Dirk Fahland Daniel A. Sadilek
Markus Scheidgen Stephan Weißleder (Eds.)

# DSML'08
# Domain-Specific Modeling Languages

**Workshop co-located with Modellierung 2008**
**Berlin, Germany, March 14, 2008**
**Proceedings**

*Editors' addresses:*
Humboldt-Universität zu Berlin
Department of Computer Science
Unter den Linden 6
10099 Berlin, Germany

{fahland | sadilek | scheidge | weissled}@informatik.hu-berlin.de

## Preface

In recent years, computer science produced a multitude of modeling languages. They are used to describe, analyze, and understand systems, components, problems, and solution algorithms. Systems in domains distant from computer science base upon concepts that are substantially different from the implementation technology. The main problem is to relate domain-specific concepts and technical details in a sound and conceivable manner.

Domain-specific modeling languages (DSMLs) are considered as one solution for this problem. They provide all domain-relevant concepts, which are based on adequate formalisms and models. Supported by corresponding tools, domain experts are able to create, analyze, and refine models with knowledge from their domain. By creating such tools and languages, experienced modelers enable domain experts to unambiguously express domain-specific concepts on a formal basis.

In the field of model-driven development (MDD), the design of a DSML is seen as a modeling task: The DSML is described by a language model written in a formal language-modeling language; Tools support the development of domain-specific tools as well as the enhancements of language, tools, and models.

The workshop Domain-Specific Modeling Languages (DSML'08) co-located with the Modellierung 2008 provided a forum for researchers and practitioners. We received eight submissions out of which we selected five for inclusion in the digital and printed proceedings. The accepted papers span from pure research to experience reports.

Garcia presents an architecture for synchronizing multiple views of software models, which allows for using DSMLs in a multi-view design environment. In this architecture, bidirectional transformations between an integrated model and multiple views are described declaratively.

Sadilek and Weißleder argue for systematic, automated testing of metamodels, which define the abstract syntax of DSMLs. The authors present a method for testing metamodels that is based on giving exemplary metamodel instances.

Wimmer, Schauerhuber, Strommer, Flandorfer, and Kappel present a model browser implemented with Web 2.0 technologies. Ajax technology is used to present ECore metamodels graphically within regular browsers.

Pilarski and Knauss analyze the possibilities to combine the advantages of graphical and tabular representations of use-cases. The paper presents common concepts between the two representations and transformations between them.

Cramer, Klassen, and Kastens describe their experiences with designing a DSML for controlling robots in industrial production. By emphasizing the evaluation stage of the DSML, they demonstrate how a DSML can iteratively be developed while involving its later users.

We thank the authors for their submissions and the program committee for their hard work.

February 2008

Dirk Fahland, Daniel A. Sadilek,
Markus Scheidgen, Stephan Weißleder

## Organizing Committee

Dirk Fahland, HU Berlin
Joachim Fischer, HU Berlin
Daniel A. Sadilek, HU Berlin
Markus Scheidgen, HU Berlin
Bernd-Holger Schlingloff, Fraunhofer FIRST
Michael Soden, IKV Technologies AG
Stephan Weißleder, HU Berlin

## Program Committee

Hajo Eichler, IKV Technologies AG
Gregor Engels, Universität Paderborn
Dirk Fahland, HU Berlin
Joachim Fischer, HU Berlin
Kathrin Kaschner, Universität Rostock
Dagmar Koss, TU München
Niels Lohmann, Universität Rostock
Roman Nagy, microTOOL GmbH
Birgit Penzenstadler, TU München
Daniel Sadilek, HU Berlin
Markus Scheidgen, HU Berlin
Bernd-Holger Schlingloff, Fraunhofer FIRST
Sebastian Schuster, Universität Karlsruhe
Michael Soden, IKV Technologies AG
Dehla Sokenou, GEBIT Solutions GmbH
Stephan Weißleder, HU Berlin
Karsten Wolf, Universität Rostock
Justyna Zander-Nowicka, Fraunhofer FOKUS

## Contents

# Bidirectional Synchronization of Multiple Views of Software Models

Miguel Garcia

Institute for Software Systems (STS)
Hamburg University of Technology (TUHH), 21073 Hamburg
http://www.sts.tu-harburg.de/~mi.garcia

**Abstract:** Current best-practices for defining Domain-Specific Modeling Languages call for metamodeling techniques, which do not take into account the future use of such languages in multiview design environments. Tool implementers have tried a variety of ad-hoc techniques to maintain views in-synch, with modest results. To improve this state of affairs, a declarative approach is elaborated to automate multiview synchronization, building upon existing metamodeling techniques and recent advances in the field of function inversion for bidirectionalization. An application of these ideas to EMOF and a discussion of the resulting Declarative MVC software architecture are also provided. A significant benefit of the approach is the resulting comprehensive solution to a recurrent problem in the software modeling field.

## 1 Introduction

Most modeling languages provide different visual notations to highlight different aspects of the System Under Development (SUD). Most notably, UML2 defines a total of thirteen diagram types, grouped into three categories (structure, behavior, and interaction). In general, the same situation arises for Domain-Specific Modeling Languages (DSMLs). There is thus no escape from using several notations when modeling non-trivial software systems, a fact that vendors of modeling tools acknowledge by providing multiview capabilities. At some point in the development process the issue of *inter-view consistency* [Egy06] requires automation due to the complexity of the SUD. For example, determining consistency between a sequence diagram and the statechart for a single traffic-light may be done manually. However, tool support is required for models of realistic complexity (railroad crossings, reservation systems, consumer electronis, etc.)

The definition of a modeling language that introduces views is thus expected to provide an algorithm to determine whether a set of views is consistent. Using metamodeling terminology, the check for consistency is formulated as follows: (a) for each diagram type a metamodel has been defined, whose instances constitute the views manipulated by the modeler, including geometric information; (b) each such metamodel defines its intra-view Well-Formedness Rules (WFRs); and (c) additional WFRs ensure consistency encompassing several views. Given that WFRs are boolean-valued predicates over an object population, a yes/no answer can be provided about the consistency of the *integrated model*, i.e.,

the set of all views prepared by the modeler. Unless inter-view consistency is addressed at the level of the language definition itself, disagreement will otherwise ensue. For example, the workshop series *Consistency Problems in UML-based Software Development* was devoted to overcoming such disagreement for UML 1.x.

As useful as they are, yes/no answers about consistency contribute only partially to productivity. In a multiview setting, additional use cases demand automation (*multiview synchronization*, *model refactoring* [MB05, CW07], and *model completion* [SBP07]). In this paper, we address the multiview synchronization problem (defined below), leveraging on the lessons learnt from the related problem of inter-view consistency: we rely on a formal technique and address this concern at the language definition level.

Keeping multiple views in-synch requires propagating changes in two directions: (a) change requests validated against the WFRs of the integrated model are to be reflected on views; and (b) user-initiated *view updates* are to be processed in the opposite direction. The algorithm for realizing (a) is fixed once a *view definition* is available: given that the integrated model includes geometric information, updating views amounts to evaluating a function again. The situation is not so simple for (b), where partial information is available. For example, a particular view definition may select only those items at odd-numbered positions in a list. Inserting into the view then raises the question as to where to add an item in the underlying list (which is part of the integrated model). Such kind of decision problems are not solved by the current best-practices around tool implementation: Model-View-Controller architecture (MVC), runtime evaluation of WFRs expressed in OCL, transparent undo/redo. Rather, the particular realization of (b) is left to the criteria of tool vendors, thus opening the door to non-standard implementations. Our contribution in this paper improves on this state of affairs, not by building a tool with multiview synchronization capabilities (which is a task for industry) but by disclosing the inner workings of such solution (which industry refrains from doing).

### 1.1 Benefits of the proposed approach

The lack of tool compatibility (and sometimes correctness) around multiview synchronization stems from the fact that the specific policy governing synchronization is encoded manually in the Controller module of MVC (by each tool vendor, usually in an imperative language). In contrast, a declarative formulation, available as part of the language definition itself, allows both generating such implementation as well as statically analyzing the bidirectional transformations at design time. We call this approach DMVC, for Declarative MVC. The resulting productivity gain is particularly relevant for DSMLs, as the cost of developing tooling for them has to be amortized over a much smaller number of projects than for their general purpose counterparts. The DMVC approach is in line with recent advances in the definition of visual notations, where geometric constraints are used at runtime to automate the maintenance of diagram layouts, as discussed in detail by Grundy [LHG07] and exemplified in an Eclipse-based modeling tool generator[1].

[1] Marama meta-tools, `https://wiki.auckland.ac.nz/display/csidst/Marama+Tatau`

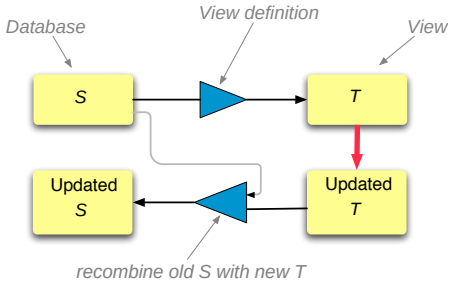


Figure 1: A bidirectional transformation, consisting of forward and backward functions [Pie06]

Our proposed architecture for DMVC builds upon a bidirectional transformation engine fulfilling formal guarantees. For example, given a view definition written by the DSML author, the engine can automatically derive its corresponding backward transformation. Importantly, the backward transformations [Ste07] thus obtained can cope with many-to-one mappings (i.e., non-injective functions, where different inputs are mapped to the same output, as for example in $f(x, y) = x + y$). This is achieved with *stateful transformations*, which track the information needed to complement that lost by the mapping (in the example, keeping a copy of either $x$ or $y$ allows handling user updates to $x + y$). Additionally, it is common practice for a backward transformation to take as input, besides the updated view, the original source. The intuition behind this scheme is depicted in Figure 1.

Statefulness and recombination distinguish our problem space from plain function inversion, which is enough in the particular case where each of (source, view) can be fully reconstructed from the other, i.e., whenever there is a one-to-one correspondence (a bijection) between source and target domains. As argued by Stevents [Ste07], such situation constitutes the exception rather than the rule in multiview modeling languages. For example, *dual syntaxes* [BMS07] do exhibit this property: a human-oriented syntax is defined for pretty-printing Abstract Syntax Trees (ASTs), while an accompanying XML-based syntax is defined for tool interchange. An MVC editor displaying a dedicated view for each representation needs no further information than the contents of an updated view in order to refresh the other, as no information is elided in the alternative syntaxes. In terms of our adopted approach, bijections are handled the same as the non-injective case (the latter being the "interesting" one from the point of view of multiview synchronization).

In summary, bidirectional transformations increase the productivity of the tooling process for DSMLs, and the quality of multiview environments. The structure of this paper is as follows. Competing methodologies around view update are reviewed in Sec. 2, followed in Sec. 3 by the application of one of them to the multiview synchronization problem, comprising the definition of EMOF-level operators for view specification (Sec. 3.1) that are well-behaved from a bidirectionalization point of view (Sec. 3.2). Given that any realistic multiview modeling tool will rely on available MVC frameworks, these practical aspects are discussed in Sec. 4. By placing the reported techniques in perspective, Sec. 5 reviews our contributions and concludes. Knowledge is assumed about language metamodeling, a background in functional programming is helpful but not required.

## 2 Candidate approaches distilled

All the approaches reviewed in this section share the common goal of enabling bidirectional transformations between pairs of data structures, with differences spanning the preferred representation (e.g., unordered trees vs. semi-structured data) and the available transformation operators (which may or may not allow user updates to alter the transformation as a side-effect). Besides highlighting the innovative aspects of each technique, their comparison allows introducing terminology to better characterize the multiview synchronization problem. However, the reader interested in the *resulting* Declarative MVC architecture may focus on Sec. 2.1 and skip the remaining subsections, moving directly to Sec. 3.

For ease of reference, candidate approaches are loosely grouped into (a) general purpose techniques (program inversion, data synchronization, and virtual view update); and (b) techniques aiming at supporting model transformations (QVT-Relations and graph-grammar based). This classification has more to do with the current level of adoption in the model-driven community than with any inherent capability of each approach.

### 2.1 Program inversion, Data synchronization, and Virtual view update

**Program inversion**. We discuss this technique first as it constitutes the basis for Declarative MVC. *Program inversion* [MHN+07] in the context of functional programming refers to determining, given a function $f(x_1, \ldots x_n)$, its inverse, so as to obtain the arguments given a result. For our problem at hand, an insight consists in choosing the building blocks for expressing view definitions such that they fulfill three *bidirectional properties*:

a) *Stability*: unmodified views are transformed back into the same source that gave origin to them (i.e., backward transformations introduce no spurious information);

b) *Restorability*: all updates on a source (that affect a view) can be canceled by updates on the view (i.e., the user has means to restore the integrated model to a previous state by just acting on the view)

c) *Composability*: the backward transformation is oblivious to the order in which updates took place (what counts is the end state).

Moreover, any composition of building blocks fulfilling these properties defines again a well-behaved bidirectional transformation. A Haskell implementation of an algorithm for the above is available[2]. We do not reproduce here the theory behind this algorithm, which is proficiently covered by Matsuda et. al. in their ICFP 2007 paper [MHN+07].

Program inversion fulfilling the above properties has been applied to particular cases: Liu [LHT+07b] presents a Java library for the bidirectional transformation of XML documents

[2]Generation of backward transformation programs based on derivation of view complement functions, `http://www.ipl.t.u-tokyo.ac.jp/~kztk/bidirectionalization/`

(the transformation operators constituting the BiXJ language). A subset of XQuery is translated into BiXJ in [LHT07a], thus allowing using a mainstream language for view definition, again with a prototype realization available[3]. Along the same lines, Xiong [XLH+07] translates a subset of ATL (Atlas Transformation Language), thus achieving bidirectionality[4].

**Data synchronization**. Algorithms developed to synchronize intermittently connected data sources (such as file systems or address books, between mobile and stationary devices) can also be applied to keep complex software artifacts in-synch. An exponent of this approach is the Harmony project [FGM+07] whose engine[5] implements Focal, a language with building blocks that allow writing only functions that always behave as *lenses*, i.e., bidirectional transformations. Focal is a low-level language operating on tree-shaped data structures (specifically, edge-labeled unordered trees). Standard encodings for mainstream data structures (lists, XML) are available, as well as libraries of higher-level lenses defined in terms of primitive ones. The design of Focal reflects its theoretical underpinnings in the field of type systems for programming languages, as static assurances can be obtained about the detailed type of inputs and outputs, to avoid runtime checks. In contrast, implementations such as BiXJ resort to returning a default value (e.g., unchanged input) or throw an exception whenever a function argument lies outside the function's domain.

