Controller ($\exists$) supervising them all (see listing OCL 1 above).

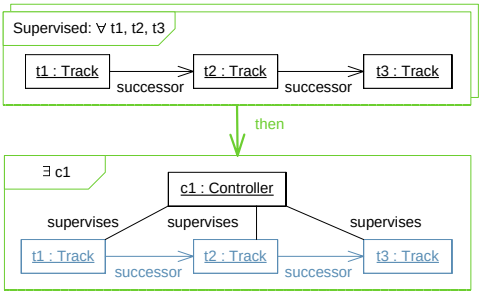


Figure 5: Connected tracks share a controller

Embedded SDD. Formal specification languages typically allow the composition of complex properties from simpler properties. In the OCL, it is possible to reference more concrete properties in the definition of a property, whereas most visual specification techniques lack this capability. The presented approach offers similar functionality that provides a visual abstraction for arbitrarily complex structural relationships and constraints.

SDDs support the composition of specifications through *Embedded Story Decision Diagrams* (ESDD). Basically, it is possible to directly embed any SDD into a host node of

another SDD. This node will then only match if the nested SDD succeeds as well. The nested SDD inherits all bindings from the host SDD, but introduces a local scope. This mechanism is the basis for *scoped nodes*, which can be used like parantheses, and *scoped AND nodes*, serving as a shorthand for two consecutive scoped nodes.

Named ESDDs can be used to encode nontrivial properties that can then be referenced by several SDDs. In order to allow their reuse in different contexts, they are defined as patterns with free variables that are bound depending on the respective current context.

An ESDD specification begins with a dedicated node type – a λ node $[Name : \lambda role_1, role_2, \ldots]$ – that defines its name and available roles. When the ESDD is invoked in a given context, the λ node binds the local variables in accordance with the provided context. The ESDD is otherwise processed like a regular SDD, eventually evaluating to $emphtrue$ or $emphfalse$.

In the host node containing the reference to named property in question, we represent the ESDD using the UML symbol for a pattern, a dashed circle. The bound elements of the host node are assigned to the roles of the ESDD by means of dashed lines labeled with the respective role name.

Figure 6 requires that all Shuttles are correctly registered with all Controllers that supervise the Track they are on. Being registered simply requires the existence of a RegistrationPattern as encoded by the ESDD registered in Fig. 7.

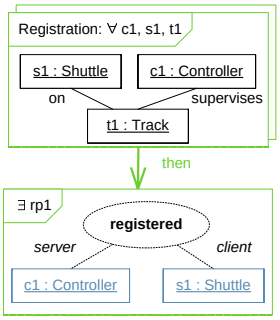


Figure 6: Shuttles must be properly registered

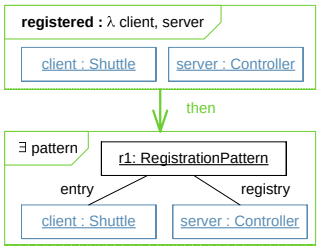


Figure 7: ESDD encoding the registered property

As ESDDs are SDDs, it is possible to have nested ESDDs. This quite naturally leads to recursively defined patterns. Figure 8 recursively defines the property that Track *to* is reachable from Track *from*. Recursion raises the question of termination. As we can assume that any instance graph consists of only a finite number of elements, there is only a finite number of distinct initial bindings that can be passed to an ESDD's λ node. By adopting the restriction that, in any recursion, each initial binding is evaluated at most once, we can thus guarantee termination. If an ESDD accepts numerical parameters, e.g. if the above example accepted a lower and an upper bound that allowed selecting all reachable tracks that were between 2 and 5 links away, termination requires proving additional termination conditions, e.g. that the upper bound is strictly decreasing in this example.

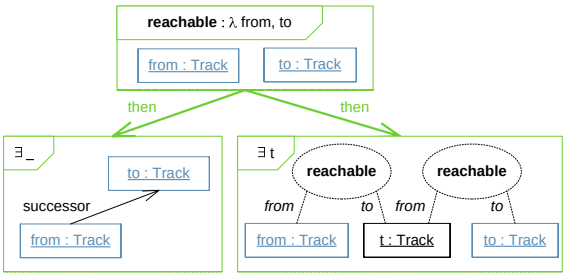


Figure 8: *to* is reachable from *from*

Transformations. SDDs allow specifying more complex (pre-)conditions, i.e. enhance the LHS of Story Patterns. The SDDPs in universal and existential nodes never have side effects. Transformation nodes ($\rightarrow +var_i^+; -var_i^+$) add the ability to specify transformations, i.e. RHS. They replace (1) leaf nodes. When a binding reaches the node, it is stored, and when the SDD evaluates to true, all transformations are applied. For existentially quantified properties, this simply means that once the first binding reaches a transformation node, it is applied. Universally quantified transformations are only applied when a transformation node has been reached for all required bindings, which is different from iteration over the bindings.

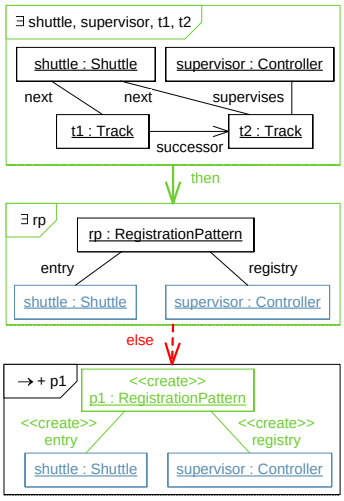


Figure 9: Creating a registration pattern

Figure 9 provides a simple example, creating a pattern provided that no matching pattern already exists. As this is a very common idiom, it is possible mark the node with $\rightarrow \exists\, var_i^+$ as a shorthand to indicate that the node should check whether what it is creating already exists, i.e. merely ensure that the postcondition holds, thus eliminating the need to explicitly model the second node in the example.

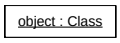
4. SYNTAX REFERENCE

In this section, we list the complete syntax of SDDs. We start with the syntax of the patterns themselves (SDDP), which slightly extends and modifies the existing Story Pattern (SP) syntax, and then present the syntax of the surrounding SDDs. For each element, we present the abstract syntax, with optional or implied elements in square brackets. Remember that color is always redundant.

4.1 Story Decision Diagram Patterns

SDDPs may contain objects, links between objects, constraints and ESDDs.

4.1.1 Objects

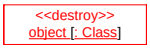


Quantified object. Black. Unbound object. May specify attribute constraints.

Bound object. Slate Blue. Bound by a previous SDDP in the same scope. May specify additional attribute constraints.

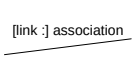
Locally quantified object. Grey. Becomes bound for the local scope.

Created object. Green. Side effect - newly created object. May specify attribute values for initialization.



Destroyed object. Red. Side effect - object will be deleted.

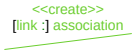
4.1.2 Links



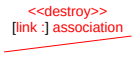
Quantified link. Black. Previously unbound link. It is possible to assign a link identifier (used for quantification).

Bound link. Slate blue. Previously bound link.

Locally quantified link. Grey. Becomes bound for the local scope.



Create link. Green. Side effect - creates a link.



Destroy link. Red. Side effect - destroys a link.

4.1.3 Constraints

Constraint (attributes). Constraints concerning the equality or inequality of multiple attributes of an object or different objects. May use built-in operators, e.g. arithmetic, on attributes, or call functions without side effects (UML query functions).

Constraint (collaboration). In transformation nodes, they enable SDDPs to call methods on bound objects, in the order indicated by the leading numbers. In existential nodes, they serve as guards.

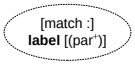
Constraint (link). Optionally, it is possible to connect two objects that are constrained by a common constraint or collaborate with a (directed) curved constraint link, which simply serves as a visual cue for relating the objects.

Homomorphism link. Like SP, SDDPs are matched using graph isomorphism by default. The link indicates that the two objects *may* represent the same instance.

Identity constraint. In connection with homomorphism links, indicates the case where both objects are actually identical.

4.1.4 Embedded SDDs

Role for ESDD. Assigned to a quantified or bound object.



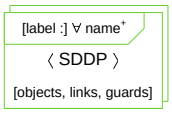
ESDD. An Embedded SDD, with its name, an instance identifier and an optional list of parameters.

4.2 Story Decision Diagrams

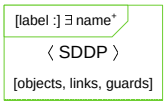
SDDs consist of five classes of nodes and three types of connectors.

4.2.1 Node types

All non-leaf nodes use UML 2.0 boxes with a header field. Except for transformation nodes, which are always black, these boxes are typically green, but may automatically turn red if they are used to model forbidden properties.

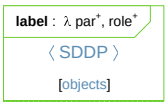


Universally quantified node. Universal quantifier, iterates over a set of bindings, AND joins the results. Contains at least one quantified element. May have multiple then connectors, the else connector is implied.



Existential node. Existential quantifier, creates one binding at a time, backtracks if necessary, i.e. OR joins the results. Contains at least one quantified element. Like all existential nodes, may have (multiple) then and else connectors. If all quantified elements are anonymous, this is indicated by an underscore.

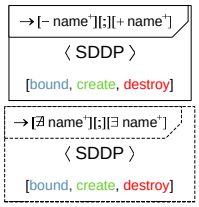
Existential guard node. Marked by a bold dot. Acts as a filter and can only reduce the number of eligible alternative bindings. Contains only bound elements.



Lambda node. Initial node of an ESDD definition. The node label defines the ESDD's name for referencing it in SDDPs. Contains only bound objects defining the types of the ESDD's roles. Additionally may contain a list of typed parameters.

Scoped guard node. A scoped node introduces a local scope and matches if the nested SDD matches, i.e. acts as a filter. Useful in connection with universally quantified subexpressions.

Scoped AND node. Provides a convenient syntax that is equivalent to two consecutive scoped nodes. Purely syntactical enhancement.



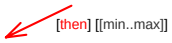
Transformation node. For specifying side effects. May only contain bound, create and destroy objects and links, and destroy objects and links must be previously bound. Appears in place of *true* leaf nodes and is applied to a selected valid binding (set).

True leaf node. Indicates that a binding has successfully satisfied this branch of the SDD.

False leaf node. Indicates that a binding has failed to satisfy this particular branch of the SDD.

4.2.2 *Connector types*

Then connector. Green. Connects a node to the implication that must follow if the node's SDDP matches. Cardinalities can be used to restrict the acceptable number of bindings. Multiple connectors specify alternatives.



Then connector. Red. Identical semantics, serves to underline modeler's intent. Turns red when connected to a *false* leaf node, or if the node's else connector is green.

Else connector. Red. Connects a node to the implication that must follow if the node's SDDP does not match at all. No cardinality, as no bindings were created.

Else connector. Green. Identical semantics, serves to underline modeler's intent. Turns green when connected to a *true* leaf node, or if the node's then connector is red.

Merge connector. Black. Joins transformation nodes into a single pattern, exists to reduce redundancies in the diagram.

5. FORMAL SEMANTICS

Language definitions that focus on expressiveness and intuitive semantics often run into problems when it comes to defining the formal semantics, which the OCL itself illustrates. On the other hand, languages that are constructed starting from a set of formally motivated operators with precise semantics often suffer in terms of expressiveness and especially practical applicability. We therefore now show that the operational, control-flow oriented informal semantics we have used above to introduce the specification technique can be mapped to a formal semantics that allows us to analyze and reason about the matching process.

Foundations. UML object diagrams as well as Story Pattern can be formally expressed using graphs. A *graph* is defined as a tuple $G = (N_G, E_G, src_G, tgt_G)$, where N_G is a finite set of nodes, E_G is finite set of edges, $src : E_G \to N_G$ is the source function that assigns a source node to each edge and $tgt : E_G \to N_G$ is the target function that assigns a target node to each edge.

A *graph labeling* (over two fixed alphabets Ω_N and Ω_E for node and edge labels, respectively) for a graph $G = (N_G, E_G, src_G, tgt_G)$ is a tuple $l_G = (ln_G, le_G)$, where $ln_G : N \to \Omega_N$ is the node labeling function that assigns a label to each node, and $le_G : E \to \Omega_E$ is the edge labeling function that assigns a label to each edge. By labeling graphs in accordance with a labeled *type graph* $\mathcal{T}$, we can define the notion of *type conformant* graphs.

For two graphs SG and G, we say that SG is a *subgraph* of G ($SG \leq G$) iff $N_{SG} \subseteq N_G$, $E_{SG} \subseteq E_G$, $src_{SG} = src_G \mid_{E_{SG}}$, $tgt_{SG} = tgt_G \mid_{E_{SG}}$, $ln_{SG} = ln_G \mid_{N_{SG}}$, and $le_{SG} = le_G \mid_{E_{SG}}$.

If we want to compare instance situations w.r.t. their structure without taking the identity of the nodes and edges into account, we have to use graph morphisms instead of simple subgraph notions for graphs (cf. [8]): A *graph morphism* $m : G_1 \to G_2$ is a pair of functions $m := \langle m_N : N_{G_1} \to N_{G_2}, m_E : E_{G_1} \to E_{G_2} \rangle$, which preserves sources, targets and labels, i.e. which satisfies $m_N \circ tgt_{G_1} = tgt_{G_2} \circ m_E$, $m_N \circ src_{G_1} = src_{G_2} \circ m_E$, $l_{N_{G_1}} = l_{N_{G_2}} \circ m_N$, and $l_{E_{G_1}} = l_{E_{G_2}} \circ m_E$. A *graph isomorphism* *iso* is a graph morphism where both functions m_N and m_E are injective.

For graph rewrite rules $r = (LHS, RHS)$, we require a graph morphism between the left hand side of the rule and the host graph. We use isomorphism except where homomorphism is explicitly indicated. We choose this interpretation because, in most cases, we have found the principle that different pattern elements map to different instances to be more intuitive.[3] Informally, when r is applied to G, all elements (nodes and edges) that are contained in the left hand side as well as in the right hand side are preserved, elements that are only contained in the left hand side are deleted, and elements that belong only to the right hand side are added to G by using appropriate morphisms. As the description of behavior is not in the focus of this paper, we refer to [2] for a complete formalization of the rules.

Using the concepts introduced above, we can thus employ *graph transformation systems* (GTS), a type of state transition system where every state is represented by a graph and every transition is described as a graph rewrite rule, to formally define the behavior of UML models: A *typed graph transformation system* S is a tuple $(\mathcal{T}_S, \mathcal{G}^i_S, \mathcal{R}_S)$ with $\mathcal{T}_S$ a type graph, $\mathcal{G}^i_S$ the set of all type conformant initial graphs of the system, and $\mathcal{R}_S$ a finite set of type conformant graph transformation rules.

SDDs can then be used to specify required invariants of the system that must hold in every reachable state of the system, i.e., match every graph that is generated by the GTS. SDDs with side effects can also be employed to specify graph transformation rules.

Patterns (SDDP). We first consider the patterns contained by individual nodes (SDDP), whose semantics very closely resemble those of Story Patterns. Mapping both the instance situations and the patterns onto graphs permits us to define that a pattern P matches a given graph G if an appropriate morphism exists.

Bindings. As we later need to relate matches of different patterns from the same SDD to each other, we label the nodes, edges, and attributes of P with variables so that each node, edge, and attribute in the SDDPs refers to a

[3]Consider Fig. 4 – using homomorphism, the pattern would flag every single shuttle in the system as a collision with itself unless an additonal constraint $s1 \neq s2$ was added.

corresponding variable from the set of vertex variables V_N, edge variables V_E, and attribute variables V_A of the SDD S. By means of a *variable binding* ξ, SDDPs then can then access those instances that have already been matched and bound by the preceding SDDPs in S. A binding for the node, edge, and attribute variables of S is then a function $\xi = (\xi_N, \xi_E, \xi_A)$ with $\xi_N : V_N \to N$, $\xi_E : V_E \to E$, and $\xi_A : V_A \to \mathbb{N}$ with N the set of all nodes and E the set of all edges of G. τ is the empty binding that maps all variables to $\perp$.

We use $var(P)$ to denote the triple of sets of node, edge, and attribute variables of a pattern P. To map the node, edge, and attribute bindings to a host graph G, we define $P[\xi]$ to be the graph which results from substituting all nodes, edges, and attributes of P with the elements assigned to the corresponding variables by ξ. We use $\mathcal{X}_S[G]$ respectively $\mathcal{X}[N, E, V_N, V_E, V_A]$ to denote the set of all possible bindings (for N the set of all nodes of G, E the set of all edges of G, and variables V_N, V_E, and V_A of S). Together, the labeling of P with variables and the binding ξ define a graph isomorphism between P and $P[\xi]$. In order to match P in G, we then only require that a *valid* binding ξ exists for which P's isomorph $P[\xi]$ is a correct subgraph of G ($P[\xi] \leq G$).

Nodes (SDD). For an SDD S, we define $\mathcal{N}_S$ as the set of its nodes. For each node $n \in \mathcal{N}_S$, P_n is the pattern contained by n, $\mathbf{pre}(n)$ is the set of parent nodes, $\mathbf{then}(n)$ and $\mathbf{else}(n)$ are the set of nodes connected to n by then resp. else connectors, $var(n) := var(P_n)$ the triple of variable sets and $free(n, \xi)$ the variables in $var(n)$ that are not bound by ξ, i.e. mapped to $\perp$. The root node of an SDD is denoted by λ_S.

Witnesses. When evaluating an SDD S, only a subset of the possible bindings $\mathcal{X}_S$ is (potentially) valid. As this subset depends on the considered node, a binding's context is relevant. We define an application ζ as a pair (n, ξ) of a node n and a binding ξ. We call a valid application a *witness*. An application is valid if a path from λ_S to n exists so that ξ is valid for all nodes on the path (excluding n) but binds no additional variables, i.e. $\omega(n, \xi) := \exists (n_1, ..., n_k) \in \mathcal{N}_S{}^* : (n_1 = \lambda_S \wedge n_k = n \wedge \bigwedge_{i=1..k-1}(n_i \in \mathbf{pre}(n_{i+1}) \wedge P_{n_i}[\xi] \leq G) \wedge \forall v \notin \bigcup_{i=1..k-1} var(n_i) : \xi(v) = \perp)$. The set of possible witnesses is then $\mathcal{Z}_S := \{(n, \xi) | n \in \mathcal{N}_S, \xi \in \mathcal{X}_S, \omega(n, \xi)\}$. We further define the truth value $eval(\zeta)$ of a witness as *true* if n is a (1) or transformation node, *false* if n is a (0) node, and else $\perp$.[4]

Candidate sets. When informally introducing the semantics of SDDs above, we used an operational interpretation where we iteratively propagated individual bindings across the SDD. In order to define the formal semantics using set-based logic, we need to consider sets of bindings. As each binding that a universal node n creates needs to satisfy the SDD S, the node creates a *candidate set* of witnesses. We define a candidate set as $\mathcal{C} \in \wp(\mathcal{Z}_S)$. $\mathcal{C}$ only satisfies S if all witnesses $\zeta \in \mathcal{C}$ satisfy S. The truth value of $\mathcal{C}$ is thus defined as $eval(\mathcal{C}) := \bigwedge_{\zeta \in \mathcal{C}} eval(\zeta)$. Nested universal nodes would introduce candidate sets of candidate sets; however, we flatten these nested sets into a single set, which is achieved by simply expanding the existing sets with the new elements instead of creating a new subset.

[4]Boolean operators ($\wedge$, $\vee$ and $\neg$) applied to $\perp$ yield $\perp$.

Alternative candidate sets. An existential node or the presence of multiple then or else connectors can create multiple alternative ways to extend the binding ξ of a witness ζ, only one of which needs to satisfy S. Again, we could place a set of alternative witnesses inside the candidate set, adding a new level of nesting for each further node. Instead, we prefer a flattened structure. We define $\mathcal{A} \in \wp(\wp(\mathcal{Z}_S))$ as a set of alternative candidate sets and add a new candidate set $\mathcal{C}^i$ to $\mathcal{A}$ for each alternative extension. As $\mathcal{C}$ is a set of witnesses, each of which may have alternative extensions, the generated number of alternatives depends on the Cartesian product of the extensions for each witness. As one valid candidate is sufficient, the truth value of $\mathcal{A}$ is defined as $eval(\mathcal{A}) := \bigvee_{\mathcal{C} \in \mathcal{A}} eval(\mathcal{C})$. The set of all witnesses occuring in a set of candidate sets $\mathcal{A}$ is denoted by $\mathcal{W}_\mathcal{A} := \bigcup_{\mathcal{C}_i \in \mathcal{A}} \mathcal{C}_i$.

Propagation. For each node n, we define the propagation function $apply_n(G, \mathcal{A}) \to \mathcal{A}'$, which basically removes obsolete candidates and adds appropriately extended versions. $apply_n$ initializes $\mathcal{A}' = \mathcal{A}$. For each witness $\zeta = (n_\zeta, \xi_\zeta)$ for n from $\mathcal{W}_{\mathcal{A}'}$ (i.e. $\zeta \in \mathcal{W}_{\mathcal{A}'} \wedge n_\zeta = n$), the following steps are then performed: (1) The possible extensions of the binding ξ_ζ are computed. We define $\mathcal{X}^1_\zeta := \{\xi'_\zeta \mid P_n[\xi'_\zeta] \leq G \wedge \xi_\zeta \leq \xi'_\zeta \wedge \forall v : \xi'_\zeta(v) \neq \xi_\zeta(v) \implies v \in free(n, \xi_\zeta)\}$, i.e. those ξ'_ζ that are valid for P_n and extend ξ_ζ with the variables introduced by P_n,[5] and $\mathcal{X}^0_\zeta := \emptyset$. If no such ξ'_ζ exists, $\mathcal{X}^1_\zeta := \emptyset$ and $\mathcal{X}^0_\zeta := \{\xi_\zeta\}$. (2) The corresponding witnesses are computed, i.e. the generated bindings are propagated along all applicable connectors. We define $\mathcal{W}^+_\zeta := \{(n', \xi') \mid n' \in \mathbf{then}(n), \xi' \in \mathcal{X}^1_\zeta\} \cup \{(n', \xi') | n' \in \mathbf{else}(n), \xi' \in \mathcal{X}^0_\zeta\}$. (3) The set of alternative candidate sets $\mathcal{A}'$ is updated. (3a) If n is universal, we define $\mathcal{A}'_\forall := \{\mathcal{C}' \mid \exists \mathcal{C} \in \mathcal{A}' : (\zeta \notin \mathcal{C} \wedge \mathcal{C}' = \mathcal{C}) \vee (\zeta \in \mathcal{C} \wedge \mathcal{C}' = \mathcal{C} \setminus \zeta \cup \mathcal{W}^+_\zeta)\}$, i.e. extending each candidate set with the new witnesses. (3b) If n is existential, we define $\mathcal{A}'_\exists := \{\mathcal{C}' \mid \exists \mathcal{C} \in \mathcal{A}' : (\zeta \notin \mathcal{C} \wedge \mathcal{C}' = \mathcal{C}) \vee (\exists \zeta' \in \mathcal{W}^+_\zeta : (\zeta \in \mathcal{C} \wedge \mathcal{C}' = \mathcal{C} \setminus \zeta \cup \zeta'))\}$, i.e. adding a new alternative candidate set for each new witness.

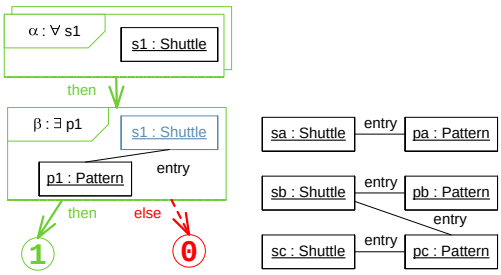
Story Decision Diagram Semantics. We can now define the semantics of an SDD S. For leaf nodes, we have $[\![(1)]\!]^G_\mathcal{A} := \mathcal{A}$ and $[\![(0)]\!]^G_\mathcal{A} := \mathcal{A}$. For non-leaf nodes, we define $step(\mathcal{N}, \mathcal{A}) := [\![n]\!]^G_{step(\mathcal{N} \setminus n, \mathcal{A})} \mid n \in \mathcal{N}$ and $step(\emptyset, \mathcal{A}) := \mathcal{A}$ and then have $[\![n]\!]^G_\mathcal{A} := step(\mathbf{then}(n) \cup \mathbf{else}(n), apply_n(G, \mathcal{A}))$ (i.e. n's second child is evaluated on the result of n's first child etc.). Finally, we define for the whole SDD: $[\![S]\!]^G := [\![\lambda_S]\!]^G_{\{\{(\lambda_S, \tau)\}\}}$, i.e. we start at the root node λ_S of S with a single candidate set consisting of the empty binding τ at λ_S. The truth value of an SDD S is then $eval(S) := eval([\![S]\!]^G)$.

In Fig. 10, we present a basic example for illustrating the introduced semantics. The SDD S in Fig. 10(a) is evaluated on graph G in Fig. 10(b). Figures 10(c)-10(g) then list the witness ζ, the set $\mathcal{A}$, and the set of witnesses $\mathcal{W}_\mathcal{A}$ for each iteration of the propagation functions.[6]

Embedded SDDs. ESDDs can simply be seen as an extended form of guard. Accordingly, they are processed in step (1) of the *apply* function. When computing $\mathcal{X}^1_\zeta$, we additionally require for each $\xi'_\zeta \in \mathcal{X}^1_\zeta$ that for each ESDD E with host node n holds $eval([\![E]\!]^G_{\ell(\xi'_\zeta)}) = true$. The map-

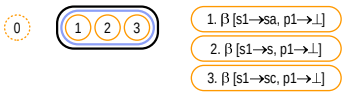
[5]As there are no new variables in guard nodes, $\xi'_\zeta = \xi_\zeta$ and is either valid for P_n or not.

[6]The witnesses in $\mathcal{A}$ are represented by numbers in circles referencing the corresponding elements of $\mathcal{W}_\mathcal{A}$.



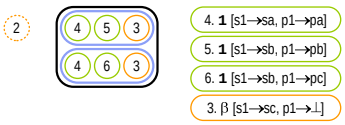
(a) The SDD S (b) An instance graph G

(c) (0) (i.e. τ) is the only witness at root node α

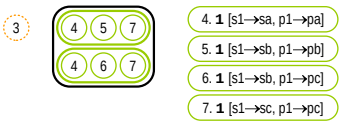


(d) $apply_{\alpha}(G, \mathcal{A})$, $\zeta = (0)$: extends the candidate set to $((1), (2), (3))$

(e) $apply_{\beta}(G, \mathcal{A})$, $\zeta = (1)$: creates witness (4)



(f) $apply_{\beta}(G, \mathcal{A})$, $\zeta = (2)$: creates witness (5) and (6), introducing two new alternative candidate sets



(g) $apply_{\beta}(G, \mathcal{A})$, $\zeta = (3)$: creates witness (7). $eval(S)$ evaluates to true, both candidate sets satisfy S, even.

Figure 10: Successful evaluation of a basic example

ping function $\ell : \mathcal{X}_S \to \mathcal{X}_E$ performs the task of E's λ node, binding E's variables in accordance with ξ'_ζ. As the semantics of the ESDD E with λ node λ_E, we define $[\![E]\!]^G_\xi := [\![\lambda_E]\!]^G_{\{\{(\lambda_E,\xi)\}\}}$. For recursively defined E, we define $[\![E]\!]^G_\xi$ as the least fixed point.

Transformation nodes. In the presence of transformation nodes, we randomly pick a candidate set $\mathcal{C} \in [\![S]\!]^G | eval(\mathcal{C}) = true$ after matching the SDD. For each witness $\zeta = (n, \xi) \in \mathcal{C}$, we then apply the transformation specified by n to $P_n[\xi]$ using standard Story Pattern semantics (Single Pushout).

6. EVALUATION

Assessing the intuitiveness of a visual notation would require an extensive field study with engineers from different disciplines, which was not feasible for us. However, we informally tested the notation on developers who had previous experience with Fujaba. In several iterations, we asked different developers to interpret a set of unrelated diagrams, without introducing the notation first. Results with the finally adopted notation where encouraging, suggesting that most aspects of the notation are intuitively accessible.

In order to assess whether SDDs could successfully be applied with the proposed semantics, we have manually transcribed several SDDs to the input format of the GTS model checker *GROOVE* [7], which required splitting each SDD into multiple, less expressive GTS rules. The above examples where successfully evaluated for a set of correct and incorrect non-trivial instance graphs. Based on code generated from Story Patterns by Fujaba, we have also written Java and C++ programs for evaluating the SDDs.

7. CONCLUSION AND FUTURE WORK

We have presented a visual language that extends Story Patterns with concepts that make them more expressive, especially when they are employed for the specification of requirements.

We are currently implementing the notation as a plug-in for Fujaba4Eclipse. We intend to fully integrate them, i.e. make it possible to not only specify stand-alone SDDs, but to offer the option to use them whereever Story Patterns can currently be used. This will include appropriate code generation capabilities. We also consider automating the export to GROOVE for use in the context of GTS verification.

8. REFERENCES

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

[2] H. Giese and D. Schilling. Towards the automatic verification of inductive invariants for infinite state models. Technical report, Universität Paderborn, 2004.

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

[4] S. Kent and J. Howse. Constraint trees. In T. Clark and J. Warmer, editors, *Object Modeling with the OCL*, pages 228–249. Springer, 2002.

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

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

[7] A. Rensink. Towards model checking graph grammars. In M. Leuschel, S. Gruner, and S. L. Presti, editors, *Workshop on Automated Verification of Critical Systems (AVoCS)*, Technical Report DSSE–TR–2003–2, pages 150–160. University of Southampton, 2003.

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

On Semantic Issues in Story Diagrams

Matthias Tichy, Matthias Meyer, and Holger Giese
Software Engineering Group
University of Paderborn, Germany
[mtt|mm|hg]@uni-paderborn.de

ABSTRACT

Story Diagrams provided by Fujaba are a powerful visual formalism for the specification of structural transformations. Their visual appearance is based on UML activity and UML collaboration diagrams. The semantics draws on graph transformation systems and therefore has a solid foundation. However, as reported in this paper, there are several subtle problems with the semantics and the code generation of Story Diagrams. These problems have been discovered during our efforts to make Story Diagrams applicable for embedded real-time systems and to verify Story Diagrams. We will outline a selection of identified semantic problems in this position paper and will discuss several possible solutions we see which seem more appropriate than the present solution.

1. INTRODUCTION

One cornerstone of the Fujaba Tool Suite and its standard notation to describe behavior are Story Diagrams [2]. They provide a powerful yet intuitive visual formalism for the specification of structural transformations. A central element of Story Diagrams which describe the single transformation steps are Story Patterns which extend UML collaboration diagrams in a natural manner to describe pattern for partial instance situations which should be queried and the side effects which should take place when a match for the pattern has been found. These Story Patterns are then composed to complex transformation by means of UML activity diagrams which describe the control flow between the basic actions.

The semantics of Story Patterns and Story Diagrams [4] is based on graph transformation systems. Graph transformation systems provide a direct mapping for most of the concepts in Story Patterns and Story Diagrams. However, more complex UML concepts such as multiplicities or qualified associations have to be covered differently.

In this position paper, we will report about several subtle problems with the semantics and code generation for Story Diagrams and Story Patterns which have been discovered.[1] We at first looked into ways to make their benefits available in more demanding application areas such as embedded real-time systems where the in the paper outlined present "solutions" are often not acceptable. In addition, we started looking into the problem of verifying Story Patterns [1] and Story Diagrams which lead to the discovery of further problems of semantic issues.

[1]We considered the current version of Fujaba 4 and its built-in Java code generation.

We will outline a selection of detected semantic problems in this position paper and will discuss several possible solutions we see. We will therefore initially review what we consider a appropriate semantics and the resulting requirements the code generation should fulfill in Section 2. To further structure our discussion of semantic issues, we at first look into an number of problems which arise already for Story Patterns in Section 3. Then, we consider in Section 4 problems which relate to the control flow between the Story Patterns when combining them in an overall Story Diagram. Finally, we close the paper with a discussion of our finding and an outlook on planned future work.

2. PREREQUISITES

What is a good semantics or more specifically in our case what are semantic weaknesses which have to be avoided when defining a language and supporting it with a tool? Let us first clarify that we do *not* want to discuss the definition of the semantics or its elegance or appropriateness but rather only the resulting semantics of the language relevant for those who use a language.

Our guideline for a "good" formal semantics can be characterized by the following phrase:

> A formal semantics should assign an unambiguous meaning to each syntactically allowed phrase of the language.

Using this simple statement we can derive the following required properties for the formal semantics:

- The semantics should not be *incomplete*. This means that the semantics must assign a meaning to all syntactically correct instances of a language.
- Each instance of the language should have a unique meaning and thus the semantics should not be *ambiguous*.

While these criteria are important, in the considered cases the semantic issues are not only related to the formal semantics. We also have a *compiler semantics* which is determined by the behavior resulting from the code generation. In addition, we have something like an *intuitive semantics* characterized by the intuitive understanding of an average developer. Therefore, consistency between these semantics is of crucial importance.

From a purely formal point of view we could ignore the intuitive semantics and only require that the compiler semantics respects the formal semantics (otherwise we have a

semantically incorrect code generation). However, in practice it is important to ensure that both the formal semantics as well as the compiler semantics are not at odds with the intuitive semantics. Otherwise, the resulting language is highly misleading and prone to error.

Using our former requirements as starting point, we can then derive the following weaknesses which should be avoided if possible:

- We have an *unexpected instance* if the intuitive understanding cannot map any reasonable meaning to it while it might have a nicely defined formal semantics or compiler semantics.[2]
- We have an *unexpected error* if there are instances of the language which have a proper intuitive meaning. However, the formal or compiler semantics are erroneous for this instance of the language.
- There should be no instances of the language where the formal semantics is in conflict with the intuitive understanding of the average developer. Nevertheless, this formal semantics can be objectively considered as reasonable. We name such ambiguities due to the conflict between intuition and defined semantics *counterintuitive*.

In the following, we present the semantic issues in Story Patterns and Story Diagrams as well as classify them by the above defined weaknesses.

3. ISSUES IN STORY PATTERNS

3.1 Undefined Order of Creations

One of the key features of Story Patterns is the ability to model dynamic changes of object structures. Story Patterns are frequently used to specify the deletion or creation of objects and links between them.

3.1.1 Problem Description

The upper part of Figure 1 shows a class diagram modeling two classes A and B and a one-to-one association ab between them. The Story Pattern below first creates two objects b1 and b2 of type B. Next, links of type ab are created between both new B objects and the bound object a (of type A).

Since ab is a one-to-one association, object a can only be connected to one B instance at a time. Thus, the Story Pattern is intuitively invalid and classifies as an *unexpected instance*. Another problem is, that the order in which the links are created is not defined. Consequently, if only one B object is connected with a after the execution of the pattern, it is not clear which (b1 or b2).

The order in which objects are created is not defined either. In general, the order of object creations does not matter. However, in situations where the creation of objects fails, e.g. because of insufficient memory (cf. Section 4.2), it is unclear which of the creations have been successful. The same is true for failed link creations.

[2]In this case it would be better to have certain constraints which must hold for a valid instance in addition to the syntactical correctness by means of explicit *well-formedness rules*. However, due to computational limits, for very expressive languages not all well-formedness rules required to exclude unexpected instances can be effectively checked.

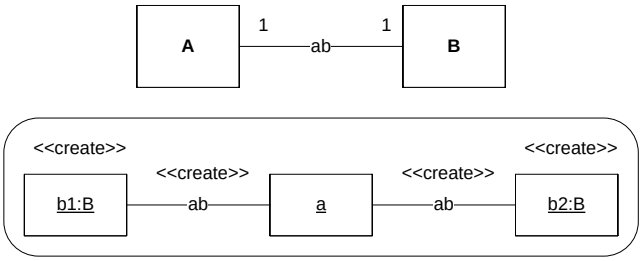


Figure 1: Undefined Order of Creations

3.1.2 Current Semantics

According to the semantics definition in [4], in case of a to-one association, an existing link is replaced upon creation of a new one. In addition, all object creations are executed first and all links are created afterwards. The current code generation adheres to these definitions.

The order in which objects or links are created is undefined and thus non-deterministic.

3.1.3 Similar Problems

Similar problems arise in case of ordered to-n associations, where linked objects are stored in a list. Assume the ab association in Figure 1 is an ordered one-to-n association. Then, both B objects are connected to a after the execution of the Story Pattern. However, it is undefined again, whether b1 will be before b2 in the list of linked objects or the other way round.

Furthermore, the language definition in [4] allows for inserting single objects into an ordered association at specific positions. In Figure 2, the first object linked to a via the ordered ab association is bound to b1. Then, a new object b2 is created and linked to a as direct successor of b1 (indicated by the next arrow). This is in accordance with the formal semantics. The current code generation, however, does not support this feature.

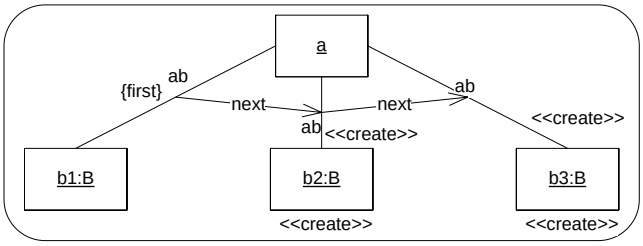


Figure 2: Inserting a sequence into an ordered association

In the above figure, another object b3 is created and linked to a as successor of b2. This insertion of a sequence of objects in a defined relative order is not supported at all.

3.1.4 Possible Solutions

Conflicting link creations in case of to-one associations can be detected at specification time and should simply be forbidden.

The designer should be able to explicitly define the order of creations by assigning an index to each creation. Further-

more, it should be possible to mix object and link creations by defining a global order of creations. Thereby, the complete creation of more important structures could be favored over other creations.

For the insertion of objects into ordered associations at specific positions, the current code generation has to be fixed. In order to enable the insertion of whole sequences of objects, the semantics definition as well as the code generation would have to be extended.

3.2 Matching links between sets

Story Patterns allow the binding of an arbitrary number of objects of the same type to a single multi-object or set variable [4]. Set objects greatly simplify the manipulation of multiple objects. Creating a link to a set object, for example, results in the creation of a link to each contained object. The change of an attribute value specified at a set object is done to all its members.

3.2.1 Problem Description

The class diagram in the upper part of Figure 3 defines three classes A, B, and C. Instances of B may be connected to an arbitrary number of instances of A and C via associations ab and bc, respectively. Furthermore, A and C participate in the n-to-n association ac.

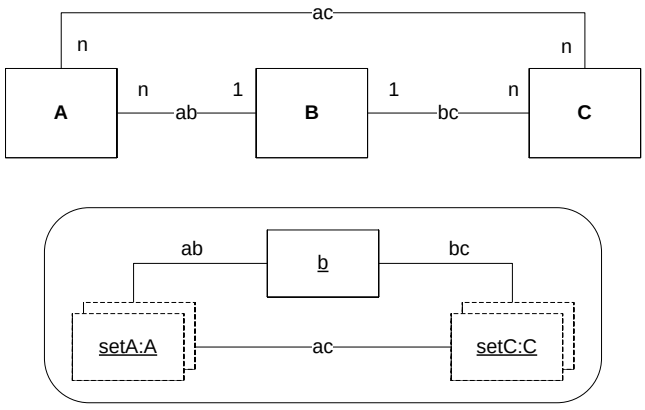


Figure 3: A link between two sets

The Story Pattern shown below the class diagram binds all objects of type A which are connected via ab to the bound object b (of type B) to the set object setA. All objects of type C which are linked to b via bc are bound to the set object setC.

In addition, a link of type ac is required between the two set objects. The problem here is, that the meaning of this link between the two sets is not clear. Since there is an intuitive understanding which is neither supported by the formal semantics nor by the compiler semantics (cf. below) we classify this problem as an *unexpected error*.

3.2.2 Intuitive Semantics

In fact, there are at least two interpretations of a link between two set objects which are equally intuitive:

1. the link requires each object in one set to be connected with all objects in the other set, which we call a *total* connection between the sets.
2. the link requires that each member of one set is connected with at least one member of the other set, which we call an *any* connection between the sets.

If the underlying association is bidirectional, the above definitions hold in both directions.

3.2.3 Current Semantics

The semantics definition given in [4] explicitly forbids links between sets. However, Fujaba allows their specification but the current code generation is not able to handle them and aborts throwing an error message.

3.2.4 Possible Solutions

The two intuitive interpretations described in 3.2.2 could be formally defined and supported by the code generation. The desired interpretation of a particular link could be specified by assigning a stereotype - ≪total≫ or ≪any≫ - to it.

3.2.5 Similar Problems

In case of the Story Pattern presented in Figure 3, the two sets are bound independently starting from b and the link between them has to be checked afterwards. However, the same interpretations for links between sets are applicable to the binding of one set starting from another set as well. This would require additional code generation strategies.

When more than one link is specified between the same two sets, the connections between all their members have to be in accordance with all specified links. In order to determine the sets, they have to be filled initially. Then, for each link, all objects which are not connected appropriately are removed from the sets. This process is repeated until both sets are stable which has to occur eventually since no objects are added anymore.

Other situations, such as more than two connected set objects or when single objects can be bound only via a (chain of) set objects, require further investigation.

3.3 Qualified Associations

Qualified associations resemble data structures which use keys to store and retrieve values, e.g. hash tables. Fujaba supports two types of qualified associations. The first type implicitly uses an attribute of the associated class as key. In addition, association roles can be used as keys. The second one allows free keys which must be explicitly specified for the links in the story patterns.

3.3.1 Problem Description

Figure 4 shows an abstract example of three classes A, B, and C. The classes A and C are associated with B via two qualified associations of the first type using the role attributes as keys. If we consider the Story Pattern in the lower part of Figure 4, we have the following problem concerning the link creation: If the link between objects a and b is created first, the key b.c is null but this key is used to add object b to the qualified association. Only after the link between b and c is created, the role attributes in b have the correct value. But then the null-key is already used for the first link. The problem source is the usage of specific access methods for the create link of the qualified association. These access methods implicitly call the read access method on the qualifier which return the wrong or null key.

We classify this problem as *unexpected error* according to the classification in Section 2.

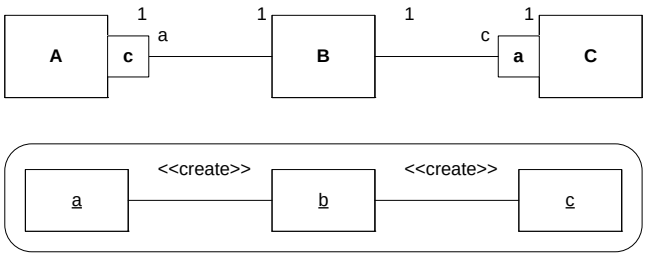


Figure 4: Double Qualified Associations

3.3.2 Intuitive Semantics

As both qualifiers are bound in the Story Pattern, a correct creation of the links is indeed possible. The implicit call of the read access method on the qualifier should be replaced by simply using the known bound qualifier.

3.3.3 Current Semantics

In the current implementation, the wrong code implicitly using the read access methods on the qualifier are used. There is no semantics given in [4]

3.3.4 Possible Solution

The intuitive semantics should be implemented.

3.3.5 Similar Problems

If we only consider a single qualified association with a qualifier based on another association, there is a similar problem. Here, the order of the different link creation actions is important. If all normal links are created before the qualified link, everything is fine. If not, the wrong key (null) would be used. A related problem occurs when links are deleted. For the special case of the `removeYou`-method this behavior has been implemented recently.

4. ISSUES IN STORY DIAGRAMS

4.1 For-Each Activities

For-each-Activities are used to perform activities on all possible bindings. Basically, they describe a loop where *for each* binding the loop body is executed.

4.1.1 Problem Description

The problem is here what happens when the loop body changes the set of possible bindings by creating or deleting elements.

4.1.2 Current Semantics

In [4], a *fresh matches* semantics is employed for for-each loops. The *fresh matches* semantics means that if during execution of the for-each loop new bindings are created, those bindings are additionally matched in the loop condition. Thus, the loop may not terminate which is not feasible.

We classify this semantics as *counterintuitive* as for-each suggests an independent processing (without side effects in between the different body incarnations and no side-effects between the body and the initial set of bindings) according to the classification in Section 2.

Unfortunately, the *fresh matches* semantics is not always correctly implemented in the current version of Fujaba. The behavior is dependent on the employed association implementation. If for example a linked list is used in the implementation of a to-many association, new objects are always placed at the end of the linked list. The linked list is used for iterating over all elements in the to-many association in the binding process. Thus, newly created bindings are matched in the for-each loop. Actually, linked lists are used in combination with hash sets for ordinary to-many associations to guarantee that new elements are found in the bindings.

Sorted and ordered associations are problematic as new links can be arbitrarily inserted into both association's data structures. Thus, a newly created link can potentially be inserted before the current iterator element in the data structure. Consequently, this potentially new binding will not be found in the for-each loop.

In a hard real-time system, the compliance with certain deadlines is of utmost importance. If Story Diagrams are employed in hard real-time environments, a worst case execution time must be known in order to guarantee that the deadlines are satisfied. The fresh matches semantics is thus not suitable in the context of hard real-time systems as new bindings are also processed by the for-each loop and thus the execution time may be arbitrarily prolonged (cf. [3]).

4.1.3 Intuitive Semantics

It is debatable whether the employed *fresh matches* semantics is intuitive. Zündorf decided to employ the fresh matches semantics but said that iterated story patterns should not create new matches.

> "To avoid these kinds of vague semantics one should not write iterated story patterns that create new matches or that destroy more than one match at a time. If this rule is regarded, the semantics becomes clear and simple. Due to our experience, it is easy to respect this rule in practice and iterated story pattern have proven to be very handy in various situations."

Consequently, we propose that new matches should not be regarded in for-each loops.

4.1.4 Possible Solution

Zündorf even mentioned a different semantics, the *pre-select* semantics [4]. Here, all possible bindings are stored before the application of the for-each loop. Newly created matches are then not considered. Unfortunately, this behavior requires additional memory in order to store all matches bindings. We propose to employ the *pre-select* semantics [4] to avoid the inherent problems of the *fresh matches* semantics.

Another solution could be to provide a (probably pessimistic) heuristic which can decide whether the story patterns in the for-each loop can potentially create a new binding. An easy approach is to forbid the creation of the objects and links which can create a new binding. Though, this approach may be overly pessimistic.

4.2 Unsuccessful Creations

Story Patterns allow the specification of creation of links and objects. The creation of a link typically also involves the creation of objects inside the employed data structure and its accompanying algorithms. If Story Patterns are employed in environments with tight resources, this object creation may fail. This problem is often neglected in standard

applications, too. Even there, the memory has a fixed upper bound and, consequently, an object creation may also fail.

4.2.1 Problem Description

If a Story Pattern contains a set of object and link creation operations, any one of these object creations may fail leaving the graph in an undefined state. Unfortunately, the developer has no chance to react to or even know about a failed object creation.

We classify this problem as *incomplete* and *counterintuitive* according to the classification in Section 2. The semantics is counterintuitive as the execution of a Story Pattern is regarded as atomic. The semantics is incomplete as failures are not appropriately considered.

4.2.2 Current Semantics

In the current formal semantics, all object creations and deletions are executed whenever the left hand side matches. Failures are not considered. The code generation resembles this formal semantics. Tough, the code generation implies a certain ordering of the creation and deletion actions. Consequently, if a failure occurs during execution of those actions, all actions prior to the failed one are executed successfully, while the remaining ones are not executed due to the thrown OutOfMemoryError in case of the current Java code generation. As the generated code does not catch the OutOfMemoryError the Story Diagram is left, too.

4.2.3 Possible Solution

We propose creation checks as additional syntactic elements (see Figure 5 from [3]). A creation check is executed after binding the left hand side of a Story Pattern. If the creation check fails, the Story Pattern is left via a creation check failure transition. The developer can then react to a creation check failure e.g. by removing other objects and/or links. Furthermore, we recommend refinements for the creation check failure transitions which allow to distinguish for which object or link the creation check failed.

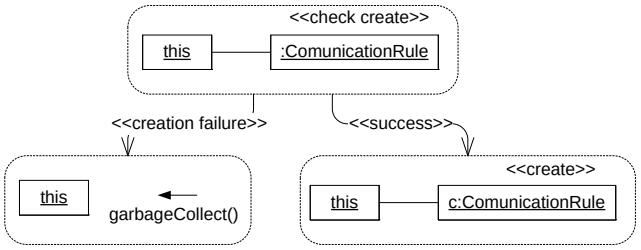


Figure 5: Creation check syntax proposal.

An alternative way of handling the creation problem could be to adopt transaction behavior (Atomicity). Then, the atomicity property guarantees that either all or none of the creation operations are executed. This naturally also holds for deletions of objects or other transformations from the left to the right hand side of the Story Pattern. If the Story Pattern is left via the success transition, a binding has been found *and all* transformations have been successfully applied.

5. CONCLUSIONS

We discussed several semantic issues of Story Diagrams in this paper. While some are easily addressed like the presented problem of qualified associations, others need a more thorough discussion in order to come up with a suitable solution (see for-each loops). For these problems, we gave a starting point for subsequent discussions. Maybe, different solutions are required for the different domains in which Story Diagrams are used. Nevertheless, we believe that all of those problems are solvable. Then, appropriate solutions increase the power of Story Diagrams even further.

6. REFERENCES

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

[2] 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, pages 296–309. Springer Verlag, November 1998.

[3] M. Tichy, H. Giese, and A. Seibel. Story Diagrams in Real-Time Systems. In *Proc. of the 4th International Fujaba Days 2006, Paderborn, Germany*, September 2006. submitted.

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

Story Diagrams in Real-Time Software

Matthias Tichy, Holger Giese, and Andreas Seibel
Software Engineering Group
University of Paderborn, Germany
[mtt|hg|aseibel]@uni-paderborn.de

ABSTRACT

Software plays an increasingly important part in todays embedded systems. Development efforts for embedded software must consider the trade-off between fast development, maintainable code, correct as well as high-performance software. Graphical object-oriented languages can help in creating more maintainable code, while also providing better means to ensure correctness. Predictable real-time behavior w.r.t. the execution time of operations is additionally of particularly importance in the embedded domain. We present in this paper a graphical language employing graph transformations as a formal foundation. The language is especially geared for event-driven transformations of data structures. Additionally, we present our approach for worst case execution times estimation to predict software behavior in the time domain. We evaluate our approach using an example from the railway domain.

1. INTRODUCTION

Software plays an increasingly important part in todays embedded systems. As many embedded systems are used in a safety-critical context (e.g. cars, medical systems, trains), they must adhere to strict quality requirements. The software must not only work correctly, but also be predictable w.r.t. the execution time of operations. Additionally, product cycles are becoming shorter. Thus, software development has to finish in even shorter time frames.

Many approaches try to tackle the aforementioned problems. Block diagrams are employed for a graphical specification of control algorithms in tools like Matlab/Simulink. Object-oriented and graphical languages are used to specify event-driven software with complex object structures. They can in principal address the aforementioned problems but are seldom used in embedded software development. This stems partly from problems in ensuring worst case execution times (WCET) which is essential for correctness w.r.t. real-time specifications.

Graph transformations are used in many variants as a language for expressing structural transformations [17, 18, 7, 21]. Several approaches exist [16, 1] for ensuring the functional correctness of graph transformations. As mentioned above, WCET computation is required for ensuring correctness w.r.t. real-time specifications but current approaches for WCET estimation like [20, 5] are not applicable for software which executes arbitrary data structure changes.

In previous works, we have shown (1) how to estimate and optimize WCET for a graph transformation variant called Story Pattern [4], and (2) how to formally verify inductive invariants on sets of Story Patterns [1]. Both approaches help to develop software which satisfies strict quality requirements. In this paper, we draw on these previous works and present a graphical language for event-driven structural transformations whose usage is feasible in embedded and safety critical software development.

In the next section, we present the railcab research project[1] and the running example of this paper which stems partly from the railcab project. We describe the different parts of our language in Section 3. In Section 4 we introduce our approach for WCET estimation of programs which are specified in our language. Section 5 employs an evaluation which compares our estimated WCET with actually measured worst case execution times. In Section 6, we review some related approaches. We conclude the paper in Section 7 and present research directions for future work.

2. EXAMPLE

A new transport system called Railcab is developed in Paderborn. The transport system utilizes autonomous vehicles which drive on a nearly standard railway system. Communication is required between the shuttles for coordination purposes, for example for building convoys to reduce the air resistance and thus the general power consumption.

In previous works [9, 4], we addressed a situation as shown in Figure 1 when two shuttles move towards the same joining switch. The shuttles need to coordinate how to pass the switch in order to avoid a possible collision. Obviously, this has to be finished before they reach the switch. Thus, this coordination problem is subject to hard real-time requirements. This simple scenario considers only two shuttles approaching a single switch.

Train stations typically employ several switches and numerous shuttles which concurrently drive in the station area. Consequently, we need to consider a scenario with multiple switches and multiple shuttles. Additionally, shuttles have to communicate and coordinate not only with shuttles which they meet at the next switch but also with shuttles which they may meet at later switches. Figure 2 shows the extended scenario based on the above stated requirements.

The embedded software searches an internal data model for shuttles which it may meet two tracks ahead in order to initiate a communication and coordination between the shuttles. The communication, required to keep this data model up to date, is out of the scope of this paper. Details can be found in [9, 10].

[1] www.railcab.de/en

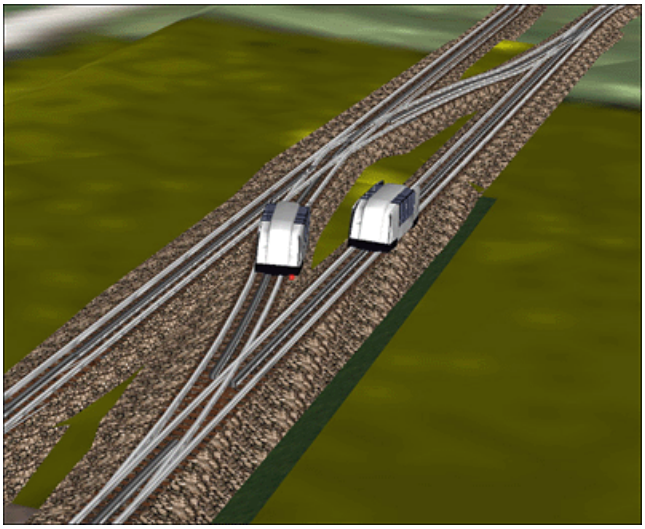


Figure 1: A possible collision of two shuttles at a switch

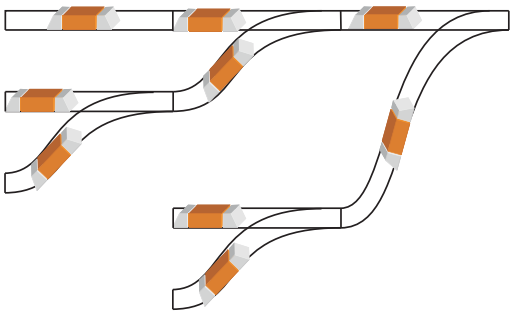


Figure 2: Extended scenario

In the following section, we present the language which can be used to specify the structure and behavior of the software for this task.

3. LANGUAGE PRESENTATION

We present the language constructs for structural specification as well as behavioral specification for real-time systems. We start with the specification of the structure in Section 3.1. Then we continue describing the behavioral parts of the language in Section 3.2.

3.1 Structural Specification

We employ UML class diagrams [14] as a standard notation for the specification of structures. Because our approach must be applicable in the embedded system domain, we have to deal with restricted computation capabilities, especially less memory.

Consequently, we refine UML class diagrams in order to employ them in this embedded domain. In embedded systems, it is required to know the exact amount of required memory beforehand to avoid unsuccessful memory allocation. The number of class instances and the multiplicities of associations are the sources for unknown memory requirements in UML class diagrams.

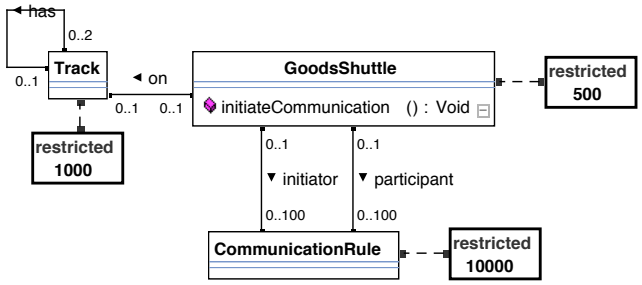


Figure 3: Extended class diagram of the depot system

It is obvious that we need to require the specification of an upper bound for the number of class instances. In the generated code, this upper bound is enforced by an implementation of the factory design pattern [8]. During initialization of a factory of a specific class, all instances of the specific class are allocated. This avoids the allocation of heap memory at runtime. The factory is then employed to *create* new class instances and reuse *deleted* class instances.

Figure 3 shows the UML class diagram refinements considering our shuttle system case study. Note that the defined maximal number of instances can also be described by OCL 2.0 [15] constraints. The following OCL 2.0 constraint specifies the maximal number of instances for the class Track.

```
context Track inv:
   Track.allInstances().size() <= 1000
```

Secondly, unbounded association multiplicities lead to unknown memory requirements for the data structures which implement the association. Thus, we prohibited the specification of unbounded association multiplicities and require fixed ones. In addition, the generated code for the association uses the upper bound to provide a static size of the data structure at the code level.

3.2 Behavioral Specification

Based on the structural model, we use the Story Diagram [7, 22] formalism for the specification of structural transformations and the control flow between the transformations. Story Diagrams are an extension of UML activity diagrams. In addition to standard UML activity diagrams, activities in Story Diagrams can be Story Patterns which specify structural transformations. Story Patterns are based on the graph transformation formalism [17]. The class diagrams presented above resemble a type graph. Story Patterns are the formalism which are used to transform instances of this type graph.

3.2.1 Real-Time Story Diagrams

We have shown how Story Patterns can be used in a real-time environment in [4]. In the following, we present a number of extensions in order to use Story Diagrams in a real-time context.

3.2.1.1 Transformation Feasibility Check.

The basic requirements of Real-Time Story Patterns are fixed maximum multiplicities for association roles and class instances in the class diagram. These requirements are

needed for both guaranteeing a fixed WCET and ensuring memory requirements.

Creations of links and objects are specified in the right hand side of a Story Pattern. We want to support the standard way of specifying instance and link creations in Story Patterns but must cope with creation failures when the above mentioned constraints would be violated. A creation failure leads to the situation where the left hand side has been successfully matched but the matched subgraph is not or only partly transformed to the right hand side.

We propose a syntax element which is used for the specification of checks whether an object or an link can indeed be created. This check is done during matching the left hand side. Only if all creation checks succeed in addition to the normal binding, the left hand side of the Story Pattern is considered to be matched. Thus, it is guaranteed that a subsequent creation of an appropriate object or link will be successful.

We add a new transition type Creation Failure to Story Diagrams to give the developer the possibility to react on a failed creation check. That transition is only taken when the left hand side has been matched, but at least one creation check has failed (see Figure 4).

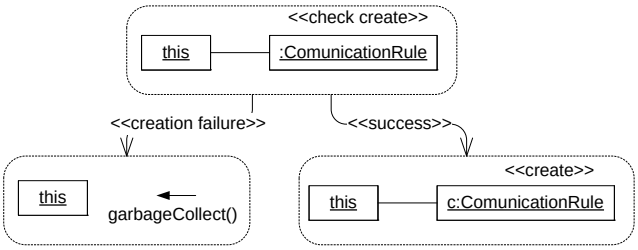


Figure 4: Creation Check Extension

Another solution is to implicitly execute the creation check before each object or link creation. We decided against that solution in order to give the developer explicit control about the situation. We are currently evaluating whether we need different transitions for different failed link and object creations. Then, the developer can specify different behavior for each possible creation check failure. We are currently in the process of adding the proposed syntax element into Story Diagrams and its accompanying code generation as well as integration into the worst case estimation.

3.2.1.2 Loops.

UML activity diagrams allow the specification of loops in the control flow. As Story Diagrams share the control flow concepts of UML activity diagrams, loops can also be specified in Story Diagrams. It is often required that one or a set of graph transformations should be applied to all possible bindings of a left hand side of a graph transformation. For this special case, the special loop concept for-each has been introduced in Story Diagrams. The for-each loop is executed for all possible bindings of the left hand side (see Figure 5).

Loops impose a challenge for WCET estimation as the WCET of a Story Diagram obviously depends on the maximal number of loop iterations. In the general case, the maximal number of iterations need to be obtained from the loop code. Fortunately, the for-each-loop construct allows us to compute the maximal number of iterations as this corresponds to the

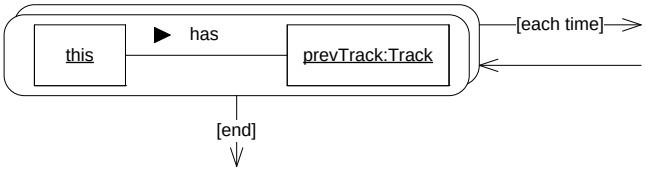


Figure 5: Example of a for-each-loop

maximal number of bindings for the left hand side of the for-each Story Pattern. The maximal number of bindings is computable from the specified maximal multiplicities of associations and classes which we already required due to memory constraints (see Section 3.1). Considering Figure 5, we compute that at most two bindings can exist for the left hand side based on the association's multiplicity specified in the class diagram. Consequently, the activities which are reached via the each time transition can only be executed two times.

In [22], a *fresh matches* semantics is employed for for-each loops. The *fresh matches* semantics means that if during execution of the for-each loop new bindings are created, those bindings are additionally matched in the loop condition. Thus, the number of loop iterations cannot be deduced from the class diagram and the loop may even not terminate at all. This behavior is obviously not suitable in the context of real-time systems. Thus, we currently prohibit that the activities inside the for each loop do create additional bindings. Additionally, we propose to employ the *pre-select* semantics [22] to avoid the inherent problems of the *fresh matches* semantics which are presented in detail in [19].

We allow other loops in the control flow, but we rely then on input from the developer about the maximal number of iterations.

3.2.1.3 Example.

The Story Diagram specifies how to find possible hazardous situations which have to be avoided. Possible collisions of two shuttles are hazardous situations. A situation needs to be detected where still enough time is left to initiate avoidance mechanisms. We simplify this by declaring that enough time is left, if two shuttles may meet at one of the next two switches.

To avoid a possible collision, we require that a shuttle which is detecting the possible hazardous situation, creates a communication rule. This communication rule guarantees, that no collision happens. The complete collision avoidance is not part of this paper, but details can be found in [11].

3.2.2 Real-Time Story Charts

Story Charts [13, 22] are a formalism which blends the specification of event-based behavior (e.g. Statecharts [12], UML state machines [14]) with the specification of structural transformations based on graph transformations. The structural transformations are executed in order to react to incoming events.

In previous works, we refined UML state machines for the specification of event based behavior in the hard real-time domain [2, 3] into the Real-Time Statechart formalism. In order to blend Real-Time Statecharts with graph transformations similar to Story Charts, we can simply use the above described Story Diagrams as side effects of transitions

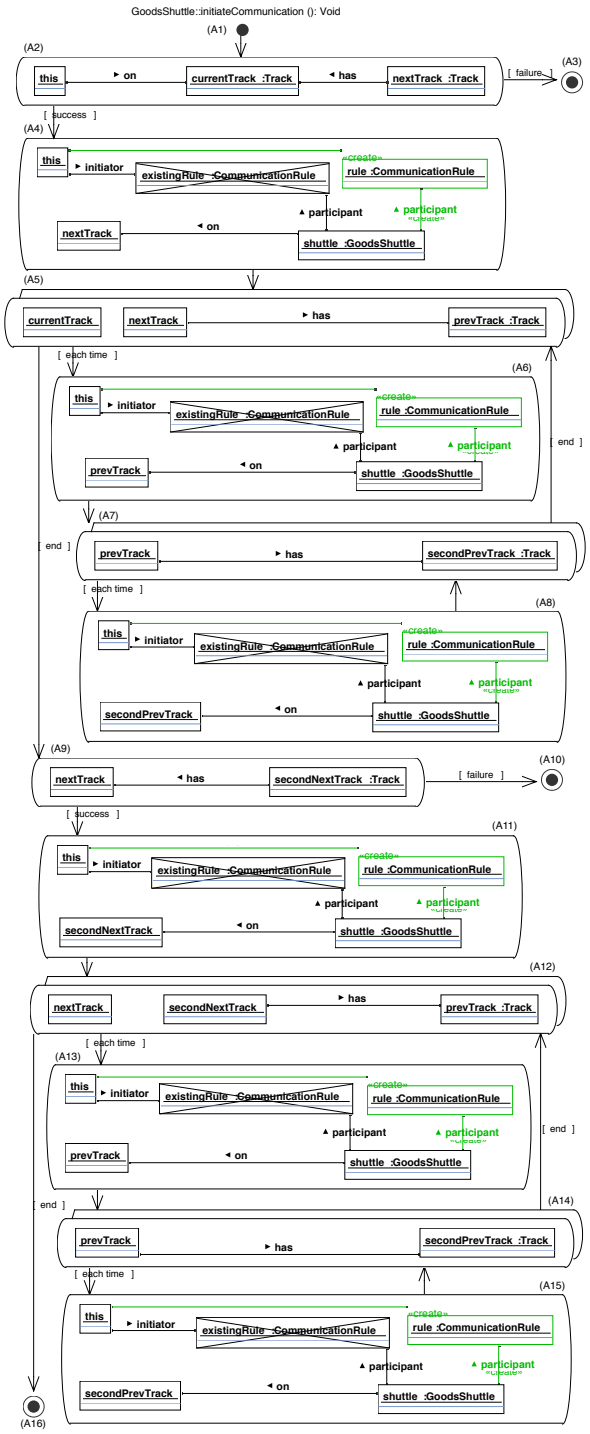


Figure 6: Story Diagram for collision avoidance

as well as entry and exit actions.

The Real-Time Statechart formalism requires that for each side effect the WCET is known. If this requirement is fulfilled, schedulability of the specified behavior can be checked and code can be subsequently generated. In the following, we present how WCET's can be estimated from Story Diagrams.

3.2.2.1 *Example.*

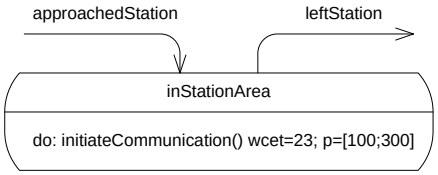


Figure 7: Part of the Real-Time Story Chart Diagram for the shuttle behavior

Figure 7 shows an extract of the event-driven behavior of a shuttle. The state inStation is via an incoming approachedStation event. This event stems from another component of the shuttle. The initiateCommunication story diagram of Figure 6 is executed periodically (every 100 to 300 ms) as long as this state is the current state. The state is left after the leftStation message is received.

4. WORST-CASE EXECUTION TIMES ESTIMATION

The WCET estimation of Story Diagrams is composed into two parts. First, the WCET of every Story Pattern is estimated. In previous work [4], we have presented the analysis steps which are required to estimate a WCET for Story Pattern. This approach is based on a platform specific *profile* which contains WCET's for a set of basic operations like checking whether an object is contained in a linked list or not. Our *profile tool* does a fully automated measurement for creating this profile before analyzation – at the moment it supports C++ and the phyCORE-MPC555 board. Another possibility to obtain these WCET's is to use a special tool like the aiT Worst-Case Execution Time Analyzer [20]. It may also be gathered from several tests on the target platform under certain circumstances. The WCET of the complete Story Diagram can then be computed based on the individual Story Pattern's WCET's.

But the WCET of every Story Pattern only, is not sufficient. We need the worst case number of possible bindings of loop condition Story Pattern in order to compute the number of iterations of the for-each loop. The worst case number of possible bindings is identical to the worst case number of iterations (WCNI) described in [4]. For other loops, the maximal number of loop iterations specified by the developer is used (see Section 3.2.1.2).

Figure 8 visualizes the entire WCET estimation process.

4.1 WCET Estimation for Story Diagrams

After the first step of estimating the WCET and the WCNI of every Story Pattern, the analysis can proceed estimating the WCET of the Story Diagram. Our static approach for estimating the WCET of Story Diagrams is path based on the control flow of the Story Diagrams. Story Diagrams are an extension of UML activity diagrams. As UML activity diagrams, every Story Diagram consists of exactly one start activity and several stop activities. Between these activities there are story activities, which could be Story Patterns or branches (NOP Activities), connected by transitions which

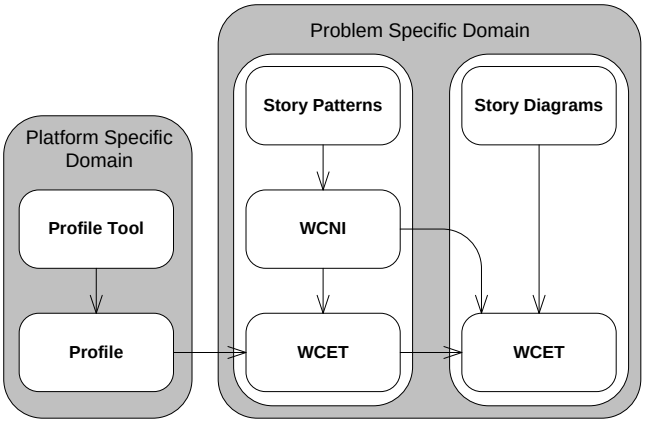


Figure 8: Analysis process

describe a control flow (see Figure 6). Other approaches using low level path based analysis have to extract a control flow graph (CFG) from the executable assembler code, for example [20]. Our model already implies a CFG with all necessary informations for WCET estimation.

To estimate the WCET of every stop activity, we use a recursive algorithm which estimates the WCET's for the longest path from the start activity to each stop activity by summing the WCET of every Story Pattern on any path. The path with the maximum WCET then determines the WCET which is associated with the stop activity.

Loops are recognized by the algorithm. We use the number of possible bindings of the loop condition Story Pattern (the for-each Story Pattern) in order to determine the number of loop iterations. This information is given by the WCNI of a for-each Story Pattern which is the initiator of a loop. A for-each Story Pattern has two outgoing transitions: (1) every time a binding has been found for the for-each Story Pattern, the each-time transition is taken, (2) if all bindings have been processed, the end transition is taken.

For a non-nested loop the WCET of the for-each Story Pattern and all contained Story Patterns is simply multiplied by the number of loop iterations.

Loops can be nested. Thus, we additionally need to recursively increase the WCNI of the inner loop by multiplication with the WCNI of the outer loop. This principle is called for-each compensation and is depicted in Figure 9.

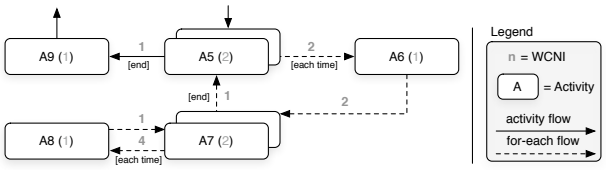


Figure 9: Example of the for-each compensation process

This figure shows a simplified version of the Story Diagram of Figure 6. The activities are numbered from the top to the bottom with A1 to A16. The for-each compensation process maps the WCNI's of the for-each initiator Story Patterns to the transitions of the according for-each flows. Initially, every transition has a WCNI of 1. The for-each compensation multiplies the WCNI of the for-each Story Pattern with every transition on the for-each flow except three situations. First, the end-transition is ignored. Second, transitions which lead back to the initiating for-each Story Pattern are ignored. At last, transitions that go out of another for-each Story Pattern are ignored. The WCNI is multiplied with the WCET of a story activity which is part of the for-each loop.

Because in some cases the user does not want to iterate over all matched instances on the left hand side of a for-each Story Pattern, the user may define an object as already bound as done in our example in Story Pattern A5 and A12 with currentTrack and nextTrack. This has, for example, the effect that the matchable objects for nextTrack contain an object that is equal to the currentTrack in Story Pattern A5. Due to the isomorphic binding of Story Patterns, this object will not considered in further matchings of the Story Pattern again. This reduces the possible number of matchable objects which implies a reduction of the number of iterations of the Story Pattern. Our analysis recognizes such patterns and corrects the WCNI of the affected Story Pattern automatically.

5. EVALUATION

Because we do not consider the issues of multi-threading, process scheduling and processor caching in our approach, we needed an appropriate hardware and software platform. Consequently, we used a phyCORE-MPC555 with integrated 40MHz 32-Bit PowerPC micro-controller from Motorola and the embedded operating system DREAMS[2][6] for the runtime measurements of our example of Section 2. We use the Story Diagram of Figure 6 as the scenario for our evaluation.

5.1 Worst-Case Instance Situation

To show the quality of our WCET estimation of Story Diagrams, a worst-case instance scenario has to be created in order to measure the WCET of the executed example application. Therefore, an initial instance situation has to be created in a way that every Story Pattern of the Story Diagram of Figure 6 executes as most as possible operations. It is not enough that the left hand side is successfully matched and the right hand side successfully executed. Also, the employed data structures have to be initialized in a way that searching an element results in the data structure operation's WCET. Figure 10 shows a snapshot of the initial instance situation of our generated example application.

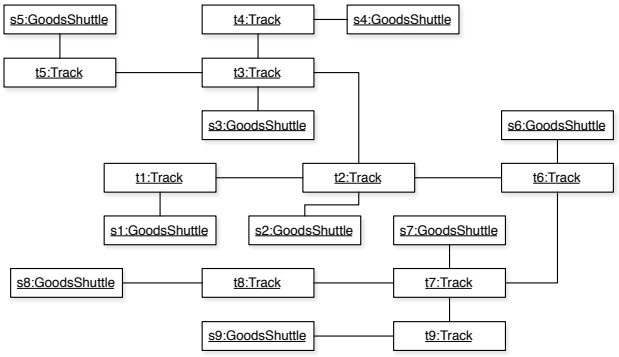


Figure 10: Initial worst-case instance situation

[2]Distributed Extensible Application Management System

For a clearer presentation, Figure 10 does not include objects of the type CommunicationRule. GoodsShuttle s1 is the initiator of 92 other CommunicationRule objects. This leads to the WCET for the creation of a link between s1 and another CommunicationRule object since link creation implies searching for a possible duplicate in the data structure. We cannot initially create 100 CommunicationRule objects because the Story Diagram is going to add 8 more CommunicationRule objects to the associated data structure of the initiator association of s1 during successful execution.

Normally, we should leave 14 slots free of the initiator's associated data structure in s1, but we use the optimization concerning isomorphic checks (see Section 4.1). The Story Pattern A5 and A12 use isomorphic checks to prevent further infeasible matches like currentTrack in Story Pattern A5 which is already bound in Story Pattern A2 and checked for a possible collision in Story Pattern A4. Therefore, nextTrack would be matched in Story Pattern A5 and then checked for a possible collision in Story Pattern A6 again. The for-each loops initiated by Story Pattern A5 and A12 therefore only match one object of the type Track instead of two possible matches. The analysis recognize the isomorphic matches and reduce the for-each iteration number by the number of isomorphic situations in the initiator for-each Story Pattern. Thus, only 8 CommunicationRule objects can be created and added to the initiator associated data structure of s1 during execution of the Story Diagram.

Also every GoodsShuttle object except s1 is a participant of 99 other CommunicationRule objects. Those CommunicationRule objects are not connected by an initiator link with s1. This results in the WCET for creating a participant link between a GoodsShuttle object and a CommunicationRule object and creating a link between this CommunicationRule object and s1. Additionally, this implies the worst case for the negative application condition (NAC) check between a GoodsShuttle and the existingRule object of the type CommunicationRule.

The Story Patterns A4, A6, A8, A11, A13 and A15 of Figure 6 require a situation where another GoodsShuttle object on a neighboring Track object is located which has a participant link to another CommunicationRule object. Thus, most of the code of the NAC is being executed, but not finished with true. This means that the left hand side can be matched with most of the checks being executed. Thus, every GoodsShuttle object except s1 is a participant of another CommunicationRule object.

5.2 Analysis and Measurement

The evaluation has to show the quality of our WCET estimation from our analysis in direct comparison to the measured WCET of the example application. We present three of our experiments with the Story Diagram of Figure 6.

In the first experiment, we looked at the WCET of both the left hand side and the right hand side (LHS + RHS). The second experiment does only consider the left hand side of the Story Diagram without the transformation concerned by the right hand side (LHS). The difference between these experiments is the WCET of the right hand side only (RHS).

Every experiment has been executed 10 times. In every step, we increased the multiplicity's of the associations participant and initiator at the CommunicationRule side by 10. We started with multiplicity 1 and finished with the multiplicity 100. The experiments with multiplicity 1 must be distinguished from the ones with multiplicity > 1, because one-to-one associations and different matching sequences as well as other operations are used inside the NAC's binding objects and during the execution of the right hand side (cf. [4]).

We only increase the multiplicity's of the association initiator and participant. This has the effect that the WCET is increasing linear with increasing multiplicity, because we do not increase the number of nested loop iterations. Thus, we can interpolate the analyzed and measured data as linear functions of the form $f(x) = a \cdot x + b$ with $x \in \mathbb{N} \backslash \{0, 1\}$. We introduce three linear functions which describe the WCET of our analysis. $a(x) = 0.2321 \cdot x + 0.5035$ which is the analysis of the complete example, $a_l(x) = 0.2045 \cdot x + 0.2577$ describes the analysis of the left hand side only and $a_r(x) = 0.0276 \cdot x + 0.2458$ describes the right hand side only.

Also the measurement data of every experiment can be interpolated. Thus, we present the experimental results as a linear function. $m(x) = 0.1968 \cdot x + 0.4447$ which is the WCET of the complete example, $m_l(x) = 0.1695 \cdot x + 0,2355$ is the WCET of the left hand side only, and $m_r(x) = 0.0273 \cdot x + 0.2092$ is the WCET of the right hand side only.

Figure 11 plots these functions in a cartesian coordinate system. The x-axis is the multiplicity of both associations initiator and participant. The y-axis is the WCET in ms.

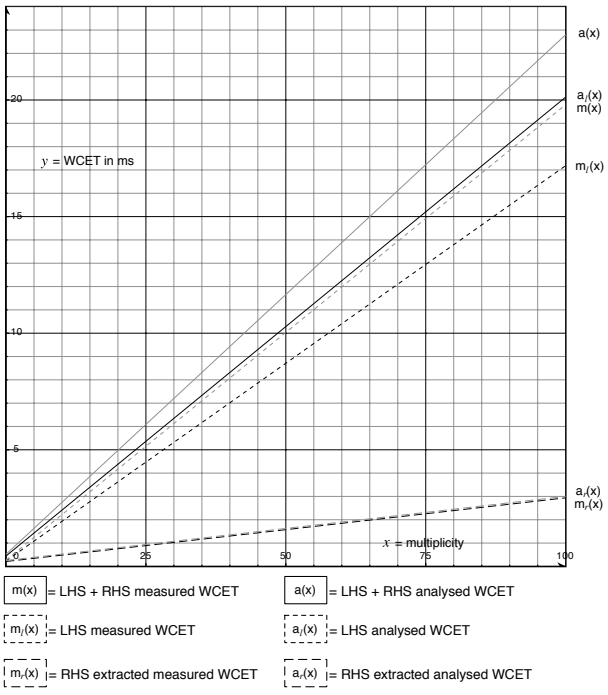


Figure 11: Evaluation chart

To get a better overview about the cornerstones of the analyzed and measured data, Figure 12 shows the WCET of every experiment at the first- and last step only.

5.3 Evaluation Conclusion

The difference between the experiments with $x = 1$ and $x > 1$ is a result of different operations which are used for matching and transformation as described in Section 5.2. For the $x = 1$ case we see a remarkable smaller relative difference between the measured and analyzed RHS exper-

	Measurement	Analysis	Difference (A-M)
LHS + RHS (x = 1)	0.643 ms	0.686 ms	0,043 ms (6.6%)
LHS (x = 1)	0.323 ms	0.324 ms	0.001 ms (0.03%)
RHS (x = 1)	0.320 ms	0.362 ms	0.042 ms (13.1%)
LHS + RHS (x = 100)	20.124 ms	22.788 ms	2.664 ms (13.2%)
LHS (x = 100)	17.171 ms	19.772 ms	2.601 ms (15.1%)
RHS (x = 100)	2.953 ms	3.016 ms	0.063 ms (2.1 %)

Figure 12: Evaluation table

iment as the $x > 1$ case shows. This may occur because at the moment we do not have optimized every code fragment of the profile measurement. We lose 5.25 μs per Story Pattern which contains a right hand side. This overestimation is caused by three operations: one object creation and two link creations. The object creation operation is already optimized in the profile tool, so that the overestimation per create link operation should be 2.625 μ. The LHS experiment of the $x = 1$ case does only contain optimized operations. The $x > 1$ case does contain more unoptimized operations than the $x = 1$ case. Several code fragments contain too much measured code, at the moment. This could be a reason for the overestimation of the LHS experiment in the $x > 1$ case. Different to the RHS of the $x = 1$ case, the $x > 1$ case does only use optimized code fragments which should be the reason for a lower aberration between measurement and analytical results. The existing aberration for every experiment of the $x > 1$ case could be the time which is caused by the instance situation not being the worst-case (cf. Section 5.1).

In general, it has to be said that the clock of the system is not as precise as needed. A tick with has a execution time of 800 ns. Some code fragments oscillate between one and two ticks which cause roundoff errors. The code fragment WCET estimation optimization is reducing this error by executing the code δ times. With increasing δ, we are able to get a refined timing with decreasing roundoff errors. Our evaluation used $\delta = 100$ for the code fragments.

6. RELATED WORK

Several approaches and tools for WCET estimation, like aiT from AbsInt[3], already exist. Most of them handle low-level analysis to estimate the WCET of executable code [20]. These approaches have to obtain flow-graph analysis from the executable code in order to appropriate cycles and recursions in the code flow. The user can obtain upper bounds for determined cycles. Also the code language is often restricted to procedural languages like C in aiT, for example. Using executable code offers a finer granularity in WCET estimation because parts of the flow-graph can be described more detailed. Our approach obtains the WCET for whole code fragments which is a well defined sequence of code. The main aspect of our approach is to obtain an high-level analysis to appropriate WCET's. Therefore obtaining basic WCET's from the code fragments is satisfying our necessities. The input of our analysis is the model which is an equivalent to the flow-graph and a target profile containing the code fragments WCET's.

Curatelli and Mangeruca present in [5] a method to compute the number of iterations in data dependent loops. They present a formal model for the loop condition as well as the loop body. The formal model of the loop body is very expressive due to a set of loop index counters and transformations of the loop index counters which include other loop index counters on the right side of the assignment.

The body of the loops of Story Diagrams as well as the generated loop bodies for binding the left hand side of Story Pattern can be mapped to this formal model. For example, the binding-loops of Story Pattern can be mapped to a set of loop index counters for each to-many association. In the loop body this index counters are then increased. Unfortunately, the formal model of the loop condition is not expressive enough for our task as the loop condition does not allow to set individual maximum values for each index counter but only a linear constraint containing all index counters.

7. CONCLUSIONS

Embedded software must satisfy strict quality requirements. i.e the software must work correctly w.r.t. functional requirements as well as real-time requirements. Many different textual and graphical approaches exist to help in developing those software. We presented an approach for the special case of graph transformations in embedded real-time systems.

Our approach contains means for structural specification in form of refined UML class diagrams. Structural transformations are specified using Story Diagrams. Additionally, we propose a syntactic extension which allows to check whether creation of elements is indeed possible despite tight memory constrains.

We present how to estimate worst case execution times for Story Diagrams in order to ensure temporal correctness. The WCET estimation of Story Patterns is based on [4]. The WCET of a Story Diagram is then computed based on the Story Pattern's WCET's. Loops in Story Diagrams are specifically treated based on known maximum number of iterations of the loop. Thus, Story Diagrams can be used as side effect in Real-Time Statecharts.

The provided evaluation shows that our approach is a safe (pessimistic) approximation of the WCET's of the presented example scenario.

The presented approach is part of the Fujaba Real-Time Tool Suite. We are currently implementing the mentioned language extension concerning the checking of create operations. We offer a C++ code generation which was used to generate the code for our example scenario.

7.1 Future Work

We are currently looking into reducing the difference between the measured execution time and the estimated one. One aspect is the refinement of the profile which is used to compute the WCET of a Story Pattern.

The estimation can be improved by an improved analysis of the behavior of the Story Pattern. For example, if an object is created in the Story Pattern, there are two cases to consider: (1) if the object creation of a class A succeeds, only maxInstances-1 instances of A can be matched in the left hand side and (2) if the object creation fails, the right hand side would not be executed at all. Another example is a series of object creations in different Story Patterns of a

[3]http://www.absint.com/ait

Story Diagram. It remains to be seen whether an improved analysis scales for bigger Story Diagrams.

The used data structures (linked lists) could be replaced by sophisticated ones which have a better execution time in the worst case. Additionally, we will continue to evaluate the approach in other scenarios in order to improve the quality of the estimation.

Acknowledgment

We thank Anatol Derksen who helped in working with the phyCORE-MPC555 running the DREAMS operating system and for his great assistance during the practical evaluation step.

8. REFERENCES

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

[2] S. Burmester, H. Giese, M. Hirsch, D. Schilling, and M. Tichy. The Fujaba Real-Time Tool Suite: Model-Driven Development of Safety-Critical, Real-Time Systems. In *Proc. of the 27th International Conference on Software Engineering (ICSE), St. Louis, Missouri, USA*, pages 670–671. ACM Press, May 2005.

[3] S. Burmester, H. Giese, and W. Schäfer. Model-Driven Architecture for Hard Real-Time Systems: From Platform Independent Models to Code. In *Proc. of the European Conference on Model Driven Architecture - Foundations and Applications (ECMDA-FA'05), Nürnberg, Germany*, Lecture Notes in Computer Science, pages 25–40. Springer Verlag, November 2005.

[4] S. Burmester, H. Giese, A. Seibel, and M. Tichy. Worst-Case Execution Time Optimization of Story Patterns for Hard Real-Time Systems. In *Proc. of the 3rd International Fujaba Days 2005, Paderborn, Germany*, pages 71–78, September 2005.

[5] F. Curatelli and L. Mangeruca. A method for computing the number of iterations in data dependent loops. *Real-Time Systems*, 32(1-2):73 – 104, February 2006.

[6] C. Ditze. *Towards Operating System Synthesis*. Number 157 in HNI-Verlagsschriftenreihe. University of Paderborn, Germany, 2000.

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

[8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns, Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[9] H. Giese, S. Burmester, F. Klein, D. Schilling, and M. Tichy. Multi-Agent System Design for Safety-Critical Self-Optimizing Mechatronic Systems with UML. In B. Henderson-Sellers and J. Debenham, editors, *OOPSLA 2003 - Second International Workshop on Agent-Oriented Methodologies*, pages 21–32, Anaheim, CA, USA, Center for Object Technology Applications and Research (COTAR), University of Technology, Sydney, Australia, October 2003.

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

[11] H. Giese, M. Tichy, S. Burmester, W. Schäfer, and S. Flake. Towards the Compositional Verification of Real-Time UML Designs. In *Proc. of the 9th European Software Engineering Conference*, pages 38–47. ACM Press, September 2003.

[12] D. Harel. STATECHARTS: A Visual Formalism for complex systems. *Science of Computer Programming*, 3(8):231–274, 1987.

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

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

[15] Object Management Group. *Object Constraint Language Specification 2.0*, May 2006.

[16] A. Rensink. Towards Model Checking Graph Grammars. In M. Leuschel, S. Gruner, and S. L. Presti, editors, *Workshop on Automated Verification of Critical Systems (AVoCS)*, Technical Report DSSE–TR–2003–2, pages 150–160. University of Southampton, 2003.

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

[18] G. Täntzer. AGG: A Tool Enviroment for Algebraic Graph Transformation. In *Proc. of Applications of Graph Transformation with Industrial Relevance (AGTIVE2000), Kerkrade, The Netherlands*, LNCS. Springer Verlag, 2000.

[19] M. Tichy, H. Giese, and M. Meyer. On Semantic Issues in Story Diagrams. In *Proc. of the 4th International Fujaba Days 2006, Paderborn, Germany*, September 2006. submitted.

[20] I. Wenzel, B. Rieder, R. Kirner, and P. Puschner. Automatic timing model generation by cfg partitioning and model checking. In *DATE '05: Proceedings of the conference on Design, Automation and Test in Europe*, pages 606–611, Washington, DC, USA, 2005. IEEE Computer Society.

[21] A. Zündorf. Graph Pattern Matching in PROGRES. In *Proc. of the* 5^{th} *International Workshop on Graph-Grammars and their Application to Computer Science*, LNCS 1073. Springer Verlag, 1996.

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

Visual Specification of Structural and Temporal Properties*

Holger Giese and Florian Klein†
Software Engineering Group, Department of Computer Science,
University of Paderborn, D-33098 Paderborn
hg|fklein@uni-paderborn.de

ABSTRACT

The UML has become the de-facto standard in software engineering. Due to the visual nature and accessibility of its structural diagrams, it is widely accepted as the tool of choice for structural modeling. However, for specifying structural properties that go beyond cardinalities, the UML only provides a textual specification language, the OCL. For mixed structural and temporal properties, only proprietary combinations of OCL with temporal logic exist today. The intricate nature of both OCL and temporal logic already causes problems for many software engineers. When communicating with people without a computer science background, e.g. domain experts, employing OCL, any dialect of temporal logic, or a mix of both is usually impracticable. In this paper, we propose a visual language for specifying requirements including structural as well as temporal aspects. Based on an extension of Story Patterns, our approach will allow specifying scenarios that contain requirements concerning structural dynamics within Fujaba. In addition, we present a scheme for turning a specification into a powerful behavioral monitor, enabling us to verify dynamic structural properties of models at run-time or in a model checker.

1. INTRODUCTION

The popularity of the UML is in large part due to its visual nature and accessibility of its structural modeling concepts. However, for specifying more detailed structural properties, the UML only provides a textual specification language, the OCL [20]. The writing of OCL properties requires that the developer translates his/her concrete ideas about the required structural properties from the familiar structural view in form of class and object diagrams into an often intricate textual syntax. When reading OCL, a complicated and error prone translation in the opposite direction is required.

This mental mapping problem of textual OCL is already problematic in most standard software engineering environments, where OCL is therefore rarely employed. Important structural properties are often not documented, and information to this effect is lost in the course of the development process, as no tool besides natural language seems to be able to capture them, at least not economically.

For temporal logic such as LTL or CTL [7], the situation is even graver. As reported in [8], developers (even experts) have serious problems handling the intricate nature of these logics. Even in projects with very well trained experts, employing them is often impossible, as the resulting property specifications will usually be unintelligible to domain experts from other disciplines that need to participate in the effort.

For example, this problem becomes a serious hinderance when software engineers develop the software for complex mechatronic systems which also involve complex control engineering, mechanical engineering, and electrical engineering. As part of the trend towards more intelligent, efficient, and flexible mechatronic systems, dynamic software architectures which permit structural adaptation at run-time are beginning to displace static architectures and models. While this permits building systems that change in response to current needs, designing and validating such adaptable systems poses new challenges to software engineers, as the involved structural and temporal aspects are closely intertwined.

In this paper, we discuss how visual languages can be used for specifying structural as well as dynamic properties. First, we show how *Story Decision Diagrams* (SDD) [12] can be used to capture structural requirements. SDD are an extension of Story Patterns [17], combining the intuitive concept of matching structural patterns with decision diagrams, which foster a consecutive if-then-else decomposition of complex properties into comprehensible smaller ones. We then introduce *Timed Story Scenario Diagrams* (TSSD), a new notation inspired by the Visual Timed Event Scenario approach [1], as a way of capturing dynamic properties. They provide conditional timed scenarios describing the partial order of specific structural configurations. In addition, we present a scheme for turning specifications into powerful monitors which enable the verification of models w.r.t. dynamic structural properties using a model checker that supports structural evolution.

After reviewing and discussing the current state of the art in Section 2, we introduce our application example from the mechatronic domain in Section 3. The concepts for modeling structural properties are introduced in Section 4 and combined with the ones for modeling temporal properties in Section 5. By providing a mapping from the property specification to operational detector behavior in Section 6, we can demonstrate the practical validation of these properties. Finally, the paper provides a conclusion and an outlook on planned future work.

†supported by the International Graduate School of Dynamic Intelligent Systems.
*This work was developed in the course of the Special Research Initiative 614 – Self-optimizing Concepts and Structures in Mechanical Engineering – University of Paderborn, and was published on its behalf and funded by the Deutsche Forschungsgemeinschaft.