The capabilities of EMOF-based modeling infrastructures (in particular undo/redo and evaluation of OCL invariants) grant a large degree of tolerance to inconsistent input, a feature that proves extremely valuable during the initial exploratory phases of DSML language engineering (which comprises the definition of transformations for each view). Moreover, experience shows that modelers frequently perform a series of editing operations which temporarily result in WFRs being broken. We aim at preserving this flexibility, to avoid usability problems similar to those that plagued syntax-directed text editors. In summary, we strike a balance between static assurances and ease of use by relying on runtime checks to capture side conditions not enforceable at design-time. If needed, static assurances beyond those amenable to static type checking can still be obtained by offline *model checking*, as shown in the case studies of [ABK07] (for transformations expressed as LHS $\rightarrow$ RHS production rules) and [GM07] (for imperative transformations).

**Update of virtual views in databases**. The view update problem has been studied in the context of databases, where the mechanism to define views is taken as given (relational algebra or calculus) and the kinds of view updates that may be propagated back without loss of information are determined. Recent work focuses on updating virtual and materialized XML views (in the latter case, incrementally). Most results have been incorporated into the program inversion and data synchronization techniques [MHN+07, FGM+07].

Given that the EMOF data model is richer than its relational counterpart (because of object IDs, ordered collections, reference handshaking), techniques targeting relational databases are overly restrictive when adapted to EMOF, limiting the operators available for view definition. The incremental maintenance of materialized views in OO databases is studied by Ali [AFP03].

[3] BiXJ and Bi-CQ, `http://www.ipl.t.u-tokyo.ac.jp/~liu/`

[4] Bi-ATL, `http://www.ipl.t.u-tokyo.ac.jp/~xiong/modelSynchronization.html`.

[5] Harmony Project, `http://www.seas.upenn.edu/~harmony`.

### 2.2 QVT-Relations and Graph-grammars

**QVT-Relations**. QVT-Relations was designed to encode input-output relationships by means of pattern-matching guarded by preconditions. At any given point in time, all but one of the models participating in a transformation are considered as non-updatable, thus constraining the solution space to a well-defined set of (updates, instantiations, deletions) on the *target model*.

Erche et. al. [EWH07] point out that metamodel-based language specs do not specify the connection between concrete and abstract syntax and propose QVT-Relations to bridge that gap. Given that such transformations are bidirectional, their architecture aims at solving the same problem space as Declarative MVC. There is no detailed discussion in [EWH07] on whether every QVT-Relations transformation is well-behaved in terms of the conditions defined by Matsuda et. al. [MHN$^+$07] (Sec. 2.1). Irrespective of the particular transformation mechanism adopted, Duboisset [DPKS05] recognizes that not all geometric constraints relevant for concrete visual syntaxes can be expressed in EMOF + OCL metamodels, offering as an example topological constraints in spatial databases. It is clear that QVT-Relations can support roundtripping over one-to-one mappings, however a discussion of its capabilities to back-propagate updates on non-injective views is missing in the literature. Our approach around geometric constraints is covered in Sec. 4.

**Triple Graph Grammars (TGGs)**. TGGs [GGL05] build upon directed typed graphs and graph morphisms. Informally, a TGG transformation rule consists of three graphs (left, interface, and right) and two morphisms (from the interface graph to each of left, right) which together describe the correspondence between embeddings of these graphs in source and target. In other words, such rule also states the inter-consistency conditions between source and target, besides specifying a transformation. Figure 2 depicts an example, the compilation of `if-then-else` into lower-level constructs (conditional jumps). Before a TGG transformation can be applied, its *positive* and *negative application conditions* are evaluated. These conditions demand a required context (certain nodes or edges must exist) or forbid a context (certain nodes or edges must be absent) connected in a certain topology.

An extension of TGG transformations to accomodate $N$-way relations is offered in [KS06]. For our purposes, this capability is not necessary as our architecture revolves around a single integrated model (i.e., to synchronize $N$ different view types $N$ bidirectional transformations are defined, as depicted in Figure 3). Graphical IDE support is available[6], and modularization has been proposed to cope with large-scale transformations. Similar to other rewriting techniques, the control flow aspect of a complex transformation (when to apply which rules) suggests breaking up large transformations into several more focused ones, to be applied sequentially.

As with the data synchronization approach, an encoding of EMOF models is necessary (in this case, into directed typed graphs), as well as expressing transformations in terms of graph morphisms guarded by application conditions. In our setting, some features of

[6]Some TGG-based tools: (a) MOFLON, `http://www.moflon.org/`; (b) MoTE/MoRTEn (as FUJABA plugins), `http://wwwcs.uni-paderborn.de/cs/fujaba/projects/tgg/`; (c) AToM3, `http://atom3.cs.mcgill.ca/`

the program inversion approach (Sec. 2.1) prove beneficial over TGGs: (a) OCL expressions in view definitions can be used directly by Matsuda's bidirectionalization algorithm [MHN+07], and (b) no explicit rules need be declared to delete view elements not supported anymore by source elements. The runtime overhead of encoding EMOF models into graphs can be reduced with the *Adapter* design pattern, at the cost of an indirection level (as with any approach, these design decisions would need to be revisited if the modeling infrastructure natively managed models in the format of the transformation engine).

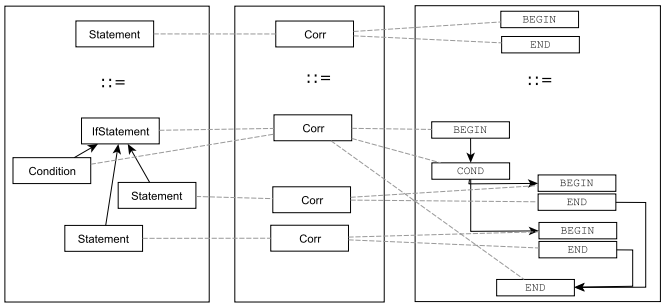


Figure 2: Sample TGG-based translation (`if-then-else` into conditional jumps, [Lei06])

## 3 Integration in an EMOF-based modeling infrastructure

After settling on the bidirectional program inversion technique [MHN+07], the interfacing of its functional inversion algorithm with current metamodeling infrastructure has to be addressed. A canonical approach consists in encoding EMOF models into inductive data types to automatically apply the inversion algorithm to each view definition, expressed as an affine function in treeless form [MHN+07]. Alternatively, a *fixed catalog* of bidirectional operators can be defined for EMOF models, fulfilling the three stated bidirectional properties. Both alternatives are explored, in Sec. 3.2 and Sec. 3.1 resp. Briefly, the advantage of the canonical approach is the open-ended set of base operators that can be defined, while the existing EMOF-based ones can only be recombined. On the other hand, adopting the EMOF-based operators avoids the detour to the inductive-data-types representation. Besides the performance gain, usability is also improved, as modelers are accustomed to conceptualizing transformations in terms of EMOF-level constructs. In any case, the approaches are not mutually exclusive, and any of them can be adopted to define views (injective or not) as part of the Declarative MVC (DMVC) architecture (Sec. 4).

As with Bidirectional-XQuery [LHT07a] and Bidirectional-ATL [XLH+07] a possibility consists in finding a subset of QVT-Relations amenable to encoding with bidirectional operators. Given that DSML authors already master the concepts required to understand the building blocks of bidirectionalization, directly using them results in making available their full expressive power. In a next step, subsets of OCL and QVT-Relations, which are already EMOF-aware, can be recast in terms of the operators in the next two subsections.

### 3.1 Operators for two-way transformations in Ecore: TwEcore

Each operator consists of a *forward* and a *backward* function. Borrowing notation from [HLM+06], $[\![X]\!]_F(s)$ stands for the application of the (possibly composite) operator $X$ to the source $s$ in the forward direction, resulting in a view $t$. The backward function, $[\![X]\!]_B(s, t')$ takes as argument the *unmodified* source $s$, the *updated* view $t'$, and returns a pair $(s', X')$ consisting of an updated source $s'$ as well as a possibly updated operator $X'$, to be used in further invocations. This statefulness is exemplified by $X$ = `twRenameProp(old, new)`, to rename the property `p` named `old`, where $s$ denotes an EMOF class. In this case $[\![X]\!]_F(s)$ is a clone of $s$ save for renaming the cloned property `p` from `old` to `new`. In turn, $[\![X]\!]_B(s, t') = (s', X')$ where $t'$ may have user updates, including renaming of property `p` itself. The backward function returns in $s'$ such changes save for any renaming of `p`, whose name is restored to `old`. An updated property name `new'` provided by the user on the view $t'$ is recorded instead in the state of $X'$ = `twRenameProp(old, new')`. Therefore, a successive application of $X$ will involve again the latest name entered by the user.

The basic example of a composite transformation is function composition, represented by $X$ = `twSeq(`$X_0 \ldots X_n$`)`, where the simpler transformations $X_0 \ldots X_n$ are applied so that $s'_i$ is the updated source for $t_i$ $(0 \leq i \leq n-1)$. The definition for this generic operation is reproduced from [HLM+06]:

$$
\begin{array}{rcl}
[\![X]\!]_F(s) & = & t \\
[\![X]\!]_B(s, t') & = & (s', \ \texttt{twSeq}\ [X'_0 \ldots X'_n]) \\
\texttt{where}\ t_0 & = & [\![X_0]\!]_F(s) \\
\ldots & & \\
t & = & [\![X_n]\!]_F(t_{n-1}) \\
(s'_{n-1}, X'_n) & = & [\![X_n]\!]_B(t_{n-1}, t') \\
\ldots & & \\
(s', X'_0) & = & [\![X_0]\!]_B(s, s'_0)
\end{array}
$$

In addition to the operator definitions, an EBNF-based concrete syntax is necessary to facilitate the discussion and exchange of view definitions (an area for future work in TwEcore). In terms of implementation, such syntax proves useful as it enables the interpretation of ad-hoc, or dynamically generated, view-definition scripts. In fact, this use case was foreseen by the authors of [MHN+07] and is supported in a Haskell-based bidirectional XML editor where users can update not only sources and views, but also transformations connecting them.

### 3.2 Encoding of EMOF models using inductive data types

This subsection explores the implications (for multiview synchronization) of implementing an EMOF infrastructure using functional programming (FP) instead of Java. This

exercise is not as far-fetched as it might seem at first sight because: (a) several bidirectionalization approaches are naturally expressed with FP; (b) functional programs are more amenable to static analysis than their OO counterparts; and (c) most of the proposed new language features for post-Java languages originate in FP[7]. The Declarative MVC architecture does not impose a functional realization, with this subsection serving as outlook for readers sharing an interest in functional programming.

Porting an EMOF infrastructure to the functional paradigm comprises devising encodings for (a) EMOF data structures, and (b) algorithms for views and transformations in EMOF. Regarding (a), given that EMOF models are typed, labelled graphs, the encoding proposed by Erwig is applicable [Erw01]. Regarding (b), the algorithms to port fall into two categories: (b.1) those already formulated in terms of OO concepts (e.g., written in QVT-Relations, ATL[8], or Java); and (b.2) those written as affine functions in treeless form, as expected by the algorithm for well-behaved bidirectionalization of Matsuda et. al. [MHN+07]. For (b.1) an encoding style is required that at least preserves the type-checking capabilities of the OO representation. The OOHaskell approach (Kiselyov and Lämmel [KL05]) fulfills these requirements by exploiting the type checking and type inference mechanisms of Haskell. As a result, Haskell-based processing following an OO style never results in a runtime errors like "method not found" that the OO version would have detected at compile-time.

While the pragmatic approach of TwEcore (and BiXJ, Bi-XQuery, and Bi-ATL) accelerates the construction of proofs of concept for DMVC tools, the same benefits could be easily achieved in a modeling infrastructure based on functional programming.

## 4 Diagram-based views and Geometric Constraint Solvers

Existing MVC frameworks for modeling infrastructures (e.g., EMF.Edit) support out-of-the-box a particular case of synchronization between view and model, namely bijections, where updates originating in a view are applied as-is to the single correspoding item in the associated model. The Declarative MVC architecture leverages this *data binding* capability (for EMF[9], GUI widgets[10], and the Eclipse Graphical Modeling Framework[11]). Interactions of this kind do not have to cope with non-injectiveness (as in $f(x, y) = x + y$). For comparison, they are shown bracketed with (A) in Figure 3, while those requiring bidirectionalization are marked with (B).

In case an update to the integrated model requires adding figures to a diagram view, default values have to be provided for the figure's position, size, layer, color, etc. While these values cannot be computed by the bidirectional transformation engine, they can still be managed declaratively with the help of a *geometric constraint solver* (e.g., [MC02]) which assumes the role of a *local Controller* in one of the MVC subsystems depicted in Figure 3

[7] Scala programming language, `http://www.scala-lang.org/`
[8] ATL, ATLAS Transformation Language, `http://www.eclipse.org/m2m/atl/`
[9] EMF Data Binding, `https://bugs.eclipse.org/bugs/show_bug.cgi?id=75625`
[10] JFace Data Binding, `http://wiki.eclipse.org/index.php/JFace_Data_Binding`
[11] Eclipse GMF, `http://www.eclipse.org/gmf/`

(i.e., it processes a subset of the view-level change requests, forwarding the non-filtered ones to the main Controller). Constraint solvers are responsible for enforcing geometric invariants, such as: (a) ensuring area inclusion between substates and their parent state in a statechart diagram, (b) ensuring non-overlap of the 2D regions for different figures.

The constraints on the layout of figures mandated by a *visual syntax* do not usually comprise the heuristics (such as crossings minimization) that distinguish a diagram with a comfortable layout from another which is hard to decipher. After computing a layout that fulfills those cognitive quality measures, small user edits should not cause a full re-arrangement, as becoming familiar with a new layout places a cognitive load on the user. This dynamic aspect is not normally considered in graph layout algorithms [Dub06]. Moreover, capturing all relevant *visual aesthetics* of a given visual notation is nowhere near straightforward, as their relative weight on diagram understanding may be discovered only with empirical studies [Dub06, p. 5]:

> A followup study reveals a visual aesthetic not previously considered by the graph drawing community. This new aesthetic, continuity, is the measure of the angle formed by the incoming and outgoing edges of a vertex. For the task of finding a shortest path between two vertices, continuity can become even more important than edge crossings.

The example reveals that well known visual aesthetics for graph layout overlook constraints that can be specified declaratively. Geometric constraint solvers are widely used in CAD tools (feature-based parametric modeling, [HJA05]) and are starting to be adopted by graphical frameworks for metamodeling [LHG07].

## 5 Conclusions

The complexity around keeping views in-synch in multiview authoring environments requires a comprehensive solution. As shown in this paper, one such solution relies on metamodeling, well-behaved bidirectional transformations, and geometric constraint solvers. Current EMOF infrastructures have paved the way for functional extensions, such as bidirectionalization (which requires moderate integration effort) and geometric constraint solving (which is only now starting to be considered part of metamodeling [LHG07]).

Existing tools for general-purpose modeling have been developed following a traditional (non-declarative) MVC architecture, and are not expected to migrate overnight to a new paradigm. Instead, the primary candidates to benefit from Declarative MVC are Domain-Specific Modeling Languages (DSMLs). More generally, we argue that applying to DSMLs the same (metamodel-based) definition techniques as for UML 1.x will impair their adoption, as such techniques overlook the connection between concrete and abstract syntax, do not handle multiview synchronization, and lack precise semantics for backpropagating updates from non-injective views. The techniques brought together in this paper address those weaknesses identified in previous efforts around the definition and tooling of DSMLs.