2. RELATED WORK

Visual Structural Properties. Constraint diagrams [15] visualize constraints as restrictions on sets using Euler circles, spiders and arrows. To compensate for the decrease in expressive power w.r.t. the OCL, constraint trees [16] combine them with the idea of parsing an OCL statement into a tree, replacing only selected constraints with constraint diagrams. The downside is that while quantification on sets is intuitive, structural constraints quickly result in intricate, visually complex diagrams with little relation to the original UML specification.

VisualOCL [3] is an approach that focuses on mapping OCL syntax to a visual format as closely as possible, thus facilitating the parsing of structural constraints. Based on the theory of graph grammars, Story Patterns (cf. [17]), an extension of UML Object Diagrams, are an alternative approach which can also be used for specifying constraints. Like most approaches that extend UML Structure Diagrams, they are very accessible, but in turn have deficits when it comes to quantification and negation.

Visual Temporal Properties. Several notations for scenarios as a means to visually describe temporal behavior have been proposed: UML 1.x sequence diagrams or message sequence charts have been employed to specify and check timed properties (cf. [19]). However, they are usually considered as not expressive enough, as only a set of runs or one specific run of the system, but no conditional properties, can be described. Therefore, the interpretation w.r.t. the system is usually unclear. This limitation has been tackled by a number of approaches such as live sequence charts (LSC) [14] or triggered message sequence charts (TMSCs) [24], which add the ability to describe conditional behavior in a sequence diagram style notation. To some extent, these enhancements found their way into UML 2.0 sequence diagrams (cf. [21, p. 444] assert block).

Other approaches such as the Visual Timed Event Scenario approach [1] focus on scenarios for pure events, rather than the interaction of predefined units. Therefore, they provide a more intuitive notion of temporal ordering than sequence diagrams, which require specifying a sequence of interactions that "enforces" this ordering.

Another related thread of research are specification patterns for temporal properties. As outlined in [8], the overwhelming number of temporal properties can be covered using a rather small set of specification patterns. This idea has been extended and applied to real-time systems in [18].

However, all these scenario-based or specification pattern-based approaches focus on the purely temporal aspect of behavior, abstracting from its structural aspects. Statements concerning the required temporal behavior of expressive structural properties are not supported.

Most approaches which permit combining structural and temporal properties are extensions of the OCL towards the description of dynamics. Through the introduction of additional temporal logic operators in OCL (e.g., eventually, always, or never), modelers are enabled to specify required behavior by means of temporal restrictions among actions and events, e.g., [4]. Temporal extensions of the OCL that consider real-time issues have been proposed for events in OCL/RT [6] and for states in RT-OCL [10]. As temporal logic alone already causes an even more demanding mapping problem (cf. [8]), integrating the OCL and some temporal logic concepts at the textual level does not yield a sufficiently comprehensible solution.

Story Diagrams [9] extend UML Activity Diagrams with Story Patterns to provide them with operational semantics. Though visually similar to TSSDs, their purpose is different: They strive to specify exactly *how* something happens, while TSSDs focus on mechanisms to specify *what* and *when* should result.

3. APPLICATION EXAMPLE

Motivation. The RailCab R&D project is developing a system of autonomous shuttles travelling on a railway network, with the intent to combine the advantages of railways and automobiles, providing fast, safe, energy-efficient and convenient individual transportation. In order to achieve significant improvements over existing systems, the project combines traditional mechanical and electrical engineering with software engineering techniques. The project is representative of a new class of advanced mechatronic systems [5] using sophisticated control and coordination techniques such as structural adaptation, ad-hoc collaboration, or self-optimization in complex real world situations. The promise of more intelligent, efficient, and flexible systems has led to an increased interest in such mechatronic systems, notably in the automotive sector. However, these improvements come at a cost, as designing the required more complex software poses new challenges to software engineers. Advanced mechatronic systems typically run concurrently and with real-time requirements, are often distributed and heterogeneous, the relevant context for decisions is often characterized by complex structural properties, and their physical nature makes them safety-critical almost by default. Approaches for handling the additional levels of complexity and verifying system safety are thus required.

Throughout this paper, we will use an example that is inspired by the RailCab project. In previous work, we have used related examples to demonstrate the compositional verification of real-time coordination patterns [13], modular system coordination using social structures, and the verification of safety properties that are inductive invariants of the system [2]. Here, we focus on specifying the associated structural and behavioral system requirements in a manner that is expressive, accessible to domain experts, and yet operational and compatible with existing model checking and verification techniques.

Structure. The railway network is modeled as a graph of small track segments, each about as large as a shuttle. Tracks are unidirectional, they have one or two (branch) successors and are successor to one or two (join) tracks. Shuttles are located on one track and may have next relationships with other tracks to indicate where they are travelling.

Tracks are monitored by associated controllers. Shuttles can execute a registration pattern with a controller. The registration pattern is a real-time coordination pattern which ensures that a shuttle keeps the controller informed about its exact position and is in turn informed about the position of all other shuttles in the controller's area of responsibility in regular intervals. This pattern is the foundation upon which another coordination pattern, the convoy pattern, operates. This pattern ensures that two shuttles in close proximity safely coordinate their behavior, which provides shuttles with the ability to reduce drag by forming

contact-free convoys. Figure 1 provides an overview of these elements.

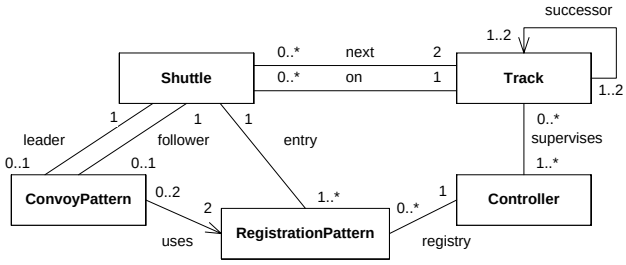


Figure 1: The elements of the shuttle system

The primary requirement we are considering is the absence of accidents. As the continuous control aspects are (correctly) encapsulated in the coordination patterns, we can analyze the safety of the system on a discretized world model by checking whether the correct coordination patterns exist in all specific instance situations, i.e. evaluating the structural correctness of the system.

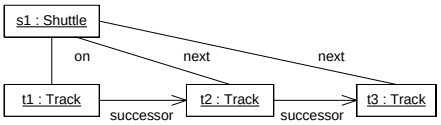
Properties. We now derive the properties we will formalize below. First of all, no two shuttles may share a track, as this would correspond to a collision. In order to make shuttle behavior predictable, shuttles need to mark the next two tracks they will use. Furthermore, if there is a shuttle right in front of another, the shuttles have to execute the convoy pattern in order to avoid collisions. As the convoy pattern depends on the registration pattern, both shuttles need to be registered with the same controller beforehand. Therefore, shuttles are required to register with all available controllers for their current position. To avoid problems when moving from one controller's area to another, these areas overlap - we require that for each shuttle, there always exists a controller that covers both the shuttle and its two next tracks. Finally, we impose a structural constraint that the system contains no dead ends and all tracks are reachable from any other track.

4. STRUCTURAL PROPERTIES

The fundamental abstraction that our approach is based upon is the idea of interpreting instance situations of an object-oriented system as graphs. Informally, this seems intuitively plausible, as UML Object Diagrams as a common way of describing instance situations already have a graph-like structure. More specifically, we map each object to a node and each attribute/association to an edge of a labeled graph. The theory of graph transformation systems (cf. [23]) then provides the formal semantics that are typically missing from UML-based notations, which allows reasoning about states and behavior of object-oriented systems modeled using a visual notation.

Story Patterns are an extended type of UML object diagram (cf. [17]) that allow expressing properties and transformations, especially structural changes. They consist of a precondition, the left hand side (LHS), and a postcondition, the right hand side (RHS). It is possible to define negative application conditions by crossing out elements of the diagram; however, it is not possible to forbid groups of elements or forbid only elements with specific associations.

Story Patterns without side effects, i.e. with identical LHS and RHS, can be used to describe and allow testing for system properties. E.g., the Story Pattern in Fig. 2 matches if a shuttle's on and next associations point to adjacent tracks in the proper order. A translation into OCL is provided below the figure. For our example, we would like this property to be a positive invariant of the system that is true for all shuttles. However, there is no way to make this explicit in the pattern.




```
context s1: Shuttle inv
  s1.next→exists(t2 |
    s1.on.successor→exists(ta | ta = t2))
      and
        s1.next→exists(t2 |
          s1.next→exists(t3 |
            t2.successor→exists(tb | tb = t3)))
```

Figure 2: Story pattern: a simple positive invariant

In [2], we used Story Patterns to specify invariants of the system that represented forbidden states (accidents, hazards), which could then be formally verified. This required the implicit convention that all patterns represented *negative* invariants of the system, which could not be indicated explicitly. The resulting restriction to negative invariants entailed the use of unintuitive multiple negations, i.e. representing a required element as a forbidden element of a forbidden pattern.

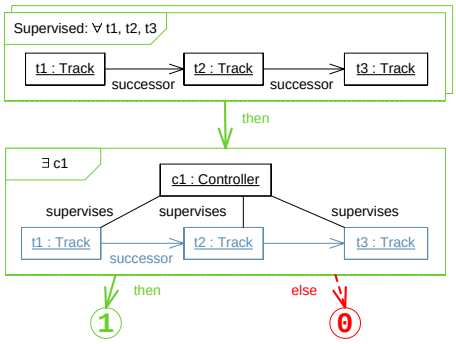
Story Decision Diagrams (SDD) [12] are an extension of Story Patterns that allow expressing more complex properties while retaining or surpassing the intuitiveness of the original visual notation. The most significant enhancements they provide are quantors, implication, alternatives, negation of complex properties, and a concept for modularity.

An SDD is a directed acyclic graph (DAG). Each node contains a Story Pattern without side effects that specifies some property. The nodes are processed starting from the root node. Each node in the DAG can essentially be seen as a local if-then-else decision for a binding. If a match is found, we follow the solid then connector; if no match is found, we follow the dashed else connector. New class and association bindings, i.e. successfully matched elements, are added to the current binding and propagated to subsequent elements. There are two special leaf nodes, (1) signifying *true* and (0) signifying *false*. When a binding reaches a leaf node, it evaluates to true or false, respectively.

Figure 3[1] encodes the requirement that for any three consecutive Tracks (∀), there must be a Controller (∃) supervising them all. The *forall* quantifier indicates that every binding for the first node needs to reach a (1) node eventually, whereas the *existential* quantifier only requires that one valid binding exists.

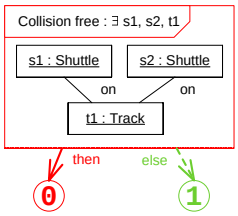
Figure 4 marks the existence of an accident (two Shuttles on the same Track) as an undesired instance situation. The negation is modeled by switching the then and else connectors, i.e. a match leads to failure and no match leads to success. There are only positive nodes in the patterns, which facilitates their interpretation. Complex negative conditions can be expressed by chaining the corresponding nodes.

[1]Color is optional and only encodes redundant information.




```
context Track inv
  Track.allInstances()→forAll(t1, t2, t3 |
    (t1.successor→exists(tx | tx = t2)
      and t2.successor→exists(ty | ty = t3))
    implies (Controller.allInstances()→exists(c1 |
      c1.tracks→exists(ta, tb, tc |
        ta = t1 and tb = t2 and tc = t3))))
```

Figure 3: Connected tracks share a controller




```
context s1 : Shuttle inv
  NOT Shuttle.allInstances()→exists(s1, s2 |
    s2.on = s1.on)
```

Figure 4: No two shuttles on the same track

Finally, SDDs provide modularity through the concept of **Embedded SDD** (ESDD) that encode nontrivial properties in a reusable fashion. ESDDs behave like patterns with free variables that can be bound depending on the respective current context. By inserting the ESDD into an SDD and binding concrete node instances to these variables, the ESDD can then be employed as a compact notation for the specified property. It is also possible to have nested ESDDs. This quite naturally leads to recursively defined patterns.

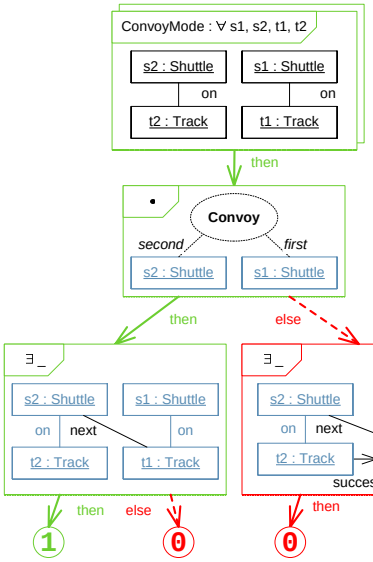
Figure 5 requires that a Shuttle executes a correct ConvoyPattern with a Shuttle on its immediate next Track, and does not execute a correct pattern if the other Shuttle is not on one of its next Tracks. The existence of a correct ConvoyPattern is encoded by the ESDD Convoy in Fig. 6.[2] It requires a common controller and matching RegistrationPatterns.

5. TEMPORAL PROPERTIES

So far, we have used graphs to express complex static properties. It is tempting to apply the same approach to temporal properties, e.g. in order to describe the structural adaptation of a system. In addition to the outlined concepts for structural properties, we then need to consider their occurrence and temporal ordering.

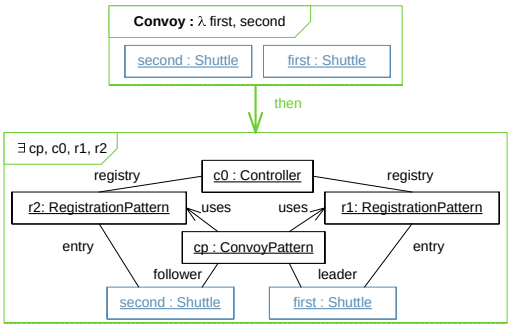
The temporal behavior of a system can be described as a sequence of states. Extending our former considerations, each state is described as a graph. The identity of nodes and edges is preserved between two system states, while attribute values are anonymous.

[2]Standard leaf nodes can be omitted.




```
context Shuttle inv
  Shuttle.allInstances()→forall(s1, s2 |
    (s1.on <> s2.on)
  implies
    ((Shuttle::convoy(s1,s2)
      implies
       (s1.next→exists(ta | ta = s2.on)))
    and
      (NOT Shuttle::convoy(s1,s2)
        implies
          NOT
           (s1.next→exists(ta | ta = s2.on)
             and
               s1.on.successor→exists(tb |
                 tb = s2.on)))))
```

Figure 5: Shuttles must or must *not* form a convoy




```
context Shuttle::convoy(first : Shuttle, second : Shuttle)
  ConvoyPattern→allInstances()→exists(cp |
    (cp.leader = first and cp.follower = second)
      and
       Controller→allInstances()→exists(c0 |
         c0.registrationPattern→exists(r1, r2 |
           r1.entry = first and cp.uses→exists(ra | ra = r1)
             and
               r2.entry = second
                 and
                   cp.uses→exists(rb | rb = r2))))
```

Figure 6: ESDD: The shuttles form a convoy

5.1 Timed Story Scenario Diagrams

Observations. The idea behind *timed story scenario diagrams* (TSSD) is to use the ordering of structural observations in order to specify valid orderings and temporal properties. Each such observation is made by an *observation node* containing an SDD.

Edges. Two structural observations can be related to each other using temporal ordering and constraint edges:

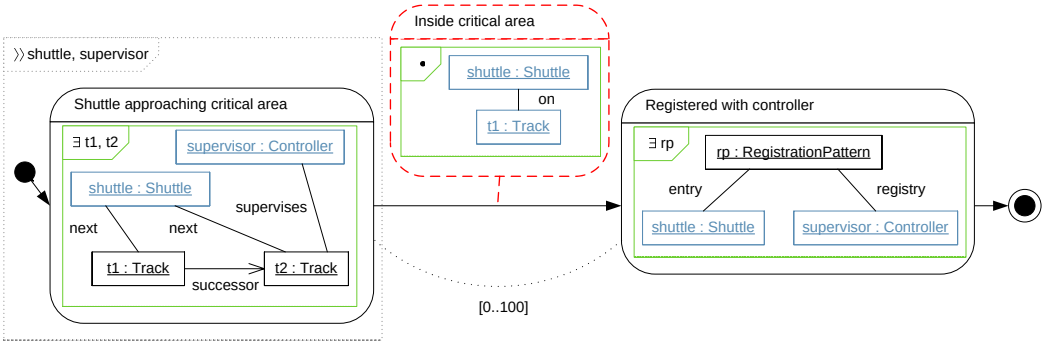


Figure 7: TSSD Syntax - A Shuttle registers with a Registry

(a) The *temporal ordering edge* ($A \longrightarrow B$) denotes that observation A is made before observation B. Note that this includes that observation A and B may occur at the same point in time.

(b) The *until temporal ordering edge* ($A \xrightarrow{C} B$) denotes that observation A is observed before observation B and that the structural property C is valid for all states between the observation A and the observation B. If observations A and B occur at the same time, structural property C is never evaluated and does not need to be fulfilled.

(c) In addition to the temporal ordering edges, *time constraint edges* with lower bound a and upper bound b ($A \overset{[a\ldots b]}{\cdots\cdots} B$) constrain the permitted time difference between the occurrence of two observations A and B.

AND, OR, NOT. Observations are only partially ordered. If the TSSD branches, both paths progress independently and in parallel. If an observation node has multiple ingoing temporal ordering edges, all preceding observations need to occur before the observation is considered (AND).

The final node ⦿ indicates a successful match. More formally, a system trace π fulfills a TSSD if ⦿ has been reached for a prefix of π. By using multiple ⦿ nodes on independent paths, disjunction (OR) can be expressed.

The final node ⊗ indicates a violation. Replacing ⦿ with ⊗ turns a path into a forbidden scenario (NOT), just as switching the leaf nodes in an SDD is used to express negation. If a trace reaches both types of final node at the same time, failure ⊗ takes precedence over success ⦿, resulting in a violation.

Triggers. However, TSSDs are not limited to simply recognizing and chronicling observations over time. Arbitrary initial fragments of a TSSD can be combined into a trigger. As long as the trigger sequence has not been completely detected, the resulting TSSD provides no constraint for the correct temporal behavior. However, once the trigger has been detected, the remaining elements of the TSSD define temporal properties which have to be fulfilled.

Figure 7 is a basic TSSD presenting the key elements of the syntax. When a Shuttle is approaching a Controller's supervised area, they have between 0 and 100 time units to launch a RegistrationPattern. In the mean time, the Shuttle must not yet have entered the critical area, which is indicated by the (forbidden) state on the transition.

Subscenarios. Modularity is again of paramount importance for practical scalability. We therefore provide the ability to invoke a subscenario (see Fig. 8) that is defined

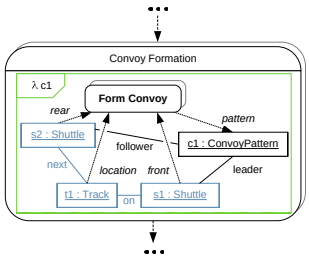


Figure 8: Invoking a subscenario (fragment)

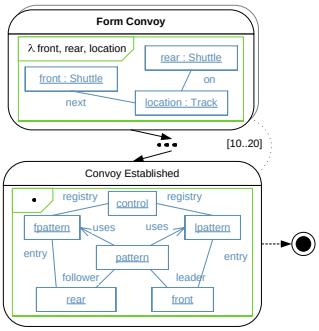


Figure 9: Definition of a subscenario (fragment)

elsewhere (see Fig. 9). In the example fragment, we use the subscenario to abstract from the details of convoy formation. Invocation works just like ESDD evaluation. However, as a subscenario might create bindings that are needed later on in the scenario, the invocation itself takes place inside a λ-node that allows exporting arbitrary bindings (e.g. *pattern* $\rightarrow$ *c1*) back from the subscenario.

Quantification. A subtle aspect of TSSD is that they provide three different levels of quantification. First of all, observations are matched by SDDs and can thus be structurally equivalent but distinct instances of the same pattern. This is very different from typical event- or message-based approaches that do not consider structure and cannot differentiate between multiple (concurrent) instances of the same event. Consider a simple scenario where a student's undergraduate program (trigger) requires her to choose a class and eventually complete it. Assume the class and its completion are existentially quantified in the SDDs; we therefore bear with her as she starts and abandons several classes un-

til she finally completes one. Multiple matches thus become part of a set of alternative bindings, only one of which needs to succeed. Secondly, we quantify over triggers, as seen in Fig. 7. We want the scenario to be triggered and successfully completed for *all* cases when a new matching Shuttle-Controller-pair is detected. Finally, we quantify over time: we require untriggered scenarios to match at least once (*existential TSSD*), whereas triggered scenarios always need to be fulfilled (*universal TSSD*).

5.2 Formal Semantics

Clear and intuitive semantics are of paramount importance for temporal properties. It is well known that temporal logics such as LTL or CTL [7] are hard to understand and even harder to write for any non-trivial property (cf. [8]).

If we want to be able to specify that a specific detected structural constellation will result in a different specific structural constellation within a certain time bound (see Figure 7), we not only have to consider individual states in form of a graph, but all sequences of states in form of graphs as generated by a graph transformation system (GTS) representing our system model. Graph-Interpreted Temporal Logic [11] provides a propositional temporal logic for such systems.

For a path π, we denote its potentially infinite length by $l(\pi)$ and refer to its i-th state graph as $\pi[i]$. A function $T(\pi, i)$ denotes the time when the i-th state has been reached and thus provides a concrete notion of time.

We can exploit the DAG property of all TSSDs to define their semantics. We propagate the path π and bindings $\mathcal{X}$ between variables and instances that have been created by previous observations along the temporal and constraint edges to determine the valid bindings for each node. Then, the semantics for a node n for time offset t_s (written $[\![n]\!]_{\mathcal{X}}^{\pi,t_s}$) is given by those pairs $(t_n, \mathcal{X}_n)$ of time t_n and bindings $\mathcal{X}_n$ where t_n fulfills certain conditions and $\mathcal{X}_n$ are valid bindings for n's SDD sdd_n that extend the bindings generated by n's direct predecessor nodes. We have

$$(t_n, \mathcal{X}_n) \in [\![n]\!]_{\mathcal{X}}^{\pi,t_s}$$

iff (a) a set of pairs $(t_{n'}, \mathcal{X}_{n'})$ for all regular predecessor nodes $N = \{n' | n' \longrightarrow n \vee n' \xrightarrow{n''} n\}$ exists whose combined bindings permit matching the SDD sdd_n of node n:

$$\forall n' \in N : \exists (t_{n'}, \mathcal{X}_{n'}) \in [\![n']\!]_{\mathcal{X}'}^{\pi,t'_s} : X_n \in [\![sdd_n]\!]_{comb(\{\mathcal{X}_{n'} | n' \in N\})}^{\pi[t_n]}.$$

with $comb(\{\mathcal{X}_1, \ldots, \mathcal{X}_n\})$ the combination of the different bindings. In addition (b), the time offset t_n is uniquely determined for the $t_{n'}$ and $t_m = \max\{t_{n'} | n' \in N\}$ by the following conditions: (1) The node n cannot be fulfilled earlier by the same binding.

$$\forall t' : t_m \leq t' < t_n : \mathcal{X}_n \cap [\![sdd_n]\!]_{\mathcal{X}'}^{\pi[t']} = \emptyset$$

(2) In case of an until edge $n' \xrightarrow{n''} n$, the SDD for n'' must be satisfied until n is reached.

$$\forall n', n' \xrightarrow{n''} n \; \forall t' : t_{n'} < t' < t_n : \; [\![sdd_{n''}]\!]_{\mathcal{X}_{n'}}^{\pi[t']} \neq \emptyset$$

(3) For a time constraint $n' \overset{[a \ldots b]}{\text{- - -}} n$, the chosen time offsets must satisfy $[a, b]$.

$$\forall n', a, b, \; n' \overset{[a \ldots b]}{\text{- - -}} n : \; a \leq |T(\pi, t_n) - T(\pi, t_{n'})| \leq b$$

If no predecessor node exists ($pred(n) = \emptyset$), we only require that the detection of the binding is recognized as early as possible. If t_n is not zero, we thus require in this case:

$$\forall t, t_s \leq t < t_n : \mathcal{X}_n \cap [\![sdd_n]\!]_{\mathcal{X}}^{\pi[t]} = \emptyset$$

A pair $(t, \mathcal{X})$ is a solution for a TSSD td for a path π and the time offset t_s iff a final success node $\bigodot_i$ of the TSSD exists with $(t, \mathcal{X}) \in [\![\bigodot_i]\!]_{\mathcal{X}}^{\pi,t_s}$ such that for all $\bigotimes_i$ and $(t', \mathcal{X}') \in [\![\bigotimes_i]\!]_{\mathcal{X}}^{\pi,t_s}$ hold that $t' > t$ or $\mathcal{X}|_{var(\mathcal{X}')} \cap \mathcal{X}'|_{var(\mathcal{X})} = \emptyset$. We write $[\![td]\!]_{\mathcal{X}}^{\pi,t_s}$ to denote all solutions, i.e., all successful bindings reaching a success node before any failure node is reached.

Given this definition for $[\![td]\!]_{\mathcal{X}}^{\pi,t}$, we can further define the semantics for universal and existential TSSD for a given GTS M and TSSD td with free variables $free(td)$ as follows:

- A GTS M fulfills an *existential TSSD* $\phi = \exists\ td$ iff there exists a trace π, a time offset t, and an initial binding $\xi \in X[free(td)]$ for which td can be satisfied:

  $$\exists \pi \in [\![M]\!], t \in [0, l(\pi)], \xi \in X[free(td)] : \; [\![td]\!]_{\{\{\xi\}\}}^{\pi,t} \neq \emptyset.$$

- A GTS M fulfills the *universal TSSD* $\psi = \forall\ td_t : \; td_b$ iff for all traces π, all time offsets t, and all initial bindings $\xi \in X[free(sd_t)]$ holds that if a matching binding for td_t has been found, then for the same binding, td_b will also eventually become true:

  $$\forall \pi \in [\![M]\!], t \in [0, l(\pi)], \xi \in X[free(td_t)] :$$
  $$\forall (t', \mathcal{X}') \in [\![td_t]\!]_{\{\{\xi\}\}}^{\pi,t} : \; [\![td_b]\!]_{\mathcal{X}'}^{\pi,t'} \neq \emptyset.$$

6. PROPERTY DETECTORS

Specifications using SDDs and TSSDs are not merely a tool for communicating and reasoning about structural and temporal properties, but can be used for verification or run-time monitoring the specified properties by running a property detector in parallel with the system. There are two fundamentely different ways to implement such a detector open to us. The first possibility is to start from the graph-based semantics, translate the property detector into a GTS and execute it inside a graph model checker, which is useful for verification and evaluating prototypes. The second, more efficient possibility is to use Fujaba to generate the detector as a Java or C++ program capable of monitoring an application. Here, we focus on the first option, which enforces a more rigorous approach, for assessing the feasibility of such detectors. Code generation is discussed in [12].

GROOVE is a GTS model checker [22], capable of simulating GTS and generating state spaces. For a GTS specifications, GROOVE can compute all reachable states of the transformation system, optionally bounded by the occurrence of a forbidden graph. We have previously developed a Fujaba plugin for exporting Story Patterns from Fujaba into GROOVE. While exporting SDDs is much more complex, we used this foundation to manually derive story detectors for evaluation.

6.1 Structural Properties

For structural properties, we have the choice of computing all alternative bindings exhibiting the property or just one set of bindings that fulfills the SDD. Pragmatically, we have chosen to stop the search as soon as the first solution is

found. A single SDD is transformed into many small graph rules (the *ConvoyMode* SDD resulted in 44 rules) following a repetitive pattern that could easily be automated, especially since the largest part of the rules are identical for all SDDs.

The basic algorithm works by implicitly traversing the SDD from its root to the leaves. On the way up, *markers* are used to store bindings and activate and inhibit the appropriate nodes. When a marker reaches a leaf, it is in turn marked up with the result, which is then propagated back down towards the root, along with a set of witnesses. Once a result marker reaches the root, the evaluation of the SDD is complete and all markers and results are cleared. There is only one type of marker; markers are identified by their position relative to the root element and connected through 1 and 0 edges, corresponding to then and else connectors. Each marker only stores the additional variables it binds; the complete binding is thus defined by the path from the marker to the root element. Figure 10 shows a snapshot of the matching process for the property from Fig. 2 – the shuttle to the left has already been marked as correct; the detector will next try to bind tracks t1, t2, t3 using the rule in Fig. 11 for the shuttle on the right.

Rule priorities are very important for the correct execution. (A) Cleanup has the highest priority. It is triggered by a result marker at the root element. (B) Next come the immediate propagation rules, i.e. success for existential and failure for universal nodes. Here, nodes closest to root have the highest priority to ensure immediate propagation. (C) Then the rule groups containing the main part of the rules follow. Here, nodes that are farther away from the root node have higher priority to enforce depth-first traversal. Inside each group, there are three rules per node: (2) if no matching 1 marker is present, the *then* rule tries to match the actual pattern and create a 1 marker (see e.g. Fig. 11). (2) Rules creating result markers at leaf nodes operate at this level as well. (1) The *else* rule creates a 0 marker if no 1 or 0 marker is present. (0) The *return* rule propagates failure (success) for existential (universal) nodes. (C') Nodes containing ESDDs have a fourth rule with highest priority (3) that triggers the ESDDs by creating a root marker with a binding for the free variables. The (2) *then* rule then additionally checks the result marker of the ESDD. The whole ESDD has higher priority than the current SDD.

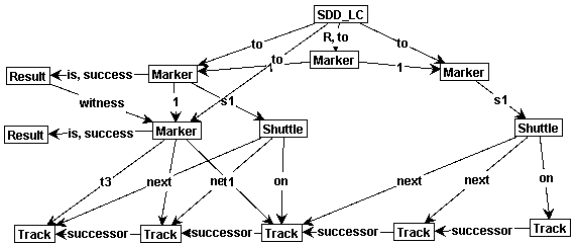


Figure 10: *Located*: Markers while processing

When all rules were translated, we created a set of correct and incorrect instance graphs (see Fig. 12) and ran the SDD property detectors on them. All examples were evaluated correctly, yielding the expected results. Notwithstanding the large number of rules, evaluation was fast and efficient, as the individual rules were mostly small and only a limited number of them was active at any one time.

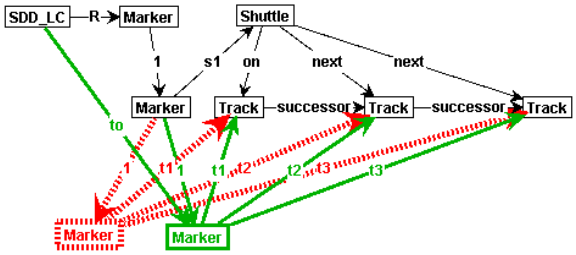


Figure 11: *Supervised*: Graph rule for second node

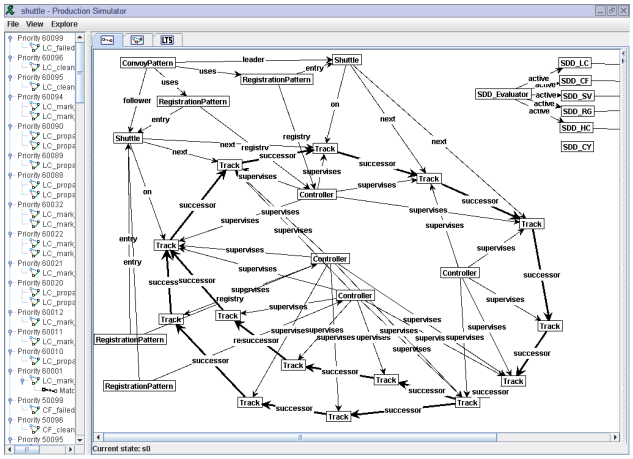


Figure 12: The test scenario in GROOVE

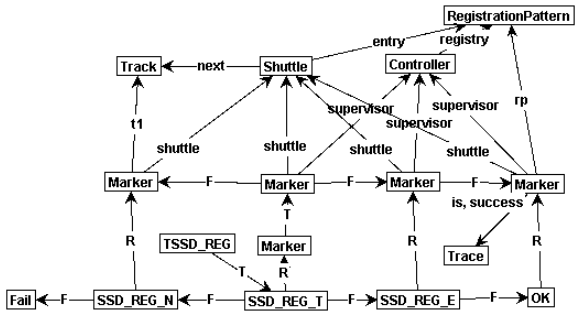


Figure 13: *Register Shuttle*: Scenario matched

6.2 Temporal Properties

At the operational level, verifying temporal properties is remarkably similar to verifying structural properties with ESDDs. Whenever a positive result has been propagated down to an observation's root marker, new root markers are created for all subsequent observations and the bindings are copied from the witness to the new root markers. However, the successful observation itself also remains active, ready to create new bindings. In Fig. 13, the scenario in Fig. 7 has just reached the success node. The result will now be propagated back down the F edges to the trigger, and all root markers will be deleted.

In our experiments, we chose not to implement the time constraint, which would have required an unwieldy auxiliary construction as GROOVE does not support time natively. However, we added a marker representing a time step that

inhibits all SDD rules in order to give the shuttle the opportunity to actually move between subsequent matching attempts of the SDD rules.

7. CONCLUSION AND FUTURE WORK

We have presented a visual approach for the specification of temporal and structural properties. We have shown how extensions of UML object diagrams can be employed for the description of complex structural conditions. Based on these notations, we have introduced timed scenarios as a natural way of specifying the temporal order and time constraints for a sequence of observations. The approach thus combines the specification of detailed structural properties and requirements concerning structural dynamics using a clear and intuitive visual notation. The presented formalization provides the required solid foundation for the soundness of the approach. The operational realization as Story Detectors that detect the specified properties based on the GTS model checker GROOVE show that the approach even works under the restricted conditions of a model checking engine.

When specifying invariants using Fujaba in our previous work, we had to misappropriate Story Diagrams for that purpose. We intend to extend Fujaba with a systematic way to explicitly attach (visually specified) constraints to models and model elements. We are currently working on an implementation of the new notations as a plug-in for Fujaba4Eclipse. We also plan to automate the export of SDDs and TSSDs into GROOVE and provide code generation for constraint monitors in Java and C++, extending the existing code generation facilities.

8. REFERENCES

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

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

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

[4] J. Bradfield, J. Kuester Filipe, and P. Stevens. Enriching OCL Using Observational mu-Calculus. In R.-D. Kutsche and H. Weber, editors, *Fundamental Approaches to Software Engineering (FASE 2002), Grenoble, France*, volume 2306 of *LNCS*. Springer, April 2002.

[5] D. Bradley, D. Seward, D. Dawson, and S. Burge. *Mechatronics*. Stanley Thornes, 2000.

[6] M. Cengarle and A. Knapp. Towards OCL/RT. In L.-H. Eriksson and P. Lindsay, editors, *Formal Methods – Getting IT Right, International Symposium of Formal Methods Europe, Copenhagen, Denmark*, volume 2391 of *LNCS*, pages 389–408. Springer, 2002.

[7] E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, Jan. 2000.

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

[9] 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, pages 296–309. Springer Verlag, November 1998.

[10] S. Flake and W. Mueller. An OCL Extension for Real-Time Constraints. In *Object Modeling with the OCL: The Rationale behind the Object Constraint Language*, volume 2263 of *LNCS*, pages 150–171. Springer, February 2002.

[11] F. Gadducci, R. Heckel, and M. Koch. A fully abstract model for graph-interpreted temporal logic. In H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors, *TAGT*, volume 1764 of *Lecture Notes in Computer Science*, pages 310–322. Springer, 1998.

[12] H. Giese and F. Klein. Visual Specification of Structural and Temporal Properties. Technical Report tr-ri-06-276, Computer Science Department, University of Paderborn, September 2006. To appear.

[13] 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.

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

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

[16] S. Kent and J. Howse. Constraint trees. In T. Clark and J. Warmer, editors, *Object Modeling with the OCL*, pages 228–249. Springer, 2002.

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

[18] S. Konrad and B. H. C. Cheng. Real-time specification patterns. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 372–381, New York, NY, USA, 2005. ACM Press.

[19] X. Li and J. Lilius. Timing Analysis of UML Sequence Diagrams. In R. France and B. Rumpe, editors, *UML'99 - The Second International Conference on The Unified Modeling Language Fort Collins, Colorado, USA*, volume 1723 of *Lecture Notes in Computer Science*, October 1999.

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

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

[22] A. Rensink. Towards model checking graph grammars. In M. Leuschel, S. Gruner, and S. L. Presti, editors, *Workshop on Automated Verification of Critical Systems (AVoCS)*, Technical Report DSSE–TR–2003–2, pages 150–160. University of Southampton, 2003.

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

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

UML-Based Rule Modeling with Fujaba

Sergey Lukichev[1], Gerd Wagner[1]
[1]Institute of Informatics, Brandenburg University of Technology at Cottbus, Germany
Lukichev@tu-cottbus.de, G.Wagner@tu-cottbus.de

ABSTRACT

In this paper we describe visual rule modeling tool Strelka, which is implemented as a Fujaba plugin. The modeling tool supports a UML-Based Rule Modeling Language (URML). It extends standard UML metamodel with a concept of a rule. We discuss an issue of a UML-based rule modeling, present rule metamodel, describe implementation of a Fujaba plugin and give examples of business rules, modeled using Strelka.

1. INTRODUCTION

Rules are becoming increasingly important in business modeling and requirements engineering, and as a high level programming paradigm. They are widely recognized to play an important role in the Semantic Web and are a critical technology component for the early adoption and applications of knowledge-based techniques in e-business, especially enterprize integration and B2B e-commerce. A lot of work has been conducted in the area of visual representation of business vocabularies and the mainstream technology is MOF/UML, which allows visualization of domain concepts by means of, for instance, UML class diagrams. In the area of emerging rule technologies for the Semantic Web and for Business Rules there are relatively few approaches and tools for visual rule modeling. We argue, that rule modeling language should allow visual rule expressions, which can be understood by domain experts or by existing software engineers without extensive technical training. We base our modeling approach on UML since it is widely adopted modeling language. Since rules are built on facts, and facts are built on concepts as expressed by terms[1], a UML-based rule modeling approach is a natural extension of UML. To support rules in UML class diagrams we extend the UML metamodel with a concept of a rule. The request for a UML-based rule modeling tool for the Semantic Web comes from the industry. Many companies claim that even if they understand benefits of using Semantic Web technologies like ontologies and rule languages, it is difficult for them to start since ontology architects and rule experts are quite expensive. A UML-based rule modeling approach for the Semantic Web will facilitate the use of the Semantic Web technologies by traditional UML modelers. The actuality of the proposed modeling approach also comes from the rules standardization efforts of W3C (http://www.w3.org/2005/rules/) and OMG (http://www.omg.org), which need rules modeling methodologies and tools.

In Section 2 we give a short overview of existing rule modeling approaches. In Section 3 we describe a metamodel and modeling language for derivation rules. In Section 4 we describe our rule modeling plugin for Fujaba, called Strelka and discuss modeling examples in Section 5. In Section 6 we summarize our results and describe future plans.

2. RELATED WORKS

The Protege tool provides facilities for ontology and rules modeling. In particular, it supports modeling in RDF [9] and OWL [8] as well as modeling of SWRL [3] rules. Protege requires a significant knowledge of ontology modeling. Moreover it is doubtful that it can be easily adopted in enterprizes, which already use UML technologies for software engineering.

There are ontology language specific tools for visual representation of ontologies, for instance, SemTalk from Semtation GmbH, which provides a visual language for modeling OWL ontologies. The approach of defining visual language for a particular ontology language has a lack of flexibility and scalability (since they provide visual notation for only one ontology language, for instance, OWL), while our UML-based approach has a power of MDA and allows obtain rules in language-independent manner.

There is a work on defining UML profile for ontologies and rules[10]. The approach defines a UML profile for SWRL rules and can be used for modeling of OWL ontologies and SWRL rules. Our approach is more general, since it supports not only SWRL-like (integrity and derivation) rules, but production rules and reaction rules as well. In addition, we provide special visual notation for rules, which cannot be obtained by the UML profile approach.

3. METAMODEL FOR RULES

In order to model rules with UML we have developed a UML-Based Rule Modeling Language (URML)[1]. This language supports modeling of derivation rules, production rules and reaction rules. In this paper we focus more on derivation rules, but briefly describe the general rule metamodel.

3.1 Rules

The general rule metamodel is depicted on Figure 1. A rule extends a UML TypedElement and belongs to a Namespace. We consider three main rule types:

[1]The URML on I1 website http://www.rewerse.net/I1 or in REWERSE I1 deliverable D8.

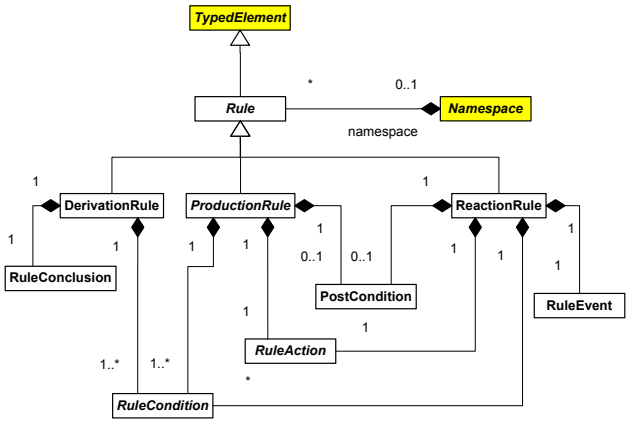


Figure 1: The URML rule metamodel

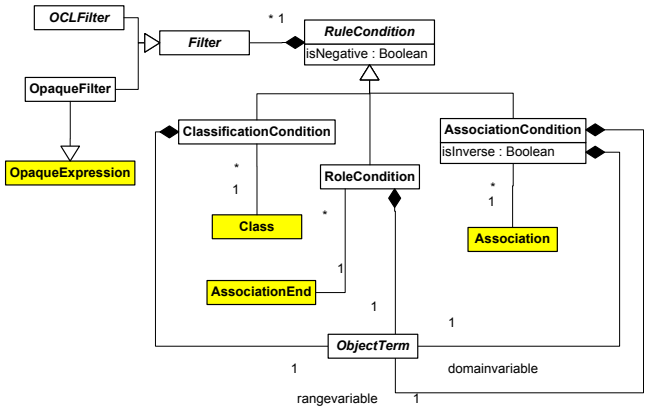


Figure 2: URML condition metamodel

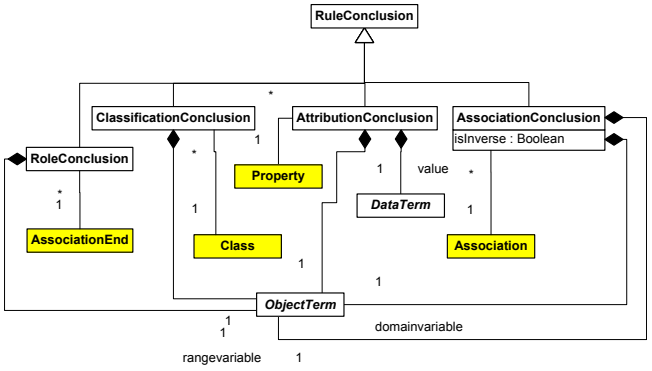


Figure 3: URML conclusion metamodel

DerivationRule has at least one condition and a conclusion. Such rule defines how model element can be derived. Example of derivation rule in the natural English language is: "*A gold customer is a customer with more than $1Million on deposit.*" This rule derives a concept of a *gold customer*, if condition "*more than $1Million on deposit*" is hold;

ProductionRule has at least one condition, one rule action and an optional post condition. Such rule performs an action if conditions are hold. For instance, the rule "*Exempt an investment from tax on profit if the stocks have been bought more than a year ago*" is a production rule, since if condition "*stocks have been bought more than a year ago*" is hold, then the action "*exempt an investment from tax on profit*" is performed;

ReactionRule may have several conditions, a triggering event, one rule action and an optional postcondition. Such rule formalizes event-condition-action behavioral model, where the action is executed on event with a condition satisfied. An example of such rule is "*When a share price drops by more than 5% and the investment is exempt from tax on profit, then sell it*". Event "*is share price drops by more than 5%*", condition is "*the investment is exempt from tax on profit*" and the action is "*sell it*".

3.2 Rule Conditions

A rule condition is a ClassificationCondition, RoleCondition or AssociationCondition (Figure 2).

ClassificationCondition refers to a UML Class, which is a condition classifier, and consists of an ObjectTerm, which is an object variable or an object;

RoleCondition refers to a UML AssociationEnd, which is a condition classifier, and consists of an ObjectTerm, which is an object variable or an object at the association end;

AssociationCondition refers to a UML Association, which is a condition classifier, and consists of two ObjectTerm's as a domain and a range, which are object variables or objects of classes from corresponding association ends of the Association.

A rule condition may have a filter expression. A filter is used to filter instances of a conditioned classifier. URML offers an option to express filters in OCL syntax, but an OpaqueFilter may be used for some vendor-specific implementation and syntax of filter expressions. Strelka tool supports OCL filters by integrating Dresden OCL Toolkit [7].

3.3 A Rule Conclusion

A rule conclusion is a RoleConclusion, ClassificationConclusion, AttributionConclusion, and AssociationConclusion (Figure 3). It is used in derivation rules in order to define a UML concepts (classes, associations, attribute values).

RoleConclusion refers to a UML AssociationEnd, which is a conclusion classifier, and consists of an ObjectTerm, which is an object variable or an object at the association end;

ClassificationConclusion refers to a UML Class, which is a conclusion classifier, and consists of an ObjectTerm, which is an object variable or an object;

AttributionConclusion refers to a UML Property and consists of an ObjectTerm, which is a context object or an object variable of the property and DataTerm, which is a value of the property;

AssociationConclusion refers to a UML Association, which is a conclusion classifier, and consists of two ObjectTerm's as a domain and a range object variable or object from corresponding association ends of the Association.