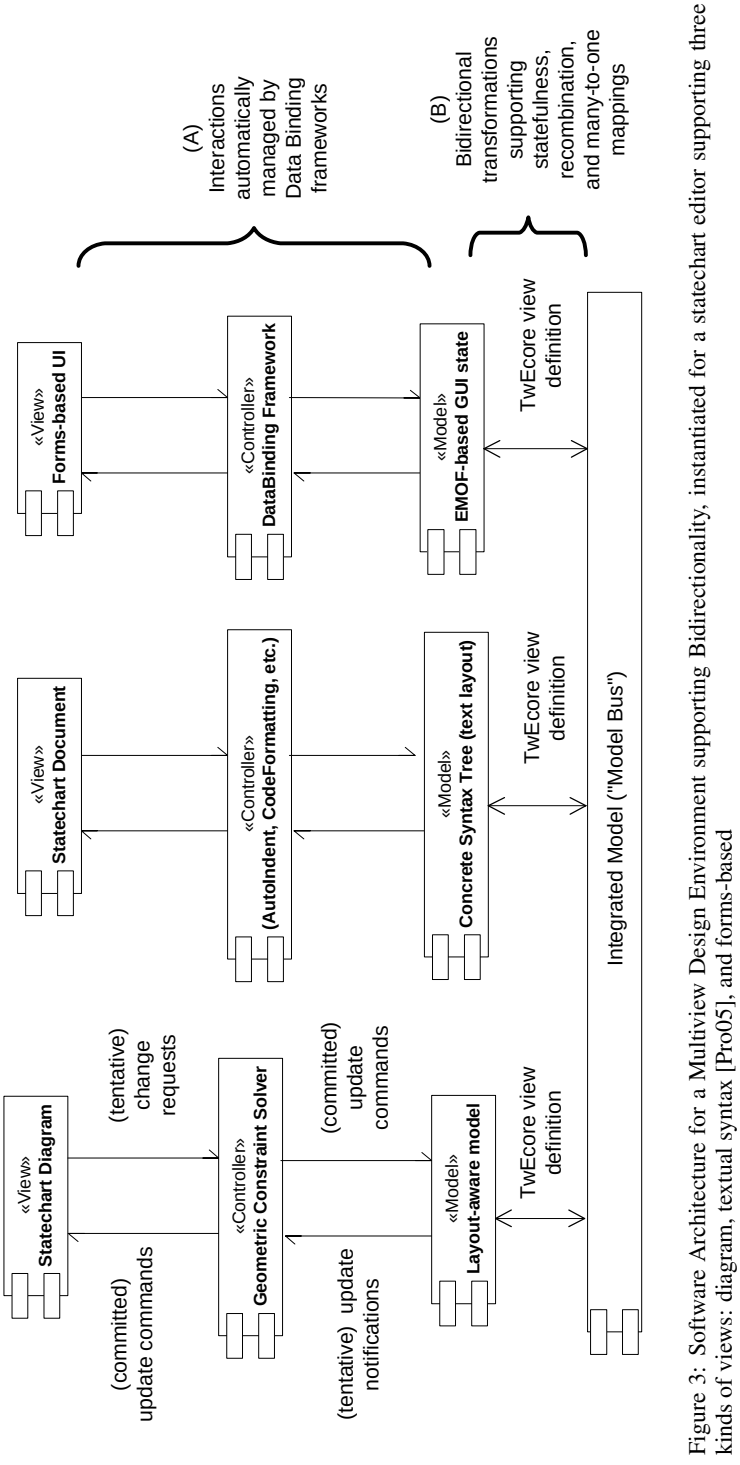


Figure 3: Software Architecture for a Multiview Design Environment supporting Bidirectionality, instantiated for a statechart editor supporting three kinds of views: diagram, textual syntax [Pro05], and forms-based

## References


[ABK07] Anastasakis, K, Bordbarand, B, and Küster, J. M. Analysis of Model Transformations via Alloy. In Faivre, B. B. A, Ghosh, S, and Pretschner, A, editors, *4th MoDeVVa workshop Model-Driven Engineering, Verification and Validation*, pages 47–56, In conjunction with MoDELS07, Nashville, TN, USA, 2007.

[AFP03] Ali, M. A, Fernandes, A. A. A, and Paton, N. W. MOVIE: an incremental maintenance system for materialized object views. *Data Knowl. Eng.*, 47(2):131–166, 2003.

[BMS07] Brabrand, C, Møller, A, and Schwartzbach, M. I. Dual Syntax for XML Languages. *Information Systems*, 2007. Earlier version in Proc. 10th International Workshop on Database Programming Languages, DBPL '05, Springer LNCS vol. 3774. pp. 27-41.

[CW07] Correa, A and Werner, C. Refactoring Object Constraint Language Specifications. *Software and Systems Modeling*, 6:113–138, 2007.

[DPKS05] Duboisset, M, Pinet, F, Kang, M.-A, and Schneider, M. Precise Modeling and Verification of Topological Integrity Constraints in Spatial Databases: From an Expressive Power Study to Code Generation Principles. In Delcambre, L. M. L, Kop, C, Mayr, H. C, Mylopoulos, J, and Pastor, O, editors, *ER*, volume 3716 of *LNCS*, pages 465–482. Springer, 2005. `http://www.isima.fr/~kang/pinet/ER_paper.pdf`.

[Dub06] Dubé, D. Graph Layout for Domain-Specific Modeling. Master's thesis, School of Computer Science, McGill University, Montreal, Canada, 2006. `http://moncs.cs.mcgill.ca/people/denis/files/thesis_HREF.pdf`.

[Egy06] Egyed, A. Instant consistency checking for the UML. In *ICSE '06: Proceeding of the 28th International Conference on Software Engineering*, pages 381–390, New York, NY, USA, 2006. ACM.

[Erw01] Erwig, M. Inductive graphs and functional graph algorithms. *J. Funct. Program.*, 11(5):467–492, 2001.

[EWH07] Erche, M, Wagner, M, and Hein, C. Mapping visual notations to MOF compliant models with QVT relations. In *SAC '07: Proceedings of the 2007 ACM symposium on Applied computing*, pages 1037–1038, New York, NY, USA, 2007. ACM.

[FGM+07] Foster, J. N, Greenwald, M. B, Moore, J. T, Pierce, B. C, and Schmitt, A. Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. *ACM Trans. Program. Lang. Syst.*, 29(3):17, 2007.

[GGL05] Grunske, L, Geiger, L, and Lawley, M. A Graphical Specification of Model Transformations with Triple Graph Grammars. In Hartman, A and Kreische, D, editors, *ECMDA-FA*, volume 3748 of *LNCS*, pages 284–298. Springer, 2005.

[GM07] Garcia, M and Möller, R. Certification of Transformation Algorithms in Model-Driven Software Development. In Bleek, W.-G, Räsch, J, and Züllighoven, H, editors, *Software Engineering 2007*, volume 105 of *GI-Edition LNI*, pages 107–118, 2007. `http://www.sts.tu-harburg.de/~mi.garcia/pubs/2007/se2007/GarciaMoeller.pdf`.

[HJA05] Hoffmann, C. M and Joan-Arinyo, R. A Brief on Constraint Solving. Unabridged; abridged version in CAD&A, 2005. `http://www.cs.purdue.edu/homes/cmh/distribution/papers/Constraints/ThailandFull.pdf`.

[HLM+06] Hu, Z, Liu, D, Mei, H, Takeichi, M, Xiong, Y, and Zhao, H. A Compositional Approach to Bidirectional Model Transformation. Technical Report METR 2006-54, The University of Tokyo, Bunkyo-Ku, Tokyo, Oct 2006. `http://www.keisu.t.u-tokyo.ac.jp/research/techrep/data/2006/METR06-54.pdf`.

[KL05] Kiselyov, O and Lämmel, R. Haskell's overlooked object system. Draft; Submitted for publication; online since 30 Sep. 2004; Full version released 10 September 2005, `http://homepages.cwi.nl/~ralf/OOHaskell/paper.pdf`, 2005.

[KS06] Königs, A and Schürr, A. MDI - a Rule-Based Multi-Document and Tool Integration Approach. *Journal of Software & System Modeling*, 5(4):349–368, December 2006.

[Lei06] Leitner, J. Verifikation von Modelltransformationen basierend auf Triple Graph Grammatiken, March 2006. Diplomarbeit. TU-Berlin und Universität Karlsruhe (TH).

[LHG07] Liu, N, Hosking, J. G, and Grundy, J. C. MaramaTatau: Extending a Domain Specific Visual Language Meta Tool with a Declarative Constraint Mechanism. In *VL/HCC*, pages 95–103. IEEE Computer Society, 2007.

[LHT07a] Liu, D, Hu, Z, and Takeichi, M. Bidirectional interpretation of XQuery. In *PEPM '07*, pages 21–30, New York, NY, USA, 2007. ACM. `http://www.ipl.t.u-tokyo.ac.jp/~liu/PEPM2007.pdf`.

[LHT+07b] Liu, D, Hu, Z, Takeichi, M, Kakehi, K, and Wang, H. A Java Library for Bidirectional XML Transformation. *JSSST Computer Software*, 24(2):164–177, May 2007. `http://www.ipl.t.u-tokyo.ac.jp/~hu/pub/jssst-cs06-liu.pdf`.

[MB05] Marković, S and Baar, T. *Proc of the 8th Intnl Conf MoDELS 2005*, volume 3713 of *LNCS*, chapter Refactoring OCL Annotated UML Class Diagrams, pages 280–294. Springer Verlag, October 2005.

[MC02] Marriott, K and Chok, S. S. QOCA: A Constraint Solving Toolkit for Interactive Graphical Applications. *Constraints*, 7(3-4):229–254, 2002.

[MHN+07] Matsuda, K, Hu, Z, Nakano, K, Hamana, M, and Takeichi, M. Bidirectionalization transformation based on automatic derivation of view complement functions. In Hinze, R and Ramsey, N, editors, *ICFP*, pages 47–58. ACM, 2007. `http://www.ipl.t.u-tokyo.ac.jp/~hu/pub/icfp07.pdf`.

[Pie06] Pierce, B. C. The Weird World of Bi-Directional Programming. Invited talk at ETAPS, 2006. `http://www.cis.upenn.edu/~bcpierce/papers/lenses-etapsslides.pdf`.

[Pro05] Prochnow, S. H. KIEL: Textual and Graphical Representations of Statecharts. `http://rtsys.informatik.uni-kiel.de/~rt-kiel/kiel/documents/talks/oberseminar-0511-spr/talk.pdf`, Nov 2005.

[SBP07] Sen, S, Baudry, B, and Precup, D. Partial Model Completion in Model Driven Engineering using Constraint Logic Programming. In *INAP'07 (International Conference on Applications of Declarative Programming and Knowledge Management)*, Würzburg, Germany, 2007.

[Ste07] Stevens, P. Bidirectional Model Transformations in QVT: Semantic Issues and Open Questions. In *MoDELS*, volume 4735 of *LNCS*, pages 1–15. Springer, 2007.

[XLH+07] Xion, Y, Liu, D, Hu, Z, Zhao, H, Takeichi, M, and Mei, H. Towards Automatic Model Synchronization from Model Transformations. *22nd IEEE/ACM International Conference on Automated Software Engineering*, pages 164–173, November 2007. `http://www.ipl.t.u-tokyo.ac.jp/~xiong/papers/ASE07.pdf`.

# Towards Automated Testing of Abstract Syntax Specifications of Domain-Specific Modeling Languages

Daniel A. Sadilek and Stephan Weißleder

Humboldt-Universität zu Berlin
Department of Computer Science
Rudower Chaussee 25
12489 Berlin, Germany

{sadilek|weissled}@informatik.hu-berlin.de

**Abstract:** The abstract syntax of domain-specific modeling languages (DSMLs) can be defined with metamodels. Metamodels can contain errors. Nevertheless, they are not tested systematically and independently of other artifacts like models or tools depending on the metamodel. Consequently, errors in metamodels are found late—not before the dependent artifacts have been created. Since all dependent artifacts must be adapted when an error is found, this results in additional error correction effort. This effort can be saved if the metamodel of a DSML is tested early. In this paper, we argue for metamodel testing and propose an approach that is based on understanding a metamodel as a specification of a set of models. Example models are given by a user to test if the set of models specified by a metamodel is correct. We present an example from the domain of earthquake detection to clarify our idea.

## 1 Introduction

Metamodels are a common way to describe the structure of domain-specific modeling languages (DSMLs). Tools for a DSML like editors, interpreters, or debuggers base on this metamodel. Like every other artifact, metamodels contain errors (e.g. wrong specification of classes or associations between them). When errors in a metamodel are found late, dependent models and tools must be adapted. Hence, *detecting errors* in a metamodel early can save time and money.

In software engineering, *testing* is the primary means to detect errors. In this paper, we advocate *testing metamodels* and present an approach for automated testing based on the specification of positive and negative example models.

In Sec. 2, we describe how to *specify* metamodel tests with example models and we describe how to *execute* them in Sec. 3. In Sec. 4, we substantiate our approach with an exemplary development process of a simple DSML. We discuss related work in Sec. 5. We conclude and give an overview of future work in Sec. 6.

## 2 How to Test Metamodels?

### 2.1 Example Models and Test Models

How can a metamodel be tested? To answer this question, we have to consider the nature of metamodels. A metamodel is the specification of a set of possible or desired models. What does it mean that a metamodel contains an error? It means that the specified set of models either contains a model that is undesired or that it does not contain a model that is desired.

The set of models specified by a metamodel is a subset of all instances of all possible metamodels expressible as instances of the meta-metamodel used[1]. Figure 1 shows an Euler diagram visualizing this idea. To test a set specification, one could give all elements of the set and check if the set does contain these and only these elements. Since metamodels generally specify an infinite set of models, this is impossible. Instead, representative elements can be given. Each representative element can be either an element or not an element of the set.

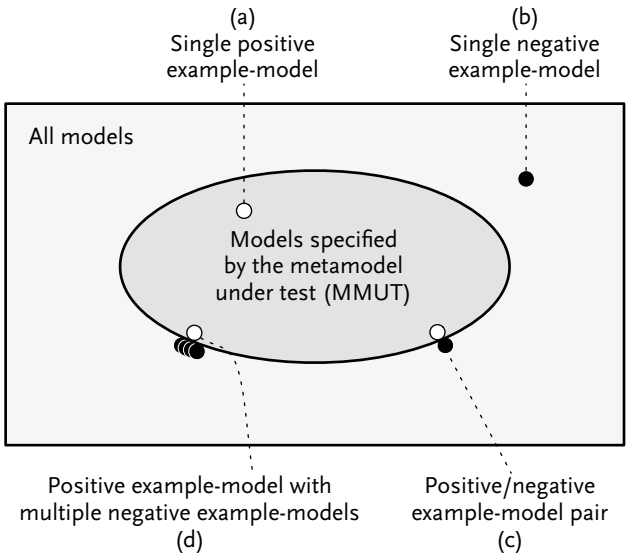


Figure 1: Metamodel as a set specification; example models as elements.

In the following, we describe the relationship between such representative elements and a *Metamodel Under Test (MMUT)*.

For metamodels, we call representative elements *example models*. Example models that are elements of the set of desired models should be correct instances of the MMUT—hence

[1] We consider metamodels that are instances of MOF.

we call them *positive* example models (Fig. 1a). Example models that are not elements of the set of desired models should not be instances of the MMUT—we call them *negative* example models (Fig. 1b).

Single example models can be anywhere inside or outside the set specified by the MMUT. But, for high discriminatory power of the tests, we propose to give example models as pairs of a positive and a negative example model that differ only in one aspect, for example by an attribute value or by the existence of some object or reference. The positive/negative example model pairs then demarcate the boundary of the set specified by the MMUT (Fig. 1c). The more example model pairs are given by a user, the more precise the boundary is demarcated. This resembles the common testing technique of boundary testing [Bei90].

A positive example model and its negative counterpart differ only slightly. If a user has to specify them separately, this introduces a lot of redundancy. Therefore, we propose to specify them in only one *test model*. A test model is a positive example model extended with *test annotations* that describe which model elements have to be added or removed to make it a negative example model. Thus, a test model allows a user to specify a positive/negative example model pair without redundancy.

We propose to allow the user to annotate more than one model element. Then, one test model can describe one positive and multiple negative example models (Fig. 1d).

## 2.2 Test Metamodel

### 2.2.1 Motivation for an Additional Metamodel

Technically, models cannot be created and stored without a corresponding metamodel. Which metamodel should be used for test models? Can we use an existing one, for example the MMUT or the metamodel of UML object diagrams?

Unfortunately, the MMUT cannot be used. The reason is that the MMUT does not allow to express test annotations. Also, a user may want to create test models before the MMUT exists—for example, to sketch how instances may look like or to follow a *test first* approach like in test-driven development [Bec02].

Test models describe instances of the MMUT. UML object diagrams can be used to describe instances of arbitrary classes. Could they be used to describe instances of the MMUT's classes? Unfortunately, the metamodel for UML object diagrams does not contain elements to express test annotations, i.e. there is no way to describe a combination of several example models in one object diagram. Also, UML object diagrams explicitly reference the classes that the modeled objects instantiate. This again forbids to create test models before the MMUT exists.

Therefore, another metamodel for test models is needed. We call it *test metamodel*. Figure 2 shows the test metamodel we propose and other artifacts of our approach: A test model is an instance of the test metamodel and it specifies one positive example model

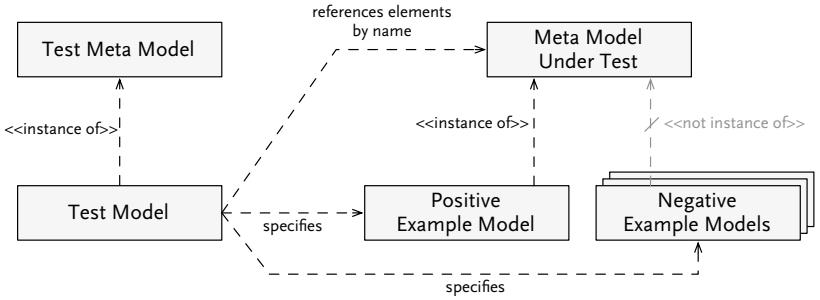


Figure 2: Relations between artifacts of our approach.

and an arbitrary number of negative example models. A test model references elements of the MMUT by name (explained below). For each MMUT, there can be various test models that are all instances of the test metamodel. The test metamodel is fixed, i.e. test models for different MMUTs are all instances of the same test metamodel.

### 2.2.2 Structure of the Test Metamodel

Figure 3 shows the test metamodel: *Instances* of classes are given with their class name and an optional object name. An instance can have an arbitrary number of *attributes*. Each attribute has a name and a value, which is given generically as a string literal. Instances can be connected by *references*. Reference ends can be named. The name must match an attribute of the instance's class at the opposite end of the reference.

All *model elements* (instances, attributes, and references) have an existence specification that can be *arbitrary* (default value), *enforced*, or *forbidden*. All model elements with existence specification *arbitrary* or *enforced* are part of the specified positive example model. If an element is enforced, removing this element from the model leads to a negative example model. If an element is forbidden, it is not part of the positive example model and adding it leads to a negative example model.

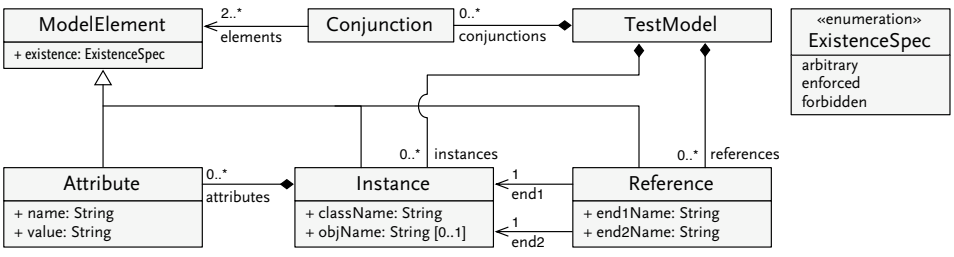


Figure 3: The test metamodel.

If multiple elements in a test model are enforced or forbidden, one test model describes multiple negative example models. If two or more elements should be enforced or forbidden conjunctionally, i.e. they should describe just one negative example model, then they can be connected by a *conjunction*.

## 3 Test Execution

In this section, we briefly describe how test models are used to test metamodels. The test execution consists of 5 steps:

1. *Resolve references to the MMUT.*
   The test model references the elements of the MMUT by name. In the first step, it is checked whether all references can be resolved. If a reference of a not forbidden element cannot be resolved, the test fails.
2. *Derive example models from the test model.*
   Each test model specifies one positive example model and multiple negative example models. The *positive example model* is derived from the test model by leaving out all forbidden elements. A *negative example model* is derived for each conjunction in the test model and for each enforced or forbidden element that is not connected to a conjunction. Let $e$ be a conjunction of elements or a single element for which a negative example model is to be derived. Then all forbidden elements except $e$ are left out when constructing the model. If $e$ itself is a forbidden element, it is added to the negative example model; if $e$ is enforced, it is left out.
3. *For all example models: Create an instance of the MMUT according to the example model.*
4. *For all example models: Check multiplicities and constraints of the created MMUT instance.*
5. *For all example models: Decide test outcome.*
   If the current example model is a positive one, constraints must *not* be violated in the previous steps; if it is a negative one, *at least one* constraint must be violated.

## 4 Example: Testing the Metamodel of a Stream-Oriented DSML

In this section, we describe the first step of an exemplary iterative development process of a simple *stream-oriented DSML* for the configuration of an earthquake detection algorithm: A sensor source generates a data stream that can be filtered and that finally streams into a sink. For this example, we use the prototypical implementation of our approach: *MMUnit* (`http://mmunit.sourceforge.net`).

The development of the stream-oriented DSML involves a language engineer and an expert of the domain, a seismologist. As a first step, seismologist and language engineer discuss some example uses of the new language. For the beginning, they concentrate on one specific detection algorithm called *STA/LTA* [Ste77]. They sketch their ideas in an informal ad hoc concrete syntax. Figure 4 shows the resulting model they have drawn on a whiteboard.

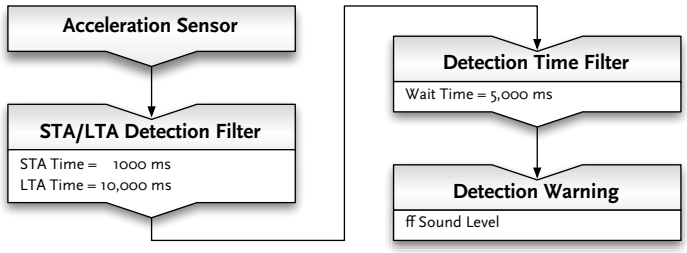


Figure 4: A first whiteboard sketch of a model expressed in a stream-oriented DSML.

The intention behind this model is as follows: Sensor readings from an *acceleration sensor* are piped through a filter that realises the *STA/LTA detection*. The filter forwards sensor readings that are considered to be the beginning of an earthquake and blocks all others. The frequency of sensor readings is limited by another filter, *detection time filter*, before they stream into a stream sink that generates an earthquake *detection warning* whenever a sensor reading streams in, for example by activating a warning horn. The filters and the sink contain attributes influencing their behavior.

The language engineer prefers a test-driven development. Therefore, he creates a metamodel test *before* he creates the metamodel. For this, he derives a test model from the model he and the seismologists sketched on the whiteboard. The result is shown in Fig. 5.[2] The four instances on the left reproduce the model sketch. The *positive* example model specified with the test model consists of only these objects. The test model also specifies three *negative* example models: (1) Each sink must have a reference to a stream source. Therefore, the language engineer sets the existence specification of the reference from *oWarning* to *oTimeFilter* to "enforced". This describes a negative example model in which the reference is missing. (2) Seismologist and language engineer discussed but discarded the idea of a motion detector filter. To ensure that the final metamodel does not support a motion detector, the language engineer adds the forbidden instance *oMotionDetector*. The corresponding negative example model contains this additional instance. (3) Each source must be referenced by exactly one sink. To test this, the language engineer adds the forbidden instance *oWarning2*. Again, the corresponding negative example model contains this additional instance.

[2]The notation we use for test models is similar to that of UML object diagrams. Additionally, enforced elements are marked with a thick border, forbidden elements with a dashed one.

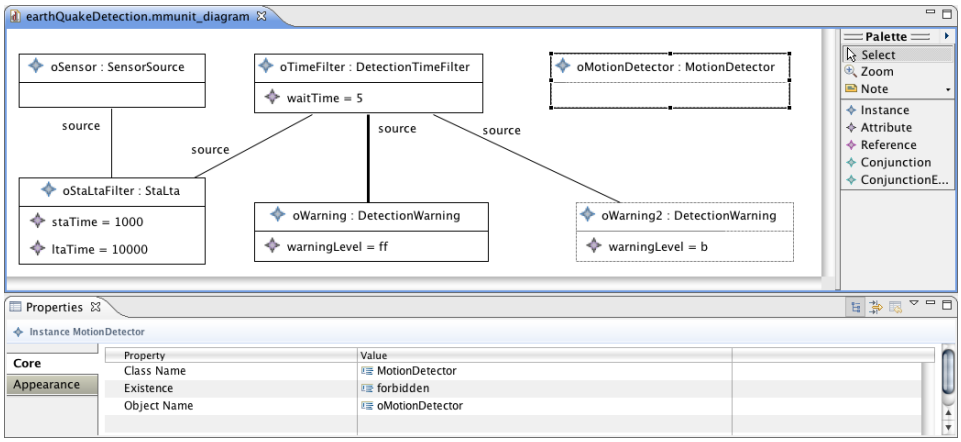


Figure 5: Screenshot of the test model for the earthquake detection metamodel. (The test model editor is part of our prototype implementation MMUnit.)

After specifying the test model, the language engineer creates a metamodel for the stream-oriented language (Fig. 6). In order to execute the tests specified by the test model, the language engineer uses MMUnit to generate corresponding JUnit test cases. The generated JUnit tests use a library that implements the test process as described in Sec. 3. Executing the JUnit tests reveals an error: The negative test model that contains the additional instance *oWarning2*, case (3), is not rejected as an instance of the metamodel. The language engineer realizes that he forgot to set the multiplicity of the association between *Source* and *Sink* to 1 on the *Sink* end. He corrects the error and executes the tests again. Now, all tests pass.

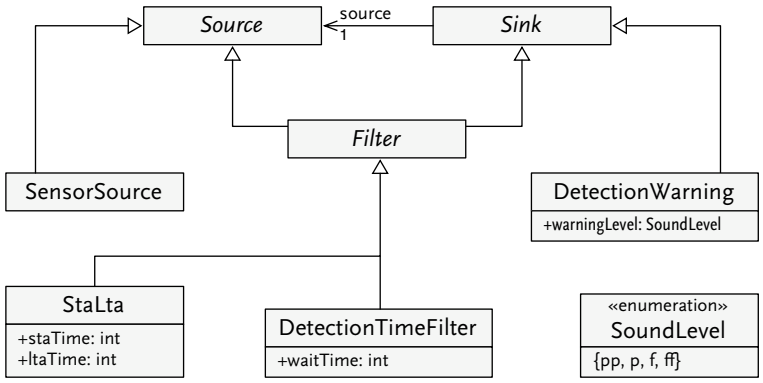


Figure 6: A proposal for a domain-specific metamodel.

## 5 Related Work

In model-based testing, many approaches use models as specifications to generate test cases for a system under test (SUT) [NF06, PP05, OA99, AO00]. The tests check if the SUT satisfies all constraints of the model. The models themselves are assumed to be correct, whereas we want to test the correctness of (meta-)models.

Tests for model transformations are handled in [Küs06, WKC06, BFS+06]. They all assume that the used metamodels are correct and they focus on testing the transformation process between them. Our approach is complementary to their approaches as it tests the metamodels they assume to be correct.

In grammar testing [Pur72], character sequences are used to test a developed grammar [Läm01]. This generic approach permits to define both words that conform to the grammar and words that do not. While our metamodel also allows to generically describe instances, we target metamodels, not grammars.

## 6 Conclusion and Future Work

**Conclusion.** Metamodels play an important role for the definition of the abstract syntax of a DSML. In this paper, we argued that metamodels should be tested systematically. We proposed an approach for testing metamodels and exemplified it with tests of a metamodel for a stream-oriented DSML. Our approach is based on understanding metamodels as set specifications. Our idea is to use example models that may lie either inside or outside of the set specified by the metamodel.

We already did a prototypical implementation of our approach, which we sketched shortly in this paper. It is based on the Eclipse Modeling Framework (EMF). The prototype is called *MMUnit* (`http://mmunit.sourceforge.net`) and provides an editor for test models and can generate JUnit tests from test models. Such a generated JUnit test reads a test model and checks if the described positive example model is an instance of the MMUT and if the described negative example models are not instances of the MMUT. If both checks pass, the test succeeds; otherwise it fails.

Metamodel tests are possible. They can be specified quite easily. By the integration with JUnit, metamodel tests can be executed automatically. Thus, metamodel tests can be integrated into existing software development processes (e.g. metamodel tests can be executed as part of a continuous integration build).

**Future work.** Currently, our implementation tests classes, attributes, and associations of a metamodel together with their multiplicities. Usually, a constraint language like OCL is used to constrain the set of possible models. We plan to extend our implementation to support the evaluation of OCL constraints during test execution.