4. STRELKA - RULE MODELING FUJABA PLUGIN

In the REWERSE Working Group I1 we have developed a Strelka tool, which supports the URML. The tool is implemented as a Fujaba plugin. Following standard Fujaba architecture for writing plugins, it connects Fujaba metamodel with the URML metamodel, introducing required rule-related concepts like conditions, conclusions, rules, etc. We have used Fujaba to model URML metamodel, described in Section 3 and to generate corresponding Java sources. Below is a part of URML visual notation for rules, supported by Strelka:

Rules are represented as circles with identifiers;

Condition arrow refers to a *conditioned model element*, which is a classifier such as a class or an association. It may come with a *filter expression* selecting instances from the extension of the condition classifier and with an explicit object variable (or object variable tuple, in the case of an association) ranging over the resulting instance collection;

Negated condition arrow is crossed at their origin. It denotes a negated condition which has to be conjoined with one or more positive condition arrows such that its variables are covered by them;

Derivation rule is represented graphically as a circle with an internal label "DR" and a rule identifier attached to it. Incoming arrows represent conditions, outgoing arrows represent conclusions;

Conclusion arrow also refers to a classifier model element. Its meaning is to state that the predicate represented by the conclusion classifier applies to any instance that satisfies all rule conditions;

Filter expression is a text annotation of a condition arrow. Filter is used to filter instances of a conditioned classifier;

Variable is a text annotation of a condition or conclusion arrow and contains a name of an instance variable of a condition or a conclusion classifier.

A more detailed description of URML is available in the REWERSE Working Group I1 Deliverable D8.

In order to visualize the URML metamodel, we have implemented a rendering class for each URML metamodel class by extending Fujaba AbstractUnparseModule. In order to deploy modeled rules into a particular rule engine or rule-based application, Strelka supports serialization of rule models into the rule markup language R2ML[6], which is an acronym for the REWERSE Rule Markup Language. This rule language has been designed as a rule interchange format between different platforms and has such features as integration of functional languages (such as OCL) with Datalog languages (such as SWRL), the ontological distinction between objects and data values, the datatype concepts of RDF and user-defined datatypes, supports actions and events. Due to its design features, R2ML is an efficient intermediary format for rules and can be used as an interchange language between different rule systems and formalisms. There is a number of subsequent projects in the Working Group I1 dedicated to the rule interchange support between UML/OCL and OWL/SWRL using R2ML, Jess and R2ML, JRules and R2ML, etc. The R2ML serialization of URML models is already supported by Strelka. Since URML filter expressions (Figure 5) have OCL syntax, Dresden OCL Toolkit [7] and OCL for Fujaba have been adopted and integrated with Strelka.

5. MODELING EXAMPLES

In this section we provide two rule examples, modeled with the URML in the Strelka tool.

5.1 Deriving Association

Let's model the following derivation rule:

> If a rental car is stored at a branch, is not assigned to a rental and is not scheduled for service, then the rental car is available at the branch.

Its rule diagram is depicted on Figure 4, which is a screenshot of the Strelka plugin. The non-standard Fujaba modeling behavior, like arrows, which start and end in the middle of associations, is implemented in the tool. The rule is represented as a circle with an abbreviation DR, which stands for "Derivation Rule" and a rule identifier, which identifies the rule in a rule set. This rule has three conditions: *rental car is stored at the branch*, *rental car is not assigned to a rental*, *rental car is not a rental car scheduled for service*, which are visualized by incoming arrows, connecting conditioned classifier (a class or an association) with the rule circle. In order to visualize negated conditions, a condition arrow is crossed. The rule conclusion *the rental car is available at the branch* is visualized as an arrow from the rule circle to the derived association *isAvailableAt*. Condition and conclusion arrows are annotated with variables: **bra**, **rc**, **ren**. The semantics of the rule is captured by the following logical formula:

$$isAvailableAt(rc, bra) \longleftarrow isStoredAt(rc, bra) \wedge$$

$$\neg RentalCarSchduledForService(ren) \wedge$$

$$\neg \exists re(isAssignedTo(rc, re))$$

5.2 Deriving Class

Let's model the following rule:

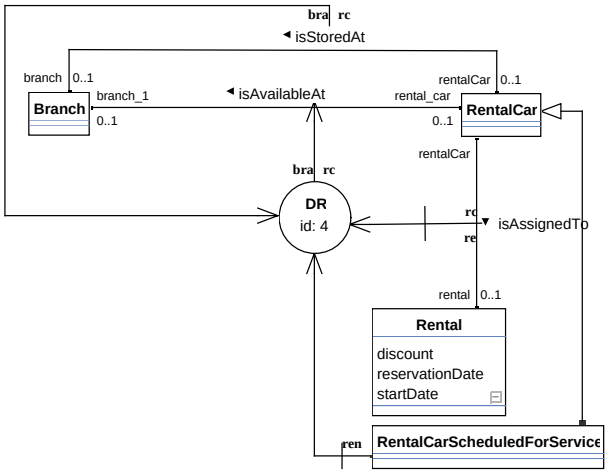


Figure 4: If a rental car is stored at a branch, is not assigned to a rental and is not scheduled for service, then the rental car is available at the branch.

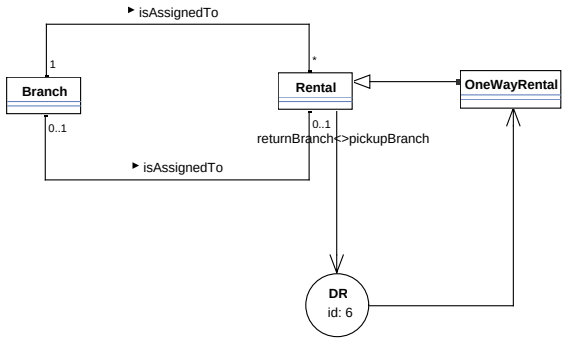


Figure 5: If return branch of a rental is different from pickup branch of a rental, then rental is a one way rental.

> If return branch of a rental is different from pickup branch of a rental, then rental is a one way rental.

Its diagram is depicted on Figure 5. This rule has only one condition with the filter expression in OCL syntax $returnBranch <> pickupBranch$.

6. EXPERIENCES AND CONCLUSIONS

In this paper we have introduced the Strelka tool for UML-based rule modeling. The tool can be used for visual modeling of different rule types. In order to support graphical notation of the URML we have added several new FSA Swing classes like double-head arrow, negated arrow, UML signal event, UML time event, etc.

From our experience, the main problem with Fujaba is a flexibility of the user interface. We have implemented several setting options in order to switch off/on methods and attributes visibility, method signatures and attribute types. The usage scenario assumes that the UML class diagram of a business vocabulary is developed first and then rules are visualized on subsequent diagrams. A subsequent diagram represents only a part of the business vocabulary, which participates in the rule. For instance, if only one of several associations between two classes participates in a rule, it is difficult to hide others from the diagram in Fujaba. One possible solution is to use Fujaba views. At the moment a modeler can choose an association for a view, but it is not shown, for instance, between which classes this association is.

During our work on URML models serialization into the R2ML we had problems with Dresden OCL toolkit integration with Fujaba. There is a project, which integrates OCL toolkit with Fujaba and we have based our R2ML serialization solution on it, but OCL support in Fujaba is still in alpha.

The future work on the tool includes integration of the rule verbalization component, which is currently under development in the Working Group I1. We are going to join Fujaba4Eclipse project and make Strelka working under Eclipse.

7. REFERENCES

[1] Ross, R. G., Principles of the Business Rule Approach. Addison-Wesley Information Technology Series (2003).

[2] Object Constraint Language (OCL), v2.0, http://www.omg.org/docs/ptc/03-10-14.pdf, last accessed Sep 4, 2006

[3] Semantic Web Rule Language (SWRL), http://www.daml.org/swrl, last accessed Sep 4, 2006

[4] Model Driven Architecture (MDA), OMG, http://www.omg.org/cgi-bin/doc?mda-guide, last accessed Sep 4, 2006

[5] W3C Workgroup on RIF Charter, http://www.w3.org/2005/rules/wg/charter, last accessed Sep 4, 2006

[6] Wagner, G., Giurca, A., Lukichev, S. (2006). A Usable Interchange Format for Rich Syntax Rules. Integrating OCL, RuleML and SWRL. In proceedings of Reasoning on the Web Workshop at WWW2006, May 2006.

[7] Dresden OCL Toolkit, http://dresden-ocl.sourceforge.net/, last accessed Sep 4, 2006

[8] Patel-Schneider, Peter F., Horroks I., OWL Web Ontology Language Semantic and Abstract Syntax, http://www.w3.org/2004/OWL, last accessed Sep 4, 2006

[9] Klyne G., Caroll J.J. (Eds.), Resource Description Framework (RDF): Concepts and Abstract Syntax, W3C, 2004.

[10] Brockmans, S., Haase, P., Hitzler, P., Studer, R., A Metamodel and UML Profile for Rule-Extended OWL DL Ontologies., In proceedings of 3rd European Semantic Web Conference, ESWC 2006, Budva, Montenegro, June 11-14, 2006, Springer Verlag.

Generating Visual Editors Based on Fujaba/MOFLON and DiaMeta

Mark Minas
Universitat der Bundeswehr Munchen
85577 Neubiberg, Germany
Mark.Minas@unibw.de

ABSTRACT

DiaMeta is a tool for generating visual editors supporting free-hand and structured editing. The visual language's abstract syntax has to be specified in terms of a meta-model. As a meta-modeling framework, *DiaMeta* has used the Eclipse Modeling Framework EMF. This paper describes how *DiaMeta* now makes use of *Fujaba/MOFLON* as a meta-modeling framework and, hence, the MOF 2.0 standard. This allows for more expressive specifications of visual languages because MOF 2.0 is more powerful than EMF's meta-model Ecore.

1. INTRODUCTION

Model driven development is generally based on domain specific models that have to be created and edited by customized visual editors. In order to reduce efforts to create such customized visual editors, several tools have been developed that allow to specify domain-specific visual languages and to generate visual editors from those specifications. Examples are MetaEdit [9], Atom^3 [5], Pounamu [16], the Generic Modeling Environment GME [8] and Tiger [6]. All of them use meta-modeling for specifications, i.e., the modeling language is specified in terms of a model that is essentially a class diagram. The meta-model actually specifies the language's abstract syntax and, at the same time, the data structure for representing models. Specification of concrete syntax is greatly simplified by the fact that those tools are restricted to graph-like visual languages. It is essentially a mapping of the internal model representation to visual representations on the screen. Editors allow to edit diagrams in a structured way, this is by selecting operations that modify the internal model and adjust its visual representation.

Directly modifications of the visual representation by the user, i.e., free-hand editing, is not supported by such editors. Moreover, languages that are not graph-like (e.g, hierarchical languages like Statecharts, Sequence diagrams, or Nassi-Shneiderman diagrams) are either not supported or supported by specific built-in extensions (e.g. nodes containing other nodes). Free-hand editing and specification of languages that are not graph-like are supported by tools that use some kind of grammar for specification of a language's abstract syntax. Examples are VLDesk (extended positional grammars) [4], Penguins (constraint multiset grammars) [3], and DiaGen (hypergraph grammars) [10, 11]. Grammars are more powerful than meta-models. However, meta-model-based tools are more successful than grammar-based tools since meta-models are simpler to use than grammars when specifying (visual) languages.

The tool *DiaMeta* for generating visual editors from a specification aims at combining the benefits of meta-model-based and grammar-based specifications. Abstract syntax is specified in terms of a meta-model. However, being a successor of the grammar-based tool *DiaGen*, *DiaMeta* is not restricted to graph-like languages, and it supports free-hand as well as structured editing. A recent paper [12] has described *DiaMeta* in a version that is based on the *Eclipse Modeling Framework* EMF [7, 2] as a meta-modeling framework. Free-hand editing is supported by internally representing a diagram by a graph that is then checked against the meta-model. This kind of syntax analysis is carried out as a constraint satisfaction problem that can be solved in an efficient way [13].

This paper describes *DiaMeta* in a different and yet preliminary version that is based on *Fujaba/MOFLON* [1] instead of on EMF. *Fujaba/MOFLON* supports MOF 2.0 [14] while EMF is based on the less powerful Ecore. MOF 2.0, hence, is better suited for meta-model-based language specifications. This is demonstrated in this paper by the (very simple) example of Petri nets.

The next section describes the syntax specification with meta-models. Moreover, the basic concepts of syntax analysis as described in [13] are outlined. Section 3 introduces the common editor architecture of each editor built using *DiaMeta* and the diagram analysis when editing a diagram in free-hand mode based on the concepts presented in section 2. Section 4 presents details of the *DiaMeta* environment, in particular on specification and code generation. Section 5 concludes the paper.

2. SYNTAX SPECIFICATION AND ANALYSIS BASED ON META-MODELS

Petri nets are used as a running example of this paper. Fig. 2 shows such a Petri net consisting of four places *P1*, *P2.1*, *P2.2*, and *P2.3* as well as two transitions *T1* and *T2*. Integers near a place or an arc express the capacity of the place or the transition, respectively. As usual, a capacity of 1 is not displayed. Fig. 2 is actually a screenshot of an editor that has been specified and generated using *DiaMeta* and *Fujaba/MOFLON*.

A Petri net's abstract concepts are *places*, *transitions*, *pre*

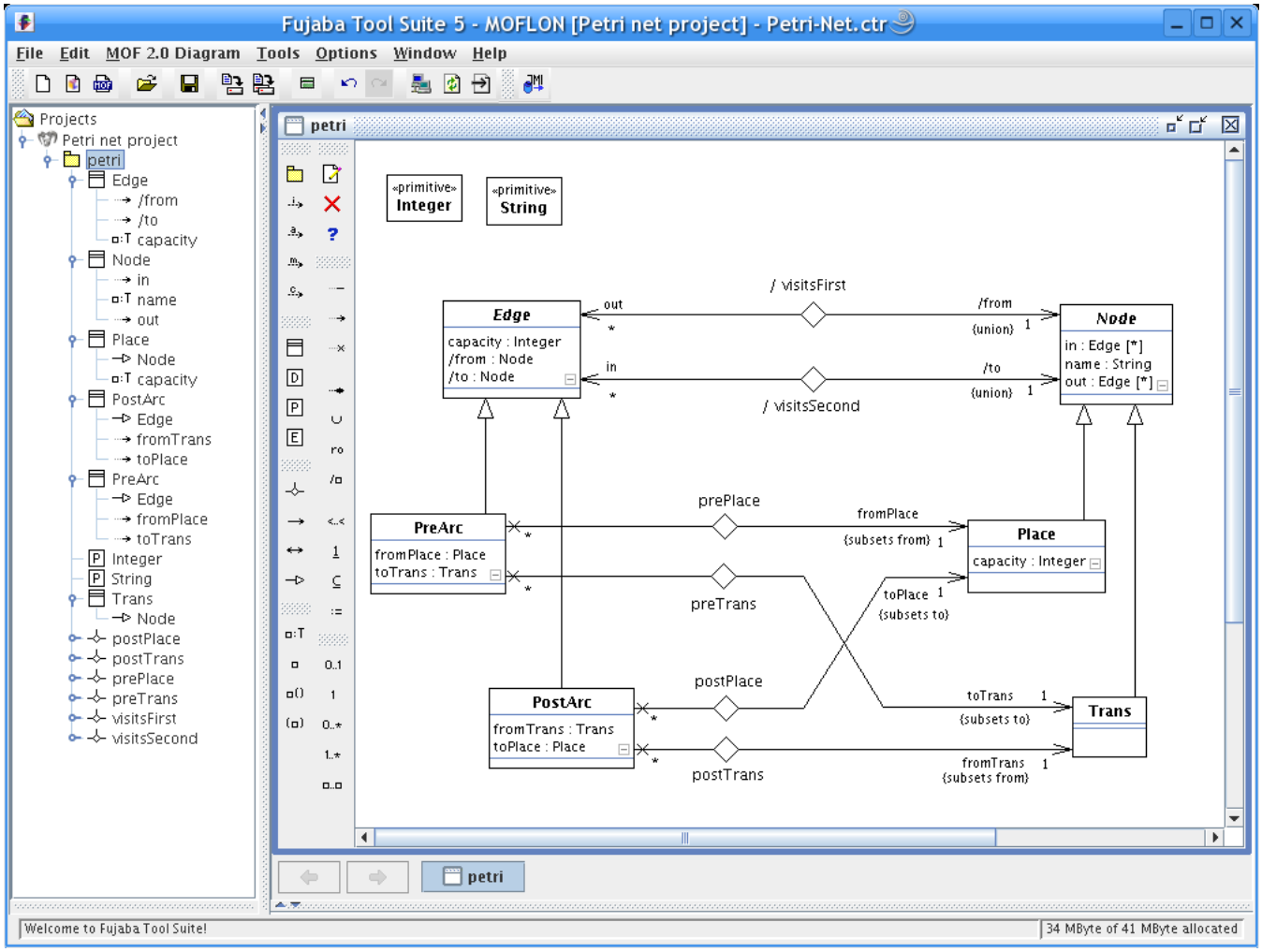


Figure 1: Screenshot of *Fujaba/MOFLON* with the meta-model of the abstract syntax of Petri nets.

arcs, and *post arcs*. The former two concepts are the nodes of a Petri net, the latter ones its edges. A pre arc is an edge from a place to a transition whereas a post arc connects a transition with a place. Places and transitions carry a name, and places, pre arcs as well as post arcs have an integer capacity. These concepts and their interrelation are specified in terms of a meta-model that is shown in the screenshot of *Fujaba/MOFLON* in Fig. 1.

Meta-models specified in *Fujaba/MOFLON* follow the MOF 2.0 standard [14]. Concepts are specified by classes, e.g., *Place*. Attributes of a class are drawn similar to attributes in UML class diagrams. E.g., class *Place* has the integer attribute *capacity*. Subclassing is used to create subtypes that inherit all features of their superclasses. E.g., class *Place* is specified as a subclass of the abstract class *Node*, and *Place* inherits all attributes (and associations) of *Node*, e.g., string attribute *name*. Relations between concepts, i.e., classes, are represented by associations shown as diamonds with connections (*member ends*) to the related classes. E.g., association *visitsFirst* is an association between classes *Edge* and *Node*, indicated by the navigable member ends with roles *from* and *out*. Navigable member ends are drawn as arrows whereas non-navigable member ends are crossed out (e.g. the member end of association *prePlace* to class *PreArc*). Multiplicities are indicated as usual. E.g., cardinalities of the member ends of association *visitsFirst* express that an arbitrary number of edges can be related to a single node, but that each edge is related to exactly one node. This association, actually, represents the source node of an edge.

Note that the member end *from* of association *visitsFirst* (and member end *to* of *visitsSecond* in an analogous way) is specified as *derived union* (*derived* is indicated by a slash, and *union* by {*union*}) which corresponds to the member ends *fromPlace* and *fromTrans* of associations *prePlace* and *postTrans*, resp., that are specified {*subsets from*}. This is a specific feature of MOF 2.0 that is not available in EMF with Ecore. That way, associations of subclasses can be specified more precisely. In the example, the meta-model describes that each edge visits exactly two nodes. However, concrete subclasses visit different kinds of nodes at their source and target. Whereas a *PreArc* instance visits *Place* and *Trans* instances as their source and target node, respectively, a *PostArc* instance visits *Trans* and *Place*. Without the con-

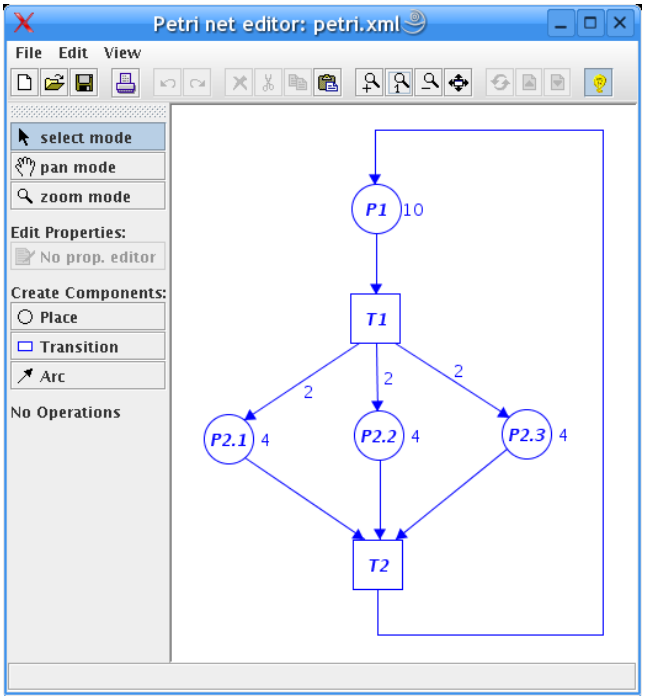


Figure 2: Screenshot of the generated Petri net editor.

cept of unions and subsets, different kind of edges could not be generalized to a common *Edge* class that has associations indicating its visited nodes. Instead, this fact would have had to be explicitely programmed by operations *from* and *to*.

It is obvious that this feature makes MOF 2.0 more powerful than EMF with Ecore. It allows to specify the precise associations of different node and edge types, but allows to access nodes and edges at the same time as abstract concepts. This can be used directly, e.g., when employing a layouter that simply uses the abstract concepts of nodes and edges as well as their interrelations that can be accessed while ignoring the concrete classes.

Additional constraints on the object structures may be used to restrict the set of all valid object structures even more. The Unified Modeling Language UML allows for expressing such constraints using the Object Constraint Language OCL. However, we will ignore additional constraints in the following.

Fig. 2 shows a valid Petri net according to the meta-model shown in Fig. 1. The Petri net is valid because the abstract representation of the Petri net, as shown in Fig. 3, conforms to the meta-model. Fig. 3 is a graph (called *instance graph*) whose rectangular nodes are instances of classes in the meta-model, oval nodes are specific attributes. Arrows indicate links, i.e., instances of associations in the meta-model. The instance graph conforms to the meta-model if nodes and links are connected as specified in the meta-model. Formally, a kind of homomorphism must exist from the instance graph to the meta-model that regards subclassing.

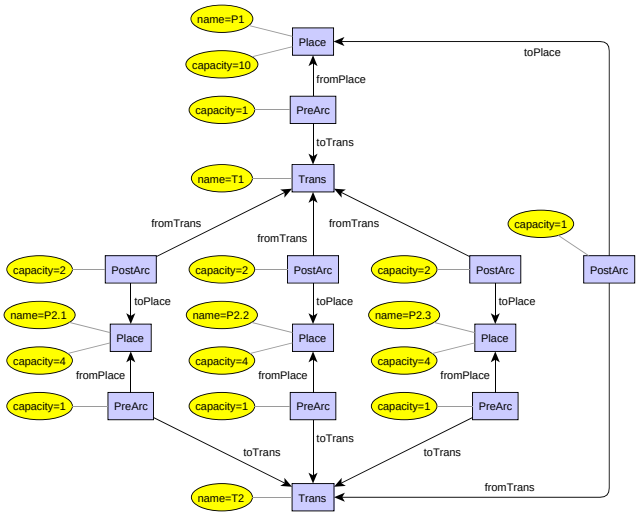


Figure 3: Instance graph of the Peri net shown in Fig. 2.

In general, as considered in more detail in [12, 13], instance graph nodes need not carry a concrete class type like in Fig. 3, but the type of an (abstract) superclass. This is so because the instance graph is built in the course of diagram analysis after modifying the visual representation of a diagram in free-hand style as described in the next section. And different concepts of the abstract syntax, i.e., meta-model, may have indistinguishable visual representations. E.g., an arrow in the visual representation of a Petri net may be a pre arc or an post arc, depending on its context.[1] In the simple example of Petri nets, these kinds of arcs are easily distinguishable as shown in section 3.4. Otherwise, the common superclass *Edge* would have had to be used instead of *PreArc* and *PostArc*. And then, checking whether the instance graph conforms to the meta-model does not only consist of checking, but also of finding concrete subclasses for such nodes such that each node carries a concrete subclass of its original label and conforms to the meta-model. In [13], this kind of syntax analysis problem has been described as a *constraint satisfaction problem* (CSP) that can be preprocessed in linear time in the size of the analyzed diagram. Experiments had shown that backtracking was never required after preprocessing the CSP, i.e., syntax analysis is efficient when using meta-model-based syntax specifications.

3. DIAMETA EDITORS

DiaMeta provides an environment for rapidly developing diagram editors based on meta-modeling. Diagram editors developed using *DiaMeta* (such editors are called "*DiaMeta* editors" in the following) always support free-hand editing. Each *DiaMeta* editor is based on the same editor architecture which is adjusted to the specific diagram language. This architecture is described in the following. *DiaMeta*'s tool support for specification and code generation, primarily the *DiaMeta Designer*, are postponed to section 4.

[1]Note that the editor shown in Fig 2 provides a single button (labeled *arc* and shown with an arrow icon) for creating arrows inside the editor control on the left. The concrete type of an arrow is recognized by the editor automatically.

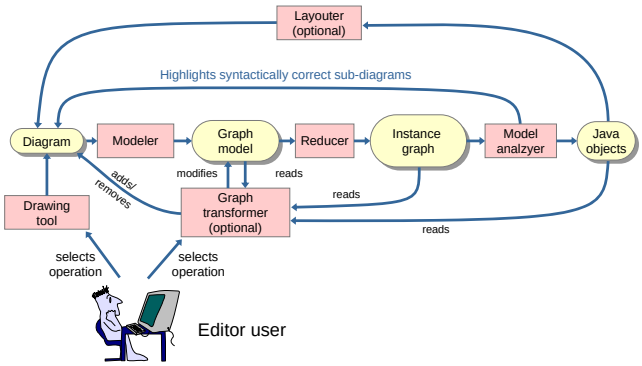


Figure 4: Architecture of a diagram editor based on *DiaMeta*.

3.1 DiaMeta editor architecture

Since *DiaMeta* is actually an extension of the diagram editor generator *DiaGen* [10, 11], *DiaMeta* editors have a similar structure like *DiaGen* editors. Fig. 4 shows the structure which is common to all *DiaMeta* editors and which is described in the following paragraphs. Ovals are data structures, and rectangles represent functional components. Flow of information is represented by arrows. If not labeled, information flow means reading reps. creating the corresponding data structures. The structure of *DiaMeta* editors is very similar to *DiaGen* editors; they most prominently differ in the method of abstract syntax specification and, hence, syntax analysis. Moreover, *DiaGen* requires a specification of attribute evaluation for creating abstract diagram representations. Since meta-models as abstract syntax specification are also a specification of abstract diagram representations, *DiaMeta* does not require such an additional specification.

The editor supports free-hand editing by means of the included drawing tool which is part of the editor framework, but which has been adjusted by the *DiaMeta Designer*. With this drawing tool, the editor user can create, arrange and modify diagram components which are specific to the diagram language. Editor specific program code, which has been specified by the editor developer and generated by the *DiaMeta Designer*, is responsible for the visual representation of these language specific components. The drawing tool creates the data structure of the diagram as a set of diagram components together with their attributes (position, size, etc.).

The sequence of processing steps necessary for free-hand editing starts with the *modeler* and ends with *model analyzer* (cf. Fig. 4): The modeler first transforms the diagram into an internal model, the *graph model*. The *reducer* then creates the diagram's *instance graph* that is analyzed by the *model analyzer*. This last processing step identifies the maximal sub diagram which is (syntactically) correct and provides visual feedback to the user by drawing those diagram components with a certain color; errors are indicated by missing colors. However, the model analyzer not only checks the diagram's abstract syntax, but also creates the object structure of the diagram's syntactically correct sub diagram.

The *layouter* modifies attributes of diagram components and thus the diagram layout based on the (syntactically correct sub diagram's) object structure. The layouter is necessary for realizing syntax-directed editing: Structured editing operations modify the graph model by means of the *graph transformer* and add or remove components to resp. from the diagram. The visual representation of the diagram and its layout is then computed by the layouter. However, layouters and structured editing are not considered in this paper.

The processing steps necessary for free-hand editing are described in more detail in the following.

3.2 Diagram components

Each diagram consists of a finite set of diagram components, each of which is determined by its attributes. For the diagram language of Petri nets, there are places, transitions, and arrows. Each place is a circle whose position is defined by its *xPos* and *yPos* coordinates, its inscribed name by a *text* attribute, and its capacity shown near the border. Transitions are simply squares with their inscribed name, and each arc is an arrow whose position is defined by its two end points, i.e., by two coordinate pairs *xPos1* and *yPos1* resp. *xPos2* and *yPos2*. Arcs, moreover, have an integer attribute representing the arc's capacity. All of these attributes are necessary for completely determining a diagram component, e.g., when storing it to a file or retrieving it again. However, only some of these attributes are essential for a diagram's abstract syntax, too. In the Petri net example, only the inscribed text of nodes and the capacity of places and arcs are part of the abstract syntax as shown in Fig. 1. Position attributes are solely member of the concrete syntax.[2]

Diagrams are checked for their correctness in terms of their instance graphs that are the result of a translation process from the diagram's concrete syntax, i.e., mainly the arrangement of is diagram components. This process is described in the following, and it requires an intermediate, uniform representation of the analyzed diagram, its *graph model*.

3.3 Graph model

Arrangements of diagram components can always be described by spatial relationships between them. For that purpose, each diagram component typically has several distinct *attachment areas* at which it can be connected to other diagram components. An arrow representing an arc in a Petri net, e.g., has its two end points as attachment areas. Connections can be established by spatially related (e.g., overlapping) attachment areas as with Petri nets where an arrow has to end at the border of a place's circle or a transition's square in order to be connected to the the place resp. the transition.

DiaMeta uses directed graphs to describe a diagram as a set of diagram components and the relationships between attachment areas of "connected" components. Each diagram

[2]The EMF version of *DiaMeta* allows to specify all attributes, even those of the concrete syntax, in the meta-model. The meta-model, hence, specifies the language's abstract syntax and, at the same time, some aspects of the concrete syntax [12].

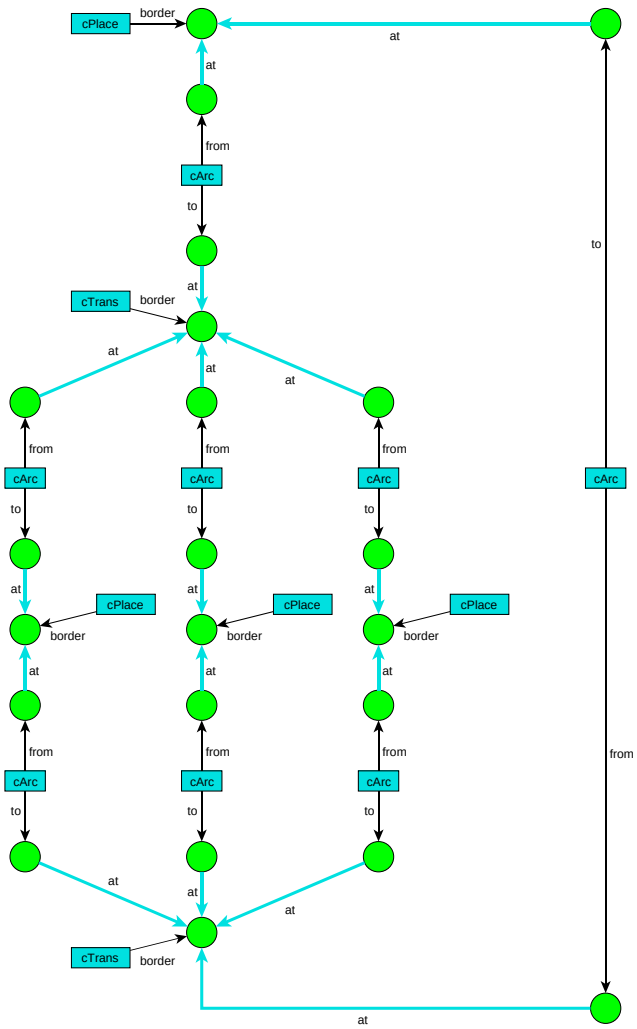
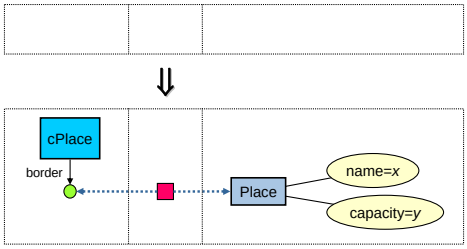


Figure 5: Graph model of the Peri net shown in Fig. 2.

component is modeled by a node (called *component node*) whose type is determined by the kind of represented diagram component. Attachment areas are also modeled by nodes (called *attachment nodes*). Edges (called *attachment edges*) connect component nodes with attachment nodes for all attachment areas that belong to a component. Edge labels are used to distinguish different attachment areas. Relationships between attachment areas are modeled by edges (called *relationship edges*) connecting the corresponding attachment nodes. Relationship edges carry the kind of relationship as edge type.

Fig. 5 shows the graph model of the Petri net in Fig. 2. Attachment nodes are depicted by circles, component nodes by rectangles. Thin arrows show attachment edges, thick arrows relationship edges. Each place (type *cPlace*) and transition (type *cTrans*) has its *border* as its only attachment area, whereas arcs (type *cArc*) have attachment areas *from* and *to*. The only relationship type *at* indicates that the end point of the corresponding arrow is at the border of the connected place resp. transition.



x = cPlace.name;
y = cPlace.capacity;

Figure 6: Reducer rule for places.

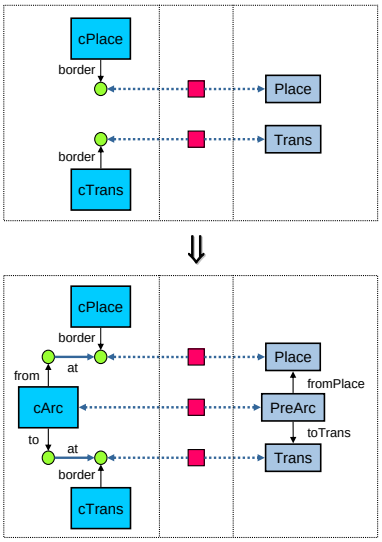


Figure 7: Reducer rule for arcs from a place to a transition.

3.4 Instance graph

The *reducer* is responsible for translating the graph model to the corresponding instance graph. The translation process has to be specified in the *DiaMeta Designer* in terms of *reducer rules*. Fig. 6 and Fig. 7 show two of the four rules specifying the reducer for the diagram language of Petri nets. Reducer rules are triple graph grammar rules [15] that are used for translating graph models to instance graphs. Dotted borders in Fig. 6 and 7 indicate how the graph model (left compartment), interface (center compartment), and instance graph (right compartment) are modified.

Fig. 6 shows that a *cPlace* component node together with its attachment node are translated into a *Place* node of the instance graph. The attributes *name* and *capacity* of the *Place* node are set depending on the diagram component's corresponding attributes. An analogous rule is responsible to translate transitions.

Fig. 7 translates arcs that are pre arcs into *PreArc* nodes that are connected correctly. Note that the corresponding nodes of left-hand side and right-had side have not been indicated explicitely in order to avoid cluttering of the representation. An analogous rule is responsible to translate post arcs.

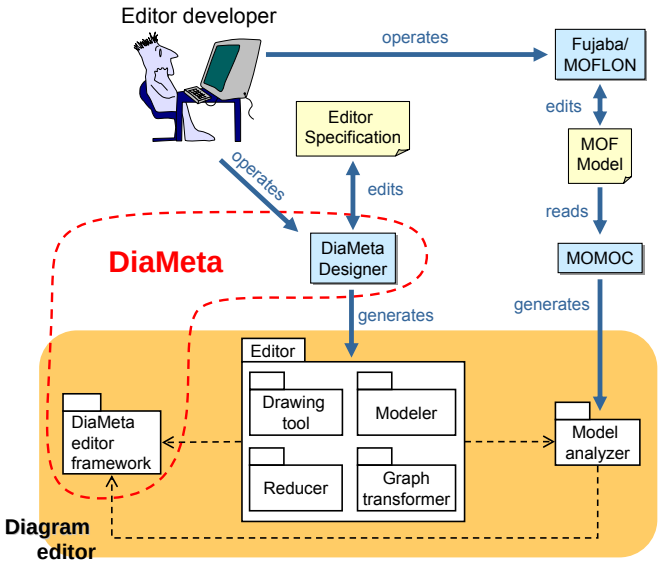


Figure 9: Generating diagram editors with *DiaMeta* and *Fujaba/MOFLON*.

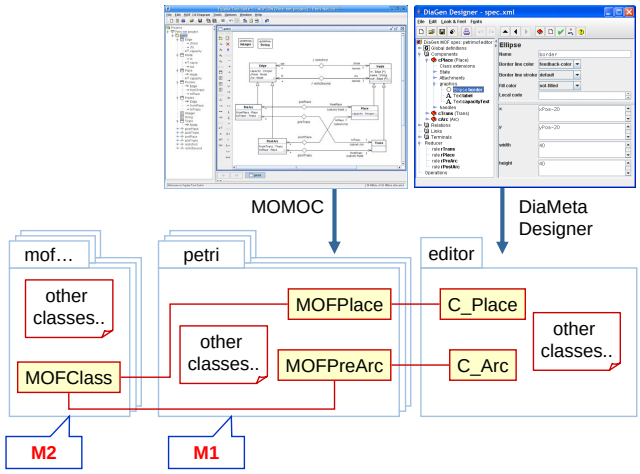


Figure 10: Generated Java classes from the specification.

3.5 Model analyzing

Finally, the instance graph is checked against the diagram language's meta-model as described in [13] and briefly outlined in section 2. By solving a constraint satisfaction problem, the model analyzer tries to identify a maximal subgraph of the instance graph that corresponds to the meta-model. For the simple language of Petri nets, simply checking all constraints is sufficient because the concrete class of each instance graph node has already been recognized by the reducer.

The identified maximal subgraph is used to instantiate the classes specified by the meta-model; the obtained structure of Java objects is an abstract representation of the diagram that can be used when integrating the editor in a larger environment. As a demonstration of this concept, Fig. 8 shows an XMI serialization of the object structure created for the Petri net shown in Fig. 2. Each object representation carries a unique numerical identifier `xmi:id`. Links between objects are represented by referencing the other object by its identifier.

The subgraph, moreover, corresponds to a subgraph of the graph model and, hence, a subset of diagram components that form a syntactically correct subdiagram. Based on this information, feedback to the user is provided as described in section 3.1.

4. DIAMETA ENVIRONMENT

This section completes the description of *DiaMeta* and outlines its environment that supports specification and code generation of diagram editors that are tailored to specific diagram languages. The *DiaMeta* environment shown in Fig. 9 consists of an editor framework and the *DiaMeta Designer*. The framework is an extension of the *DiaGen* framework and, as a collection of Java classes, provides the generic editor functionality which is necessary for editing and analyzing diagrams. In order to create an editor for a specific diagram language, the editor developer has to provide two specifications: First, the abstract syntax of the diagram language in terms of a meta-model, and second, the visual appearance of diagram components, the concrete diagram language syntax, the reducer rules and the interaction specification.

DiaMeta as described in this paper uses *Fujaba/MOFLON* as a meta-modeling framework. *Fujaba/MOFLON* consists of a MOF editor that edits the MOF model and a code generator *Momoc* that reads the MOF model and generates Java classes that implement the specified meta-model. The generated classes, which are shown as package *Model analyzer* in Fig. 9, are responsible for model analyzing as shown in Fig. 3.1 and described in Sec. 3.5.

The editor developer uses the *DiaMeta Designer* for specifying the concrete syntax and the visual appearance of diagram components, i.e., that Petri net places are drawn as circles with inscribed name and arcs as arrows. The *DiaMeta Designer* generates Java code from this specification. This code, which is shown as package *Editor* with sub-packages *Drawing tool*, *Modeler*, *Reducer*, and *Graph transformer* in Fig. 9, implement the language specific aspects of the corresponding components in Fig. 4. Together with package *Model Analyzer* generated by *Momoc* and the editor framework, this code implements an editor for the specified diagram language.

Note that *DiaMeta Designer* and *Fujaba/MOFLON* do not share any information in the current state of the *DiaMeta* environment. However, the generated code has to cooperate such that the instance graph created by the reducer can be analyzed by the model analyzer. This is possible as long as labels of nodes and edges created by reducer rules that are specified using the *DiaMeta Designer* correspond with class and association names specified using *Fujaba/MOFLON*. However, there is no tool support yet that checks this correspondence.

Fig. 10 shows some concrete Java classes and packages that are taken from the framework resp. that are generated by

```
<xmi:XMI xmlns:xmi="http://www.omg.org/XMI" xmi:version="1.0">
  <petri.PreArc capacity="1" from="21023091" fromPlace="21023091" to="22116759" toTrans="22116759" xmi:id="19538521"/>
  <petri.Trans in="29012646" name="T1" out="19898081 31989064 2763497" xmi:id="19253663"/>
  <petri.PreArc capacity="1" from="6066422" fromPlace="6066422" to="19253663" toTrans="19253663" xmi:id="29012646"/>
  <petri.Place capacity="4" in="31989064" name="P2.3" out="4316281" xmi:id="32844677"/>
  <petri.Place capacity="4" in="2763497" name="P2.1" out="3802505" xmi:id="26678250"/>
  <petri.Place capacity="4" in="19898081" name="P2.2" out="19538521" xmi:id="21023091"/>
  <petri.PostArc capacity="2" from="19253663" fromTrans="19253663" to="26678250" toPlace="26678250" xmi:id="2763497"/>
  <petri.PostArc capacity="1" from="22116759" fromTrans="22116759" to="6066422" toPlace="6066422" xmi:id="29165216"/>
  <petri.PostArc capacity="2" from="19253663" fromTrans="19253663" to="32844677" toPlace="32844677" xmi:id="31989064"/>
  <petri.PreArc capacity="1" from="26678250" fromPlace="26678250" to="22116759" toTrans="22116759" xmi:id="3802505"/>
  <petri.PreArc capacity="1" from="32844677" fromPlace="32844677" to="22116759" toTrans="22116759" xmi:id="4316281"/>
  <petri.Trans in="3802505 19538521 4316281" name="T2" out="29165216" xmi:id="22116759"/>
  <petri.PostArc capacity="2" from="19253663" fromTrans="19253663" to="21023091" toPlace="21023091" xmi:id="19898081"/>
  <petri.Place capacity="10" in="29165216" name="P1" out="29012646" xmi:id="6066422"/>
</xmi:XMI>
```

Figure 8: XMI representation of the object structure that the generated editor creates from the Petri net shown in Fig. 2.

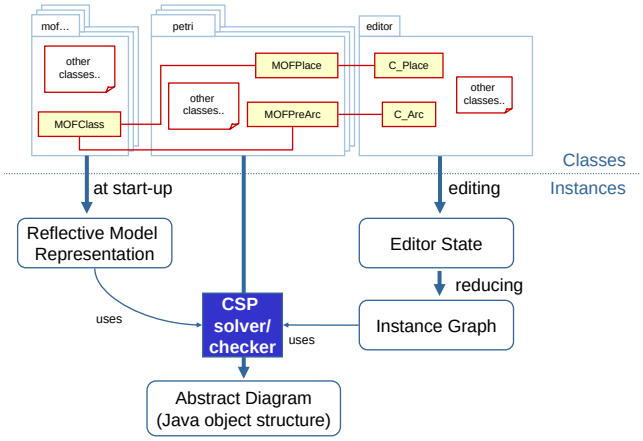


Figure 11: Using the reflective model representation for checking the diagram's syntax by constraint satisfaction resp. checking.

Momoc and the *DiaMeta Designer* for our example of Petri net diagrams. *Momoc* creates actually some packages (called *petri* in Fig. 10) that contain all classes corresponding to the language's meta-model. The *DiaMeta Designer* generates the package *editor* together with the classes that are responsible for visualizing diagram components, interacting with them in the editor, and language specific classes for diagram analysis as outlined in Fig. 9.

The classes in packages *petri* implement the meta-model and, hence, the abstract syntax of the specified diagram language. However, *Momoc* does not only generate these classes, but also code that sets up a reflective meta-model representation at start-up time as shown in Fig. 11. The reflective model representation represents the meta-model using the meta-meta-model of MOF 2.0. This reflective representation allows to inspect the meta-model, i.e., the abstract diagram at editor runtime. In terms of the meta-modeling hierarchy, the abstract syntax of a specific diagram is on the M_0 level. Its meta-model is on the M_1 level. The meta-meta-model is the model of all meta-models and, hence, on the M_2 level. These meta-meta-model classes of the M_2 level are instantiated at start-up time such that these instances provide a runtime data structure representing the specified meta-model. The model analyzer makes use of this runtime data structure in order to inspect the meta-model and to find all possible concrete classes for each node of the instance graph and all possible associations for each edge in the instance graph.

5. CONCLUSIONS

The paper has described *DiaMeta* a tool for generating visual editors that support free-hand and – at the same time – structured editing. However, structured editing has not been considered in this paper. The new contribution of *DiaMeta* consists of the combination of free-hand editing and a diagram language specification based on a meta-modeling approach. In other approaches, meta-model-based editors are restricted to structured editing, and specified visual languages must be graph-like. *DiaMeta* first translates the uniform graph model of a diagram into another graph representation (the *instance graph*). This separation of diagram representation and syntax analysis allows for specification of visual languages that are not graph-like as already demonstrated by *DiaMeta*'s predecessor *DiaGen*. The focus of this paper lies on a new version of *DiaMeta* using *Fujaba/MOFLON* as a meta-modeling framework. Different from the *DiaMeta* version based on EMF, this allows to use the MOF 2.0 standard for specifications of visual languages which is more powerful than EMF with Ecore.

There are several lines of current and future work. The described version of *DiaMeta* based on *Fujaba/MOFLON* is still preliminary. As shown in Fig. 8, the object structure does not contain position and size information belonging to a diagram's concrete syntax. However, this information is necessary if a diagram is going to be restored from the object structure, e.g., after some modifications have been made by another tool. The version of *DiaMeta* based on EMF supports such object models already [12]. Similar support will be added to the described version based on *Fujaba/MOFLON* as soon as some required extensions are available in *Fujaba/MOFLON*.

The paper has described the architecture of visual editors generated by *DiaMeta*, and diagram analysis that checks the correctness of freely drawn diagrams and translates them – if they are correct – into some object structure. However, there are still unsolved questions. The predecessor of *DiaMeta*, *DiaGen*, that used a grammar-based syntax specification, could easily identify maximal subdiagrams that are syntactically correct; *DiaMeta* with its model-based syntax specification and its syntax analysis based on a constraint satisfaction problem does not yet provide as satisfying results as *DiaGen*.

Moreover, *DiaMeta* does not yet support additional constraints on the object structures. Such constraints are required if the diagram language is not fully specified by a class diagram. OCL support by *Fujaba/MOFLON* will automatically allow for specifying and checking such constraints when editing diagrams in generated editors.

The current *DiaMeta* implementation makes use of MOF 2.0 for modelling diagram languages and for providing an implementation. The languages of the UML and MOF itself is specified by MOF models, too. An apparent application of *DiaMeta*, hence, is generating editors for UML languages and MOF 2.0; only the corresponding specifications by the *DiaMeta Designer* are missing.