Another restriction in our current approach is that a test model always describes exactly one positive example model. We think that one may also want to describe multiple positive

example models that differ only slightly or one may also want to describe negative example models only. For this, we could extend the test metamodel with an attribute that states whether the test model describes a positive or a negative example model as the base case. Furthermore, we could add another enumeration value for existence specifications that allows for specifying that a model element can be removed or left in the example model without influencing whether the example model is a positive or a negative one.

We left open whether metamodel tests pay off economically? To answer this question, systematic case studies are necessary.

**Acknowledgments.** We would like to thank the reviewers for valuable comments. This work was supported by grants from the DFG (German Research Foundation, research training group METRIK).

## References

[AO00] Aynur Abdurazik and Jeff Offutt. Using UML Collaboration Diagrams for Static Checking and Test Generation. In *UML 2000*. University of York, UK, 2000.

[Bec02] Kent Beck. *Test Driven Development: By Example*. Addison-Wesley Professional, November 2002.

[Bei90] B. Beizer. *Software Testing Techniques*. John Wiley & Sons, Inc., 1990.

[BFS+06] Erwan Brottier, Franck Fleurey, Jim Steel, Benoit Baudry, and Yves Le Traon. Metamodel-based Test Generation for Model Transformations: an Algorithm and a Tool. In *ISSRE '06: Proceedings of the 17th International Symposium on Software Reliability Engineering*, pages 85–94, Washington, DC, USA, 2006. IEEE Computer Society.

[Küs06] Jochen M. Küster. Definition and validation of model transformations. *Software and Systems Modeling*, V5(3):233–259, 2006.

[Läm01] Ralf Lämmel. Grammar Adaptation. In José Nuno Oliveira and P. Zave, editors, *FME'01*, volume 2021 of *LNCS*, pages 550–570. Springer, 2001.

[NF06] Clementine Nebut and Franck Fleurey. Automatic Test Generation: A Use Case Driven Approach. *IEEE Trans. Softw. Eng.*, 32(3):140–155, 2006.

[OA99] Jeff Offutt and Aynur Abdurazik. Generating Tests from UML Specifications. In *UML'99 — The Unified Modeling Language*, volume 1723 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 1999.

[PP05] Wolfgang Prenninger and Alexander Pretschner. Abstractions for Model-Based Testing. *Electr. Notes Theor. Comput. Sci.*, 116:59–71, 2005.

[Pur72] Paul Purdom. A sentence generator for testing parsers. *bit*, 12(3):366–375, 1972.

[Ste77] S. W. Stewart. Real time detection and location of local seismic events in central California. In *Bull. Seism. Soc. Am.*, volume 67, pages 433–452, 1977.

[WKC06] Junhua Wang, Soon-Kyeong Kim, and David Carrington. Verifying Metamodel Coverage of Model Transformations. In *ASWEC'06*, 2006.

# How Web 2.0 can leverage Model Engineering in Practice

Manuel Wimmer, Andrea Schauerhuber, Michael Strommer,
Jürgen Flandorfer and Gerti Kappel
Business Informatics Group
Institute for Software and Interactive Systems
Vienna University of Technology, Austria
{wimmer|schauerhauber|strommer|flandorfer|kappel}@big.tuwien.ac.at

**Abstract:** Today's online model repositories offer to download and view the textual specifications of e.g. metamodels and models in the browser. For users, in order to efficiently search a model repository, a graphical visualization of the stored models is desirable. First attempts that automatically generate class diagrams as bitmaps, however, do not scale for large models and fail to present all information. In this paper, we present our Web 2.0 MetaModelbrowser, a model visualization service which provides an Ajax-based tree-viewer for efficiently browsing Ecore-based metamodels and their models. As a main contribution of this work the MetaModelbrowser is complementary to existing model repositories in that its visualization service can be integrated into them. The MetaModelbrowser, furthermore, allows zooming in and out of the details of arbitrarily sized models as necessary. Furthermore, we have done some case studies on the one hand how to extend the MetaModelbrowser, e.g., for creation, update, and deletion of model elements as well as supporting model weaving, and on the other hand how to incorporate the MetaModelbrowser in current versioning systems.

## 1 Introduction

With the rise of model-driven development, model repositories are intended to facilitate research in model engineering and consequently in domain-specific modeling. Model repositories are central places where all kinds of modeling artifacts (e.g., meta-metamodels, metamodels, models, and possibly transformation models) are stored and coordinated. They can serve as a platform for making available the specification of metamodels to others (typically necessary for domain-specific modeling languages) and for exchanging models, as well as a resource for teaching/learning materials.

There have been started some intiatives for building model repositories, e.g., zoomm.org, www.kermeta.org/mrep, or the Atlas MegaModel Management (AM3) [1]. The latter one is hosted within the popular Eclipse environment and is a subproject of the Generative Modeling Technologies (GMT) project. The artifacts present in this model repository, furthermore, are organized into sets of models of similar nature called zoos, e.g. a zoo for metamodels and a zoo for transformations [4]. The AM3 zoos are continuously growing and provide a respectable source of information in the meantime.

However, a more popular way of storing and organizing modeling artifacts is probably