6. REFERENCES

[1] C. Amelunxen, A. Königs, T. Rötschke, and A. Schürr. MOFLON: A standard-compliant metamodeling framework with graph transformations. In A. Rensink and J. Warmer, editors, *Model Driven Architecture - Foundations and Applications: Second European Conference*, volume 4066 of *Lecture Notes in Computer Science*, pages 361–375, Heidelberg, 2006. Springer Verlag, Springer Verlag.

[2] F. Budinsky, D. Steinberg, E. Merks, R. Ellersick, and T. J. Grose. *Eclipse Modeling Framework*. The Eclipse Series. Addison Wesley, 2004.

[3] S. S. Chok and K. Marriott. Automatic construction of user interfaces from constraint multiset grammars. In *Proc. 1995 IEEE Symp. on Visual Languages, Darmstadt, Germany*, pages 242–249. IEEE Computer Society Press, Sept. 1995.

[4] G. Costagliola, V. Deufemia, and G. Polese. Towards syntax-aware editors for visual languages. *Electronic Notes in Theoretical Computer Science*, 127(4):107–125, Apr. 2005. Proc. Workshop on Visual Languages and Formal Methods (VLFM 2004).

[5] J. de Lara, H. Vangheluwe, and M. Alfonseca. Meta-modelling and graph grammars for multi-paradigm modelling in atom3. *Software and Systems Modelling*, 3(3):194–209, Aug. 2004.

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

[7] EMF web site. http://www.eclipse.org/emf/, 2006.

[8] Generic Modeling Environment (GME) web site. http://www.isis.vanderbilt.edu/projects/gme/, 2006.

[9] Metaedit+ documentation on the MetaCase web site. http://www.metacase.com/, 2006.

[10] M. Minas. Concepts and realization of a diagram editor generator based on hypergraph transformation. *Science of Computer Programming*, 44(2):157–180, 2002.

[11] M. Minas. VisualDiaGen – a tool for visually specifying and generating visual editors. In J. L. Pfaltz, M. Nagel, and B. Böhlen, editors, *Applications of Graph Transformation with Industrial Relevance, Proc. 2nd Intl. Workshop AGTIVE'03, Charlottesville, USA, 2003, Revised and Invited Papers*, volume 3062 of *Lecture Notes in Computer Science*, pages 398–412. Springer-Verlag, 2004.

[12] M. Minas. Generating meta-model-based freehand editors. Appears in Proc. 3rd International Workshop on Graph Based Tools (GraBaTs'06), Natal (Brazil), September 21-22, 2006.

[13] M. Minas. Syntax analysis for diagram editors: A constraint satisfaction problem. In A. Celentano and P. Mussio, editors, *Proc. of the Working Conference on Advanced Visual Interfaces (AVI'2006), May 23-26, 2006, Venice, Italy*, pages 167–170. ACM Press, 2006.

[14] Object Management Group. *Meta Object Facility (MOF) Core Specification*, version 2.0 edition, Jan. 2006. Document - formal/06-01-01.

[15] A. Schürr. Specification of graph translators with triple graph grammars. In G. Tinhofer, editor, *20th Int. Workshop on Graph-Theoretic Concepts in Computer Science*, volume 903 of *Lecture Notes in Computer Science*, pages 151–163, Heidelberg, 1994. Springer Verlag.

[16] N. Zhu, J. Grundy, and J. Hosking. Pounamu: A meta-tool for multi-view visual language environment construction. In *Proc. 2004 IEEE Symposium on Visual Languages - Human Centric Computing (VL/HCC'04)*, pages 254–256, 2004.

Integrating Fujaba and the Eclipse Modeling Framework

Jendrik Johannes
TU Dresden
Fakultät Informatik
Institut für Software- und Multimediatechnologie
D-01062 Dresden
jendrik.johannes@mailbox.tu-dresden.de

Ilie Savga
TU Dresden
Fakultät Informatik
Institut für Software- und Multimediatechnologie
D-01062 Dresden
ilie.savga@tu-dresden.de

Tobias Haupt
TU Dresden
Fakultät Informatik
Institut für Software- und Multimediatechnologie
D-01062 Dresden
s0500251@inf.tu-dresden.de

ABSTRACT

With Fujaba4Eclipse the Fujaba Tool Suite [18] is intergrated into the Eclipse Platform [5, 13]. Through this integration, Fujaba benefits from the stable and well-documented Eclipse infrastructure and from re-use of some Eclipse tools, such as the Graphical Editing Framework (GEF) [16]. Still, there are other powerful Eclipse technologies, which could be enabled in Fujaba. We show, how this can be done by the Eclipse Modeling Framework (EMF) [11, 3]. More exactly, we demonstrate, how the EMF's metamodel Ecore is used to define EMF-complient Fujaba metamodels. The benefit is twofold. First, Fujaba technologies (e.g., pattern recognition) can be applied to EMF models directly in Eclipse. Second, existing EMF-based tools and applications become applicable to Fujaba models.

1. INTRODUCTION

By integrating Eclipse and Fujaba, both platforms can profit from each others technologies. Fujaba4Eclipse is a first step of that integration. It relies on the Eclipse plugin mechanism and on the Eclipse Graphical Editing Framework (GEF). Still, it uses its own representation of modeling concepts, which limits the interoperability with other Eclipse based modeling tools.

To alleviate the problem of interoperability, we propose to improve modeling (and metamodeling) in Fujaba towards standards such as the OMG metamodeling hierarchy (Figure 1). This hierarchy is a foundation for most CASE environments and enables compatibility and exchangeability of models. Such environments can be developed, for instance, using the Eclipse Modeling Framework (EMF), which complies to OMG standards.

"EMF is a modeling framework and code generation facility for building tools and other applications based on a structured data model."[11] Models defined in EMF have a common metamodel called *Ecore*, which is essentially a subset of OMG's MOF standard [8]. This, in turn, implies a use of Ecore as a metametamodel (i.e. a model to define metamodels)[1]. In fact, Ecore is defined in terms of itself, too.

The integration with Ecore opens Fujaba tooling to a broader community by allowing distinct Fujaba features, like the *pattern recognition* mechanism or support for *story driven modeling*, to be used with models developed in other Ecore aware applications. On the other hand, Fujaba models can be passed to other applications for further use. Such applications are, for example, EclipseUML [10], Rational Rose [7] or E-Comogen [6], the latter being a software composition tool developed at our institute that employs Ecore models to describe languages.

As a consequence of the integration, Fujaba's modeling can further profit from other standards implemented by the EMF community. Those include, among others, a common UML2 metamodel implementation [15], a common *Ontology Definition Metamodel* implementation [14] and OCL integration [17] (both of the latter are part of the *EMF technologies* subproject [12] which contains more such standard implementations).

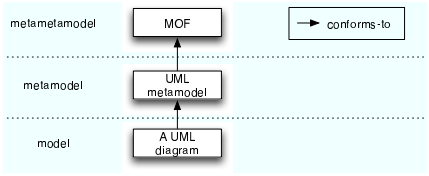


Figure 1: The OMG's metamodeling hierarchy.

To gain compliance to the metamodeling hierarchy, we propose to use Ecore in two ways. First, we require Fujaba metamodels to be defined in Ecore (using Ecore as metametamodel). There is no explicit Fujaba metametamodel at the moment, all implemented metamodels use Fujaba's *metamodel framework*. We show that there does exist an (implicit) metametamodel and how it corresponds to Ecore. Second, we argue that there are correspondences between Ecore as a metamodel and Fujaba metamodels (such as the Fujaba's UML metamodel). Once these correspondences are defined, models (and metamodels) can be exchanged between the two platforms.

[1]One could argue, that, when used to define other models, Ecore serves as a (meta) language, rather than a metamodel. For the sake of simplicity, in this paper we intentionally avoid discussions of relations between modeling languages and the metamodel hierarchy. See [4] for a deep discussion on that matter.

The rest of the paper is organized as follows. Section 2 gives an overview of Ecore. Sections 3 and 4 present our two main ideas of how to approach the integration of Eclipse modeling and Fujaba. After discussing the evaluation of our proposal in Section 5, we shortly present related work in Section 6 and conclude in Section 7.

2. ECORE: OVERVIEW

Ecore, based on the object-oriented paradigm, is used to define models (then it is a metamodel, comparable to the UML metamodel) or metamodels (then it is a metametamodel, comparable to MOF). Many elements found in Ecore are equivalent to elements found in UML class diagrams. Thus, one way to specify Ecore models is by means of UML class diagrams. As mentioned, Ecore is expressed in terms of itself. Figure 2 shows how Ecore's basic subset, called kernel, is defined in UML and figure 3(a) shows a simple Ecore model, also in UML.

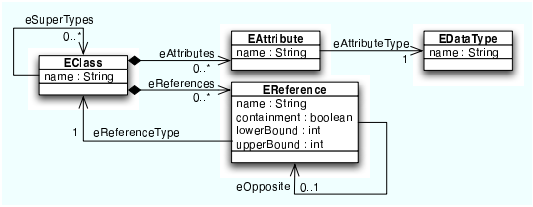


Figure 2: The Ecore kernel [3].

Alternatively to using UML class diagrams, one can define Ecore models by Java interfaces and classes (Figure 3(b)). Such interfaces contain Javadoc annotations to identify model elements and include further constrains which can not be expressed directly in Java. A model expressed in annotated Java can be translated to an equivalent UML representation and back, as both are serializable to XMI (XML Metadata Interchange) [9].

From a model (regardless of how it is expressed), annotated Java code is generated by the code generation module. For the annotated interfaces, an implementation of the interfaces and factory classes are generated automatically. It is still possible to create classes manually or modify generated ones. Generated and manual written code is distinguished by special annotations to avoid overwriting of manual classes in case of a repeated code generation.

(a) UML (b) Java

Figure 3: An Ecore model expressed in UML and Java.

3. ECORE AS METAMETAMODEL: DEFINING FUJABA METAMODELS

In this section, we discuss the use of Ecore as an explicit metametamodel in Fujaba (i.e., how to describe any Fujaba metamodel in terms of Ecore concepts). By this, we obtain an uniform metamodel representation inside of Fujaba as well as between Fujaba and other Ecore aware tools. We consider two scenarios:

- Creation of new Fujaba metamodels. New metamodels for Fujaba can be easily defined by using EMF model editors and their implementation can be generated usign EMF's code generation.
- Re-using existing Fujaba metamodels. To avoid the re-definition of metamodels, we want to extrapolate Ecore models out of available Fujaba metamodel implementations. By carefully annotating the implementation of a Fujaba metamodel (e.g., Fujaba's metamodel for UML class diagrams) with Ecore terms, we extend it to represent a valid Ecore metamodel (Figure 4). We are aware of a possible complexity of this task. Moreover, we expect to discover conceptual flaws in existing metamodels, which may needed to be corrected.

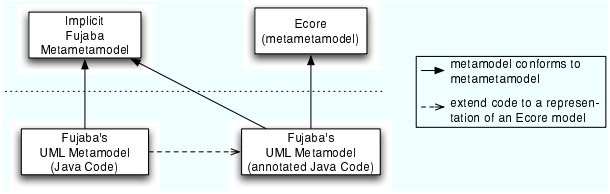


Figure 4: Fujaba's UML metamodel code extended to conform to Ecore as metametamodel.

4. ECORE AS METAMODEL: RELATING TO FUJABA METAMODELS

Once Ecore is introduced as a metametamodel (Section 3), we can approach an even tighter integration of Ecore into Fujaba. In this section, we go one step down in the metamodeling hierarchy and discuss how Ecore as a metamodel relates to Fujaba metamodels. We also discuss how this relationship can be realized in the Fujaba implementation to achieve our goal of integration.

In Fujaba, interfaces to express concepts of object-oriented languages exist (e.g., *FClass* or *FMethod*). We will call the set of these interfaces the *abstract metamodel*, since it is implemented in different Fujaba metamodels, like Fujaba's UML metamodel (e.g., *UMLClass* implements *FClass*). Ecore models the same concepts of object-oriented languages (e.g., *EClass* and *EOperation*). Table 1 enumerates similar elements found in Fujaba's abstract and UML metamodel and in Ecore.

These similarities suggest to use Ecore as a metamodel in Fujaba as well. This has the advantage that Fujaba can be used to define Ecore models and that it can treat Ecore models defined elsewhere.

Ecore	*Abstract Metamodel*	*Fujaba UML*
EClass	FClass	UMLClass
EAttribute	FAttr	UMLAttr
EReference	FAssoc	UMLAssoc
EOperation	FMethod	UMLMethod
EClassifier	FType	UMLType
EDataType	FBaseType	UMLBaseType
EString	FBaseType.STRING	UMLBaseType.STR
EInteger	FBaseType.INTEGER	UMLBaseType.INT
EBoolean	FBaseType.BOOLEAN	UMLBaseType.BOOL

Table 1: Relations between elements of Ecore and Fujaba metamodels.

To achieve this, a Fujaba metamodel should refer to Ecore comparable to how it refers to the abstract metamodel. However, the difference is that Ecore should be referred by delegation, while the abstract metamodel is referred by inheritance.[2] In the case of UML class diagrams, for instance, every element in the model references a corresponding Ecore element, which content accessable from Fujaba. The interface of the Fujaba UML metamodel stays unchanged. A Fujaba UML model contains Fujaba specific information and functionality not contained in the delegated Ecore model — it *refines* the Ecore model. Such Fujaba specific features are, for example, navigation and access capabilities required by different parts of Fujaba (e.g., graph transformation based on Story Driven Modeling). The Ecore model, on the other hand, can be shared with other applications.

In fact, one can also replace the inheritance relationship between a Fujaba metamodel and the abstract metamodel by a delegation relationship (Figure 5). From that viewpoint, the abstract metamodel becomes redundant, because it represents similar concepts as Ecore and does not add any model information. We can remove the abstract metamodel and let the UML metamodel refine Ecore directly (Figure 6).

The downside is, that all Fujaba code and plugins based on the abstract metamodel, if any, will have to be rewritten to use Ecore interfaces instead. For the beginning one could also keep the Fujaba interfaces as deprecated and base new plugins and metamodels on Ecore with the option of a full integration in the future.

Metamodels not based on the abstract metamodel, like the metamodel for UML state diagrams, can also refine Ecore, although they do not have directly related concepts. A state can, for example, be modeled by a class using the state design pattern. In this case, a model element of type *State* should have a reference to an element of the type *EClass* that represents the corresponding state. Identical approaches could be developed for a story diagram metamodel as needed for Story Driven Modeling.

[2]We chose to use delegation for a cleaner separation of Ecore and Fujaba modeling concepts. In addition, an Ecore model is represented by concrete classes, while the Fujaba's abstract metamodel consists of interfaces only.

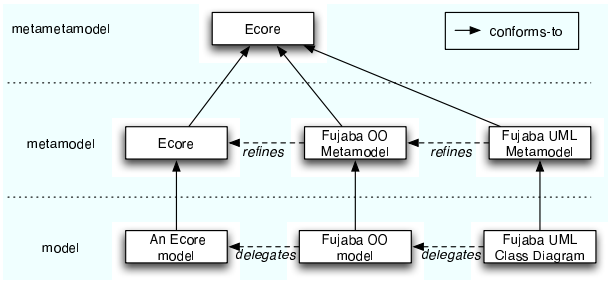


Figure 5: Ecore as a metamodel refined by Fujaba's abstract metamodel.

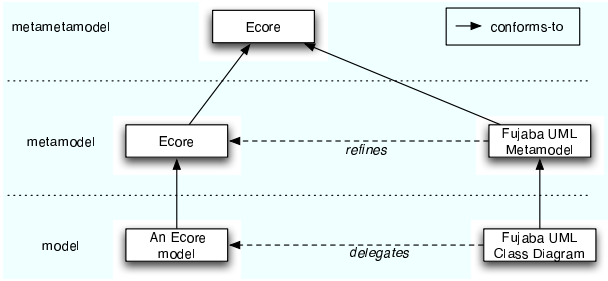


Figure 6: Ecore as a metamodel directly refined by Fujaba's UML metamodel.

5. EVALUATION

While prototypical testing our ideas, we have encountered certain difficulties concerning structural and semantic incompatibility of EMF generated code and Fujaba's core implementation. A big problem poses the fact that the classes implementing metamodels in Fujaba also include functional code concerning, for example, the visual representation of diagrams. To maintain this functionality, manual alterations to generated EMF metamodel code are required.

We also encountered difficulties with the fact that Fujaba misses a clean separation between an interface and an implementation layer, while this is provided in EMF generated code. That makes it hard to identify metamodels and the metametamodel in Fujaba code. As a consequence, annotating Fujaba's metamodel implementations (as proposed in Section 3) is difficult.

Another problem concerns the pattern recognition. The structure of the EMF generated metamodel code does not comply with the structure expected by the engine. Especially, the class naming scheme of EMF's code-generation is by default different from Fujaba's. One needs to overcome these and other minor issues before a full integration of Fujaba and EMF can be realized.

6. RELATED WORK

Amelunxen [1] reports about a tool for editing MOF diagrams, implemented as a Fujaba plug-in. The tool allows to specify MOF metamodels and generate Java metamodels, which are complient to the SUN's Java Metadata Interface (JMI) standard. Despite its relation to MOF, this work does

not concern metamodeling in Fujaba's core and does not aim at modeling interoperability.

Amelunxen et al. [2] discuss, how Fujaba can be adapted to serve as a core of a metamodeling framework. Fujaba is regarded as a good foundation for this purpose, due to its modeling features (diagrams and OCL constraints) and functionality, like code generation and a powerful graph rewriting system. Eclipse is proposed as an integration framework by a means of its plugin mechanism and a number of other Eclipse tools like its user interface and editor frameworks. However, they do not consider to integrate EMF to support standardized modeling.

7. CONCLUSIONS

Our vision is that a tight integration of Fujaba with the Eclipse Modeling Framework will bring profits for both communities. To achieve a necessary degree of integration, the metamodeling approaches presented here are applicable. Converging modeling in Fujaba towards EMF and Ecore also means improving the modeling towards standards, since EMF and related Eclipse projects employ OMG standards like MOF and XMI. That increases the quality of Fujaba and the ability to combine it with other tools using the common standards. Furthermore, the Model-Driven Engineering principles in Fujaba can be improved by support of such standards as MOF, XMI, OCL, CWM and SPEM.

8. REFERENCES

[1] C. Amelunxen. A MOF 2.0 Editor as Plugin for FUJABA. In H. Giese, A. Schürr, and A. Zündorf, editors, *Proc. 2nd International Fujaba Days*, volume tr-ri-04-253, pages 43–48. Universität Paderborn, 2004.

[2] C. Amelunxen, A. Königs, T. Rötschke, and A. Schürr. Adapting FUJABA for Building a Meta Modelling Framework. In H. Giese and A. Zündorf, editors, *Proc. 1st International Fujaba Days*, volume tr-ri-04-247, pages 29–34, 10 2003.

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

[4] J.-M. Favre. Foundations of meta-pyramids: Languages vs. metamodels – episode ii: Story of thotus the baboon. In J. Bezivin and R. Heckel, editors, *Language Engineering for Model-Driven Software Development*, number 04101 in Dagstuhl Seminar Proceedings. Internationales Begegnungs- und Forschungszentrum (IBFI), Schloss Dagstuhl, Germany, 2005.

[5] T. E. Foundation. Eclipse platform technical overview, April 2006.

[6] T. D. S. E. Group. E-CoMogen webpage. `http://web.inf.tu-dresden.de/~jh30/work/rewerse/comogen`, July 2006.

[7] IBM. Rational Rose homepage. `http://www-306.ibm.com/software/awdtools/developer/rose/index.html`, July 2006.

[8] Object Management Group. MOF homepage. `http://www.omg.org/`, July 2006.

[9] Object Management Group. XML metadata interchange (XMI) specification. `http://www.omg.org/technology/documents/formal/xmi.htm`, July 2006.

[10] Omondo — the live UML company. EclipseUML homepage. `http://www.omondo.de/`, July 2006.

[11] The Eclipse Foundation. Eclipse modeling framework — EMF. `http://www.eclipse.org/emf/`, July 2006.

[12] The Eclipse Foundation. Eclipse modeling framework technologies — EMFT. `http://www.eclipse.org/emft/`, July 2006.

[13] The Eclipse Foundation. Eclipse project. `http://www.eclipse.org/`, July 2006.

[14] The Eclipse Foundation. EMF based ontology definition metamodel implementation. `http://www.eclipse.org/emft/projects/eodm/`, July 2006.

[15] The Eclipse Foundation. EMF based UML2 metamodel implementation. `http://www.eclipse.org/uml2/`, July 2006.

[16] The Eclipse Foundation. Graphical editing framework — GEF. `http://www.eclipse.org/gef/`, July 2006.

[17] The Eclipse Foundation. OCL integration for EMF. `http://www.eclipse.org/emft/projects/ocl/`, July 2006.

[18] University of Paderborn — Software Engineering Group. Fujaba tool suite webpage. `http://www.fujaba.de/`, July 2006.

Building Graphical Editors with GEF and Fujaba

Thomas Buchmann
University of Bayreuth
Chair of Applied Computer Science I
Universitaetsstrasse 30
95440 Bayreuth, Germany
thomas.buchmann@uni-bayreuth.de

Alexander Dotor
University of Bayreuth
Chair of Applied Computer Science I
Universitaetsstrasse 30
95440 Bayreuth, Germany
alexander.dotor@uni-bayreuth.de

ABSTRACT

The Fujaba tool suite supports the development of models and the generation of code, but it does not cover the development of tools with graphical user interfaces. In this paper, we report on ongoing work in constructing Fujaba-based graphical editors with GEF, an Eclipse-based framework for constructing graphical editors.

1. INTRODUCTION

Fujaba [11] is a powerful environment for developing executable models with the help of class, story, and state diagrams. However, support is confined to the *application logic* and does not cover the user interface.

GEF [6] is an Eclipse-based framework for constructing graphical editors, which may be run either as Eclipse plugins or as stand-alone tools. GEF assumes some model that has to be visualized and edited, but does not prescribe a specific modeling framework.

Combining GEF and Fujaba appears attractive for several reasons: (1) Reusing GEF reduces the effort of constructing a graphical editor considerably. (2) In the context of Eclipse, GEF is a main-stream framework which will be extended and improved. (3) The Ecplise framework for tool integration may be exploited. (4) GEF/Fujaba based editors run as Eclipse plugins nicely complement the Eclipse plugin version of Fujaba.

2. PROJECT0R

Our long-term goal is the construction of modular, customizable tools for managing development processes. In the context of this research, we have been developing *project0r*, a management tool based on dynamic task nets. *Dynamic task nets* (DTNs) are enactable process model instances, each instance representing a set of development activities and their relationships. As we use DTNs only as a showcase in this paper, we recommend [7] for a detailed description.

The screenshot in Figure 2 shows a DTN example drawn in *project0r*, as our GEF-Plugin for Eclipse is called. The DTN consists of tasks depicted as boxes. Each task contains two lines of text (name and current state). Solid and dashed arrows represent control and feedback flows, respectively. The DTN is not hierarchical to keep it simple, therefore it doesn't contain subtask edges.

The DTN is displayed in the main *editor pane*. On the right the *palette* is shown, which supplies the user with various functions: The first two functions allow to select either one or several elements in the pane. The next three elements

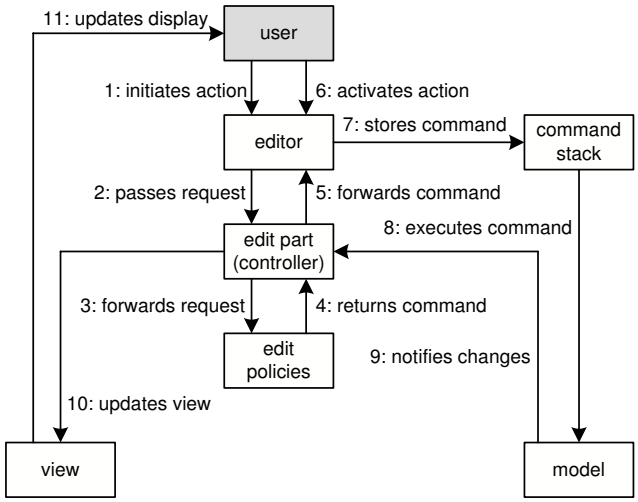


Figure 1: Command cycle in GEF

allow to draw three different types of edges: control flows, feedback flows, and subtask edges. The last element for creating a task is part of the `Tasks` palette folder - depicted by the folder symbol above. Note that the palette and the editor belong together and form the main extension point for the project0r-Plugin into Eclipse.

Right of the main pane a second extension point of the project0r-plugin is available in form of a *view*. The upper part of the view shows the outline, while the lower part shows an overview of the diagram in the editor pane. Both outline and editor pane operate on the same model: if one changes the name of a task in the outline the editor pane is updated and vice versa.

Finally, at the bottom the *property view* is displayed. It shows properties of tasks such as name and state. Properties may be edited via the property view.

3. THE ARCHITECTURE OF GEF

The architecture of GEF employs the *model-view-controller* design pattern. In GEF, controllers are called *edit parts*. For each pair of model element and view element, a corresponding edit part is established. In addition, there is a *root edit part* for coupling the containers of the model and the view.

Fig. 1 illustrates the main components of the GEF architecture as well as the basic *command cycle*. Note that we discuss only the creation of nodes. GEF handles different

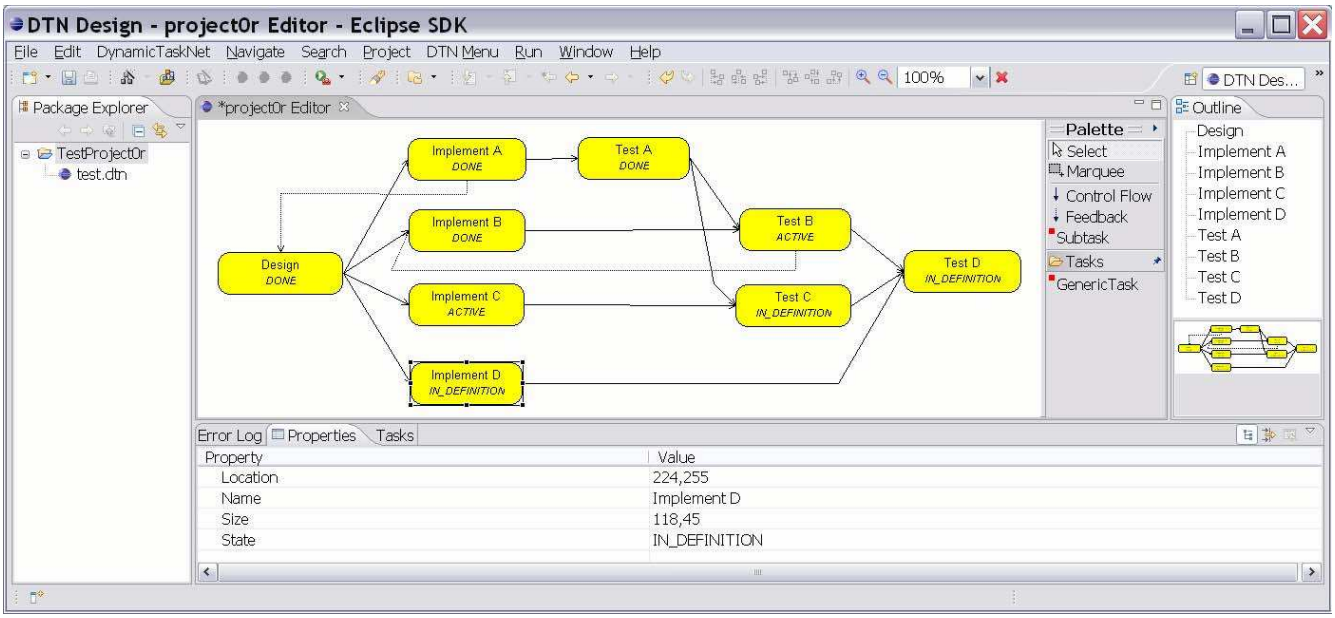


Figure 2: Screenshot of the project0r plugin

classes of commands in different ways, but space restrictions do not allow to discuss them all.

When the user initiates an action by selecting an item in the palette (1), the *editor* passes a corresponding request to the *root edit part* (2), which consults its *edit policies* (3) in order to determine a matching *command*. A command object is returned to the edit part (4), which forwards the command to the editor (5). The command is executed only after the user has activated it (6) by clicking on the canvas. To this end, the editor appends the command to the *command stack* (7), which supports the execution (8) as well as undo and redo of user commands. Command execution modifies the model, which notifies changes to the edit part (9). The edit part updates the view (10), which finally updates the display at the user interface (11).

Please note that the model is external to the GEF framework. GEF can handle any model, provided it supports the Java Beans standard. Thus, the model has to provide `get` and `set` methods for properties and implement a listener interface for change notifications. The listener interface is required by the edit part to update the view. Moreover, `get` and `set` methods are used e.g. in the properties view.

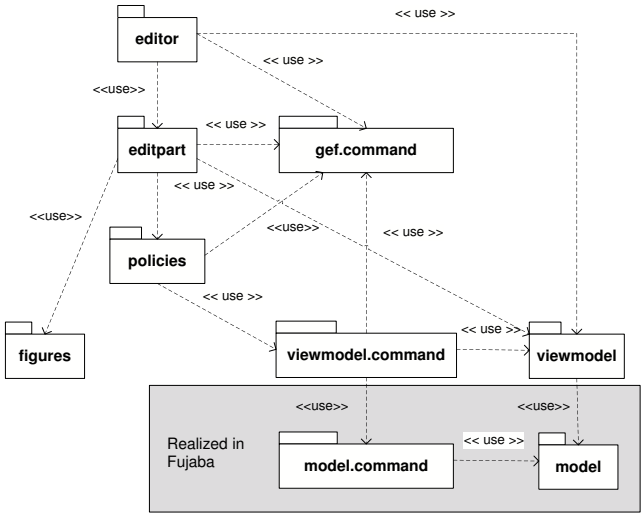


Figure 3: Package diagram of the project0r editor

4. BUILDING THE EDITOR

Figure 3 summarizes the most essential packages and usage dependencies of project0r. The packages shown in the grey box at the bottom were generated from a Fujaba model. `gef.command` is provided by GEF. All other packages were implemented in Java. Basically, we followed the instructions given in [8], which shows how to build a simple GEF editor step by step. In the following, we focus on the integration between the GEF based parts and the model.

The GEF framework does not delineate cleanly the model from its representation. In particular, in the case of graphical representations GEF assumes that representation attributes such as sizes and positions of nodes are stored in the model. Without the separation of model and representation, multiple views on the same model cannot be supported. Multiple views are quite common in the context of management systems (our focus of interest), e.g., project plans may be represented as Gantt diagrams or MPM networks.

To solve this problem, a *view model* is introduced as an intermediate data structure between the model (in Fujaba) and the view (in GEF). The view model stores the graph of nodes and edges to be displayed in the view, including representation attributes such as size and position which are specific for a view. To minimize redundancies, the view model does not store the values of model attributes such as name and state of a task.

The view model acts as a *wrapper* of the model, i.e., from the perspective of the editor components the view model behaves like an ordinary model in the GEF architecture.

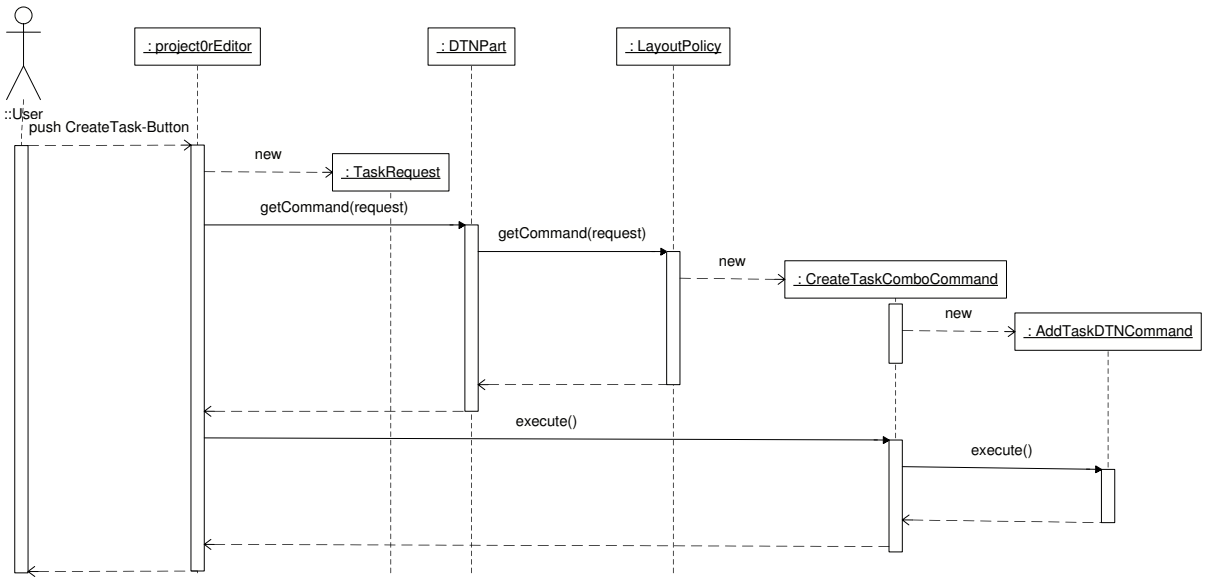


Figure 4: Sequence diagram for creating a task

Changes to the model are triggered via the view model, using the view model commands. Please note that the view model commands also include commands which affect only the layout, such as e.g. moving a node. If a view model command does afffect the model, it holds a reference to a model command. View model commands are stored in the GEF command stack. Execution, undo, and redo of a view command triggers the respective operations on the referenced model command. Please note that model commands are not GEF commands, i.e., they do not inherit from the GEF `Command` class.

The view model acts an *observer* of the model, and the edit part listens to the view model. When the model is changed via a model command, the changes to the model are caught by the view model, which updates itself and notifies the edit part, which in turn updates the view. In particular, this architecture allows to cope with model commands which perform complex changes to the model (beyond insertion/deletion of single nodes and edges). Complex changes are eventually reduced to base operations, which are observed by the view model. For example, when all active tasks in a task hierarchy are suspended transitively, the view model is informed about all individual state changes and notifies the responsible edit parts accordingly.

Both the view model and the model comply with the Java Beans standard. With respect to the view model, conformance is required by GEF. Since the model has to provide some notification mechanism, as well, we decided to make use of Java Beans for the model, as well. To make this work, we had to adjust the CodeGen2 templates and add some lines to be able to generate code for classes that have been declared using the `JavaBeans` stereotype.

Figure 4 illustrates how the creation of a task is performed in project0r. In response to the request generated on behalf of the user, a view model command is created which wraps a corresponding model command. The editor is only aware of the view model command and triggers its execution, which in turn executes the wrapped model command. Changes in the model are propagated to the end user via the view model and the edit part (not shown because of space restrictions).

To manipulate the model via the GEF editor, model operations have to be provided as commands. To this end, we developed a `model.command` package (see Figure 3) supporting execution, undo, and redo of commands. As an example, Figure 5 shows a story diagram for undoing the creation of a connection. The embedding of the connection into the task net is deleted, but the command object still references the connection as well as its source and target task for redoing.

5. RELATED WORK

The Fujaba environment includes *Dobs* [5], a dynamic object browser for visualizing object structures at run time and for executing operations on these object structures. As part of Fujaba4Eclipse, Dobs may be run as an Eclipse plugin (*eDobs*). Both Dobs and eDobs serve as debugging tools and have not been designed as graphical editors. The graphical representation displays 1:1 the object structures maintained by the generated code; there are no mechanisms for abstraction (e.g., by representing an object and adjacent links as an edge) or for customization of the representation.

UPGRADE [1] is a universal platform for graph-based development which has originally been used with models generated from the PROGRES environment [10]. Recently, it has been integrated with Fujaba, as well. With UPGRADE, graphical editors may be generated for arbitrary graph models. These editors operate as stand-alone tools. In contrast, our work exploits technology developed in the context of Eclipse. Since our graphical editors may be run as Eclipse plugins, we may benefit from the functionality provided by the Eclipse infrastructure and integrate smoothly with other tools based on Eclipse.

The *TIGER* project [4] is dedicated to the automatic generation of GEF editors using graph transformations. TIGER is based on AGG, an approach to graph transformations based on category theory. In contrast to our work, TIGER already covers the generation of graphical editors (while

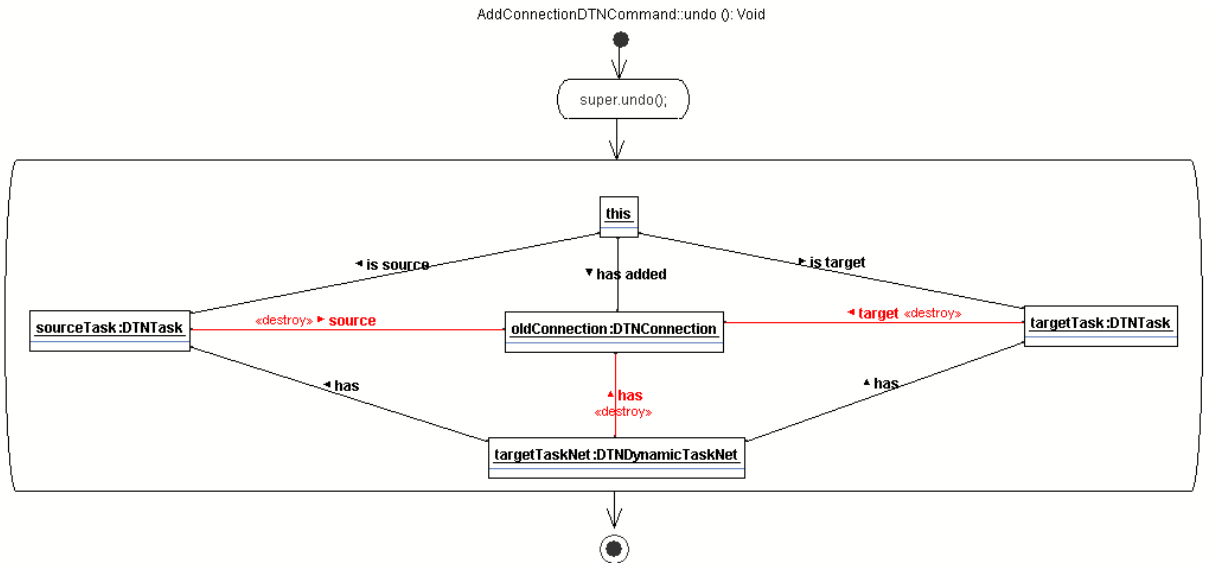


Figure 5: Storydiagram for undoing a command

we implemented the code for the editor by hand). However, TIGER addresses only a single view per model and stores graphical information as attributes in the model. In project0r, we introduced an intermediate data structure — the view model — to support multiple views.

GMF (Graphical Modeling Framework) [3] aims at integrating GEF with *EMF* (Eclipse Modeling Framework [2]). GMF assumes that the model is defined with the help of EMF. Given an EMF model, a GEF editor may be generated with the help of GMF. Like in project0r, an intermediate data structure between model and view is introduced in order to separate cleanly the model from the view. However, the view model does not wrap the model; rather, view model and model are kept consistent by synchronized updates. This architecture may break if the effect of a command on the model is not known in advance. Then, it is necessary to register the view model as an observer of the model and have it updated by notifications.

6. CONCLUSION

In this paper, we have sketched how a graphical editor based on GEF may be built on top of a Fujaba model. project0r served as an initial prototype for exploring GEF technology. Although we created the graphical editor with limited effort, usage of the GEF framework was more difficult than expected. In particular, we decided to modify the architecture assumed by GEF by introducing an abstraction layer between the model and the view. Furthermore, we had to implement a command layer on top of the model to support execution, undo, and redo of model operations. Altogether, implementation involved a great deal of manual coding. Future work will address these deficiencies, e.g., by developing or reusing a code generator and by employing a database management system supporting undo/redo (DRAGOS [9]).

7. REFERENCES

[1] B. Böhlen, D. Jäger, A. Schleicher, and B. Westfechtel. Upgrade: A framework for building graph-based interactive tools. *Electronic Notes in Theoretical Computer Science*, 72(2):113–123, 2002.

[2] Eclipse Foundation. *EMF - Eclipse Modeling Framework*, 2006. `http://www.eclipse.org/emf`.

[3] Eclipse Foundation. *GMF - Graphical Modeling Framework*, 2006. `http://www.eclipse.org/gmf`.

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

[5] L. Geiger and A. Zündorf. Graph based debugging with fujaba. *Electronic Notes in Theoretical Computer Science*, 72(2):124–131, 2002.

[6] IBM Corporation. *GEF Programmer's Guide*, 2005. `http://help.eclipse.org/`.

[7] C.-A. Krapp. *An Adaptable Environment for the Management of Development Processes*, volume 22 of *Aachener Beiträge zur Informatik*. Verlag der Augustinus Buchhandlung, Aachen, Germany, 1998.

[8] M. Scarpino, S. Holder, S. Ng, and L. Mihalkovic. *SWT/JFace in action*. Manning, Greenwich, CT (USA), 2005.

[9] E. Schultchen, U. Ranger, and B. Böhlen. Graph-oriented storage for fujaba applications. In *Fujaba Days 2006 Proceedings*. University of Paderborn, Germany, 2006.

[10] A. Schürr, A. Winter, and A. Zündorf. The PROGRES approach: Language and environment. In H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors, *Handbook on Graph Grammars and Computing by Graph Transformation: Applications, Languages, and Tools*, volume 2, pages 487–550. World Scientific: Singapore, 1999.

[11] A. Zündorf. *Rigorous Object Oriented Software Development*. PhD thesis, University of Paderborn, Germany, 2001.

A PlugIn for the Development of Resource Aware Components with Mechatronic UML*

Holger Giese, Stefan Henkler, and Martin Hirsch†
Software Engineering Group
University of Paderborn
Warburger Str. 100
D-33098 Paderborn, Germany
[hg|shenkler|mahirsch|@uni-paderborn.de

ABSTRACT

One promising thread of research for self-optimizing embedded software is the assignment of resources to multiple system functions at run-time in order to optimize the overall system performance. Embedded software components that are capable of adapting their functionality and resource requirements to different context requirements are a crucial prerequisite for such self-optimizing embedded systems. In this paper we present a Fujaba plugin for resource aware components for the Mechatronic UML approach, which facilitates the development of such components by means of extended modeling capabilities and the automated derivation of resource profiles. In addition, the paper evaluates the run-time overhead of the solution with respect to memory usage.

1. INTRODUCTION

A promising research direction for future generations of embedded systems is self-optimization. In self-optimizing systems software may optimizes the distribution of the available resources between different system functions at run-time. This self-optimization evaluates the current system context and system status and then optimizes the overall system performance by assigning resources depending on the relevance of each of these functions.

Resource aware embedded software components which can adapt their resource requirements by adjusting the quality of service or even reducing the provided functionality are a crucial prerequisite for such self-optimizing embedded software systems. They provide the basic building blocks to realize complex self-optimizing software and offer different so called *resource profiles* to their context.

To realize a self-optimizing resource management assuming such resource aware components, run-time support in form of a *profile manager* (in form of an operating system or run-time kernel element) is required, which can take advantage of the resource profiles and is capable of manage the flexible assignment of resources in a predictable manner (cf. [1]).

In practice, the additional efforts to develop resource aware instead of traditional components manually are too large. Even getting the standard behavior right is very often a timely and costly undertaking. Therefore, an extension of the model-driven Mechatronic UML approach [5] has been suggested in [2], which requires only a few additional model extensions to be able to automatically derive the resource profiles as well as component code which realize these profiles as well as the switching between them.

In this paper we present a plugin developed in [9] which realizes the envisioned tool support for resource aware Mechatronic UML components. It realizes the tool support for resource aware components as outlined in [2] for the profile management framework of [1].

The presented plugin supports the modeling of resource information and permits to equip the state machines with additional information about transitions which characterize them as required or optional. Given these additional information, a fully automated synthesis procedure can be employed to derive a set of optimal resource profiles. Each of these resource profiles relates to a different resource usage mode of the component with specific upper bounds for the resource usage.

The remainder of the paper is organized as follows: After presenting the overall approach for resource aware components and their management by the operating system in Section 2, the plugin and its architecture are presented in Section 3. The paper then closes with an evaluation of the plugin and the resulting overhead in particular in Section 4 and some final findings in the conclusion (Section 5).

2. RESOURCE-AWARE COMPONENTS

In this section we give a brief overview of the relevant concepts of Mechatronic UML. Detailed information are given in [6]. Further, we give a brief overview of the resource manager, which coordinates the assignment of the available resources at runtime. The considered flexible resource management framework [1] is developed in the context of the DReaMS project (Distributed Real-Time Extensible Application Management system). In Section 2.3 we extend the design approach of [6] as proposed in [2]. To make the concepts more understandable we introduce in Section 2.1 a running example which is extended in Section 2.3.

2.1 Mechatronic UML

Mechatronic UML applies components in the sense of UML 2.0 [8]. To support the design of complex mechatronic systems hybrid components are introduced [6]. The Mecha-

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

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

Discrete behavior of the components and patterns is specified by real-time statecharts or their hybrid extension hybrid reconfiguration charts, which embeds also (hybrid) components. To denote the fading, we add so called fading transitions to hybrid reconfiguration charts. The provided embedding concepts enable the specification and modular verification of reconfiguration across multiple components.

To outline our approach we employ an active suspension system and its optimization, which is originated from the RailCab[3] research project at the University of Paderborn.

The suspension module is based on air springs which are damped actively by the displacement of their bases and three vertical hydraulic cylinders which move the bases of the air springs via an intermediate frame – the suspension frame. The vital task of the system is to provide the passengers a high comfort and to guarantee safety when controlling the shuttle's car body. In order to achieve this goal, multiple feedback controllers are used with different capabilities in matters of safety and comfort [7].

We focus on three controllers which provide the shuttle different comfort: The Reference controller provides sophisticated comfort by referring to a trajectory describing the required motion of the coach body in order to compensate the current track's unevenness, slopes, etc. To guarantee stability, all sensors have to deliver correct values. In case of e.g. incorrect values the less comfortable Absolute controller has to be applied, which requires only the vertical acceleration as input. If this sensor fails, our Robust controller, which provides the lowest comfort, but requires just standard inputs to guarantee stability, has to be applied.

We now give an overview of the structure and behavior of the example, modeled with Mechatronic UML (see also Figure 1).

The structure consists of a Monitor component, which embeds a Sensor, Storage and a Body Control (BC) component. The BC component embeds the Reference, Absolute, and Robust Controller (Continuous Components) and is responsible for the chassis control. The Sensor delivers the measured values to the Storage. The Storage stores the measured values from the Sensor and additionally a reference curve received from other shuttles which drive over the same track before and forward their experience in form of the reference curve. The reconfiguration between the three aforementioned controllers is coordinated by the Monitor.

The Monitor hybrid reconfiguration chart is shown in Figure 1. As aforementioned, the BodyControl component, Sensor component and Storage component are embedded in form of instances. The BodyControl component is embedded as BC[Reference] instance in the AllAvailable state, BC[Absolute] instance in the AbsAvailable state, BC[Robust] instance in the NoneAvailable and RefAvailable state. The fading transitions are visualized as bold transitions.

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

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

[3]http://www-nbp.upb.de/en/

2.2 Profile Framework

The purpose of the flexible resource management framework is to optimize resource utilisation in a system. The basic idea behind this framework is related to quality of service (QoS) approaches, nowadays widely used in the field of network and multimedia computing. When using this framework, applications have to define multiple service alternatives, each one of them demanding different amounts of resources and providing a specific quality level.

Profiles

In the context of the Flexible Resource Manager (FRM), the aforementioned application configurations are called profiles. It is assumed that each task defines a nonempty set of profiles. At any time, only one profile can be active per task. Each profile has to provide the following information:

- **Resource requirements:** Each profile has to define minimum and maximum requirements for each resource type (e.g. memory, network bandwidth, etc.).
- **Maximal assignment delay:** Maximal delay for the asignment of the requested resource.
- **Switching conditions:** Every profile must define under which circumstances a switch to which other profiles may be forced and how long each switch will take.
- **Profile quality:** A profile's quality is defined as $q_p \in [0, 1]$ and is used by the FRM to determine the best possible overall system performance.

Approach

Based on information provided by the applications, the FRM generates a set of profile configurations. Each one represents a different combination of active profiles. The configuration containing all active profiles is called active configuration. For the optimisation of the system's quality, a quality value is calculated for each configuration. A value is assigned to each task representing the relative importance of a task compared to the other ones.

Configurations are classified into two distinct classes. A system is running in a guaranteed allocation state if all worst case requirements of a configuration's profile can be fulfilled simultaneously. Accordingly, a system is in a so called overallocation state if the sum of worst case requirements of at least one resource exceeds the system's limits. Running a system in such a state must not result in the violation of any of the tasks timing constraints, if it can be ensured that a guaranteed allocation state can be reached in time.

Depending on the current resource situation, the FRM may have to force one or more tasks to switch to another profile in order to optimize resource utilization and quality. Accordingly, information about which profile is reachable from another one, and how long a switch will take, has to be specified by each task. The profile reachability information of the individual tasks can be combined to a so-called configuration reachability graph.

The degree of resource optimization depends for the most part on the maximum assignment delay that can be granted

to the resource manager. To keep this value as high as possible, the FRM applies a demand/acquire paradigm for resource allocation. Before allocating any resources, applications have to send a demand to the FRM as soon as the need for additional resources becomes apparent. They may then perform further operations until they acquire the demanded resources, which become available after the specified maximum delay at the latest.

2.3 Designing Resource-aware Components

To design resource-aware components, we extend the presented design approach of Mechatronic UML in Section 2.1 with additional semantic information to derive profiles automatically. We extend the approach by adding memory consumption (cf. [2]), quality ranking for rating, and transition flags.

Memory Consumption

In order to optimize the resource demand of an application only those resources effectively needed should be allocated. However, it is inappropriate to analyze an application's resource behavior at finest granularity, as the goal is to identify service alternatives with relatively large differences in their resource requirements. Reasonable entities for determining resource demands are the single states of a hybrid reconfiguration chart. Because of the different controllers that might be integrated in each state, requirements may vary significantly from one state to another; furthermore they may stay constant for an adequate, not too short period of time. For that reason, resource requirements are determined on a per-state basis in this approach, using worst case assumptions for individual sub-components.

Resource requirements for a particular state of a hybrid reconfiguration chart solely depend on the component instances embedded in it. Thus, worst case requirements of the respective components have to be determined and added up. Computer-aided software engineering (CASE) tools, which provide seamless support for component modeling and code generation, as our approach, should in general be capable of making this determination. Alternatively, external tools may be used to determine the requirements and the results have to be annotated manually. This way, the resources needed for component instances employed in a certain state can be determined. The actual requirements eventually propagated to the operating system then result from the component instances' requirements plus the requirements needed for the statechart representation itself.

As fading transitions are just only a compact view of fading states, the memory requirements for the fading needs to be summed up to the source and target states. In the case of an atomic transition we need only to sum up the target state plus the state vector.

The memory requirements for the considered example are shown in Figure 1.

Quality

The profile approach additionally requires a quality value to be linked to each resource demand (cf. Section 2.2). Thus, quality values are associated with each state. The benefit of the application residing in a state depends on the configuration that is applied. Assigning a reasonable quality value for a configuration ought to be the task of the control engineers. However, it may be presupposed that the quality of the output of different configurations for the same task can be rated in some way. The qualities are visualized in the upper right corner of a state. As AllAvailable has the highest comfort, this state has the highest quality in contrast to NoneAvailable and RefAvailable which have the lowest quality.

Transition Flags

The profiles or service alternatives to be identified are in contrast to the full application characterized by having reduced resource requirements, while still providing the desired service, albeit at a reduced quality is necessary. In a hybrid reconfiguration chart, this context is given by the diagram's transitions, which are thus used to distinguish between needed resources to guarantee a safety behavior and additional resources for a more comfortable configuration. Therefore, new high-level constructs partitioning the set of transitions into different classes are introduced.

The first new one is the so-called required transition flag. Transitions marked as required are crucial for the application to work properly. They encompass transitions triggered by events representing environmental changes that make a reconfiguration of the system inevitable (e.g. events raised on sensor failures).

The opposite class of required transitions is identified by the blockable construct. In general, transitions of this class lead to states with a higher service quality, i.e. accomplishing the respective reconfiguration step is not vital. Examples are transitions leading to a configuration with a more sophisticated controller, once a controller that failed earlier is working again. Required and blockable transitions are used to determine valid boundaries for the profiles to be synthesized.

Furthermore, the so-called forceable transitions are introduced. Transitions endued with this construct may be triggered at any point in time regardless of current environmental conditions. Good examples for forceable transitions are again transitions triggered on sensor failures. Even if a sensor is working properly, a failure might be assumed at any time in order to force an application into an alternative configuration. This means forceable transitions can be used to force a state change during runtime. While required and blockable transitions mutually exclude each other, combinations of required/forceable as well as blockable/forceable may be specified.

In Figure 1 the transitions are marked with `(r)` to denote *required* transitions, `(b)` to denote *blockable* and `(e)` to denote *enforceable*.

3. TOOL INTEGRATION

The Fujaba Tool Suite currently offers a wide range of UML diagrams. For the design of mechatronic systems, the Fujaba Tool Suite has been extended to the Fujaba Real-Time Tool Suite [3] which provides support for component diagrams, real-time statecharts, hybrid reconfiguration charts, interface state charts, and real-time coordination patterns. Besides the modeling concepts support for code-generation and verification of hybrid reconfiguration charts and real-time statecharts is provided.

For the realization of the necessary extension for modelling and code-generation, a new Fujaba plugin called *UML-RTProfile* has been realized. It depends on several other plugins from the Fujaba Real-Time Tool Suite. It requires the *RealtimeStatechart* plugin providing modelling and code-

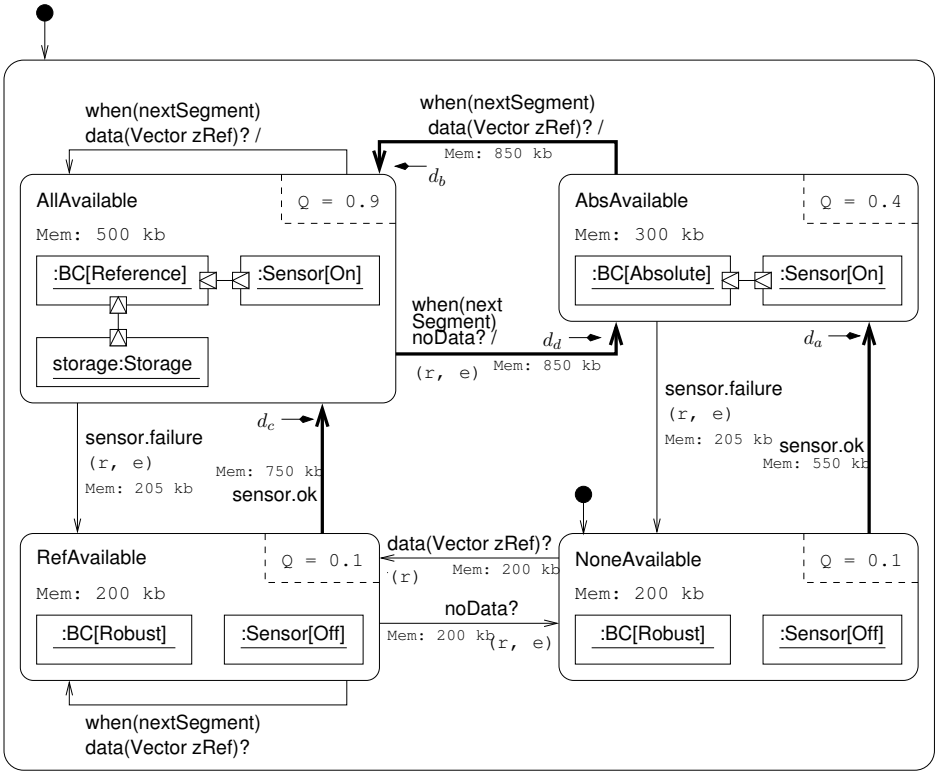


Figure 1: Monitor behavior with annotated resource requirements, quality and transition types

generation for real-time statecharts, the *UMLRT2* plugin for component modelling and its extension *HybridComponent* for modelling and code-generation of hybrid reconfiguration charts (cf. Fig 2).

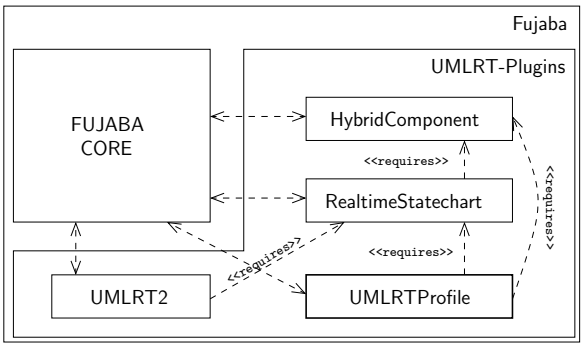


Figure 2: Tool integration

4. EVALUATION

The overall aim of our approach is to identify runtime alternatives with reduced resource requirements, allowing the system's resource manager to optimize resource utilization when executing multiple concurrent applications. The realization of profiles and dynamic resource allocation, however, increases the complexity of the application and thus influences the amount of resources needed during runtime. Consequently, the question has to be posed whether the approach can lead to a benefit at all. A general evaluation of the presented solution can hardly be made, as too many factors depend on the specific application. First of all, no forecast of the number of derived profiles, their resource requirements, or their connectivity can be made. Secondly, the number of additional states and transitions needed to realize the profile approach depends on the model, too. Last but not least, the eventual benefit depends for the most part on the amount of resources required by the individual subcomponents.

For this reason, an evaluation of the introduced suspension module example has been performed. Even if the approach is applicable for almost any resource type, the main interest will most likely be the reduction of an applications memory requirements. Therefore, the suspension controller was evaluated with regard to this resource. Table 1 shows

	# States	# Transitions	Memory (Data)[Bytes]
Standard	8	20	112
Profiles	12	38	200+8

Table 1: Memory required by the statechart during runtime

the increased complexity of the model going along with the realization of the profile approach. With the increasing number of states and transitions, the amount of memory required during runtime also increases. In [4], multiple alternatives of realizing a real-time statechart have been discussed. The best solution with regards to memory requirements and runtime was the realization of states and transitions via case

differentiation. Consequently, this variant is used in the current code-generator engine. States and transitions are identified by a unique ID (a static integer value). On a common 32-bit system, each integer requires 32 bits and thus, the memory requirements for data during runtime increase from 112 bytes to 200 bytes. Furthermore, one integer variable storing the current profiles ID and one integer variable for an additional clock representing the MIFT[4] are required.

Besides the memory needed for data, the code of the application itself has to be loaded into memory, as well. Using case differentiation leads to an significant increase of the lines of code of the generated application and thus to a larger binary (cf. Table 2). However, it must be admitted that the generated code is currently not optimized with regard to its size. For this test, the plain statechart code was

	LOC	Code[Byte]
Standard	2702	96.974
Profiles	4489	109.718

Table 2: LOC and binary size of the statechart

generated for the model with and without profile extensions and compiled under GNU/Linux using GCC 3.3.6. The size of the binary holding the statechart realization with profiles increased by about 12,5kB compared to the standard realization. Then, resource requirements of the three feedback controllers have been measured. The results can be seen in Table 3. The profile approach enables the application to

	Robust	Absolute	Reference
Memory[kB]	ca. 6	ca. 7.4	ca. 7.5

Table 3: Memory required by the feedback controllers

allocate only resources for the controller currently applied. In this case, a saving of about 13,5kB to 15kB is possible, but the binary size also increased by about 12,5kB. However, in practice, resource optimization will most likely be applied in higher dimensions, i.e. in case configurations will require significantly more resources. Thus, gaining benefit from this approach in applications with more complex configurations and higher resource demands should be possible in the general case.

5. CONCLUSION

Within this paper we present a Fujaba plugin for resource-aware components. Therefore we have presented the needed foundations of Mechatronic UML for modeling hybrid components and behavior, the flexible resource framework for optimizing resource utilization and the extensions to Mechatronic UML to derive profiles automatically. Further we present an evaluation of the approach respectively plugin.

Acknowledgements. We thank Matthias Schwarz for his work to the plugin and comments on earlier versions of this article.

REFERENCES

[4]Minimum inter-forcable time

[1] C. Böke and S. Oberthür. Flexible Resource Management - A framework for self-optimizing real-time systems. *IFIP Working Conference on Distributed and Parallel Embedded Systems (DIPES2004)*, August 2004.

[2] S. Burmester, M. Gehrke, H. Giese, and S. Oberthür. Making Mechatronic Agents Resource-aware in order to Enable Safe Dynamic Resource Allocation. In B. Georgio, editor, *Proc. of Fourth ACM International Conference on Embedded Software 2004 (EMSOFT 2004), Pisa, Italy*, pages 175–183. ACM Press, September 2004.

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

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

[5] S. Burmester, H. Giese, and M. Tichy. Model-Driven Development of Reconfigurable Mechatronic Systems with Mechatronic UML. In U. Assmann, A. Rensink, and M. Aksit, editors, *Model Driven Architecture: Foundations and Applications*, volume 3599 of *Lecture Notes in Computer Science*, pages 1–15. Springer Verlag, 2005.

[6] H. Giese, S. Burmester, W. Schäfer, and O. Oberschelp. Modular Design and Verification of Component-Based Mechatronic Systems with Online-Reconfiguration. In *Proc. of 12th ACM SIGSOFT Foundations of Software Engineering 2004 (FSE 2004), Newport Beach, USA*. ACM, November 2004.

[7] T. Hestermeyer, P. Schlautmann, and C. Ettingshausen. Active suspension system for railway vehicles-system design and kinematics. In *Proc. of the 2nd IFAC - Confecence on mechatronic systems*, Berkeley, California, USA, 9-11December 2002.

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

[9] M. Schwarz. Modeling and synthesis of ressource-flexible, mechatronic, real-time systems. Master's thesis, University of Paderborn, 2005.

A Prototype for Guideline Checking and Model Transformation in Matlab/Simulink

Holger Giese, Matthias Meyer, Robert Wagner
Software Engineering Group
Department of Computer Science
University of Paderborn
Warburger Strae 100
33098 Paderborn, Germany
[hg|mm|wagner]@uni-paderborn.de

ABSTRACT

Nowadays, the software for electronic control units in embedded systems is often developed using a chain of model-based development tools. For example, in the field of automotive systems often tools like Matlab/Simulink and related code generation tools such as dSPACE TargetLink are used. To ensure that the produced models can be processed as expected by the employed tool chains, i.e., fulfill all requirements of the later process phases, they have to adhere to a large number of often tool specific as well as organization specific guidelines. In this paper, we present a first prototype for a tool which supports the specification and checking of guidelines for the Matlab/Simulink environment. In addition, we discuss first ideas for the specification of guidelines using examples in Matlab/Simulink notation as well as the semi-automatic correction of guideline violations.

1. INTRODUCTION

Today, a model-based approach is often employed to develop the software for electronic control units (ECUs) in embedded systems such as automotive systems. Tools such as Matlab/Simulink and related code generation tools such as dSPACE Target Link permit the developer to work most of the time only at a higher level of abstraction with the models rather than the code.

To ensure that the model-based development works in practice, it is not sufficient to only use the existing tools with all their features. Instead, the employed modeling features and even the models have to be restricted to ensure that the produced models can be processed as expected across the different tools of the employed tool chain. In addition, the different models employed throughout the process should fulfill specific requirements to ensure that they are appropriate for the next process phase.

To cope with these additional constraints on the employed models in each specific process phase, the original equipment manufacturers (OEMs), suppliers, as well as tool vendors started to develop their own guidelines which capture these constraints (cf. [1]). As most of them come up with rather large catalogs of hundreds of guidelines, the developers require tool support to cope with these guideline catalogs.

First approaches for the automatic guideline checking are for example MINT [7] and the Matlab Model Advisor framework [4]. These approaches are based on the M-Script language. However, solutions based on M-Scripts or other programmed rules have severe disadvantages. The programming and maintenance of the rules is very expensive and requires detailed knowledge of the Matlab/Simulink model. Therefore, a higher abstraction description of the guidelines which is feasible also for the domain experts is more attractive.

To provide such a higher level solution, we present in this paper the required concepts and a first prototype for checking guidelines in the Matlab/Simulink environment. Based on the graph transformation capabilities of the Fujaba Tool Suite, guidelines can be specified employing graphical though formal rules which refer to a clear adapter metamodel. These specifications can be executed to detect guideline violations automatically. To ease the specification of the guidelines and make it applicable also for domain experts, the prototype further supports to derive the guideline rules from examples given within the Matlab/Simulink tool. In addition, we sketch which extensions to our prototype are required to support the automatic correction of guideline violations.

2. EXAMPLE GUIDELINE

In the automotive field, high quantities and cost pressure often require control algorithms to be implemented in fixed-point code [1]. In practice, however, a controller model is developed in floating-point arithmetic initially and transformed to enable fixed-point code generation later. Guidelines exist to assist in the transformation. Figure 1 illustrates such a guideline which will be used as a running example throughout the rest of the paper.

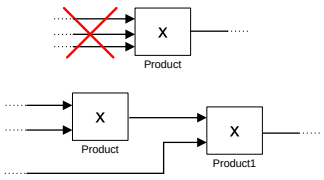


Figure 1: Guideline Violation and Correction

The presented guideline forbids the use of product blocks with more than two operands (cf. upper part of the figure). The multiplication of more than two operands produces

intermediate results. When generating fixed-point code for a model, scaling information for the intermediate results is needed, which cannot be determined automatically. Thus, product blocks with more than two operands have to be transformed into a cascade of product blocks with two operands each and proper scaling information (cf. lower part of the figure).

Violations of this guideline can be detected automatically as will be shown in the following sections. A fully automated correction is not possible, because of the missing scaling information. However, the transformation of the structure, i.e., the creation of the cascade, can be done automatically.

3. TOOL ARCHITECTURE

We developed a prototype for the automatic detection of guideline violations in Matlab/Simulink models based on the FUJABA TOOL SUITE and some extensions (cf. Figure 2). *Matlab/Simulink* is equipped with a *Java Virtual Machine*. A *Matlab/Simulink M-Script Interface* allows Java applications running inside the virtual machine to execute M-Script commands and thus to access the *Matlab/Simulink Model*. FUJABA is started from within *Matlab/Simulink* and runs in its virtual machine.

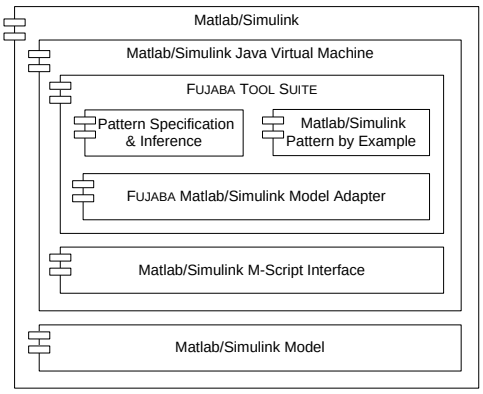


Figure 2: Tool Architecture

Guideline violations are specified formally in FUJABA by so-called pattern rules w.r.t. a Matlab/Simulink metamodel. An inference algorithm applies the rules to the model under analysis. The specification and inference is based on an approach for the recognition of design pattern implementations [6] which has been developed at our group. The *Pattern Specification & Inference* component represents the implementation of this approach by several FUJABA plug-ins which are reused by our prototype and remain unchanged.

Basically, guidelines are specified as pattern rules in abstract syntax based on the Matlab/Simulink metamodel. In addition, the *Matlab/Simulink Pattern by Example* component facilitates the derivation of pattern rules from examples given in Matlab/Simulink notation. The derived pattern rules are able to recognize structures which are equivalent to the example models in arbitrary models.

Both components use an implementation of a rudimentary Matlab/Simulink metamodel provided by the FUJABA *Matlab/Simulink Model Adapter*. The implementation conforms to FUJABA's modeling and implementation conventions and allows direct read access to the model inside the running Matlab/Simulink instance using the *Matlab/Simulink M-Script Interface*. The model adapter component allows to start the pattern inference on the Matlab/Simulink model from within FUJABA and also supports the display and highlighting of detected guideline violations in Matlab/Simulink.

4. GUIDELINE CHECKING

Guideline violations are specified by so-called pattern rules on top of a metamodel. The metamodel is presented in Figure 3 and acts as an adapter to the model in a running Matlab/Simulink instance. A *Model* is used as a root element for the pattern detection and consists of different *BlockDiagrams* which contain different *Blocks*. A *Block* is either a *Constant*, a *Product* or a *Display* block. A *Block* can be connected to another *Block* by a *Line* using the *source* and *target* associations. A selected *BlockDiagram* which is in the foreground of Matlab/Simulink can be accessed using the *current* association.

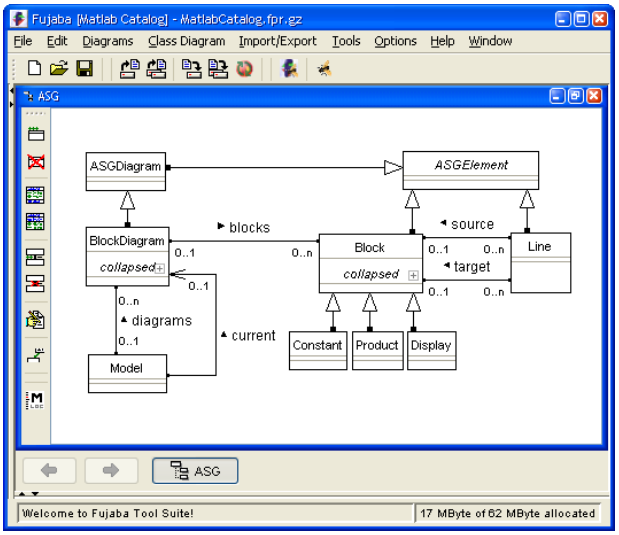


Figure 3: Metamodel

The pattern rules are based on graph grammar rules. Figure 4 gives an example of a pattern rule which detects violations of the example guideline presented in Section 2. The left-hand side of a rule requires a certain structure of metamodel elements to be present in the analyzed model. In this case, there has to be a product block that is connected to three lines which have the block as target. In addition, each of those lines is required to have a block as source.[1] If the left-hand side of a pattern rule can be found, its right-hand side creates an annotation object and links it to selected model elements to mark the pattern occurrence. In the example pattern rule, an annotation named *ProductWithMoreThan2Operands* is created and linked to the product block.

Pattern rules may require annotations created by other rules in their left-hand side thereby enabling composition and reuse of sub patterns. In addition, certain elements (including annotations) may be marked as triggers. In the example rule, the product block has been marked as a trigger and thus is displayed with a bold border. For each element in

[1]Note that the specification of the three source blocks is not really necessary here. However, they might be used to impose additional constraints.

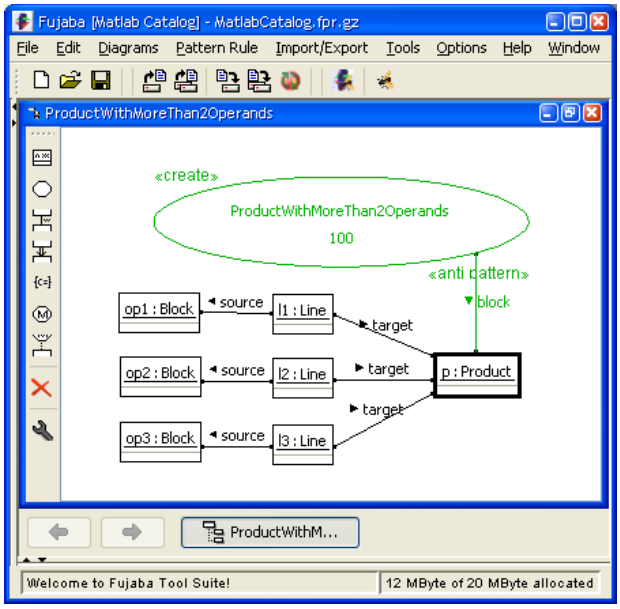


Figure 4: Guideline Violation Specification

the model that corresponds to a trigger – in this case each product block – the pattern rule is applied. By specifying annotations as triggers, the successful application of pattern rules may trigger the application of higher level rules. The actual application of pattern rules is controlled by an inference algorithm. The algorithm uses dependencies between the pattern rules and the triggers to determine which rule is applied when. This approach facilitates the analysis of even huge models which is essential since Matlab/Simulink models are expected to contain between 10.000 and 100.000 elements.

In order to exploit FUJABAs support for pattern detection and annotation, it is necessary that the metamodel is implemented according to the rules of FUJABA. The compliance with these rules allows navigating between model elements, accessing and modifying model elements, as well as creating new model elements. These kind of requirements are fundamental for FUJABA's graph rewriting algorithm - if a metamodel implementation does not fulfill these requirements, FUJABA's graph rewriting algorithm will not work with that model at all. Furthermore, the pattern detection requires the metamodel classes *Block* and *Line* to inherit from FUJABAs built-in class *ASGElement* and *BlockDiagram* to inherit from the built-in class *ASGDiagram*, respectively.

In Matlab/Simulink, the metamodel does not fulfill the requirements needed by FUJABA and it can obviously not be changed. In order to realize the checking of pattern rules anyway, we have implemented a rudimentary FUJABA compliant metamodel for Matlab/Simulink. From this metamodel specification, we generate the metamodel implementation using FUJABA's code generation facilities. The automatic code generation guarantees a FUJABA compliant metamodel that fulfills the requirements for FUJABA's graph rewriting algorithms by providing appropriate methods for the access of attributes and associations.

Up to this point, the automatically derived metamodel implementation does not provide any access to the models of Matlab/Simulink. However, this is a key requirement for the detection of guideline pattern rules in a running Matlab/Simulink instance. To establish such a linkage to the Matlab/Simulink models, we have to extend the generated metamodel to a model adapter with direct access to Matlab/Simulink models using the adapter pattern [3].

For this purpose, we remove all attributes from the generated metamodel classes and replace the generated implementation of the access methods manually using the Matlab/Simulink M-Script Interface. This built-in Application Programming Interface (API) provides direct access to Matlab/Simulink models in a running Matlab/Simulink instance. Note that we replace the method implementations without changing the method signatures in order to preserve the interface to FUJABA's graph rewriting facility. Instead of manually changing the generated implementation, another option would be to adapt the code generation. However, the implementation of the metamodel will not be generated very often and it is not clear yet how homogeneous the access method implementations will be and if an adaption of the generator is feasible.

The resulting model adapter implementation is a stateless model adapter with lazy object initialization, i.e., the adapter objects are created only on demand. Additionally, the realized adapter keeps a list of already adapted model elements and reuses them each time the model element is revisited. Thus, a model element is always represented by the same adapter object and adapter object identities are preserved. Furthermore, this guarantees a fast access to already adapted model elements of the Matlab/Simulink model and reduces at the same time the number of needed adapter objects. Of course, in the worst case, i.e., if all model elements have to be examined, this will result in one adapter object for each adapted model element anyway.

The current prototypical implementation of the model adapters handles only the metamodel shown in Figure 3. That is, of course, only a small fraction of the original metamodel. As well, the current Matlab/Simulink Model Adapter provides only read access to Matlab/Simulink models and thus, modifications to a Matlab/Simulink model are not possible yet. However, these extensions are intended as future work.

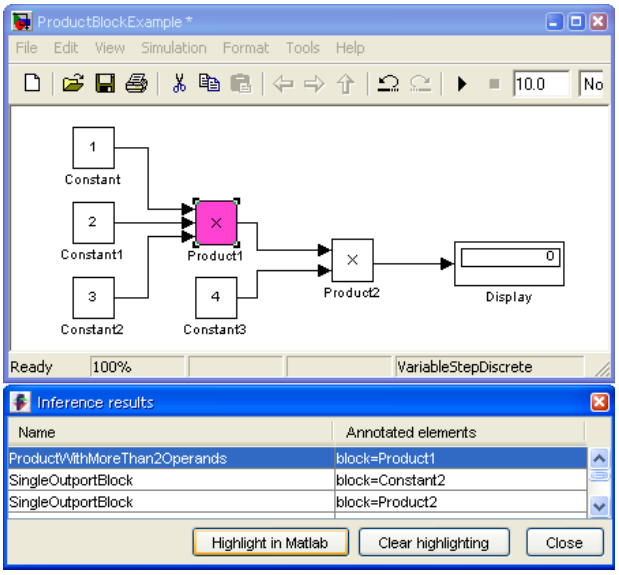


Figure 5: Result Viewer and highlighted Violation

The pattern inference obtains a *Model* instance from the Matlab/Simulink Model Adapter. Starting from the *Model* instance, model elements that trigger pattern rules are collected and the corresponding rules are applied. Each successful application of a pattern rule creates an annotation marking the pattern occurrence, i.e. a guideline violation, in the Matlab/Simulink model. The lower part of Figure 5 shows a result viewer provided by FUJABA that displays all created annotations. In addition, the user may select an annotation and choose to highlight the annotated model elements. In Figure 5 the *ProductWithMoreThan2Operands* annotation has been selected and the annotated product block with more than two operands has been highlighted and selected in the Matlab/Simulink editor.

5. SPECIFICATION BY EXAMPLE

In most cases, guidelines are not static - new guidelines have to be added or existing ones have to be adapted to special user, enterprise, project, or domain specific requirements. In the previous section, guidelines are specified as pattern rules using the abstract syntax defined by the metamodel of Matlab/Simulink. Although the presented approach enables the formalization of guidelines and the detection of their violations, a disadvantage of the approach is that the developer of a guideline rule has to be familiar with the used pattern language and the metamodel of Matlab/Simulink. However, since the complete metamodel of Matlab/Simulink is quite large and complex, this can also lead to large and complex pattern rules that are hard to read and maintain. As a consequence, it becomes infeasible for the user to change the set of guideline rules easily.

To overcome this problem, we developed an approach that enables a user to model an example for a guideline violation in the concrete syntax of Matlab/Simulink. The given guideline violation example is automatically transformed to a pattern rule in abstract syntax notation. The automatically derived pattern rule can be used for the detection of guideline violations in arbitrary models using FUJABA's inference algorithm as shown in the previous section.

Figure 6 shows an example for a guideline violation and the pattern rule as a result of the transformation. In the lower left corner of Figure 6 the user specified a guideline rule by giving an example for its violation in the concrete syntax of the Matlab/Simulink model. The presented guideline discourages from using a product block for the multiplication of two constants since this will result once again in a constant. This sort of calculations is unnecessary since the result of the multiplication can be calculated in advance. Hence, the product block and its operands can be replaced by one constant block holding the calculated result. This leads to smaller models and increases the performance of the overall calculation.

The presented example guideline violation is transformed automatically to the pattern rule shown in the background of Figure 6. The pattern consists of two objects representing the constant blocks and one object for the product block. The connections between the constant blocks and the product block are represented by line objects and appropriate links between them. Additionally, an annotation object is created and connected by links to objects representing blocks. The name of the annotation object is derived from the name of the example diagram. Note that in the guideline violation example, the product block is selected. Therefore,

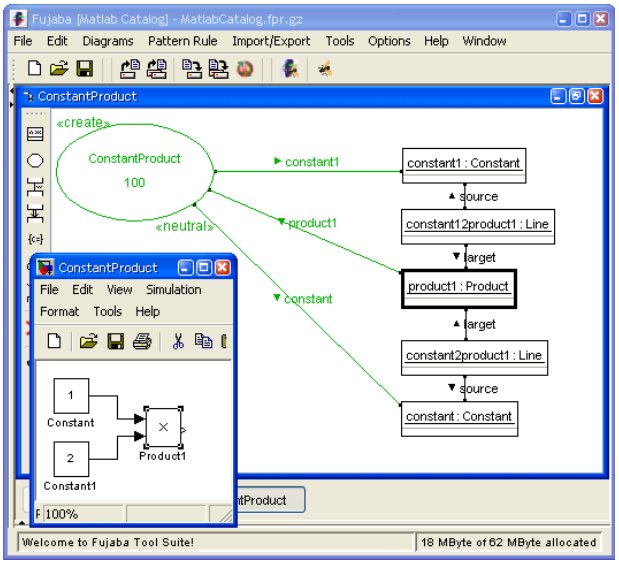


Figure 6: Guideline Specification by Example

the object that represents the product block in the pattern rule is marked as a trigger for the inference algorithm, i.e., it has a bold border in the pattern rule specification.

The transformation is realized using the technique of triple graph grammars [8]. The realized model transformation for the prototype handles the simple metamodel from Figure 3 and comprises five rules which are executed utilizing two FUJABA plug-ins [9]. For our example guideline specifications, these transformation rules generate valid pattern rules automatically. However, since our approach is still a prototype, we recommend the rule developer to inspect the extracted pattern rules and – if necessary – to adapt them manually in order to get a valid guideline violation pattern. In the future, we will study more guidelines and try to specify them by examples in order to get more experience with this approach.

6. CORRECTING VIOLATIONS

Guideline violations usually have to be corrected. How a violation has to be corrected and if this correction may be done automatically, depends, of course, on the particular guideline. In case of the example guideline presented in Section 2, a single product block with more than two operands has to be transformed into a cascade of product blocks having only two operands each and with appropriate scaling information. A fully automated correction is not possible. However, the transformation of the structure into a cascade can be automated.

We propose to formally specify corrections by graph transformations. Figure 7 shows such a transformation for the example guideline specified by a story diagram [2]. The story diagram expects a product block as argument which has been detected to violate the guideline. The given block is bound to the *product* object in the single story pattern. Next, one outgoing and three incoming lines of the *product* block are bound (objects *lOut*, *l1*, *l2*, and *l3*, respectively). If the binding is successful, a new product block (*newProduct*) is created. One of the incoming lines (*l3*) as well as the outgoing line *lOut* are reconnected to the new product block and a new line *lNew* connecting the original with the

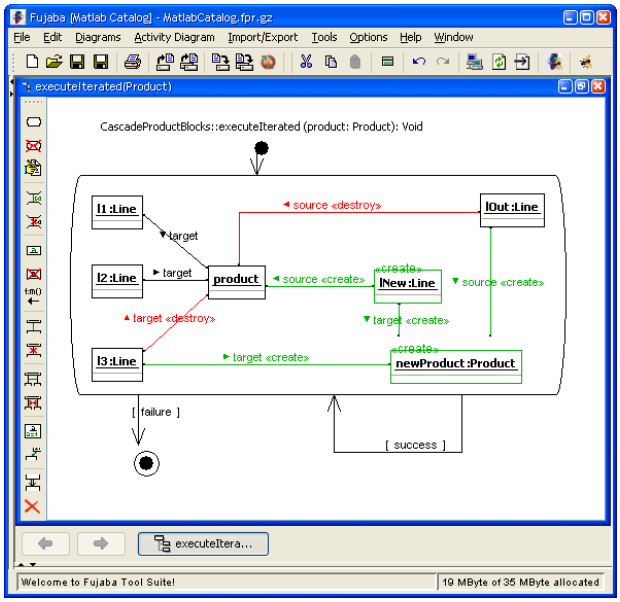


Figure 7: Partial Correction of a Violation of the Example Guideline

new product block is created.

Thus the result of the original product block and one of its former operands become the two operands of the new product block, its result is passed to the former receiver and the original product block has one fewer operand. The story pattern is repeated while the original product block has still at least three operands. Each application decreases the number of operands by one and thus the product block has only two operands left upon termination.

A semi-automatic correction of violations of the example guideline is possible. For a violation of the constant product guideline mentioned in Section 5, a correction would be to replace the two constants and the product block by one constant block with the product of the two original constants as value. This can be automated as well. However, there are many more guidelines and an analysis will be necessary in order to identify more transformations for correcting violations. It has to be analyzed if those transformations can be specified by graph transformations/story diagrams as well. In addition, the question whether the application of a transformation to a model results in a correct model again, is not addressed here but investigated in a different context (cf. [5]).

Furthermore, transformation specifications are not executable in our prototype yet because of two reasons. First, our metamodel implementation does not support write access to a Matlab/Simulink model yet. Secondly, a user interface for applying transformations to detected violations is missing.

7. CONCLUSION AND FUTURE WORK

In this paper, we have presented a prototype for detecting guideline violations in Matlab/Simulink. The novelty of the presented approach is that it is based on formal pattern rules which enable the formalization of so far informally described guidelines. In order to ease these specifications our approach allows specifying guideline violations as examples in the concrete syntax of the Matlab/Simulink modeling language. Furthermore, we have proposed to use graph rewriting for the automatic correction of guideline violations.

Up to now, the prototypically realized Matlab/Simulink adapter implements only a very small fraction of the Matlab/Simulink metamodel. In addition, the adapter does not support modifications of the model which is required for the automatic correction of guideline violations. Hence, our plan for the future is to extend the metamodel and the capabilities of the provided adapter and to realize the automatic correction based on graph rewriting.

The presented specification of guidelines using examples in the concrete syntax of the modeling language is quite promising. However, an extracted pattern rule reflects exactly the specified example. In practice, this can lead to a large number of quite similar guideline rules. To solve this problem, we have to study how a more abstract and general pattern rule can be extracted from a set of examples.

8. REFERENCES

[1] U. Eisemann. Guidelines for a model-based development process with automatic code generation. In *Preliminary Proceedings of the Workshop on Object-oriented Modeling of Embedded Real-Time Systems (OMER 3), Heinz Nixdorf MuseumsForum, Paderborn, Germany. October 13 and 14, 2005*, number tr-ri-05-261 in Technical Report, Department of Computer Science, University of Paderborn, Paderborn, Germany, October 2005.

[2] T. Fischer, J. Niere, L. Torunski, and A. Zündorf. Story diagrams: A new graph rewrite language based on the unified modeling language. In G. Engels and G. Rozenberg, editors, *Proc. of the* 6^{th} *International Workshop on Theory and Application of Graph Transformation (TAGT), Paderborn, Germany*, LNCS 1764. Springer-Verlag, 1998.

[3] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns, Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[4] The MathWorks. *MATLAB Model Advisor*, 2006. http://www.mathworks.com/products.

[5] M. Meyer. Pattern-based Reengineering of Software Systems. In *Doctoral Symposium of the 13th Working Conference on Reverse Engineering, Benevento, Italy*, October 2006. (to appear).

[6] 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, Florida, USA*, pages 338–348. ACM Press, May 2002.

[7] Ricardo, Inc. *MINT*, 2006. http:/www.ricardo.com/mint.

[8] A. Schürr. Specification of graph translators with triple graph grammars. In E. W. Mayr, G. Schmidt, and G. Tinhofer, editors, *Graph-Theoretic Concepts in Computer Science,* 20^{th} *International Workshop, WG '94*, volume 903 of *LNCS*, pages 151–163, Herrsching, Germany, June 1994.

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

Harmony - A FUJABA Plug-in for designing HL7 Messages

Jens H. Weber-Jahnke
Pervasive Primary Care Informatics Lab
Department of Computer Science
University of Victoria, Victoria, BC Canada
1-250-472-5721
jens@acm.org

ABSTRACT
Health Level 7 (HL7) is an international organization with the objective to develop standards in support of computer-based health care (eHealth). An important part of HL7's work is to design standard message types to be exchanged between interoperable medical information systems. To ensure that those messages use a standard vocabulary and format, HL7 has published a dedicated development methodology. It has been criticized that this methodology is too complex and lacks tool support. Recently, HL7 and OMG have agreed to co-operate on refining this methodology, incorporating aspects of model-driven development. Based on the FUJABA platform, we have developed Harmony, a tool to provide model-driven support for the HL7 development methodology. For this purpose, we needed to extend FUJABA with meta-modelling capabilities. This paper gives an overview of the HL7 methodology and describes the Harmony tool including any extensions made to FUJABA.

Keywords
FUJABA, model-driven design, model management, HL7, RIM, medical informatics, model transformations

1.INTRODUCTION
Many western countries have started initiatives to increase electronic information integration in the health care sector. The rationale behind such initiatives are studies indicating significant cost savings potential due to the fact that better information exchange will reduce the duplication of medical services such as lab tests and exams. Furthermore, the availability of electronic research data on medical intervention and outcomes will provide a better basis for evidence-based decision making. Experts estimate that it is the lack of such information that is responsible for many billions of dollars wasted annually on ineffective treatments. Besides the cost-savings potential, it is expected that increased medical information exchange will improve the quality of health services for patients by reducing the margin for medical errors and speeding up access to the health care system.

One of the main obstacles on the way to pervasive integration of medical information is *data heterogeneity*. Medical information systems and databases developed by different vendors, organizations, health authorities and provincial governments are using different data representations and formats. Some organizations may attempt to resolve this issue by replacing existing legacy systems with a "standard" information system solution. However, such so-called "cold-turkey migration" strategies are expensive and possible only in specific circumstances where organizations have tight control over a group of care providers of similar nature. For example, a health authority may adopt a single common information system solution for all its hospitals or an entire country may enact adoption of a single health record implementation, as is the case in the United Kingdom. However, this approach does not solve the information integration problem across jurisdictional boundaries, which is prevalent in many federative states such as Canada, USA, and Europe; Nor can these "one size fits all" solutions cover all specific data requirements in a domain as diverse and rapidly evolving as health care. Therefore, the need for health data interoperability standards will persist for some time to come.

Several standards organizations have been founded with the goal to foster the ability for medical information exchange. Perhaps the most important and internationally renowned among them is Health Level 7 (HL7). To date, HL7 has released the third major release of its interoperability standard. While earlier versions of the standard specified concrete message structures to be exchanged among compliant information systems, HL7 version 3 provides a framework and model-based methodology for "deriving" data exchange schemes such that they are standards compliant. However, the current HL7 documentation on this methodology has been criticized as too complex and ambiguous [1]. More formal and computer-supportable design steps are needed to facilitate using it in practice.

Recently, HL7 and the OMG have agreed to co-operate on refining such a methodology, incorporating key techniques of model-driven development. FUJABA is an extensible model-driven development environment that offers graph-rewriting as a formal way of defining transformations on complex object structures. In our research, we have extended FUJABA with a meta-model (OMG level M2) and the ability to perform transformations on the model level (M1). We have created the *Harmony* plug-in to FUJABA to support the HL7 development methodology. This paper gives an overview of our application (the HL7 development methodology) and describes realization and application of the *Harmony* plug-in.

2. The HL7 standard v.3
Over the past several years, HL7 has expanded its focus beyond the standardization of medical data interchange formats, covering additional areas such as functional requirements on electronic health record (EHR) systems, formal representation of computer-based practice guidelines, and other concerns related to medical informatics. In this paper, we talk about data interoperability concerns only. There are two main paradigms on how data

exchange can happen in HL7 v.3, namely the *message-oriented* paradigm and the *document-oriented* paradigm. The document-oriented paradigm is new in Version 3 of the standard. It is subject to a dedicated HL7 subordinate standard called the Clinical Document Architecture (CDA) [2]. CDA documents are self-contained units of information such as a patient health record, a lab report, or a referral letter, whereas HL7 messages are used to exchange events (and data) associated with specific business processes, such as a patient admission, discharge or billing messages. Traditionally, HL7 messages have been used for intra-organizational information exchange, e.g., to integrate information systems within hospitals. The CDA standards specification describes how HL7 messages can be used to exchange medical documents in CDA format.

CDA documents and HL7 messages are encoded in eXtensible Markup Language (XML). They derive their semantics from a common model called the HL7 *Reference Information Model* (RIM).

2.1.1 HL 7 Reference Information Model - the "mother of all"

The HL7 documentation describes the RIM as "*a critical component in the V3 development process*" and "*ultimate source from which all HL7 version 3 protocol specification standards draw their information-related content*" [3]. However, the standard also explicitly states that the RIM is not intended to serve as a data model, since it is far too abstract and generic. In other words, the RIM is clearly intended as a domain-specific *meta-model*.

The RIM has been developed using the Unified Modelling Language (UML). It extends the UML Meta Model with several HL7 specific items. At its centre, the RIM consists of a class model representing abstractions for all concepts important in health care information. In addition to the class model and auxiliary static models such as data types and terminologies, it provides models describing dynamic aspects of class instances using state-machine diagrams and interaction models.