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<ecore:EPackage xmi:version="2.0"
  xmlns:xmi="http://www.omg.org/XMI" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:ecore="http://www.eclipse.org/emf/2002/Ecore" name="SimpleUML"
  nsURI="http://SimpleUML" nsPrefix="SimpleUML">
 <eClassifiers xsi:type="ecore:EClass" name="UMLModelElement" abstract="true">
  <eStructuralFeatures xsi:type="ecore:EAttribute" name="kind" ordered="false" unique="false"
    lowerBound="1" eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EString"/>
  <eStructuralFeatures xsi:type="ecore:EAttribute" name="name" ordered="false" unique="false"
    lowerBound="1" eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EString"/>
 </eClassifiers>
 <eClassifiers xsi:type="ecore:EClass" name="Attribute" eSuperTypes="#//UMLModelElement">
  <eStructuralFeatures xsi:type="ecore:EReference" name="owner" ordered="false"
    lowerBound="1" eType="#//Class" eOpposite="#//Class/attribute"/>
  <eStructuralFeatures xsi:type="ecore:EReference" name="type" ordered="false" lowerBound="1"
    eType="#//Classifier" eOpposite="#//Classifier/typeOpposite"/>
 </eClassifiers>
[...]
```

Figure 1: XMI serialized model.

having a CVS like server software at hand. These repositories provide all means necessary to handle different versions of textual artifacts and let them compare syntactically. It seams therefore obvious to use existing tools also to store models as they can be serialized into the XMI format. Although versioning of models remains an open and challenging research field [11, 12], basic support for versioning models can be provided by CVS. CVS repositories can also be easily accessed via a standard Web browser providing an easy to use interface to users not familiar with IDE's such as Eclipse.

Generally, today's online model repositories offer either to view the models' textual specifications in standard Web browsers or their download for graphically viewing them offline with appropriate tools. For example the Eclipse Modeling Framework's (EMF) [5] allows the user to view models within a tree-viewer in its *Sample Ecore Model Editor*. For a visually more appealing presentation of models having a concrete graphical syntax defined one can use the Graphical Modeling Framework (GMF) [6] to view these models. For Ecore based metamodels such a definition and editor comes along with the GMF plug-in.

Neither viewing models in the browser nor extra downloading them is always satisfactory. On one hand the textual representation of models is hardly readable and understandable for humans. In Figure 1, we depict a snippet of an Ecore based metamodel modeling the core of UML class diagrams. This representation of models is hardly readable because of type information from the Ecore meta-metamodel itself. This is further aggravated by the structure of the XML file representing only the containment structures of the Ecore meta-metamodel and not the structure of the metamodel. Also, the heavy use of XPath expressions complicates the otherwise easy to read metamodel. Note, that we refer sometimes to both model and metamodels when we talk about models. Only if we think it is necessary to distinguish explicitly between different levels of the OMG's 4-layer metamodeling stack we will do so.

On the other hand, downloading a model for an offline graphical visualization first, requires the appropriate tools installed, second, has to be done for each model as well as detaches the downloaded models from the rest of the zoo, and third, is time-consuming.

For users, in order to efficiently search a model repository, a graphical visualization of the stored models is desirable. First attempts, such as the AM3 Atlantic Raster Zoo, that automatically generate class diagrams as PNG bitmaps, however, do not scale for large models. More specifically, the readability of these pictures is hampered by some modeling elements, e.g., associations and roles, hiding others. Furthermore, users are always confronted with the whole picture and cannot show or hide parts of the model as necessary. Finally, the class diagram representation fails to present all information of each model element. This is the case for properties of a metamodel, which have been defined in its corresponding meta-metamodel, responsible for fine tuning the semantics of a metamodel.

Visualization of models in the context of Web browsers is not only desirable for the search in model repositories but also for any attempts building a Web application for model management purposes relying on models accessible through the WWW.

## 2 Web 2.0 to the Rescue

As a solution to the above presented problems, we propose our Web 2.0 MetaModel-browser (MMB) as an alternative way of visualizing models within online model repositories. This MMB offers an Ajax-based tree-viewer for efficiently browsing Ecore-based metamodels and their models. In this respect, our goal is that users of the service get an idea of the models' structure as quick as possible. A tree-viewer, enables such efficient browsing by allowing to zoom in and out of the details of arbitrarily sized models as necessary. Based on the visualization functionality of our MMB we will demonstrate in a small case study, how the MMB can be placed into the context of model management in general.

Thus, our idea is to reproduce the user interface of EMF's Sample Ecore Model Editor (cf. Figure 2 (a)) for the Web. More specifically, parts of EMF's architecture and plug-ins are reused and the JFace/SWT-based user interface of the desktop application is replaced (cf. Section 3). Figure 2 (b) provides a screenshot of the MMB visualizing the SimpleUML metamodel, i.e., a simplified version of the UML class diagram metamodel, which is available at the AM3 AtlantEcore Zoo.

It has to be noted that in this work we seize two aspects of the Web 2.0. First, seen from the social network aspect, model repositories in fact are Web 2.0 applications that at the same time boost and are supported by the model engineering community. Second, seen from the technology aspect, the use of Ajax - often presented as the Web 2.0 enabling technology - allows for a desktop like working experience avoiding full page reloads and allowing user-transparent page updates.

Benefits of our MMB are briefly summarized in the following:

**Extension for Existing Model Repositories.** The MMB is complementary to existing model repositories in that its visualization service can be integrated into them. This is done by including parameterized links to the MMB into one's own model repository website as explained in Section 3.

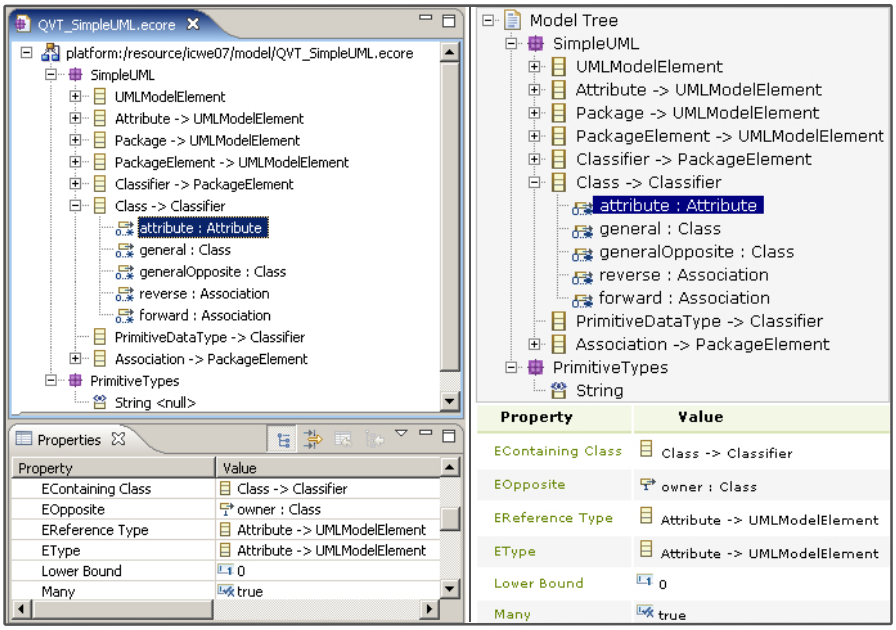


Figure 2: SimpleUML metamodel in (a) Eclipse and (b) MMB.

**Usability.** In contrast to automatically generated class diagram pictures, a tree-viewer, allows to zoom in and out of the details of arbitrarily sized models as necessary. Moreover, the MMB provides information of the models that cannot be captured in class diagrams in a separate properties table for each model element (cf. Figure 2 (b)).

**Saving of Time.** There is no need to download the models, as they can be visualized in the browser. Furthermore, the asynchronous communication through Ajax allows to request only those parts of a model which the user is interested in.

**Familiar Environment.** The MMB's user interface is heavily based on the Eclipse Sample Ecore Model Editor and thus, represents a familiar environment to EMF users (cf. Figure 2).

**Reuse.** We have built the MMB upon existing EMF plug-ins. Beside being able to browse Ecore-based metamodels, this allows automatically generating the necessary artifacts for Web 2.0 *Model*browsers. This means that users can also browse models conforming to the metamodels (cf. Section 3.3).

**Comparability of Models.** The visual representation of models fosters their comparability, e.g., metamodels representing the same language such as UML 2.0 metamodel specifications from different authors.

**Intellectual Property.** If necessary, an additional feature of our MMB allows uploading the models to the server. The models can then be browsed via an ID but the source file is not given away.

**Exchange.** We allow the user to have an alternative representation of models he/she can view per URI. This can ease the exchange of those models among partners worldwide. With developers or customers located around the world this leads to more efficient distribution of knowledge and information.

**Model Management.** The browsing and viewing of models is only one particular aspect of exploring online repositories, of course. Other features are needed to allow for substantial benefit from Ajax-based Web applications. As to this we present in Section 4.1 a case study of online model weaving, that demonstrates how the MMB can be extended to get one step closer to a desktop like model engineering tool.

**Support for Versioning Systems.** The integration of our MMB in existing versioning system browsers, e.g., for browsing SVN repositories, allows to use our service as a front-end for current state-of-the-art versioning systems in which metamodel and model versions are stored. Of course, various additional features for browsing different versions of metamodels and models could be implemented such as a diff-viewer for showing the differences between two metamodels graphically.

## 3 The Web 2.0 MetaModelbrowser

In the following sections we outline the implementation of our MMB. Its visualization service and examples can be accessed at www.metamodelbrowser.org.
In order to use our MMB for visualizing models the inclusion of the following parameterized URL in one's own model repository website is necessary:
`www.metamodelbrowser.org/BrowseTreeServlet?url=<URL_OF_MODEL>`

### 3.1 Prerequisites for MMB

Before we will go into implementation details we will give a brief overview of the framework, that constitutes our MMB, see Figure 3. As already mentioned our MMB operates on Ecore-based metamodels. Using the OMG's terminology we rely on the M3 layer on Ecore as a meta-metamodel. Based on this meta-metamodel we can handle all to Ecore conforming metamodels. Viewing of models is of course then restricted to models conforming to some well defined metamodels. These modeling artifacts have been created in the user workbench by some tool and need to be serialized in XMI format. To be able to use those models in combination with out MMB they have to be stored in some repository that is accessible through HTTP. The MMB can retrieve the models by a simple request and present them to the user. The view service for metamodels is a generic feature as the meta-metamodel is not a parameter when invoking the view request. Making the meta-

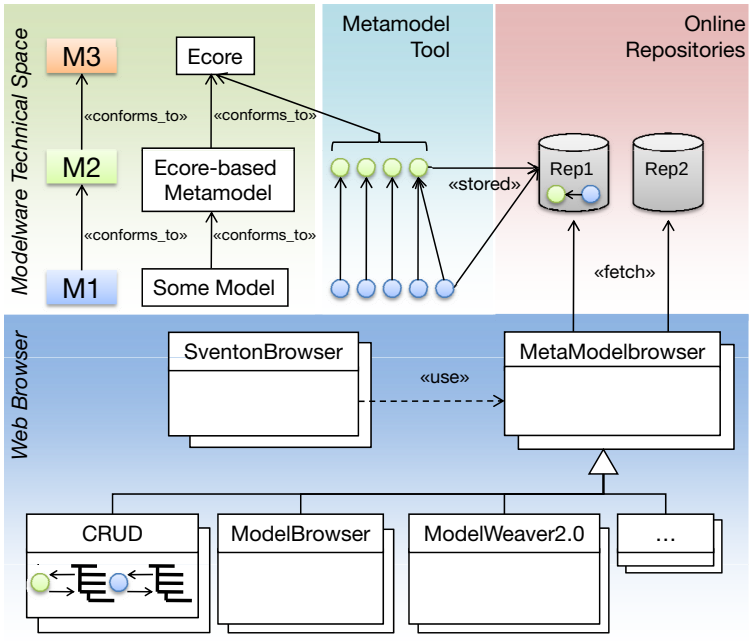


Figure 3: Overview of the MMB framework.

metamodel a variable would require to build a completely new framework for modeling, as EMF solely relies on Ecore and does not for example support MOF specified by the OMG. To view models of some modeling language is not supported in a generic way. Right now an additional plug-in for our MMB has to be generated within Eclipse by our Model Browser component manually, see Sections 3.3 and 5. The functionality of the MMB can be further extended by particular components like one that supports all four CRUD (Create, Retrieve, Update, Delete) operations. Components that support model weaving will be discussed later.

### 3.2 Architecture of the MetaModelbrowser

As a basis for our implementation task we have chosen to make use of the EMF framework whose main purpose is to allow for building tree-based model editors. EMF therefore provides for its own metamodeling language called Ecore, which can be seen as equivalent to the OMG's Essential MOF [7]. As is shown in Figure 4 (a), Ecore-based models can then be fed into the EMF code generator to automatically produce all components for a fully functional model editor. Those components are in fact three distinct Eclipse plug-ins, i.e., one representing the model, one that acts as a controller between the model and the viewer, as well as one that represents the user interface and that is integrated within the

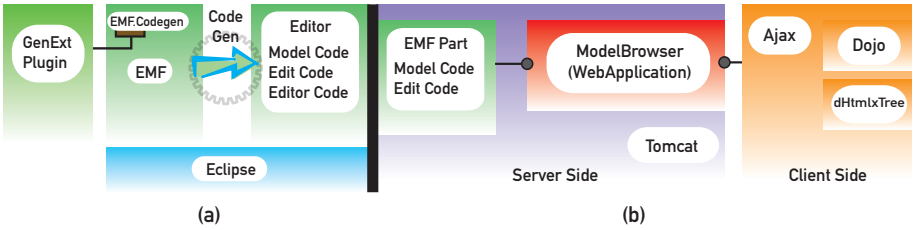


Figure 4: EMF components overview (a) and MMB architecture (b).

Eclipse workbench.

The MMB architecture, depicted in Figure 4 (b), is based on the Tomcat server infrastructure and thus relies heavily on the Java Servlet and JSP technologies. Concerning EMF we completely reuse the model and edit code from EMF's built-in Sample Ecore Model Editor as external libraries. To replace the JFace/SWT editor code we have come up with a browser-based viewer, that can be populated with content of any arbitrary Ecore-based model. This viewer is realized with the Ajax frameworks dHtmlxTree [8] and Dojo [9] as shown in Figure 4 (b). dHtmlxTree serves as a Widget for displaying the model elements in a tree. To display the properties of model elements in a table we used Dojo.

### 3.3 Browsing Models

In the above section we described how to browse metamodels by reusing existing EMF components and integrating them within the Web application. More challenging, however, is the ability to browse also models and the design decision we have had to make at this point in time. To be able to browse models one needs to install our GenExtPlug-in (cf. Figure 4 (a)), that uses an extension point of EMF.Codegen in order to apply some changes to the code generation process. After installing the plug-in the user needs to generate model and edit code and export these two artifacts as plug-ins. These plug-ins in turn have to be uploaded into the Web application and can be used as soon as they reside on the server. Our MMB then automatically determines what Modelbrowser to instantiate on the basis of the model to be visualized. For demonstration purposes we have used our GenExtPlug-in to generate the code for the SimpleUML Modelbrowser, that is available at our project site.

A screenshot of browsing a SimpleUML model is shown in Figure 5.

## 4 Extending the MetaModelbrowser Architecture

In this section, we present two case studies how to extend our browsing capabilities for metamodels and models.

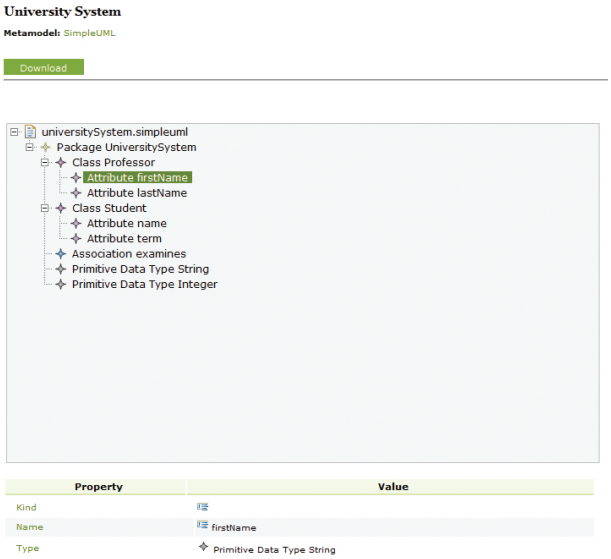


Figure 5: Browsing models conforming to a known metamodel.

The first case study deals with model management which is an important task in the software development process. Model operations therefore play a major role and are vital if the model engineering field shall be lifted to the Web. In the following we will present our early prototype ModelWeaver 2.0, which realizes tasks such as model creation, model update, model element deletion and model weaving.

The second case study is about how to incorporate our browsing capabilities in already existing versioning system browsers. The basic view service of our MMB primarily targets models residing in a simple Web directory. In order to provide also access to CVS model repositories which store metamodel and model versions, we have developed a simple extension to Sventon [2], a Java application capable of browsing Subversion repositories.

### 4.1 ModelWeaver 2.0

Based on the ATLAS Model Weaver (AMW) [14] and the MMB we have implemented a prototype, that enables the user to weave any two metamodels within the Administration Tool of the MMB. Note, that the MMB provides a simple repository by itself, which is however not open to the public yet, as we have not implemented any access control model. Our ModelWeaver 2.0 [22] builds again on the EMF to provide a mapping metamodel and let the user create, update, and browse a mapping model, that describes the dependencies between the elements of two modeling languages, i.e., metamodels. We started out with a simple mapping metamodel depicted in Figure 6. This mapping model is quite similar to

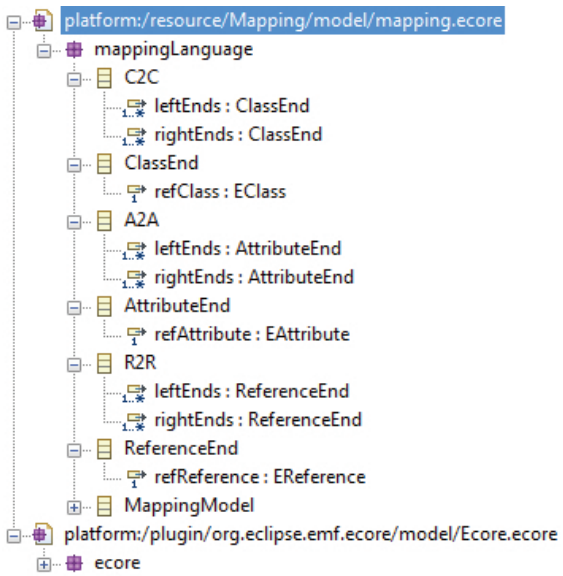


Figure 6: Mapping Metamodel for the ModelWeaver 2.0

the core weaving metamodel of the AMW. However it is more specialized in the sense that we do not define a generic weaving link, able to connect any two modeling elements. The mapping ends types are determined by the types Ecore specifies for creating metamodels. For example the *ClassEnd*s contained in a Class2Class (*C2C*) mapping are always of type *EClass*, which is defined in Ecore. Our mapping metamodel can not be strictly classified as M2 model according to the OMG's 4-layer architecture. As can be seen at the bottom of Figure 6 Ecore itself is referenced as resource into our mapping metamodel. This is because we need to access specific types of Ecore in our metamodel, which makes it in fact also an extension to Ecore, our meta-metamodel. From these perspectives it is not just a simple metamodel, although we will pretend in the following that it is just a pure metamodel.

Figure 7 shows the main view of the ModelWeaver 2.0. The view service of the MMB has been completely reused in order to visualize the two metamodels positioned on the left and the right as well as the mapping model which is located in the middle. In Figure 7 we illustrate a simple weaving application with our SimpleUML and a basic Entity Relationship language. Every concrete mapping model has a root element called *MappingModel* according to the mapping metamodel. This model element is instantiated automatically upon the start of the weaving. In a next step the user selects this element and chooses one of the three basic mapping operators available (*C2C, A2A or R2R*) from the drop down menu. Afterwards he/she can add the chosen element to the mapping model and create left and right ends for this mapping operator. Because of the drag&drop support of the dHtmlxTree Ajax framework it is then possible to drag one element from the left or right hand

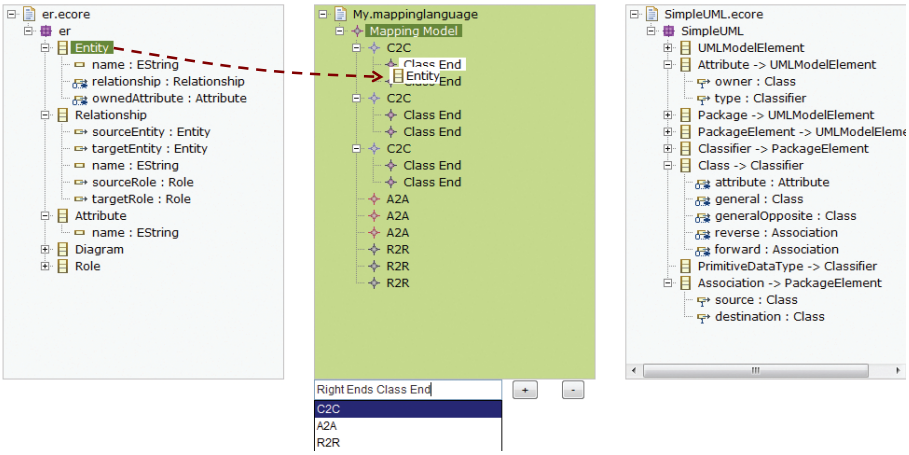


Figure 7: Weaving view of the ModelWeaver 2.0

side tree and drop it to a suitable mapping operators end. Also the deletion and modification of mapping model elements is supported. When the weaving task is finished the user has the possibility to download the created mapping model for later usage or the usage in another development environment to enable for example the creation of a transformation model. The derivation of transformation models based on the metamodel weavings is not implemented yet and remains an open issue for future work. If the user decides to get back to work later on it is also possible to load an existing mapping model into the next session provided that the same metamodels have been selected for the weaving task.

### 4.2 SventonBrowser

The most common versioning systems for documents of any kind are most likely the Concurrent Versions System (CVS) and Subversion (SVN). These systems are also used to store models to have at least the possibility to trace changes between different versions of models and compare them if necessary on a textual level such as the XMI serialization of the metamodels and models. To handle data stored within such versioning systems with our MMB framework, e.g. view these models with the visualization service, a Web interface is needed. An application providing such an interface is the Java-based Web application called Sventon, which allows to browse SVN repositories. Provided that all information needed to access an SVN repository is given, proper URLs can be passed the MMB as input to view the stored metamodels. Constructing URLs and passing them correctly to our MMB application in order to view one's own metamodels is however not very user-friendly. Due to this we have implemented a small extension to the Sventon application, which integrates our visualization service by constructing the right URLs. Our "Sventon Browser" is available under the URL:

*www.metamodelbrowser.org/sventon*

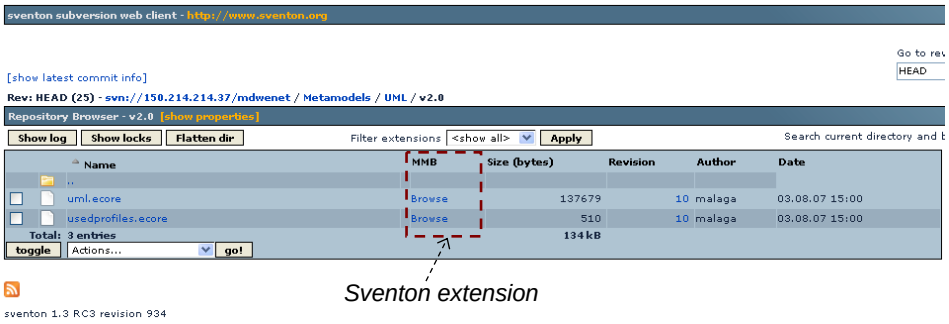


Figure 8: SventonBrowser using MMB

Unfortunatly only one repository can be configured over the Sventon Web interface at the moment. Manual entries can however be achieved through properties files in the Tomcat directory. Figure 8 shows a screenshot of our extended Sventon, that illustrates the functionality for the MDWEnet model repository. The URL's for any Web accessible repository, especially for Sventon, look like this:

```
www.metamodelbrowser.org/BrowseTreeServlet?
url=http://metamodelbrowser.org:80/
sventon/showfile.svn?path=<URI_OF_MODEL>&<some_parameters_needed>
```

With the conduction of this small case study we demonstrate how existing repositories based on CVS or SVN can be accessed and browsed using the MMB. This allows now a greater audience to benefit from our MMB framework, which might be in the near future not only limited to the model view operation.

## 5 Critical Discussion

Our MMB supports the visualization of models through the ModelBrowser plug-in. This plug-in can only be used within a running Eclipse workbench to generate the necessary model code for the Web application as described above. This solution is not very comfortable and undermines our efforts to build a platform independent application aiming at model management in general. Thus, it would be nice to have a fully automatic generation of Modelbrowser code. This means, for the user it should be possible to upload an Ecore-based metamodel and the Modelbrowser code, which allows visualizing models conforming to the metamodels, is automatically generated by the Web application. The user no longer needs to activate the code generation within EMF and upload the files to the Web server. We discovered, that this code generation is not so simple to integrate with a standalone Java application. The reason for this is that EMF uses JET (Java Emitter Templates) [3]. JET needs to be invoked within an Eclipse workbench. According to our research there seem to be workarounds to handle code generation in a standalone manner, but this would also require at least an Eclipse "Driver" available on the server side.

## 6 Related Work

With respect to our MetaModelbrowser framework we identified three major fields of related work.

*Visualization of models*. There has been a lot of work done in the area of model visualization. Especially related to us is the two dimensional graphical representation of software models. Automatic layout approaches for UML class diagrams considering certain kinds of aesthetic criteria are for example presented in [15], [16]. Based on the work on planar graphs, i.e., graphs that have no crossing edges, our MetaModelbrowser could be extended to support the visualization of large class diagrams.

*Model Repositories*. The idea of storing data as well as software building artifacts and their management dates back to the early 90s. With the Portable Common Tool Environment (PCTE) [17] a framework was developed and should integrate popular software engineering tools and support the management of large data. Today, the basis for the success of model repositories is certainly an accepted common interchange format like XMI. However, the goal of interoperability is not yet achieved. The ModelCVS project [19] therefore tries to overcome interoperability problems among different tool vendors and to provide a framework for model management.

Also, the idea of model repositories goes far beyond the software engineering community. In bio engineering, the modeling of complex organic structures becomes more and more important. Also the exchange of these models among various stakeholders gets crucial. The cellML [18], an XML-based model interchange format and repository with over 300 models, is a perfect example for the importance of online repositories and the appropriate visualization of the stored models.

*Web 2.0*. The social aspects of the Web 2.0 movement, such as participation in an informal manner, higher usability due to AJAX technology, and the lightweightness, have been its key for success. Braun et al. [20] also base their work on Web 2.0 to allow for a collaborative informal ontology engineering process. For their Web based implementation SOBOLEO, a lightweight ontology editor and social bookmarking system, they also incorporate AJAX technology. If we consider metamodels somehow equivalent to ontologies [21] it is possible to extend our MetaModelbrowser to act as a lightweight collaborative metamodel editor.

## 7 Conclusions and Future Work

In this work we have presented an Ajax-based approach to visualize metamodels and models on the Web. The approach has been implemented as a public visualization service based on existing Eclipse technologies. This visualization service is now used in the ModelCVS project [10], in the MDWEnet project [13], and for visualizing about 250 metamodels of the AtlantEcore Zoo[1].

[1] http://www.eclipse.org/gmt/am3/zoos/atlantEcoreZoo/

Also we have demonstrated by a well known model engineering technique how our MMB can be extended into a framework supporting various kinds of model management operators and how the visualization service can be integrated into existing SVN repository viewers.

The presented work reveals two major directions for future work.

*What is an intuitive visualization of models and is the tree-based view an appropriate one?* First of all, we believe that an intuitive visualization should consist of several views on the model, depending on the user's interests. The tree-based view is focused on depicting the Ecore metamodeling language's structural relationships between packages, their classes and in turn their features. Nevertheless, a class diagram can provide a better visualization for e.g., inheritance and containment relationships, under certain conditions. In this respect, a static class diagram picture is not enough. The user has to be able to influence the way and what parts of a model are to be visualized like in existing desktop modeling environments, e.g., temporarily hiding of any model element. Ideally, the class diagram view and the tree-view are combined and the latter serves as an outline of the model, e.g. in a sidebar. These ideas directly lead to the second direction for future work.

*Is it possible to realize a model editor as a Web application?* Our inital MMB allows read-only access to models but could be extended for editing functionality similar to the model weaving case study. The introduction of such functionality, however, comes along with other important issues such as versioning and concurrency which have already been researched in the database community.

## 8 Acknowledgments

We thank Jeremy Solarz for his support during implementation of the ModelWeaver 2.0.

## References

[1] ATLAS MegaModel Management, official site: http://www.eclipse.org/gmt/am3

[2] Sventon. Java Repository Browser, official site: http://www.sventon.org/

[3] JET. Java Emitter Templates, Model2Text Transformation, official site: http://www.eclipse.org/modeling/m2t/?project=jet

[4] Allilaire, F., Bézivin, J., Bruneliére, H., and Jouault, F.: Global Model Management in Eclipse GMT/AM3. Eclipse Technology eXchange workshop in conjunction with ECOOP'06, 2006

[5] Budinsky, F., Steinberg, D., Merks, E., Ellersick, R., Grose, T.: Eclipse Modeling Framework. Addison Wesley, 2003

[6] Graphical Modeling Framework (GMF), official site: http://www.eclipse.org/gmf

[7] OMG: Meta Object Facility (MOF) 2.0 Core Specification, 2003

[8] dhtmlxTree - AJAX powered DHTML JavaScript Tree component with rich API, official site: http://scbr.com/docs/products/dhtmlxTree/

[9] Dojo - The Javascript Toolkit, official site: http://dojotoolkit.org/

[10] Kappel, G., Kapsammer, E. , Kargl, H. , Kramler, G., Reiter, T., Retschitzegger, W., Schwinger, W., Wimmer, M.: On Models and Ontologies - A Layered Approach for Model-based Tool Integration, Modellierung 2006, Innsbruck, March 2006, GI LNI 82, pp. 11-27, 2006.

[11] Reiter, T., Altmanninger, K., Bergmayr, A., Schwinger, W., Kotsis, G.: Models in Conflict - Detection of Semantic Conflicts in Model-based Development. In 3rd International Workshop on Model-Driven Enterprise Information Systems (MDEIS-2007), Funchal, Madeira - Portugal, June 2007.

[12] France, R., Rumpe, B.: Model-driven Development of Complex Software: A Research Roadmap. FOSE '07: 2007 Future of Software Engineering, IEEE Computer Society, pp. 37-54, 2007.

[13] Vallecillo et al.: MDWEnet: A Practical Approach to Achieving Interoperability of Model-Driven Web Engineering Methods. In Workshop Proc. of 7th Int. Conference on Web Engineering (ICWE'07), Como, Italy, July 2007, Politecnico di Milano, pp. 246-254, 2007.

[14] Del Fabro, M.D., Bézivin, J., Jouault, F., Gueltas, G.: AMW: A Generic Model Weaver, Proceedings of the 1ère Journée sur l'Ingénierie Dirigée par les Modèles (IDM05), 2005.

[15] Gutwenger, C., Jünger, M., Klein, K., Kupke, J., Leipert, S., Mutzel, P.: A new approach for visualizing UML class diagrams, SoftVis '03: Proceedings of the 2003 ACM symposium on Software visualization, San Diego, California, pp. 179-188, 2003.

[16] Eichelberger, H.: Aesthetics and Automatic Layout of UML Class Diagrams, PhD Thesis, Julius-Maximilians-Universität Würzburg, 2005.

[17] Long, F., Morris, E.: An Overview of PCTE: A Basis for a Portable Common Tool Environment, Technical Report, Carnegie Mellon University, 1993.

[18] Cuellar, A., Lloyd, C., Nielsen, P., Bullivant, D., Nickerson, D., Hunter, P.: An Overview of CellML 1.1, a Biological Model Description Language, SIMULATION: Transactions of The Society for Modeling and Simulation International, 79(12):740-747, 2003.

[19] Kappel, G., Kapsammer, E., Kargl, H., Kramler, G., Reiter, T., Retschitzegger, W., Schwinger, W., Wimmer, M.: On Models and Ontologies - A Layered Approach for Model-based Tool Integration , Proceedings of Modellierung 2006, GI-Edition, Lecture Notes in Informatics, Eds.: H.C. Mayr, R. Breu, 22-24 March, Innsbruck, Austria, 2006.

[20] Braun, S., Schmidt, A., Walter, A., Nagypal, G., Zacharias, V.: Ontology Maturing: a Collaborative Web 2.0 Approach to Ontology Engineering, Proceedings of the Workshop on Social and Collaborative Construction of Structured Knowledge (CKC 2007) at the 16th International World Wide Web Conference (WWW2007) Banff, Canada, May 8, 2007.

[21] Kappel, G., Kargl, H., Kramler, G., Reiter, T., Schwinger, W., Wimmer, M.: Lifting Metamodels to Ontologies: A Step to the Semantic Integration of Modeling Languages, ACM/IEEE 9th International Conference on Model Driven Engineering Languages and Systems (MoDELS/UML 2006), Genova, Italy, October 2006.

[22] Solarz, J.: Model Weaver 2.0: Eine AJAX-basierte Webanwendung für Model Weaving, Master Thesis, Vienna University of Technology, 2008 (in German).

# Transformationen zwischen UML-Use-Case-Diagrammen und tabellarischen Darstellungen

Julia Pilarski
Fachgebiet Software Engineering
Leibniz Universität Hannover
julia.pilarski@arcor.de

Eric Knauss
Fachgebiet Software Engineering
Leibniz Universität Hannover
eric.knauss@inf.uni-hannover.de