The HL7 RIM is too large to include it in a figure in this paper. We refer interested readers to additional readings about HL7 RIM [4]. Fig. 1 shows an excerpt of the RIM that covers to so-called RIM foundation classes *Act*, *Entity*, and *Role* and two complex

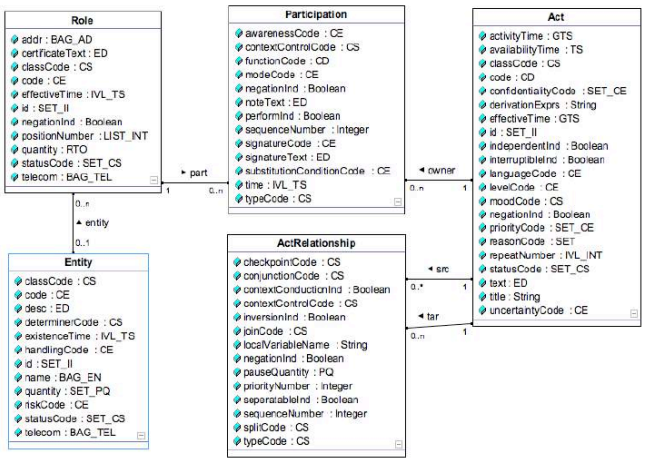


Fig. 1. HL7 RIM foundation classes

relationships among them, namely *ActRelationship* and *Participation*. Each of these foundation classes has a separate specialization hierarchy in the RIM (not shown here). Examples for specific acts are *Observations*, *PatientEncounters*, *DeviceTasks*, *DiagnosticImages*, *FinancialTransactions*, etc. Examples for specific entities are *Containers*, *Devices*, *Organizations*, *LivingSubjects*, *Persons*, *Places* etc. Examples for specializations of class *Role* are *Patients*, *Employees*, *LicensedEntities*, etc. In order to keep the RIM inheritance hierarchy from growing to large and to provide better model stability, HL7 has introduced the convention to avoid introducing new specializations of foundation classes if these specializations do not require additional data attributes. In this case, a discriminator attribute "classCode" is used to distinguish between different major subtypes, and a secondary discriminator attribute "code" is used to differentiate between minor subtypes (of the major types). The minor subtypes are provided in separate tables in the standards specifications.

The data type "CS" used for attribute "classCode" stands for "coded simple value" and is one of several data types defined in the RIM. In HL7 terminology, a "coded" value is a value in a tightly controlled and standardized vocabulary. Since, many such vocabularies exist, the comparable data type "CE" ("coded with equivalents") is similar to "CS" but enables the exchange of values using multiple equivalent coding systems. Another interesting attribute is called "moodCode" in class "Act". It determines the mood of an act, i.e., whether it is planned, recommended, observed, reported, suspected, etc. Obviously, a complete coverage of all RIM data types is beyond to scope of this paper. It is important to note the complexity of even the simplest of these data types. Each coded entry, for example, contains several fields specifying the coding system, its version, the actual codes, the informal semantics, a set of equivalent codes, etc. It is clear that this model cannot simply be implemented in a relational database in a straight-forward fashion. Refinement, scoping and normalization steps have to be performed to achieve an implementation that fulfills the specific requirements of the application and is standards conform.

The RIM contains additional model constraints formulated as narratives and using OCL. While the foundation structure of the RIM has been quite stable, details of the model are constantly being revised to accommodate additional domain specific functions in the health care area. For this purpose technical committees create so-called domain information models (DIMs), which are based on the RIM but contain domain-dependent extensions. The resulting DIMs are used to evolve the RIM in order to make it more comprehensive. This process is called *harmonization*. Moreover, the HL7 methodology enables different nations (HL7 international affiliates) to extend the HL7 v3 RIM to meet local needs - a process called localization.

2.1.2 Extensions to the UML Meta Model

HL7 has made several extensions to the UML Meta Model for the purpose of specifying the RIM. The most important extension for the purpose of this paper is the designation of model attributes as optional or mandatory.

2.1.3 The HL7 Development Methodology

HL7 has initially started their work with developing a messaging standard for medical data exchange. Therefore, its development process and methodology concentrates on deriving message specifications. Fig. 2 gives a (simplified) overview on the process

described in the HL7 Message Development Framework (MDF) document published by the HL7 Modeling & Methodology Committee [5]. The process is driven by a functional analysis of the information exchange requirements of a particular application. In this process, use cases and message interaction models are created to determine the particular information needs. These particular information needs are reflected in the derivation of a so-called Domain Information Model (DIM). The DIM is typically a subset of the RIM which targets a particular domain. A set of particular model transformations are used in the process of deriving the DIM. Essentially, DIM elements are classes and associations copied from the RIM. They can be renamed and restricted, e.g., by removing optional attributes, making optional attributes mandatory and by adding other logical model constraints (using OCL). This is done in a process called *cloning and restricting* [5]. If there are information requirements in the DIM that go beyond the information model that is provided in the RIM these extensions have to be harmonized with the RIM by submitting a harmonization proposal to the Modeling & Methodology (MnM) committee at HL7.

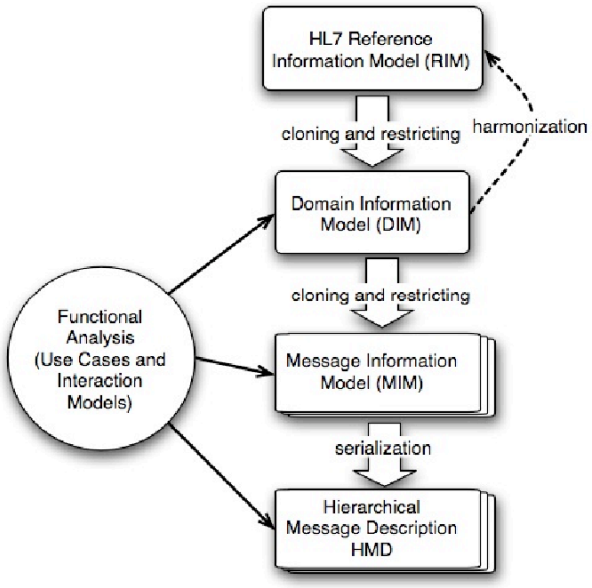


Fig. 2. HL7 Message Development Process

A similar derivation process is used to create a set of message information models (MIMs) from the DIM. Each MIM describes the specific information content of a message or family of messages. Further well-formedness restrictions apply to MIMs. For instance, it is mandatory that the classes in a MIM form a connected graph, i.e., there must be a path of associations that enables navigation from each class to each other class.

MIMs and DIMs are represented analogously to the RIM using UML class diagrams with additional annotations representing constraints. Further refinement transformations are executed on the MIM in preparation for the definition of the actual message type. Examples for refinements are the specification of navigation directions for associations in the model, specification of default values for attributes, etc. The refined MIM (called R-MIM in the HL7 methodology) contains a table that annotates each model element in the UML diagram. The R-MIM is then translated into a hierarchical message description (HMD). The HMD is a hierarchical table that specifies how a message that conforms to a certain MIM is to be formatted (serialized) into XML. Deriving an HMD from an R-MIM essentially involves specifying a "graph walk" through the R-MIM model, which serializes the necessary information into a hierarchical message.

3. Tool support for HL7 development methodology

To date, there is only very limited tool support for the HL7 development methodology as summarized in the previous section. While UML class models and tables exist in tools such as Microsoft Visio(tm) and Microsoft Excel(tm), their support does not go much beyond drawing and listing models. Design operations such as "cloning and restricting" (see previous section) are crudely supported only by relying on general built-in functionality of these tools. For example, a user would create a new UML class drawing and manually copy and rename RIM classes while deleting unwanted non-mandatory attributes in order to manually execute the "cloning and restricting" steps to arrive at a specific DIM (cf. Fig. 2). It is easy to loose consistency in such a process and the tools do not provide much support in checking or establishing consistency.

We have set out to develop better tools support for HL7 message development in our lab. In particular, we are interested in applying theories developed in the software engineering field of model-driven development to this application domain. At the heart of model-driven development is a formal concept of model transformations. Sendall and Kozaczinky categorize three approaches to defining such transformations [6]:

- *Direct model transformations* use APIs to access the internal representation of models in software tools.
- *Intermediate representation* exports models in a (standard) exchange format (such as XMI) for subsequent processing with other tools.
- *Transformational language support* provides a dedicated language to specify, execute and apply model transformations.

From the start, we have discarded the first approach because it is too difficult to use by HL7 designers. We considered using the second approach and have done various experiments with XMI-based tools. However, we registered two main difficulties, one of conceptual nature and one of technical nature: The first issue was that HL7 design transformations are done in a highly interactive manner. It is a human-driven design process and humans choose which transformations to apply and where to apply them. However, the intermediate representation approach describe above works best in "batch-oriented" automated transformation processes. It lacks in interactivity and in incrementality and thus represents a poor choice for our application. Secondly, in our experiments, we discovered serious difficulties with tool interoperability, even if they are supposed to support the same standard (e.g., XMI). Therefore, we have selected the third approach and evaluated various modelling tools with transformational language support. Important requirements in this evaluation were that the tool candidate should be easily extensible, run on any major operating system platform, and be licensed under an open-source model.

Sendall and Kozaczinky state four desirable properties to guide the selection of a transformational language for model-driven development [6]:

- *Preconditions* - the ability to describe and enforce conditions under which model transformations can be applied.
- *Composition* - the ability to compose complex model transformations from more simple ones.
- *(Native) Form* - in order to maximize comprehensibility, transformational languages should use the same form as the structure that is being transformed. If the structure is a visual graph such as a UML class model (as in our case), the transformation language should also be visual to simplify the user's cognitive load.
- *Usability* - A key aspect of usability is conciseness. Declarative languages are usually more concise and thus more usable.

Based on the above considerations, we have selected FUJABA (From Uml to Java And Back Again) [7] as the platform for developing our model-driven development tool, nicknamed *Harmony*. FUJABA is an advanced UML tool that features an extension of Activity Diagrams called *Story Diagrams*. Story Diagrams are a visual transformational language based on the theory of graph replacement systems [8]. They fulfill all desirable requirements for a transformational language in our application domain. FUJABA itself has a plug-in concept that makes extensions easy, is available under an open-source license and runs in the Java(tm) virtual machine (platform-independent).

3.1 A FUJABA-based model-driven design tool for HL7 messages

Before we describe how we constructed *Harmony*, our FUJABA-based model-driven HL7 design tool, we use the following subsection to describe the user experience we had in mind going into the construction process.

3.1.1 User experience

Fig. 3 shows that the user should be able to derive DIMs and MIMs from the RIM using a transformation-based approach. Transformations can be applied in two forms: *Initial transformations* are used to create a first version of a derived model from the model it was derived from (e.g., when creating a DIM from the RIM). Subsequent *in-place* transformations can be used on the initially derived model for the purpose of refinement. An example for an initial transformation is the aforementioned *clone* operation, while the aforementioned *restriction* operation serves as an example for an in-place transformation. Automated consistency checks would ensure that the user can only perform those model changes that do not violate consistency with other models in the derivation chain. In particular, the usual "free editing" operations of UML tools should be unavailable in this context to avoid confusion.

As the ultimate source that everything else is derived from, the RIM is a special model that should be pre-loaded at tool start-up. Nevertheless, the user should be able to update and change the RIM in order to implement revisions to the standard due to *harmonization* (cf. Section 2.1). Similarly, the library of available transformation commands should be pre-loaded at start-up, but it should also be accessible for extensions and changes when the HL7 development methodology evolves.

3.2 Construction of Harmony

We decided to construct Harmony as a plug-in to FUJABA. The plug-in mechanism easily allows us to make available custom diagram types as specializations of UML standard diagrams (such as RIM, DIM, MIM etc.). Moreover, it allows us to introduce custom operations on these diagram types (such as *clone* and *restrict*) and to disable standard operations such as *create* and *delete*.

As mentioned before, the RIM has a special role in the HL7 methodology. However, there is no technical difference on how to handle DIMs and MIMs, since these models just represent subsequent refinement steps in the development process. Therefore, we decided to create only two new diagram types, namely *RIM Diagram* and *Harmony Diagram* (to capture DIMs and MIMs). Internally, both diagram types are realized as specializations of the built-in diagram type *Class Diagram*. However, the "normal" editing operations have been made unavailable for Harmony Diagrams; they have been replaced with HL7 model transformations.

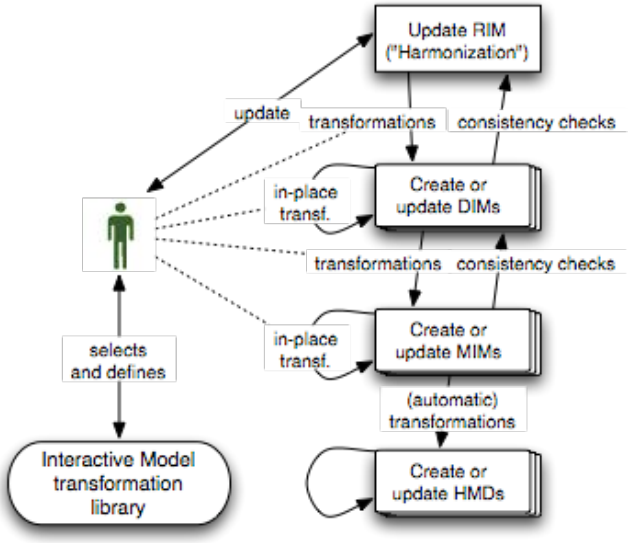


Fig. 3. Harmony User Experience

3.2.1 The RIM Diagram Type

Since our attempt to load an XMI representation of the RIM into FUJABA failed, we modelled the RIM by hand. This process was straight-forward except for the RIM concept of *optionality* in attributes (cf. Section 2.1.2). This concept is not part of the UML meta model and, thus, was not available in the FUJABA meta model either. We considered extending the FUJABA meta model, but, after talking to the FUJABA developers, we decided against it. An extension of the FUJABA core would destroy our desired notion of a purely component-based (plug-in) extension. Instead, we decided to dedicate the existing meta-model attribute *visibility* to express our *optionality* concern, since the visibility notion is not used in the context of our application. From a user's perspective, this work-around is invisible, since we created new operations to define the optionality of attributes.

3.2.2 The Harmony Diagram Type

The definition of the Harmony Diagram type involved more engineering work, since we needed to replace the standard operations on UML class diagrams with a whole new set of transformational operations. As indicated earlier, we wanted to

provide the user with the ability to use visual Story Diagrams for defining, adapting and extending the transformations available in Harmony. FUJABA has the ability to create executable Java code from story diagrams. In order to be able to do this, we had to model the part of the FUJABA UML meta model used by Harmony (class diagrams) within the FUJABA environment. A few minor changes had to be made to the FUJABA core code in order to make its meta model fully compatible with the FUJABA code generation. The FUJABA development team was very helpful in implementing these changes and they are now part of the main distribution.

With the FUJABA meta model available as a FUJABA model, we were now able to specify the HL7 model transformations in terms of Story Diagrams. We created a plug-in command within FUJABA to generate executable code for revising the transformation library in the Harmony plug-in (cf. Fig. 4).

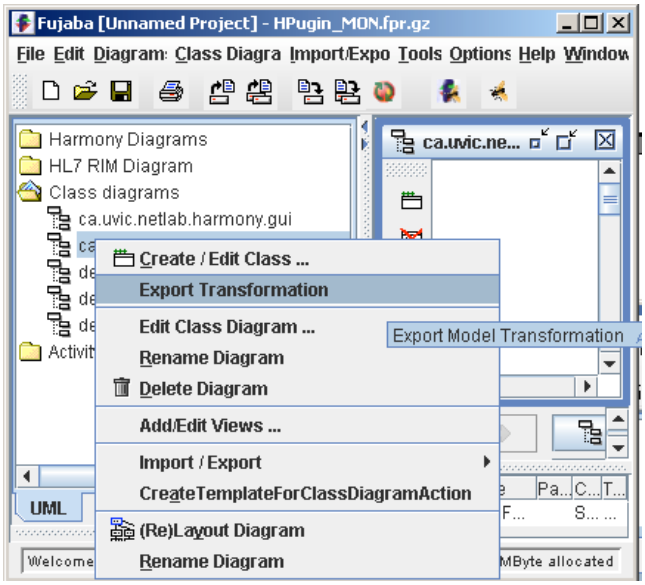


Fig. 4. Export and generation of specified transformations

The export operation would automatically compile the code generated for the new transformations and package it with the Harmony plug-in. The user just needed to restart FUJABA to load the revised Harmony plug-in. The new model transformations would then become available as additional ore revised Harmony operations (cf. Fig. 5).

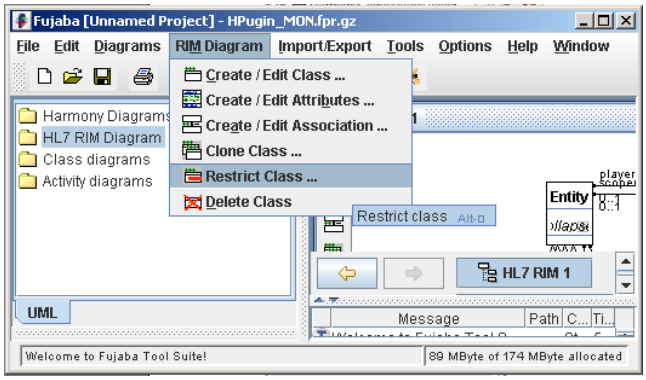


Fig. 5. Example Harmony transformation *Restrict Class*

3.2.3 Modes of working with Harmony

The preceding discussion shows that there are two principal ways of working with Harmony: the user may either *use* an existing transformation library and RIM model, or she may *update or extend* the transformation library or the RIM. In the latter case, the user needs to see all Story Diagrams that specify the current model transformations. However, in the first case, the user should not be bothered with this complexity. Therefore, we decided to enable the user to switch between a normal HL7 development session and an "advanced" mode for customizing the model transformations. Fig. 6 shows the Harmony preference panel that is used to switch between these modes.

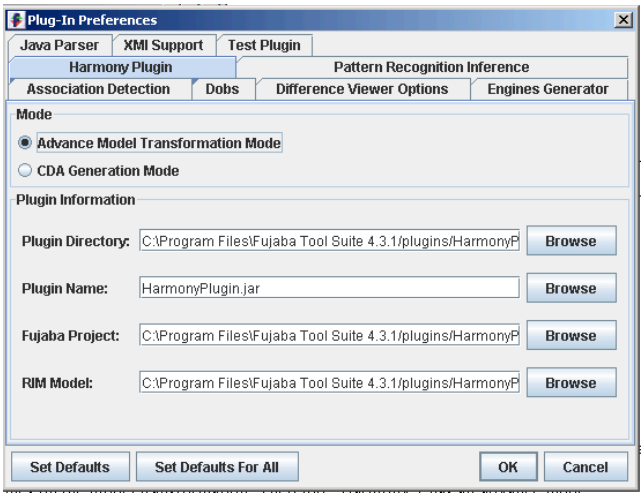


Fig. 6. Harmony preference panel

4. Evaluation and current work

We have gained in-depth practical experience with the HL 7 development process in a project called TAPAS (Technology Assisted Practice and Application Suite) (www.opentapas.org). TAPAS was funded by two western Canadian health authorities. Its mandate has been to create an open source software application to support the delivery of primary care. Its main users are general practitioners, medical office assistants and nurses. In its current version TAPAS does not implement all functions of a complete Electronic Medical Record (EMR), however it provides many if its functions.

Fig. 7 gives an overview on the requirements implemented in TAPAS. It shows three types of messages to be exchanged at the system boundaries, namely *electronic medical summaries* (e-MS), a *patient core data set* (e-CDS), and a *recommendations document*. The e-MS message is an HL7-compliant CDA document that contains all medical data relevant for referrals among GPs. Its structure was standardized within the e-MS group on Vancouver Island, B.C. The full message specification can be downloaded at www.e-ms.ca. The other two message types integrate TAPAS with an Electronic Guideline And Decision Support System (EGADSS) (www.egadss.org). The e-CDS contains many core elements of a patient's medical record, which are used by an automated decision support engine to produce point of care reminders and alerts in form of an electronic recommendations document that is sent back to TAPAS.

In our evaluation, we tried to generate parts of the DIM for the HL7 CDA used in TAPAS with Harmony. We needed several iterations to extend and update the transformation library in order to reach this goal. We could ultimately validate the feasibility of our approach in this manner. However, it became clear during the practical tool application, that this way of working top-down from the RIM is quite tedious for complex models such as the CDA DIM and MIM. After all, in many cases there is no need to start from scratch, but domain-specific HL7 standard models may actually exist (such as the HL7 CDA or the e-MS used in TAPAS). In the current version of Harmony, such models can only be loaded if they had been developed with Harmony to start with. This is because Harmony keeps track on how these models depend on other, more abstract models such as the RIM. If domain-specific models (DIMs or MIMs) were loaded, which had not been developed with Harmony, this dependency information and the corresponding consistency checks would be impossible.

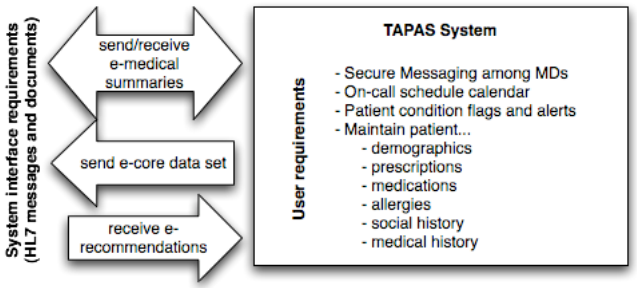


Fig. 7. TAPA requirements

From a practical standpoint, however, it would be much more convenient to be able to leverage models that had been developed with other tools. To enable such functionality, we would need a mechanism to establish the appropriate dependencies to other models, in particular the RIM. For obvious reasons, such a mechanism should be as automated as possible. In another application context, we have successfully applied Triple Graph Grammars to establish such a reverse engineered mapping between models [9]. Our future work will focus on investigating whether some of these results can be applied in the context of Harmony.

Another technical weakness of the current implementation of the Harmony tool is that it is tied to the UML meta model implemented within FUJABA. So far, this has not been a big obstacle, since the meta model used in HL7 RIM is very close to the UML meta model with only minor deviations such as the aforementioned "optionality" attribute. However, OMG has created the Meta Object Facility (MOF) to provide support for different meta models. It is expected that an adoption of the MOF by HL7 will allow their meta model to deviate more freely from the UML meta model. As a consequence, it will be harder to implement Harmony based on a UML tool. A possible solution may be a MOF-based extension to FUJABA called MOFLON, which is currently under development at the University of Darmstadt [10].

5. Related work

The current set of tools used internally by the HL7 Methodology & Modeling Committee is described in [11]. It is largely based on a collection of customized and extended off-the-shelf software components. The RIM is maintained in the Rational Rose UML tool and can be exported to an Access 2000-based model repository. A custom VisualBasic application called *RoseTree* is used to view and access this repository. A Visio 2000-based *RMIM-Designer* tool exists, which can be used to design refined MIMs. It does this by visually annotating the RIM foundation classes with specific "derivation symbols" in order to denote model transformations. RMIM-Designer makes use of *RoseTree* to perform certain consistency checks against the RIM. The transformations allowed in RMIM-Designer are more limited compared to Harmony (and the HL7 methodology) [11]. Moreover, in contrast to Harmony, DIM design, consistency checks of MIMs against DIMs and DIMs against the RIM are not supported.

More generally, our work is related to software engineering research and development carried out in the field of *model-driven development* (MDD) [12]. While most MDD research targets software development, we are mainly interested in model-driven data structure development. In particular, we are interested in maintaining and managing inter-model dependencies and constraints. A formal notion of meta-model management based on canonical transformations is explored by Bernstein [13].

6. Acknowledgements

This research was supported by the Innovation Council of British Columbia (BCIC). We thank Christina Obry for her help in implementing Harmony. Thanks to the FUJABA teams in Kassel and Paderborn for their assistance in trouble shooting problems along the way. Thanks to the TAPAS development team for allowing us to use TAPAS as a case study.

7. References

[1] R. Rogers, J. Sembritzki and P. Village, "Priorities for application of ICT standards – A project report for ethel thematic working group T1," CEN/TC 251 Secretariat: SIS - Swedish Standards Institute, SE-118 80 Stockholm, Sweden, Tech. Rep. CEN/TC 251/N02-073, 2002.

[2] R. H. Dolin, L. Alschuler, S. Boyer, C. Beebe, F. M. Behlen, P. V. Biron and A. Shabo Shvo, "HL7 Clinical Document Architecture, Release 2," J. Am. Med. Inform. Assoc., vol. 13, pp. 30-39, Jan-Feb. 2006.

[3] G. W. Beeler Jr, "On the Rim: the making of HL7's Reference Information Model," MD Comput., vol. 16, pp. 27-29, Nov-Dec. 1999.

[4] J. H. Jahnke, "Engineering component-based net-centric systems for embedded applications," Proceedings of the 8th European Software Engineering Conference Held Jointly with 9th ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 218-228, 2001.

[5] J. M. Ferranti, R. C. Musser, K. Kawamoto and W. E. Hammond, "The clinical document architecture and the continuity of care record: a critical analysis," J. Am. Med. Inform. Assoc., vol. 13, pp. 245-252, May-Jun. 2006.

[6] S. Sendall and W. Kozaczynski, "Model transformation: the heart and soul of model-driven software development," Software, IEEE, vol. 20, pp. 42-45, 2003.

[7] U. Nickel, J. Niere and A. Zundorf, "The FUJABA environment," Software Engineering, 2000.Proceedings of the 2000 International Conference on, pp. 742-745, 2000.

[8] G. Rozenberg, Handbook of Graph Grammars and Computing by Graph Transformation: Volume I. Foundations. World Scientific Publishing Co., Inc. River Edge, NJ, USA, 1997,

[9] J. Jahnke, W. Schäfer and A. Zündorf, "A Design Environment for Migrating Relational to Object Oriented Database Systems," Proc.of the 1996 Int.Conference on Software Maintenance (ICSM'96),

[10] C. Amelunxen, T. Rötschke and A. Schürr, "Graph transformations with MOF 2.0," in Proceedings of the 3rd International Fujaba Days 2005: MDD in Practice, 2005,

[11] G. W. Beeler, Jr., "Documentation of HL7 internal tooling structure, metadata,
and semantics, release 1," HL 7 Methodology & Modeling Committee, 2004.

[12] J. Bezivin, "Model driven development," in Proceedings of the 3rd International Fujaba Days 2005: MDD in Practice, 2005, pp. 15.

[13] P. A. Bernstein, "Applying Model Management to Classical Meta Data Problems," Conference on Innovative Data Systems Research (CIDR), pp. 209–220, 2003.

Graph-oriented Storage for Fujaba Applications

Erhard Schultchen, Ulrike Ranger, and Boris Böhlen
[schultchen|ranger|boehlen]@cs.rwth-aachen.de
Department of Computer Science 3
RWTH Aachen University
52074 Aachen, GERMANY

ABSTRACT
Fujaba supports the visual modeling of software applications and the generation of according Java code. During its execution, the runtime state of the generated applications can be saved and restored using the CoObRA framework. In this paper, we present the graph-oriented database DRAGOS for the persistent storage as alternative to CoObRA. Due to the extensive functionality of DRAGOS, the database offers the possibility for introducing new language features in the Fujaba modeling language. Furthermore, the usage of DRAGOS supports rapid prototyping of Fujaba specifications based on the UPGRADE framework.

Keywords
Graph Storage, Modeling Language, Rapid Prototyping

1. INTRODUCTION
Fujaba facilitates the modeling of software applications in a visual way. For this purpose, the developer draws UML-like diagrams, e.g. class diagrams for the static structure and activity diagrams for the dynamic behavior of the software application. Based on this specification, Fujaba generates according Java code, which can be executed in a *Fujaba application*.

Internally, the specified application is translated into a *graph transformation system*, thus the generated code covers appropriate data structures and graph transformations. At runtime, the application's host graph consists of Java objects representing the graph nodes. The execution of the generated transformations leads to modifications of the host graph. All Java objects together form the *state* of the application, and are managed by the Java runtime environment.

Basically, the generated Java code does not provide any support for the persistent storage of the Java objects. To fill this gap, CoObRA [6] can be used, which is a lightweight framework for the storage of Java objects. By adapting the Fujaba code generation to CoObRA, the state of a Fujaba application is automatically stored persistently. A severe drawback of the CoObRA approach is that only *changes* made to objects are registered. CoObRA does not control the *access* to objects, which is required e.g. for providing consistency in case of concurrent changes. To ensure consistency, isolation of concurrent (both reading and writing) operations has to be ensured. For this, CoObRA only provides the isolation level *read uncommitted* known from databases. An application's object structure may therefore only be accessed by a single thread, otherwise the result is not predictable. When the generated application is used by several users concurrently, e.g. if it is part of a distributed system, this restriction is not acceptable.

In this paper, we present the graph-oriented database system DRAGOS[1] [1] as alternative to CoObRA. DRAGOS has been developed at our department, and is used by applications generated with the graph rewriting system PROGRES [7]. As Fujaba also uses graph transformations internally, we have realized the integration of DRAGOS in Fujaba. DRAGOS can utilize existing database management systems (DBMS) for storing graphs. Besides, the DBMS's functionality can be used by DRAGOS, e.g. to allow concurrent access with the DBMS's isolation level . In addition to the persistent storage of runtime data, DRAGOS enables new features in Fujaba, like the evaluation of *derived attributes*.

The paper is structured as follows: Section 2 introduces the DRAGOS architecture, and describes the mapping of the application's runtime data on the database. Based on the functionality of DRAGOS, section 3 shows possible extensions of the Fujaba modeling language. Additionally, the graphical prototyping framework UPGRADE is presented, which can be used for the visualization of Fujaba applications. A summary and an outlook on future work are given in section 4.

2. GRAPH-ORIENTED DATABASE
The graph-oriented database DRAGOS is designed to store typed, and attributed graphs in a repository. In this section, we present DRAGOS and its integration in Fujaba. First, the architecture of DRAGOS is described and an introduction to the basic functionality is given. Second, we present the DRAGOS graph model describing the stored graphs. Third, the integration in Fujaba is presented.

2.1 DRAGOS Architecture
An overview of the DRAGOS architecture is given in figure 1. The DRAGOS Kernel offers the basic functionality, which comprises defining, creating, modifying and querying graphs. In detail, the DRAGOS Kernel consists of the following components:

[1]DRAGOS is an acronym for *Database Repository for Applications using Graph-Oriented Storage*, and was formerly called GRAS/GXL.

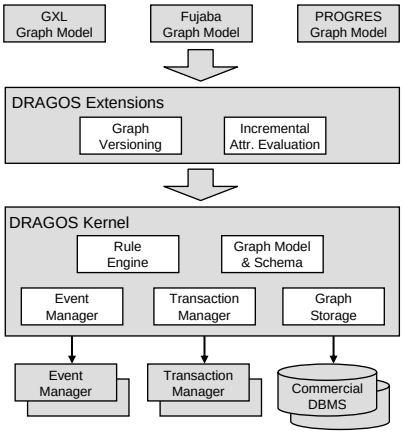


Figure 1: DRAGOS architecture overview

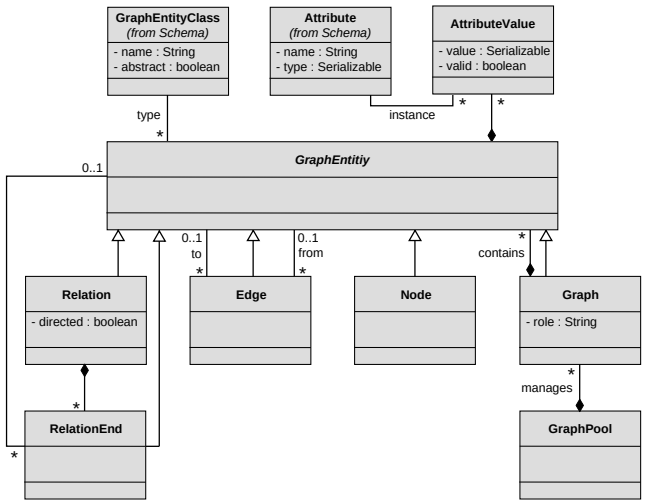


Figure 2: DRAGOS graph model

- The Event Manager propagates events raised on graph modifications to registered listeners. Users can choose between different Event Manager implementations by changing a configuration file. For example, building a distributed system consisting of several DRAGOS instances requires a special event manager. This manager has to propagate events between the different DRAGOS systems, which is not offered by the provided standard implementation.
- Using the Rule engine, applications may register event-condition-action rules to perform the defined actions when selected events are raised.
- The Transaction Manager allows to start, commit and rollback transactions. Analog to the Event Manager, the provided implementation can be exchanged easily.
- The Graph Storage provides access to an underlying data storage. For the persistent storage of graphs DRAGOS uses relational DBMS like PostgreSQL, but can also store volatile graphs in main memory[2]. Using existing DBMS allows to use their functionality, such as transactional operations with respect to the ACID properties. The DRAGOS Transaction Manager controls the DBMS transactions in this case.
- The Graph Model and the Graph Schema describe the structure of the stored graphs. The graph model is presented in detail in section 2.2.

Above the DRAGOS Kernel, extensions to the basic functionality can be implemented by extension modules. Until now, versioning of graphs (including the management of configurations) and an engine for the incremental evaluation of derived attributes are available. This modularization of the DRAGOS functionality allows applications to avoid unnecessary overhead, e.g. an application, which does not require versioning, does not activate this extension. Furthermore, DRAGOS can be easily extended by developing new extensions, which can be combined with existing ones.

On top of the DRAGOS architecture, applications are integrated by specialized graph models. This is presented in section 2.3 considering the Fujaba graph model.

[2]The in-memory storage can be serialized to disk when closing the graph pool and thus enables persistent storage, too.

2.2 Graph Model

In order to represent a graph in volatile or persistent memory, a formal notation about the entities a graph consists of is required. For this purpose, DRAGOS provides a general *graph model*, which is shown in figure 2. The interface of the Graph Storage (cf. fig. 1) is designed according to this graph model. For example, methods to retrieve all GraphEntitys or to get the source and target of an Edge are provided.

In the DRAGOS graph model, GraphPool contains a set of Graphs. A Graph holds a collection of GraphEntity, which is - according to the inheritance relation - either a Node, a binary relation (Edge), an n-ary Relation or another Graph. A Graph may be contained within another Graph, allowing hierarchical structures. Edges and Relations can connect arbitrary GraphEntitys, even if they are contained in different graphs within the same GraphPool.

Valid graph structures are defined by the graph schema, which is not shown here for the lack of space. In contrast to the graph model which defines the entities a graph consists of, the schema provides the typing system. For this, the graph schema defines classes for every element of the graph model. In figure 2, the GraphEntity refers to a type definition GraphEntityClass which is part of the schema. The schema contains a dedicated Class element for every subclass of GraphEntity.

The DRAGOS graph model allows all GraphEntitys to be attributed. As AttributeValue, serializable Java objects (including primitive data types) and references to other graph entities are supported. Attribute definitions are also part of the graph schema, as indicated by the *(from Schema)* marking in figure 2.

The DRAGOS graph model is based on the graph exchange format GXL [8]. However, GXL does not treat graphs as graph elements. As a consequence, e.g. edges are not al-

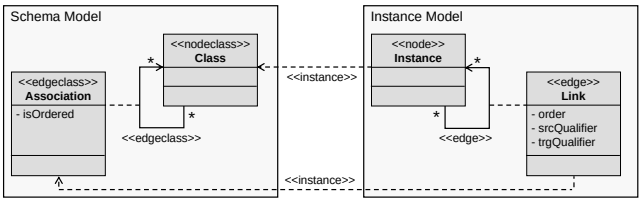


Figure 3: Fujaba graph model and schema

lowed to connect two graphs, but only elements contained in these graphs. In the DRAGOS graph model, a graph is a graph entity, too, and may therefore be incident to edges or relations. Furthermore, the DRAGOS model simplifies the GXL model to enable a more efficient implementation. For example, an attribute in the DRAGOS model cannot be attributed by another attribute, which is allowed in GXL to an arbitrary nesting depth. Following the GXL model, an attribute value would have to be treated as a graph element in order to identify it for attributing. In contrast to GXL, the DRAGOS model does not provide ordered edges or relations to reduce overhead for applications not using them. As we will present in the following, support for ordered edges can easily be added.

2.3 Graph Model Mapping

DRAGOS supports graph-oriented applications from various domains, and therefore does not tie applications to a common graph model. A common model would either restrict the applications' expressiveness or clutter applications with elements not required. Hence, DRAGOS allows applications to use a *specialized* graph model, which can be designed according to the applications' needs. The specialized graph model for Fujaba is implemented by mapping the Fujaba elements to the DRAGOS graph model. Using the tool SUMAGRAM [4], this mapping can be partially modeled graphically based on UML class diagrams.

For the support of Fujaba applications, we developed a specialized graph model reflecting object-oriented data in the graph database (cf. fig. 3), comprising a *schema model* and an *instance model*. The schema model is composed of Class and Association elements and describes the structure of a graph schema. The instance model consists of elements instantiating the schema model.

The mapping of the specialized graph model on the DRAGOS model is depicted by stereotypes enclosed in <<>>. Elements of the instance model are mapped to the DRAGOS graph model whereas elements of the schema model are mapped to the DRAGOS graph schema.

As mentioned above, the DRAGOS graph model does not provide *ordered edges*. As they are needed by Fujaba applications, these can be integrated into the specialized graph model. For ordered associations, the creation of Link instances automatically assigns an ordering index[3]. This behaviour is implemented as a method in the specialized graph model, which is depicted below in a simplified form:

```
public void createLink (Instance src, Instance trg,
                        Association type, int index) {
  Edge edge =
    this.graph.createEdge (src.getDragosNode(),
                           trg.getDragosNode(),
                           type.getDragosEdgeClass());
  if (type.isOrdered()) {
    reorderEdgesForInsert(src, trg, type, index);
    edge.setMetaAttribute(META_ATTR_ORDER, index);
  }
}
```

The method `createLink` gets two `Instance` objects and an `Association` object from the specialized graph model as parameters. In addition, the parameter `index` is passed as the insertion index for ordered associations; it is not considered for unordered ones. To create the link, according elements of the DRAGOS model are determined using the methods `getDragosNode` and `getDragosEdgeClass`. Using the returned elements of the DRAGOS model as parameters, the method `createEdge` is invoked on the `graph` object representing the application's runtime graph. A new edge of the given type is created between the two nodes by this method. If the association is ordered, existing instances connected to the same `src` or `trg` are reordered according to the insertion position `index`. After reordering, the insertion index is stored as a meta attribute of the new edge. Please note, although a Link class is depicted in figure 3, we do not create a `link` object corresponding to the `edge` object from the DRAGOS model. To fully reflect the depicted specialized graph model, the method `createLink` would have to create and return a new instance of the `Link` class. The reason this was omitted is efficency, as the Fujaba generated code does not use these objects. In fact, another method `retrieveLink` can be used to create these objects when required. This method is called by the UPGRADE framework (c.f. sec. 3.2). We omit to explain the support of qualified associations in this paper, as the implementation is similar to ordered associations.

2.4 Specialized code generation

With the specialized graph model, Fujaba applications can store their runtime state in the DRAGOS database. As next step, we have to modify the code generation to make use of DRAGOS. Based on CodeGen2 [3], we developed a plug-in for this purpose.

As basic idea, we aim to change the code generation as little as possible to preserve compatibility with existing applications. For example, we retain the accessor methods for attributes and associations, but change their implementation according to the specialized graph model. As an example, we present the set-method for a many–to–one association. With DRAGOS in use for storing the runtime state, the code is generated as follows[4]:

[3]We could have implemented the same behavior based on edges, which connect links to indicate the ordering. However, this would have increased the overhead for insertion and removal of these links.

[4]Code required for exception handling and transaction management has been excluded here.

```
public boolean setAR (AR value) { [...]
  if (value != null) {
    RepositoryManagerFactory.
      getDefaultRepository().
      createLink(this, "cR",
                 value, "aR",
                 "RM.CR:cR-aR:RM.AR");
  } else {
[...] }
```

This code block retrieves the `Repository` representing the runtime graph of the application. The `Repository` provides the method `createLink` for connecting two objects (`this` and `value`). As Fujaba uses role names to distinguish between the source and the target of a link, these are passed too (`cR` and `aR`). The association's name is passed as last parameter, which is constructed from the attached classes and the corresponding role names. The `createLink` method creates the desired edge regarding uniqueness constraints defined by the schema. If such a constraint is violated, an existing link of the same type is automatically removed. As the method `setAR(AR)` is also used to remove connections between objects by passing `null` as `value`, deletion of links has to be handled in the `else` part of the `if`-statement.

The current code generation creates accessor methods for attributes and associations. These methods could be completely abolished by using DRAGOS. Thus activity diagrams would solely utilize the methods provided by the specialized graph model for pattern matching and graph modifications. The generated code would become clearer because a large number of methods would not be created. However, other parts of the Fujaba environment such as the support for path expressions or the assignment of attributes rely on accessor methods. So, discarding accessors would require far reaching changes to the code generation. Hence we chose the more compatible approach by exchanging the accessors' implementations only.

3. APPLICATION AREAS

The usage of DRAGOS as underlying database for Fujaba applications enables several extensions. On the one hand, new features for the Fujaba modeling language can be supported, which are described in section 3.1. On the other hand, the modeled application can be visualized using the UPGRADE framework leading to a configurable graphical user interface shown in section 3.2.

3.1 Language Extensions

At runtime, Fujaba can only handle one host graph consisting of connected objects. I.e. separate graphs within one application and even isolated objects within one graph are not supported. This restriction is founded on the fact that Fujaba uses the Java runtime environment for managing its objects. Consequently, Fujaba only supports the search of object structures[5] *consisting of connected objects*. Additionally, these object structures need at least one defined *entry object* for their processing. For example, an entry object is given by the `this`-object or by a method parameter.

[5]To execute a graph transformation specified within a Fujaba activity, a certain object structure (*pattern*) has to be found within the host graph. This structure consists of all objects, which are either obligatory or have to be destroyed within the activity.

Using DRAGOS as underlying database, these limitation can be easily removed: DRAGOS supports the handling of different graphs simultaneously and of isolated objects within one host graph. For example, isolated objects can be found by querying the database for all objects of the requested object's type. This mechanism also enables the search of objects structures without any specific entry object. Another solution to enable isolated objects is to introduce a hidden (singleton) class referencing all objects. However, consistency of these references would have to be assured by the code generation.

Another limitation of Fujaba is the support of only intrinsic attributes within classes, whose concrete values are directly assigned within activities. This does not exploit the capabilities of using graphs as underlying data structure, as these also offer the possibility to define more complex attributes like *derived attributes* in PROGRES.

The values of derived attributes are not assigned directly, but are computed according to other graph objects and their attributes. These computations can be arbitrarily complex and refer to the entire host graph. The attribute is evaluated only on graph modifications, which impact the current attribute value. With the DRAGOS extension Incremental Attribute Evaluation, derived attributes can be easily integrated in Fujaba.

Fujaba allows the definition of *paths*, which aggregate an arbitrary number of associations (edges) of possible different types. Unfortunately, the usage of paths affects largely the runtime efficiency of the Fujaba application, as they have to be evaluated on every employment. The Incremental Attribute Evaluation extension also enables the definition of *static paths*, so that frequently used paths can be materialized in the database. Thus, the path is only evaluated on graph modifications, which change the path, improving the runtime efficiency.

With the support of isolated nodes, the abolishment of necessary entry objects, and the definition of derived attributes, DRAGOS offers new and useful modeling elements in the Fujaba language. Furthermore, the introduction of static paths improves the runtime efficiency of Fujaba applications.

3.2 Rapid Prototyping

Fujaba provides DOBS for executing the generated code and for visually representing the runtime state. However, DOBS allows only little customization to the representation and the user interface.

At our department, we developed the UPGRADE framework [2]. UPGRADE allows rapid prototyping by executing code generated from a PROGRES specification. The runtime graph of the generated application is presented visually. Furthermore, users can execute graph transformations and view their affects on the runtime graph. UPGRADE is highly customizable using configuration files, and can be extended through a programming interface. In addition, UPGRADE provides a filter stack to separate the user's view from the underlying data. This filter stack can be customized by the user. New filters can be implemented using the provided API. For layouting the graph structure,

UPGRADE includes a set of layout algorithms: E.g. the sugiyama layout for hierarchical graph structures and the nicolay layout for constraint-based layout.

As UPGRADE offers a lot of functionality, adapting it for Fujaba generated code is desirable. Until now, this was not possible because Fujaba applications do not offer an explicit graph model and schema. Concerning the graph schema, node types are provided through generated Java classes, but associations are not explicitly available. DOBS solves this problem by inferring associations from the respective access methods generated in the incident classes. This behavior would have to be re-implemented in UPGRADE. Concerning the graph model, UPGRADE requires methods to retrieve a list of all nodes and edges from the graph. Fujaba generated applications do not provide direct access to the graph's nodes because the Java runtime environment does not allow to list its objects. Appropriate support like an extra class referencing all objects would have to be added to the generated code. Accessing the graph's edges would require even more extensive changes to the generated code.

In addition to the graph model, UPGRADE requires events to be raised upon changes to the runtime graph for incrementally updating the display. With the standard Fujaba generated code, these events can only be obtained by registering a change listener at each node of the runtime graph. This approach is not efficient for large graphs, especially if only an excerpt of them is displayed. This might also cause memory leaks when change listeners are not properly removed from deleted nodes.

With the introduction of DRAGOS, this problem can be easily solved, as direct access to the graph elements and a central event manager are provided. A simple adapter class has to be implemented, so that UPGRADE can access the Fujaba graph model. This adapter is currently under development.

4. CONCLUSION

In this paper, we have presented the graph-oriented database DRAGOS for persistently storing the runtime data of Fujaba applications. For this purpose, we have shortly described the architecture of DRAGOS and its integration into Fujaba. With DRAGOS as underlying database, we have introduced possible extensions of Fujaba. This includes on the one hand extending the Fujaba modeling language, and on the other hand the generation of configurable graphical user interfaces for Fujaba applications.

DRAGOS can be used as an alternative to the existing CoObRA framework. The database offers more functionality than CoObRA, but still has problems in its runtime efficiency. Therefore, we are optimizing and improving the database. Additionally, we are investigating, how the presented language extensions can be integrated best into the Fujaba modeling language.

In addition to UPGRADE, we have also experimented with the re-engineering tool *RePLEX*, which is currently developed in the ECARES [5] project. The tool is implemented as a plug-in for the Eclipse IDE, using a hand-written GEF-based user interface and a Fujaba specification for the application logic. We generated code from the Fujaba specification using the modified code generation for DRAGOS. Furthermore, we modified the GEF edit parts so that change observers are no longer registered at the model objects directly, but at the DRAGOS event manager. This simplifies the management of these observers as already explained for the event management of UPGRADE. So, we provided RePLEX with a DRAGOS based persistency support with low implementation effort.

5. REFERENCES

[1] B. Böhlen. Specific graph models and their mappings to a common model. In J. L. Pfaltz, M. Nagl, and B. Böhlen, editors, *Applications of Graph Transformations with Industrial Relevance, 2^{nd} Intern. Workshop, AGTIVE 2003*, volume 3062 of *Lect. Notes in Comp. Sci.*, pages 45–60. Springer-Verlag, 2004.

[2] B. Böhlen, D. Jäger, A. Schleicher, and B. Westfechtel. UPGRADE: A framework for building graph-based interactive tools. In T. Mens, A. Schürr, and G. Taentzer, editors, *Graph-Based Tools (GraBaTs 2002)*, volume 72 of *Electr. Notes in Theor. Comp. Sci.* Elsevier Science, 2002.

[3] L. Geiger, C. Schneider, and C. Reckord. Template- and modelbased code generation for MDA-tools. In H. Giese and A. Zündorf, editors, *Proceedings of the Fujaba Days 2005*, volume tr-ri-05-259 of *Technical Report*. University of Paderborn, Germany, 2005.

[4] E. C. Maes. Development of a supporter for the mapping of graph models for GRAS/GXL. Master's thesis, University of Utrecht, May 2005. Nr.: INF/SCR-04-66.

[5] C. Mosler. E-CARES Project: Reengineering of Telecommunication Systems. In R. Lämmel, J. Saraiva, and J. Visser, editors, *Proceedings of the Summer School on Generative and Transformational Techniques in Software Engineering (GTTSE'05)*, number 4143 in LNCS, pages 437–448, Braga, Portugal, 2006. Springer-Verlag.

[6] C. Schneider, A. Zündorf, and J. Niere. CoObRA - a small step for development tools to collaborative environments. In J. Grundy, R. Welland, and H. Stoeckle, editors, *Proceedings of the Workshop on Directions in Software Engineering Environments, 26^{th} International Conference on Software Engineering*, 2004.

[7] A. Schürr. *Operationales Spezifizieren mit programmierten Graphersetzungssystemen*. Dissertation, RWTH Aachen, 1991.

[8] A. Winter. Exchanging graphs with GXL. In P. Mutzel, M. Jünger, and S. Leipert, editors, *Graph Drawing – 9^{th} International Symposium, GD 2001*, volume 2265 of *Lect. Notes in Comp. Sci.*, pages 485–500. Springer-Verlag, 2001.

Specification of Triple Graph Schemas within the Fujaba Tool Suite

Alexander Königs, Andy Schürr
Real-Time Systems Lab
Darmstadt, University of Technology
Germany
[koenigs|schuerr]@es.tu-darmstadt.de

ABSTRACT

In the context of OMG's Model driven application development (MDA®) OMG's Query/View/Transformation request for proposal (QVT-RFP) aims at the definition of languages that allow to query models, to build views, and to perform transformations on them. Triple graph grammars (TGG) as a related approach address these issues based on the formalism of graph transformations. Besides a set of declarative rules a TGG needs – as regular graph grammars do – a schema that declares the correspondences between two (meta-)models that are to be integrated. Up to now, most implementations of TGGs spend too little effort in the definition of a proper TGG schema. In this paper we elicit requirements for a dedicated TGG schema editor. Furthermore, we present our implementation of such an editor as a plug-in for the Fujaba Tool Suite. As a result we come up with an editor that enables specifiers of TGGs to specify expressive schemas in a convenient way and, moreover, support the specification of TGG rules.

1. INTRODUCTION

On an abstract level OMG's idea of Model Application Development (MDA®) [5] is about model transformation and integration in order to improve software and system development. In its request for proposals for a Query, View, and Transformation (QVT-RFP)[7] approach the OMG asks for languages that allow for the specification of such model transformations and integrations. Recently, the OMG has adopted one submission to its QVT-RFP as a standard [8]. Independently from OMG's efforts there already exists the approach of triple graph grammars (TGG) that has been invented about ten years ago by [9]. Rather than the QVT standard TGGs are founded on a well-defined formalism, i.e. graph transformations. Nowadays TGGs can be seen as an implementation of QVT. Unfortunately, implementations of TGGs still are only available as prototypes aiming at different application domains. Furthermore, most TGG approaches focus on the specification of declarative rules and strategies how to automatically derive different sets of operational rules from them. Thereby, these approaches usually disregard the schema of a TGG. A TGG schema declares the types of correspondence links which link corresponding objects of the models that are to be integrated. Additionally, a TGG schema provides a lot of additional information (i.e. constraints) beyond the scope of TGG rules. Therefore, in this paper we focus on requirements for and the implementation of a proper TGG schema editor.

Section 1 introduces a running example that is used throughout the remainder of this paper and motivates our efforts. We compare current TGG approaches with each other and the QVT standard and discuss their shortcomings with respect to their TGG schema support in Section 3. In Section 4 we derive a set of requirements that a TGG schema editor should meet and present our own approach. Our implementation as a plug-in for the Fujaba Tool Suite is described in Section 5. Finally, Section 6 concludes this paper and discusses open issues.

2. RUNNING EXAMPLE

As a running example we aim at specifying the correspondence between the (meta-)models of the requirements tool DOORS and the UML 2.0 modeling tool Enterprise Architect. In particular, we want to integrate a textual use case specification kept in DOORS (c.f. Fig. 1a) with a use case diagram from Enterprise Architect (c.f. Fig. 1b) using TGGs. In order to access the tools' data we rely on tool adapters that adapt the proprietary tool APIs to uniform and standard compliant JMI interfaces. These interfaces reflect the metamodels of DOORS (c.f. Fig. 2a) and Enterprise Architect (c.f. Fig. 2b), respectively. Observe that Enterprise Architect internally does not use the UML 2.0 metamodel but a proprietary metamodel instead. Nevertheless, Enterprise Architect provides the concrete syntax of UML 2.0.

When integrating the (meta-)models of DOORS and Enterprise Architect we want to integrate `FormalModules` from DOORS with `EAPackages` from Enterprise Architect. Furthermore, we want to integrate `FormalObjects` that represent use cases in DOORS with `EAElements` that represent use cases in Enterprise Architect. Finally, we integrate `FormalObjects` that represent include dependencies in DOORS with `EAConnectors` that represent include dependencies in Enterprise Architect.

3. RELATED WORK

The QVT standard [8] declaratively specifies the correspondence between two models based on relations (i.e. correspondence link types) which can be denoted textually or graphically. Compared to TGGs the specification of a relation in QVT looks similar to a declarative TGG rule. Although a set of relations as well as a set of TGG rules implies an underlying schema the QVT standard does not have any means for explicitly specifying a schema. Thus, the QVT standard does not provide any means for constraints which

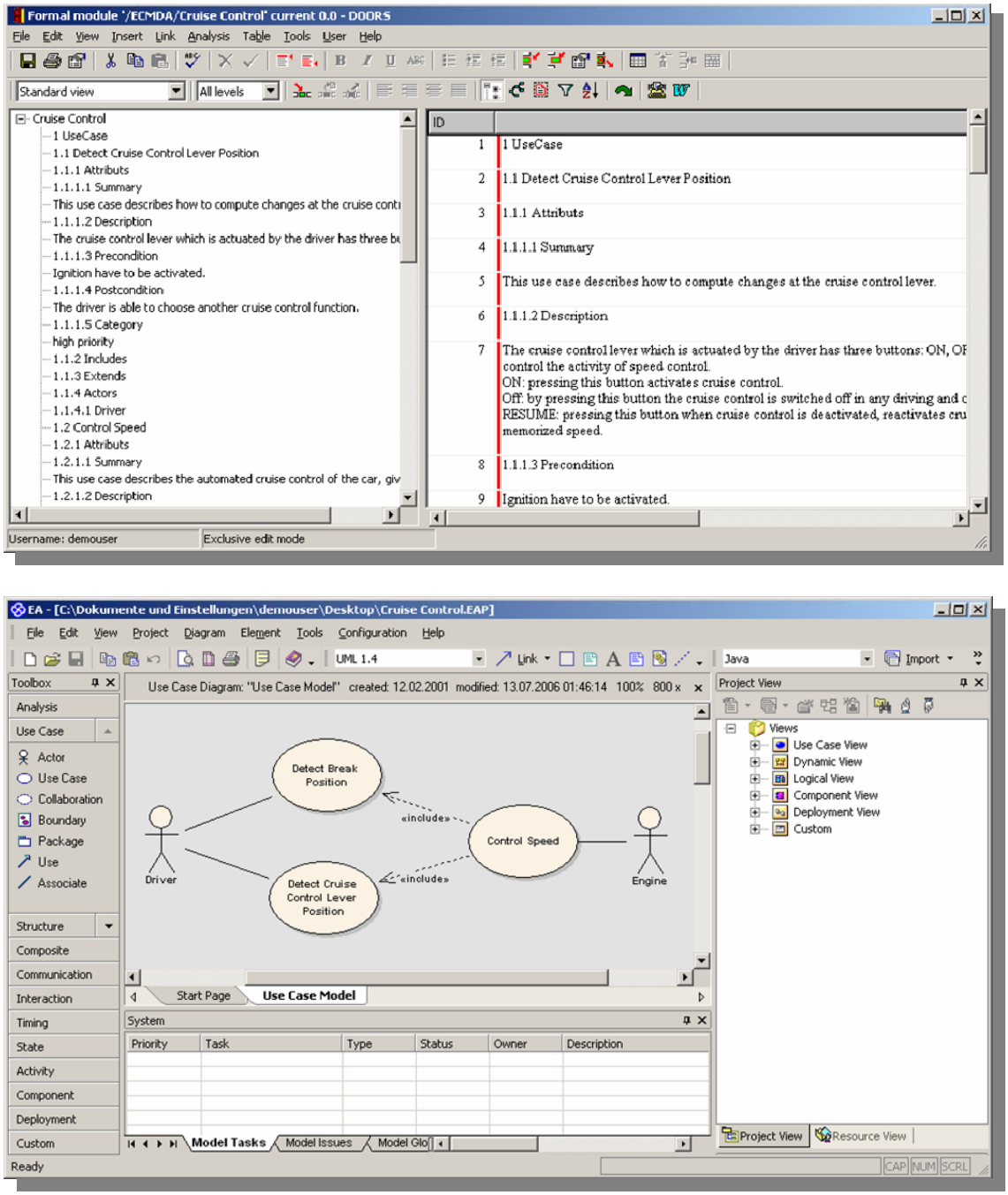


Figure 1: Screenshots of *a.* DOORS and *b.* Enterprise Architect

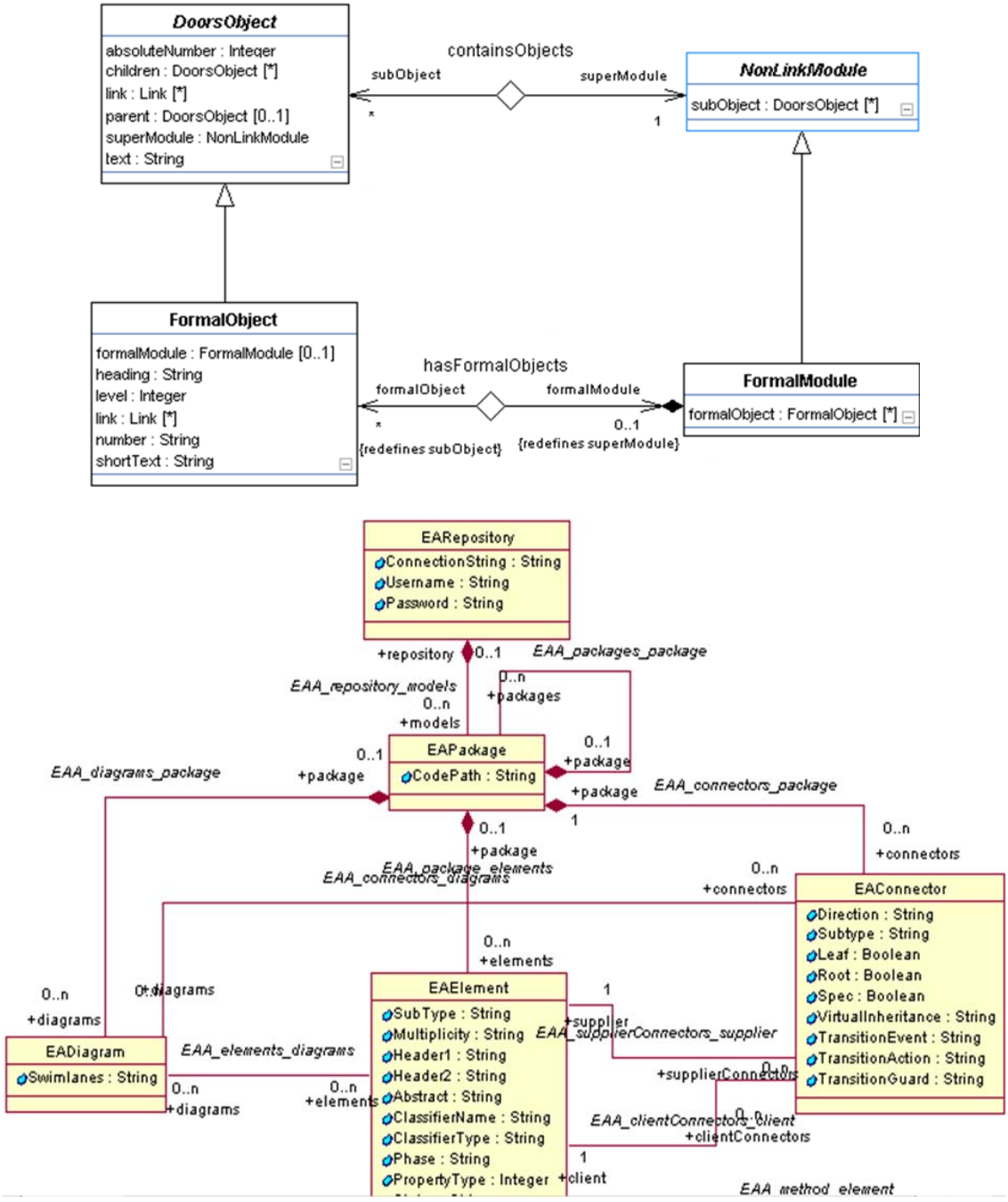


Figure 2: Parts of the metamodels of *a.* DOORS and *b.* Enterprise Architect

	QVT	Fujaba	PROGRES	Atom^3	MOFLON
Explicit TGG schema	-	+	+	+	+
Modularization of TGG schema	-	-	-	+	+
Distinct submetamodels	+	-	+	+	+
Corr. links instead of link objects	-	-	-	-	+
Attributes for corr. links	-	+	+	+	+
Multiplicities for corr. links	-	+	+	+	+
Constraints for corr. links	-	+	+	+	+
Inheritance of corr. links	+	+	+	+	-
Parametrized corr. links	-	-	-	-	+
Integration of corr. link types and TGG rules	+	-	-	+	+

Figure 3: Comparison of related approaches

should be specified at schema level (e.g. multiplicities). These constraints would be useful for consistency analysis on corresponding models. Furthermore, the QVT standard does neither support attributed correspondence links nor modularization of relations. Nevertheless, the QVT standard allows for the refinement of relations which can be compared to inheritance at the implicitly given schema level. The QVT standard assumes that there is exactly one relation for each implicitly given correspondence link type.

Currently, there is already an implementation of TGGs in Fujaba which is based on the TGG rule editor presented in [3]. This TGG rule editor already relies on a TGG schema which must be specified beforehand. The TGG schema is regarded as one single Fujaba-UML class diagram that declares all types of the metamodels that are to be integrated as well as the types of correspondence links. At schema level the distinction between classes of one metamodel and classes of the other metamodel as well as classes of the correspondence metamodel cannot be expressed. Therefore, at rule level it is necessary to decide for each object whether it belongs to the source, to the target, or to the correspondence metamodel. This is annoying since this information should already be available at schema level[1] and is immutable throughout the whole TGG specification. Nevertheless, for readability purposes it is nice to have the annotation attached to the objects to which metamodel it belongs. Furthermore, an object whose type is considered to belong to the correspondence metamodel for instance can be provided with a false annotation expressing that this object belongs to the source side for instance. Moreover, the specification of correspondence of two classes at schema level and two objects at rule level is realized by an *Association, Class, Association* construct and a *Link, Object, Link* construct. It would be more convenient to have a single *Association Class* construct. Finally, since the whole TGG schema consists of a single metamodel it is impossible to develop the source, the target, and the correspondence metamodel independently from each other. Nevertheless, the current Fujaba approach in contrast to our own approach presented in this paper does not assume that there is exactly one rule for each correspondence link type.

[1]This is not true in cases where the source and the target metamodel coincide. Nevertheless, objects belonging to the correspondence metamodel still cannot be on the source or target side.

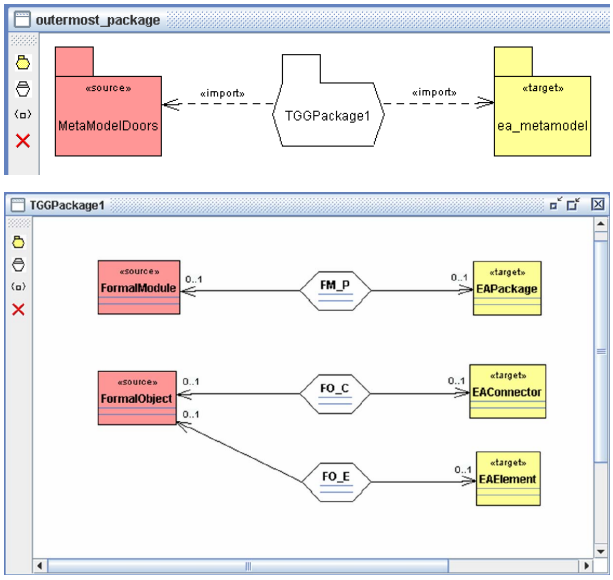


Figure 4: TGG schema: *a.* package dependencies and *b.* class correspondences

[6] presents a concept for TGG schemas in PROGRES. It proposes to integrate two separate PROGRES specifications by textually denoting the correspondences resulting in a TGG schema with distinct subschemas. The presented approach does not support attributed correspondence links, has no means for refinement at rule level, and supports no modularization at schema level. This approach has been implemented in [2].

The Atom^3 approach as presented in [4] meets our requirements of a triple graph schema very well. Atom^3 composes a triple graph schema from three distinct subschemas, allows for attributed link types, supports inheritance of correspondence link types, and allows for the definition of multiplicities and constraints at schema level. As the current Fujaba approach does Atom^3 uses an *Association, Class, Association* construct for specifying correspondence link types instead of an *Association Class* construct.

Finally, no current approach allows for the parametrization of integration rules and hence the correspondence link types. Fig. 3 summarizes our comparison. Thereby, the last column presents the properties of our own approach as part of the MOFLON framework [1].

4. APPROACH

According to the lessons learned from Section 3 we demand that a TGG schema editor meets the following requirements:

- There should be an explicit TGG schema.
- The metamodels that are to be integrated as well as the correspondence metamodel should be kept distinct.

- There should be means to modularize a TGG schema.
- It should be possible to attach multiplicities and further constraints to the correspondence link types at schema level.
- Link types should be enabled to have attributes.
- Link types should be declared as *Association Classes* rather than *Association, Class, Association* constructs.
- Inheritance of link types should be supported. It is ongoing research to verify whether this only allows for type compliance purposes or allows for the inheritance, redefinition, and extension of attached integration rules as well.
- It should be possible to parametrize correspondence link types.
- For compliance to the QVT standard each link type should have at most one declarative TGG rule attached to it.

According to these requirements we have implemented a TGG schema editor as a plug-in for the Fujaba Tool Suite. In order to specify the TGG schema and to declare the correspondence of two metamodels that are to be integrated our editor firstly asks for references to two different Fujaba projects. These Fujaba projects may be created manually or be imported from other CASE tools as Rational Rose for instance. After that the user can add TGG packages to the TGG schema and arbitrarily nest them for modularization purposes. A TGG package hierarchy need not to correspond to any package hierarchy from the metamodels that are to be integrated. Each TGG package must be provided with a package from the source metamodel and a package from the target metamodel (c.f. Fig. 4a). Thereby, the user gets access to the classes that are declared in the referenced packages. In each TGG package the user now can specify the correspondence between the accessible classes (c.f. Fig. 4b). The correspondences of two classes are expressed by *Association Class* constructs instead of *Association, Class, Association* constructs. That means that the user can specify correspondences by creating only one schema element instead of three. Furthermore, each correspondence link type may be provided with attributes like *created by whom, created when, last checked on.* Moreover, each link type carries multiplicity constraints. These constraints can be utilized for analysis purposes and express how many elements of one metamodel have to correspond to one element of the other metamodel and vice versa. Each link type may be provided with additional constraints that can be checked during model analysis. Currently, our TGG schema editor does not support inheritance on link types but this feature is planned for the next version. To this end, we have to investigate whether inheritance on link types can only be used for type compliance purposes or allow for the inheritance, redefinition, and extension of attached TGG rules as well. Finally, our TGG schema editor allows the creation of one TGG rule for each correspondence type by invoking Fujaba's existing TGG rule editor.

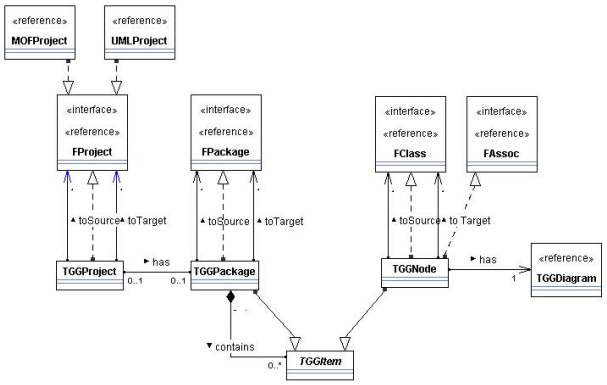


Figure 5: Simplified part of the TGG schema editor's metamodel

5. IMPLEMENTATION

We have implemented our approach we presented in the preceding section as a plug-in for the Fujaba Tool Suite version 5. Therefore, we started by modeling the metamodel (c.f. Fig. 5) of the editor by using the Fujaba Tool Suite version 4. From our metamodel specification we automatically generated Java code using Fujaba's code generation engine[2].

Among others our metamodel introduces a new project type `TGGProject` that satisfies Fujaba's `FProject` interface. The *F-Interfaces* have been added to Fujaba in version 5 to allow for abstraction from a concrete modeling language as Fujaba-UML or MOF 2.0 which have been realized as implementations of these interfaces. This project type keeps references to two `FProjects` that contains the metamodels that are to be integrated. Therefore, our TGG schema editor may integrate models defined with any Fujaba (meta-) modeling plug-in that implements the `FProject` interface, e.g. `UMLProject` or `MOFProject`. Since our TGG schema editor regards package structures we are actually limited to `MOFProjects` since `UMLProjects` does not provide a proper package structure. We regard this as a defect of `UMLProject` rather than a defect of our editor

For compliance with the Coobra plug-in which is responsible for versioning issues the metamodels that are to be integrated are added to the list of required projects of a `TGGProject`. Therefore, a `TGGProject` specification cannot be loaded until both required projects are available.

A `TGGProject` maintains a hierarchy of `TGGPackages` that allow for the modularization of the TGG schema specification. Since each `TGGPackage` satifies the `FPackage` rather than the `MOFPackage` interface we have limited means for package dependencies, i.e. there is no package merge available as in MOF 2.0. Each `TGGPackage` has two references to `FPackages` from the referenced metamodels (c.f. Fig. 4a). That means that inside a given `TGGPackage` the classes from the referenced `FPackages` are visible and can be referenced, too. Furthermore, each `TGGPackage` may contain additional

[2]Although Fujaba claims to be bootstrapped we had to do minor adjustments to the code generation mechanism in order to generate Java code that meets Fujaba's own interface specification.

`TGGPackages`.

Moreover, each `TGGPackage` may contain `TGGNodes` that declare the correspondence between two referenced `FClasses` (c.f. Fig. 4b). Each `TGGNode` satisfies the `FClass` interface as well as the `FAssoc` interface realizing an association class. That means that the correspondence of two `FClasses` can be expressed by one metamodel element rather than three, i.e. association, class, association. Furthermore, a `TGGNode` can have attributes. Finally, multiplicity constraints as well as other schema constraints can be attached to each `TGGNode`. In the next version of our TGG editor we would like to support inheritance of `TGGNodes` as well.

We integrated the existing TGG rule editor with our TGG schema editor by assigning exactly one TGG rule represented by the `TGGDiagram` type to each `TGGNode`. The drawback is that our TGG schema editor from now on depends on the TGG rule editor. Since the TGG rule editor should regard information from the TGG schema editor as well we end up in a cyclic dependency forcing us to merge both separate plug-ins into one. We believe that this is preferable rather than a real drawback.

6. CONCLUSION

Motivated by OMG's efforts in MDA and its QVT-RFP in this paper we have analyzed current implementations of TGGs with respect to their TGG schema support. We have identified shortcomings and have come up with a set of requirements a TGG schema editor should meet in order to provide useful information for the underlying TGG rules. Finally, we have introduced our own approach and described how we realized it as plug-in for the upcoming version of the Fujaba Tool Suite.

In the next version of our editor we will support the missing inheritance of correspondence link types. Furthermore, we have to investigate whether it is preferable to realize our TGG schema editor on top of the generic Fujaba interfaces rather than more specific interfaces (e.g. MOF interfaces). The realization on top of the generic interfaces provides more flexibility at the price of more specific features as MOF's package merge for instance.

Additionally, it is ongoing research to investigate how to integrate a TGG schema specification with current metamodeling standards like MOF. On the one hand we have the option to extend a common standard like MOF. This might sound peculiar and undesirable but has been done by the QVT standard as well. On the other hand we could try to automatically translate a TGG specification into a standard compliant specification similar to the translation of declarative TGG rules into operational graph rewriting rules. We believe that the latter option is preferable. Actually, we have already implemented such a translation which is out of scope for this paper.

7. REFERENCES

[1] C. Amelunxen, A. Königs, T. Rötschke, and A. Schürr. MOFLON: A Standard-Compliant Metamodeling Framework with Graph Transformations. In A. Rensink and J. Warmer, editors, *Model Driven Architecture - Foundations and Applications: Second European Conference*, volume 4066 of *Lecture Notes in Computer Science (LNCS)*, pages 361–375, Heidelberg, 2006. Springer Verlag, Springer Verlag.

[2] S. M. Becker, T. Haase, and B. Westfechtel. Model-Based A-Posteriori Integration of Engineering Tools for Incremental Development Processes. *Journal of Software and Systems Modeling*, 4(2):123–140, 2005.

[3] S. Burmester, H. Giese, J. Niere, M. Tichy, J. P. Wadsack, R. Wagner, L. Wendehals, and A. Zündorf. Tool Integration at the Meta-Model Level within the FUJABA Tool Suite. *International Journal on Software Tools for Technology Transfer (STTT)*, 6(3):203–218, August 2004.

[4] E. Guerra and J. de Lara. Attributed Typed Triple Graph Transformation with Inheritance in the Double Pushout Approach. Technical Report UC3M-TR-CS-2006-00, Universidad Carlos III., 2006.

[5] A. Kleppe, J. Warmer, and W. Bast. *MDA Explained. The Model Driven Architecture: Practice and Promise.* Addison Wesley, 2003.

[6] Lefering and Schürr. Specification of Integration Tools. In Nagl, editor, *Building Tightly-Integrated Software Development Environments: The IPSEN approach*, volume 1170 of *LNCS*, pages 440–456. Springer Verlag, 1996.

[7] OMG. Request for Proposal: MOF 2.0 Query / Views / Transformations RFP. http://www.omg.org/docs/ad/02-04-10.pdf, 2002.

[8] OMG. Meta Object Facility (MOF) 2.0 Query / View / Transformation Specification. http://www.omg.org/docs/ptc/05-11-01.pdf, 2005.

[9] Schürr. Specification of graph translators with triple graph grammars. In Mayr and Schmidt, editors, *Proc. WG'94 Workshop on Graph-Theoretic Concepts in Computer Science*, pages 151–163, 1994.

Developing Model Transformations with Fujaba

Robert Wagner
Software Engineering Group
Department of Computer Science
University of Paderborn
Warburger Str. 100
D-33098 Paderborn, Germany
wagner@uni-paderborn.de

ABSTRACT

In this paper, we present Fujaba's tool support for the specification and execution of model transformations based on the bidirectional transformation technique of triple graph grammars. The graphical though formal specification is executed incrementally and enables live model synchronization that helps to propagate changes between related models.

1. INTRODUCTION

Today's software engineering tools offer different formal and semi-formal notations for the construction of models. In addition, most tools support some kind of model-to-model transformation technology. Up to now, desired support for model transformations is missing in Fujaba and has to be realized by handcrafted code which is a quite annoying and error prone task.

The requirements and expectations regarding model transformation techniques are very demanding. A practicable solution should allow a visual specification of the transformation rules [4] with an underlying formal foundation.

In practice, the development of a software system is a quite iterative process with frequent modifications of the involved models. For example, after a model transformation often the target model has to be refined towards the final design. These modifications can have some impact on the source model and have to be propagated backward to keep the overall specification consistent. Therefore, a bidirectional model transformation technique is needed.

Moreover, modifications in the source model have to be propagated to the target model in a non-destructive manner. If a model transformation is applied from scratch, the modifications on a more detailed target model are lost. Therefore, we need an incremental transformation technique which is able to synchronize different models.

In order to meet these requirements we employ the visual, formal, and bidirectional transformation technique of triple graph grammars [9]. In this paper, we focus on the provided tool support for the development and execution of model transformations which was realized as a set of plug-ins for the Fujaba Tool Suite[1]. For a detailed description of our incremental algorithm and its evaluation we refer to [3].

The remainder of this paper is organized as follows. In the next section we introduce a simple example which will be used throughout this paper in order to explain the realized tool support. In Section 3, we give a brief and informal introduction to the concepts of triple graph grammars. Section 4 sketches our incremental transformation algorithm. The realized tool support with necessary steps for the development of a model transformation is described in Section 5. The paper closes with a conclusion and an outlook on future work in Section 6.

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

2. EXAMPLE

In order to explain the specification and execution of a model transformation, we introduce a simple example which will be used throughout this paper. The example stems from an earlier project in the domain of production control system. In this project, we combined a subset of the Specification and Description Language (SDL) [5] and the Unified Modeling Language (UML) [7] to an executable graphical language [8]. In this language, a block diagram is used to specify the overall static communication structure where processes and blocks are connected to each other by channels and signal routes. For implementation purposes, the block diagram is transformed to an initial class diagram. This class diagram can be refined and extended, e.g. using state charts, to an executable specification. Figure 1 shows a simple block diagram and the class diagram which results from a correct transformation.

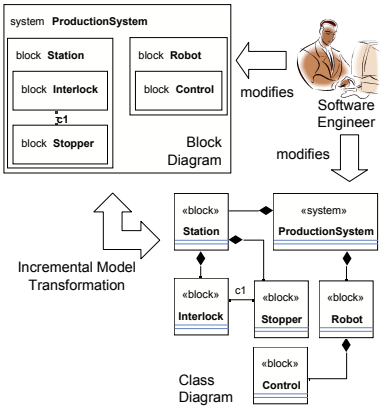


Figure 1: Application example

Basically, systems, blocks, and processes of a block diagram are transformed to classes with corresponding stereotypes. For example, the block *Station* is represented by the class *Station* with a stereotype ≪block≫, the system *Pro-*

ductionSystem as a class with the stereotype ≪system≫. The hierarchical structure of the block diagram is expressed by composition relations between the respective classes in the class diagram. The channels and signal routes of the block diagram are mapped to associations between the derived classes. In addition, each signal received by a process in the block diagram is mapped to a method of the corresponding class in the class diagram (not shown in this example).

In order to support an iterative development process without any restrictions on the order of design steps, we allow the engineer to move freely between the block and class diagram to refine and adapt both models towards the final design. In this scenario, we have to ensure that the overlapping parts and mappings between these interrelated models stay consistent to each other. For this purpose, we use the bidirectional and incremental model transformation technique of triple graph grammars. This declarative specification technique is described in the following section.

3. TRIPLE GRAPH GRAMMARS

A triple graph grammar [9] is a declarative definition of a bidirectional model transformation and consists of a graphical though formal specification of transformation rules. In Figure 2, a triple graph grammar rule in the FUJABA-notation is shown.

The rule specifies a consistent correspondence mapping between the objects of the source and target model. In particular, the presented rule defines a mapping between a block and a corresponding class. The objects of the block diagram are drawn on the left and the objects of the class diagram are drawn on the right. They are marked with the ≪left≫ and ≪right≫ stereotypes respectively. The correspondence objects in the middle of the rule are tagged with the ≪map≫ stereotype.

The rule is separated into a triple of productions (source production, correspondence production, and target production), where each production is regarded as a context-sensitive graph grammar rule. A graph grammar rule consists - as all grammars - of a left-hand side and a right-hand side. All objects which are not marked with the ≪create≫ stereotype belong to the left-hand side and to the right-hand side; the objects which are tagged with the ≪create≫ stereotype occur on the right-hand side only. In fact, these tags make up a production in FUJABA's graph grammar notation.

The production on the left shows the generation of a new sub block and linking it to an existing parent block. The production on the right shows the addition of a new class and stereotype and its linking to the class diagram. Moreover, to reflect the containment of the sub block, a composition association is created between the classes representing the parent block and the sub block. For this purpose, the rule contains additional objects representing the roles and cardinalities of the association. The correspondence production shows the relations between a block and a class and an additional constraint $\{block.name == clazz.name\}$ specifies that the block and the class have to be named uniquely.

Up to this point, the assignments and constraints to the object attributes have not been considered yet. Since triple graph grammars can be executed in both directions, the attribute constraints help to identify the objects to be matched, whereas the attribute assignments are applied only to created objects. However, since the computations of the attribute values may be more complicated than in our simple example, the assignments cannot be always derived from the constraints and have to be specified explicitly.

A graph grammar rule is applied by substituting the left-hand side with the right-hand side if the pattern of the left-hand side can be matched to a graph, i.e., if the left-hand side is matched all objects tagged with the ≪create≫ stereotype will be created. Hence, our example rule, in combination with additional rules covering other diagram elements, can generate a set of blocks along with the corresponding classes and associations in a class diagram. Though the transformation will not be executed this way, conceptually, we can assume that whenever a block is added to the block diagram, a corresponding class with an appropriate association will be generated in the class diagram. This way, the triple graph grammar rules define a transformation between block diagrams and class diagrams.

The correspondence production in the middle of the rule enables a clear distinction between the source and target model and holds additional traceability information. This information can be used to realize bidirectional and incremental model transformations that helps to propagate changes between related models, i.e., it helps to realize live model synchronization between different but related models.

4. TRANSFORMATION ALGORITHM

Before we explain our incremental transformation algorithm we have to take a closer look at the correspondence metamodel shown in Figure 3. The metamodel defines the mapping between a source and a target metamodel by the classes *TGGNode* and *Object* and its associations *sources* and *targets*. Since all classes inherit implicitly from the *Object* class (not shown here), the correspondence model stores the traceability information needed to preserve the consistency between two models. In addition, the class *TGGNode* has a self-association *next* which connects the correspondence nodes with their successor correspondence nodes. This extra link is used by our transformation algorithm.

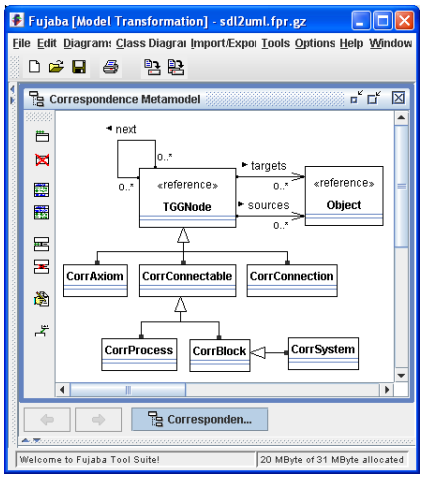


Figure 3: Correspondence metamodel

The two described classes and their associations are essential for our transformation algorithm. However, further correspondence nodes and refined associations can be added. In our example, we have added six additional correspondence nodes, including the correspondence node *CorrBlock* used in

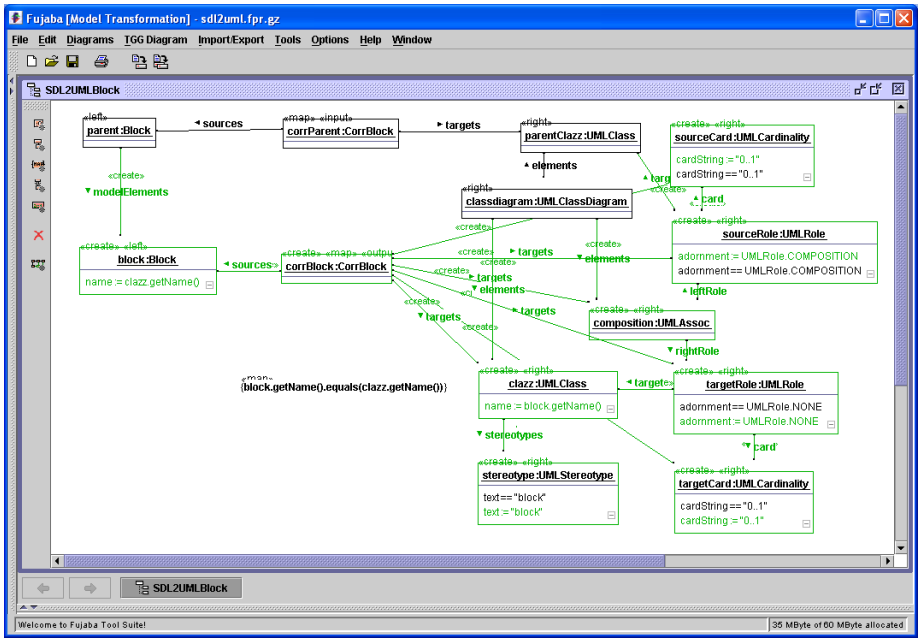


Figure 2: A triple graph grammar rule

our example rule (cf. Figure 2). The additional correspondence classes increase the performance of our transformation algorithm since for a given correspondence node type only those rules have to be checked that have the same correspondence node type on their left-hand side.

The transformation algorithm exploits the extra information from the correspondence metamodel for the incremental model synchronization. In Figure 4 an overview of the realized algorithm is shown. The incremental transformation and update algorithm traverses the correspondence nodes along the *next* links of the *TGGNode* objects using breadth-first search. First, for each correspondence node the algorithm checks whether an inconsistent situation has occurred. This is done by retrieving an applied rule (find applied rule) and checking whether it still matches to the pattern structure (check pattern structure).

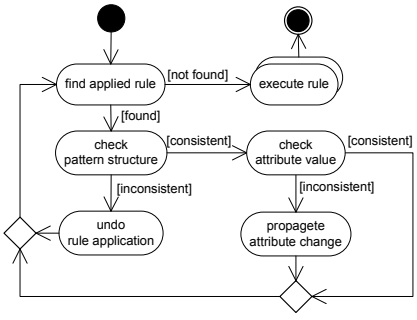


Figure 4: Transformation and update algorithm

If the rule cannot be matched anymore, e.g., due to the deletion of a model element, we have found indeed an inconsistency. In that case, the algorithm has to undo the applied transformation rule (undo rule application). This is achieved by deleting the correspondence node and all created elements. Note that by deleting the correspondence node the precondition for all successors of the deleted correspondence node will not hold anymore. As a consequence, this leads both to the deletion of the succeeding correspondence nodes and the elements in the class diagram referenced by the deleted correspondence nodes.

In the case that the structure of the applied rule still holds and only an attribute constraint evaluates to false (check attribute value), it is sufficient to propagate the attribute value change in the current transformation direction (propagate attribute change).

If all old rule applications have been checked, the algorithm searches for new model elements and transforms those elements according to the triple graph grammar specification (execute rule). Note that the transformation is executed as long as rules are applicable which is indicated by a for-each activity (execute rule).

The briefly described algorithm was implemented as a plug-in for FUJABA and is an essential part of the realized tool support which is described in the following section.

5. TOOL SUPPORT

For the visual specification of a triple graph grammar rule we use the TGGEDITOR (cf. Figure 2). This editor is a realized as a FUJABA plug-in and ensures conformance to the source, the correspondence, and the target metamodels. For this purpose, the required metamodels have to be specified in FUJABA as class diagrams.

The execution of a model transformation is done by the new FUJABA plug-ins MOTE and MORTEN. MOTE is the abbreviation for Model Transformation Engine. It is the core library for the execution of triple graph grammars and can be also used without FUJABA. MORTEN is the abbreviation for Model Round Trip Engineering. MORTEN integrates the MOTE library into FUJABA and provides a graphical user interface to setup and control several transformation tasks.

In order to execute the algorithm presented in Section 4, we derive from each rule story diagrams implementing the activities shown in Figure 4. From the automatically derived story diagrams, we generate Java code using FUJABA's code generation facilities. This code is compiled to executable transformation rules which are bundled into a single JAR archive file. In addition, the JAR file has to provide a configuration file with listed transformation rules. The archive represents the catalog of transformation rules defining the model transformation specified by a triple graph grammar. Once the catalog is available, model transformations can be carried out.

The first step to transform our block diagram is to setup an appropriate transformation task. For this purpose, we have to name the transformation task and select the catalog containing the rules for the transformation. Thereafter, we have to select the source and/or the target diagram for the transformation. For example, we can select a block diagram and transform it initially to a class diagram. Or, we can select a class diagram and transform it backward to a block diagram. If we select two diagrams, i.e., a class and a block diagram, both diagrams are checked for corresponding parts and the related parts are connected by correspondence nodes. After an initial transformation, the block and class diagram can be modified and further refined. For synchronization purposes, the incremental transformation can be re-executed each time a diagram changes (cf. Figure 5).

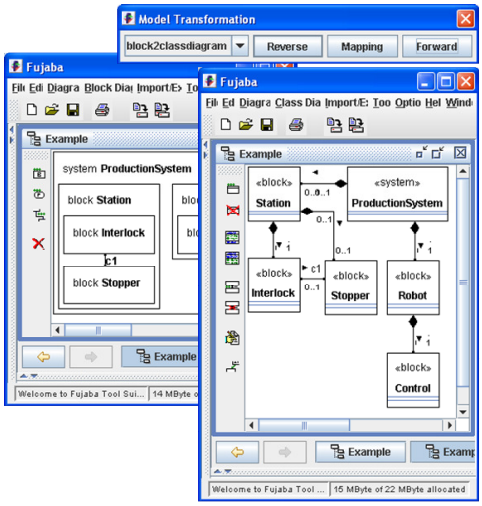


Figure 5: Block to class diagram transformation

MORTEN is intended to be used to test a specified model transformation during development. After a transformation is specified and tested, it can be integrated into any Java-based software tool with FUJABA-compliant metamodels or appropriate model adapters using the MOTE library. An example for such an integration is presented in [2] where Matlab/Simulink models are automatically transformed to pattern specifications by utilizing the MOTE plug-in.

6. CONCLUSION AND FUTURE WORK

In this paper, we have presented the realized tool support for the model-based development of model transformation rules based on the technique of triple graph grammars. The available tool support includes a FUJABA plug-in for the visual specification of triple graph grammar rules, the automatic derivation of graph rewriting rules in the form of story diagrams, and a plug-in for the incremental execution of these rules.

We are currently working on an approach which eases the specification of the triple graph grammar rules and enables the specification of rules in the concrete syntax of the appropriate models. At the same time, we are extending the model synchronization support by some auto-completion capabilities to allow a synchronization between already existing models.

As future work we plan to provide a QVT [6] compatible front-end in order to make the technology available to a broader audience. In addition, we plan to port our tools to the Eclipse platform to enable the transformation of EMF-compliant models [1].

7. REFERENCES

[1] The Eclipse Foundation. *Eclipse Modeling Framework (EMF), available at http://www.eclipse.org/emf*, 2006.

[2] H. Giese, M. Meyer, and R. Wagner. A Prototype for Guideline Checking and Model Transformation in Matlab/Simulink. In H. Giese and B. Westfechtel, editors, *Proc. of the 4th International Fujaba Days 2006, Bayreuth, Germany*, volume tr-ri-06-275 of *Technical Report*. University of Paderborn, September 2006.

[3] 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), Genoa, Italy*, volume 4199 of *Lecture Notes in Computer Science*, pages 543–557. Springer Verlag, October 2006.

[4] L. Grunske, L. Geiger, and M. Lawley. A graphical specification of model transformations with triple graph grammars. In *2nd European Conference on Model Driven Architecture - Foundations and Applications (ECMDA-FA)*, 2005.

[5] International Telecommunication Union (ITU), Geneva. *ITU-T Recommendation Z.100: Specification and Description Language (SDL)*, 1994 + Addendum 1996.

[6] OMG. *MOF QVT Final Adopted Specification, OMG Document ptc/05-11-01*. http://www.omg.org/.

[7] OMG, 250 First Avenue, Needham, MA 02494, USA. *Unified Modeling Language Specification Version 1.5*.

[8] W. Schäfer, R. Wagner, J. Gausemeier, and R. Eckes. An Engineer's Workstation to support Integrated Development of Flexible Production Control Systems. In H. Ehrig, W. Damm, J. Desel, M. Große-Rhode, W. Reif, E. Schnieder, and E. Westkämper, editors, *Integration of Software Specification Techniques for Applications in Engineering*, volume 3147 of *Lecture Notes in Computer Science*. Springer Verlag, September 2004.

[9] A. Schürr. Specification of graph translators with triple graph grammars. In E. W. Mayr, G. Schmidt, and G. Tinhofer, editors, *Graph-Theoretic Concepts in Computer Science, 20th International Workshop, WG '94*, volume 903 of *LNCS*, pages 151–163, Herrsching, Germany, June 1994.