**Abstract:** In den frühen Phasen eines Softwareprojekts steht die Modellierung in einem besonderen Spannungsfeld: Entweder sind die Modelle formal genug, um verifizieren zu können, dass sie richtig modelliert wurden, oder umgangssprachlich genug, damit beim Kunden validiert werden kann, ob das Richtige modelliert wurde. Aus diesem Grund eignen sich komplexere Modelle zur Szenario-Darstellung (z.B. UML-Sequenz-Diagramme) nicht so gut. Andererseits haben grafische Modelle den Vorteil, einen guten Überblick zu bieten. Transformationen zwischen textuellen Beschreibungen und grafischen Modellen können dieses Spannungsfeld auflösen, indem sie es erleichtern, grafische Modelle und natürliche Sprache parallel zu nutzen. Dieser Beitrag untersucht das Verhältnis zwischen den Use-Case-Diagrammen der UML und (typischerweise tabellarischen) natürlich-sprachlichen Use-Cases. Mit Hilfe eines Basismetamodells definieren wir die gemeinsamen Konzepte beider Darstellungen. Wir beschreiben entsprechende Transformationen und geben konkrete Beispiele.

## 1 Einleitung

Use-Cases (auch Anwendungsfälle genannt) setzen sich für die Anforderungsspezifikation immer weiter durch. Als Vorteil wird vor allem die Beschreibung von funktionalen Anforderungen im Kontext von Benutzerzielen gesehen. Auch die explizite Betrachtung von Ausnahmen ist eine der Stärken von Use-Cases.

Für die konkrete Ausgestaltung von Use-Cases gibt es verschiedene Vorschläge: Auf der einen Seite gibt es die modellierungsnahe Sicht (hier repräsentiert durch die UML [OMG07]). Use-Cases werden dabei in UML-Use-Case-Diagrammen benannt und die enthaltenen Erfolgs- und Sonderszenarien werden in Interaktionsdiagrammen spezifiziert. Die Vorteile sind hierbei, dass durch die definierte Semantik der Modelle Missverständnisse vermieden werden, sowie dass die Modelle durch Abstraktion eine gute Übersicht bilden können.

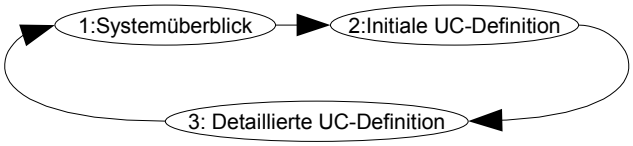


Abbildung 1. Effiziente Anwendungsfallspezifikation (nach [Bir06])

Dem steht auf der anderen Seite ein eher natürlich-sprachlicher Ansatz gegenüber [Coc01]. Hier werden die Szenarien als *User Stories* oder als Aufzählung in tabellarischen Vorlagen (im folgenden als *Template* bezeichnet) formuliert. Der Vorteil der natürlichen Sprache ist, dass Kunden ohne technischen Hintergrund einen Use-Case verstehen und kritisieren können. Dies ist eine wichtige Voraussetzung, um Use-Cases validieren zu können. Zudem hat man durch eine Menge von Formulierungsrichtlinien (z.B. [Rup06]) eine Basis geschaffen, um trotz natürlicher Sprache zu einer weitgehend eindeutigen Spezifikation zu kommen. Diese Richtlinien kann man zum Teil auch automatisch überprüfen, wie zum Beispiel in [Cri06, Kna07] gezeigt wurde. Als entscheidender Nachteil bleibt jedoch die mangelnde Übersichtlichkeit.

Hier wäre es schön, die Vorteile der UML einsetzen zu können. Abbildung 1 zeigt schematisch, wie man nach [Bir06] bei umfangreichen Anwendungsfallspezifikationen vorgehen sollte: UML-Use-Case Diagramme (1) lassen sich bei den ersten Gesprächen einsetzen, bei denen die Diskussion noch nicht ins Detail geht und ein Systemüberblick von größerer Bedeutung ist. Nachdem das Grobverhalten des Systems dokumentiert ist, können die Anforderungen in Form von textuellen Use-Case-Templates (bspw. nach [Coc01]) verfeinert werden (2). An dieser Stelle würde eine automatisierte Transformation von einem Diagramm zu einem Template die Überführung erleichtern. Anschließend folgt eine tiefere Analyse der Kundenwünsche. Dafür werden automatisch generierte Templates vervollständigt sowie neue textuelle Use-Cases angelegt (3). Erweiternd zu [Bir06] streben wir an, die vorgenommenen Änderungen wieder im Use-Case-Diagramm darstellen zu können (1). Voraussetzung dafür ist eine automatisierte Rücktransformation, da ansonsten der Aufwand für die Synchronisation und Sicherstellung der Konsistenz zu groß würde.

Wissensbasierte Ansätze im Requirements Engineering verfolgen zum Teil einen ähnlichen Kreislauf. In [FKM+01] wird die Verknüpfung mehrerer Szenarien genutzt, um eine Wissensbasis aufzubauen. Im Gegensatz dazu beschränken wir uns auf das weniger ergeizige Ziel, zwei nicht ganz deckungsgleiche Sichten auf Funktionale Anforderungen miteinander zu kombinieren. Dadurch stehen mit textuellen und grafischen Use-Case Repräsentationen zwei spezielle Modellierungssprachen mit ihren jeweiligen Stärken zur Verfügung.

Auf Basis der Vorarbeiten in [Pil07] präsentieren wir in diesem Beitrag ein Konzept, mit dem es möglich ist diesen Zyklus auf der Basis eines gemeinsamen Metamodells durchzuführen: Abschnitt 2 enthält eine systematische Analyse der gemeinsamen Konzepte

von UML-Use-Case-Diagrammen und textuellen Beschreibungen. Diese wird in einem Basismetamodell dargestellt. Nach der formalen Begriffsklärung beschreiben wir in Abschnitt 3 wie die vorgeschlagenen Transformationen auf dieser Grundlage realisiert werden können.

## 2 Ermittlung gemeinsamer Elemente

Eine Transformation zwischen zwei Modellen ist dann möglich, wenn die beiden zu transformierenden Modelle über eine Menge gemeinsamer Elemente verfügen. Die Ermittlung der Gemeinsamkeiten folgt aus dem Vergleich der Elemente beider Konzepte.

Der natürlich-sprachliche und der grafische Ansatz haben ursprünglich ähnliche Wurzeln und verfügen deshalb über eine Menge ähnlicher Elemente. Im Laufe der Entwicklung wurden beide Konzepte um neue Elemente bereichert. Infolgedessen ist es nötig zu untersuchen, inwieweit die Elemente beider Konzepte ähnlich geblieben sind und worin die Unterschiede liegen.

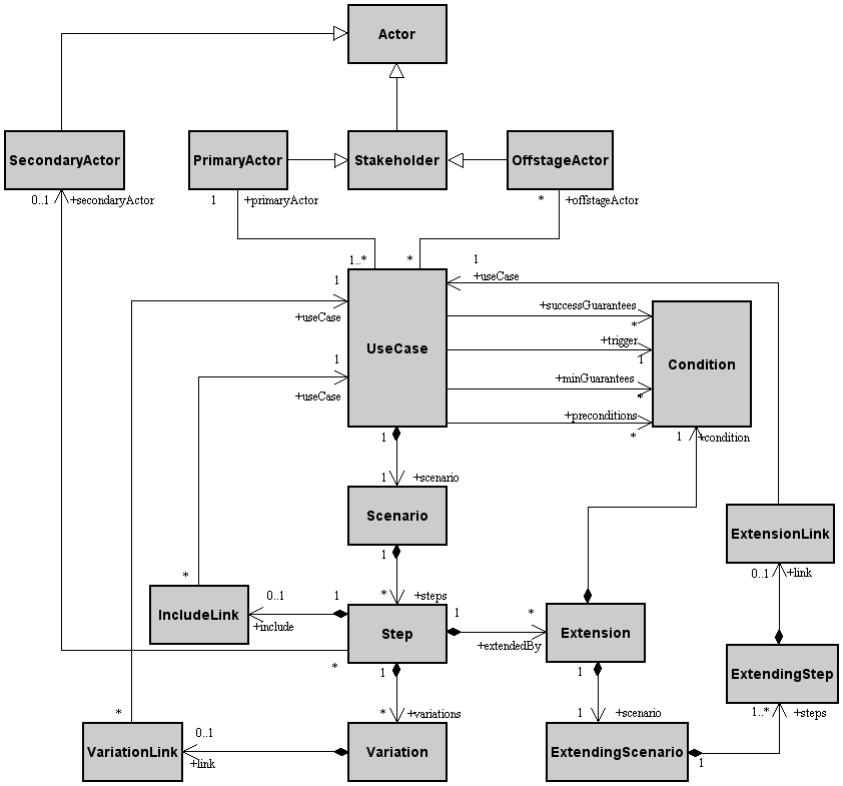


Abbildung 2: Use-Case-Metamodell

Als Ausgangspunkt der Vergleichsanalyse dient das Konzept tabellarischer Use-Cases nach [Coc1]. Im Weiteren werden Elemente dieses Konzeptes analysiert und in einem Metamodell abstrakt beschrieben. Als Notation dafür wird das UML-Klassendiagramm verwendet. Dieses Vorgehen bietet mehrere Vorteile. So wird angestrebt, eine allgemeine anwendungsunabhängige Definition tabellarischer Use-Cases festzulegen. Darüber hinaus liegt für die UML-Use-Case-Diagramme bereits ein Metamodell vor [OMG07].

Somit bildet die UML eine einheitliche sprachliche Basis, die einen Vergleich elementweise ermöglicht. Das Use-Case-Metamodell und das Use-Case-Diagramm-Metamodell werden in Abbildungen 2 und 3 gezeigt.

Das Use-Case-Metamodell wurde in Anlehnung an [Coc1] in [Lüb06, Pil07] erstellt. Ein *UseCase* ist hier das zentrale Element. Er hat eine Gruppe ihn auslösender Ereignisse und durch ihn zu gewährleistenden Garantien (*Condition*), sowie ein Szenario (*Scenario*), das aus einem oder mehreren Schritten (*Step*) besteht. Ein Schritt kann über einen *IncludeLink* auf einen anderen *UseCase* verweisen. Ein Schritt kann mehrere technische Variationen (*TechnologyVariation*) haben. Ein Schritt kann durch ein Erweiterungsszenario (*ExtendingScenario*) erweitert werden, das ebenfalls aus mehreren Schritten bestehen kann. Eine Erweiterung kann über einen *ExtensionLink* auf einen weiteren *UseCase*

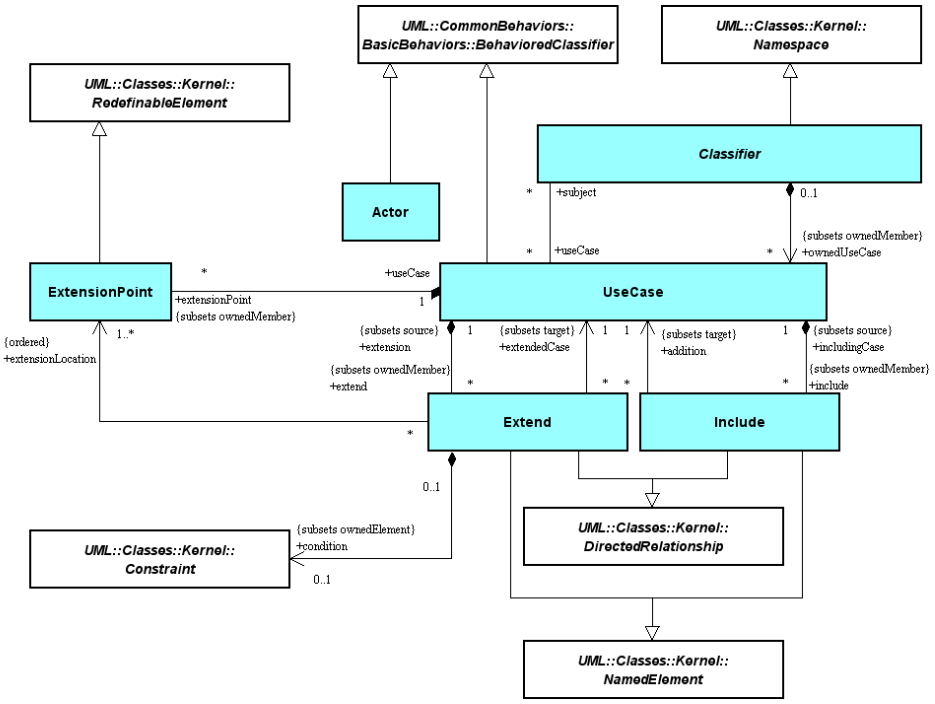


Abbildung 3: UML-Use-Case-Metamodell

zeigen. Ein *UseCase* hat eine assoziative Beziehung zu einem *Primärakteur* und mehreren *OffstageActors*.

Abbildung 3 zeigt das Use-Case-Diagramm-Metamodell, wie es in [OMG07] bereits definiert ist:

Das zentrale Element dieses Metamodells ist ebenfalls ein *UseCase*, der mehrere *«include»*- und *«extend»*-Beziehungen haben kann. Diese Beziehungen verweisen auf einen eingeschlossenen bzw. erweiterten *UseCase*. Eine Erweiterung kennt den Erweiterunspunkt (*ExtensionPoint*) des *UseCase*, in dem sie unter einer Bedingung (*Constraint*) eintritt. Als Classifier sind zwischen Akteuren (*Actor*) und *UseCases* Assoziationen, Generalisierungen und Spezialisierungen erlaubt.

Während der Analyse beider Metamodelle hat sich gezeigt, dass sich nicht alle Elemente direkt aufeinander abbilden lassen und infolgedessen ineinander nicht direkt übersetzt werden können. Als eine Abbildung wird eine Zuordnungsvorschrift bezeichnet, die jedem Element a ϵ A eindeutig ein Element f(a) ϵ B zuordnet [BSM+00]. Bereits bei dem ersten Betrachten fällt auf, dass die Anzahl der Elemente eines Template erheblich höher als die Anzahl der Diagramm-Elemente ist. Weiterhin stehen die Template-Elemente in komplizierteren Beziehungen zueinander. Einige Diagramm-Elemente können ausschließlich durch eine Gruppe von Elementen eines Template dargestellt werden.

So verfügt bspw. ein Use-Case im Use-Case-Metamodell über ein *Scenario* und Interaktionsschitte (*Step*). Im Use-Case-Diagramm-Metamodell ist lediglich ein Element *Use-*

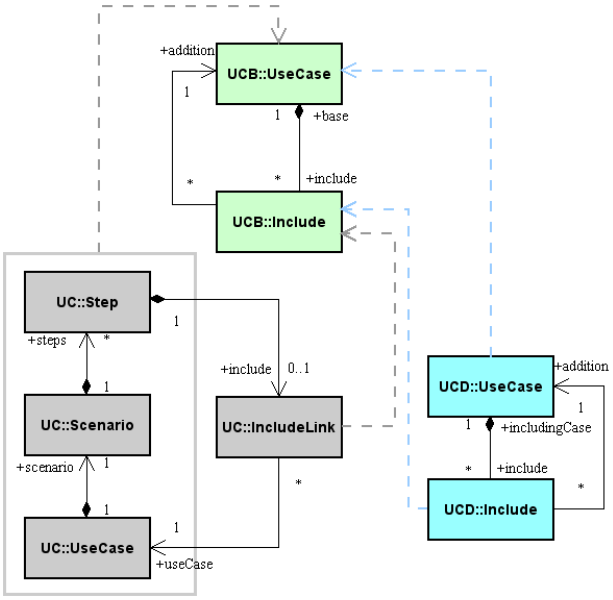


Abbildung 4. Include-Schnittmenge

*Case* zu finden. Eine *Include*-Beziehung in einem Diagramm kann aus mehreren Schritten bzw. *IncludeLinks* im Template bestehen. Ein ähnliches Verhalten stellt sich bei dem Vergleich der jeweiligen *Extend*-Beziehungen heraus. Dagegen lassen sich Technische Variationen aus dem Template im Diagramm am ehesten durch Vererbung von *UseCases* darstellen, auch weil für Templates in [Coc01] keine Generalisierung für Use-Cases vorgesehen ist (näher in [Pil07]).

Somit werden hier nicht Abbildungen, sondern Relationen zwischen Modellen betrachtet. Die Elemente bzw. ihre Beziehungen zu einander werden des Weiteren paarweise miteinander verglichen. Die gemeinsamen Elemente werden in Form eines Basismetamodells dokumentiert.

Das folgende Beispiel zeigt das allgemeine Prinzip des Vorgehens für die Ermittlung des gemeinsamen Elementes *UseCase* und seiner *«include»*-Beziehung.

Das Element *UseCase* in dem Use-Case-Metamodell enthält mehr Informationen als sein Diagramm-Verwandter. Zusammen mit einem Szenario und dessen Schritten wirkt er nach außen jedoch als ein Element. Diese Bindung ist in der Abbildung 4 durch ein Rechteck dargestellt. Zusammengenommen erfüllen diese Elemente inhaltlich die selbe Aufgabe wie ein Use-Case im Diagramm (Repräsentation eines Benutzer-Ziels). Daher kann *UseCase* als gemeinsames Element identifiziert und in das gemeinsame Basismetamodell übernommen werden.

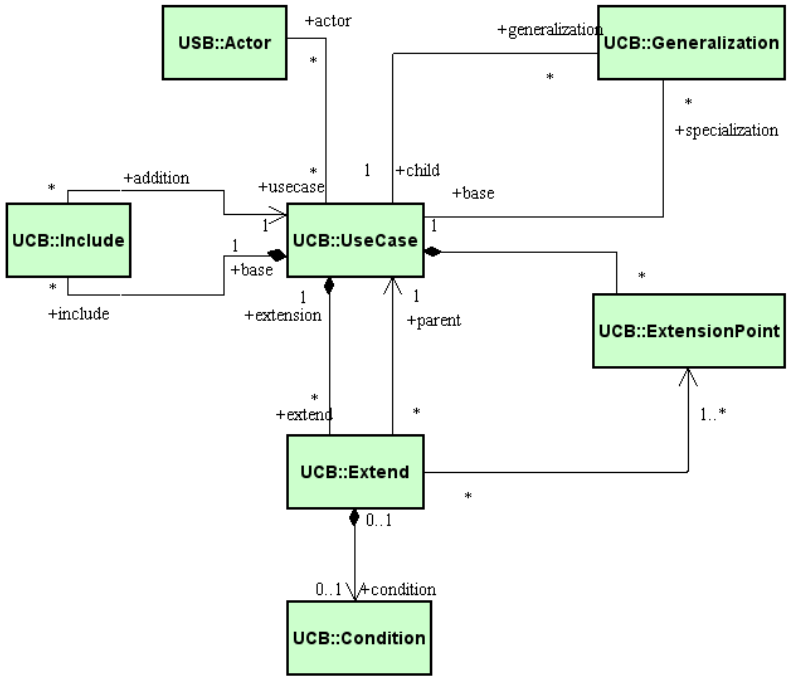


Abbildung 5. Use-Case Basismetamodell

Darüber hinaus weisen die beiden Metamodelle Verbindungen zu anderen *UseCases* auf. Das Element *UseCase* des Use-Case-Metamodells in Abbildung 2 hat dasselbe Verhalten zu dem Element *IncludeLink* wie das gleichnamige Element des UML-Use-Case-Diagramm-Metamodells zu dem Element *Include*. Ein Use-Case-Szenario kann aus mehreren Schritten bestehen. Ein Schritt kann über einen *IncludeLink* auf genau einen Use-Case zeigen. Ein Use-Case des UML-Use-Case-Diagramm-Metamodells kann ebenfalls mehrere *«include»*-Beziehungen haben. Jede dieser Beziehungen kann auf genau einen Use-Case verweisen.

Die gemeinsame Eigenschaft, durch eine Verlinkung von einem Use-Case auf einen weiteren Use-Case zu verweisen, wird ebenfalls in das Basismetamodell übernommen. Abbildung 4 zeigt das Ergebnis des Vergleiches. In der Abbildung werden folgende Kürzel verwendet: *UC* für das Use-Case-Metamodell, *UCD* für das Use-Case-Diagramm-Metamodell der UML, *UCB* für das Use-Case-Basismetamodell. Die gestrichelten Pfeile deuten an, welche Elemente Transformationen aufeinander abbilden müssen.

Zuletzt werden die einzelnen Elemente des Basismetamodells zusammengefügt. Das Ergebnis der Zusammensetzung stellt Abbildung 5 dar. Es enthält die Elemente, die sowohl für Use-Case Diagramme als auch für die Use-Case Beschreibungen relevant sind. Es fungiert als eine Art Schnittstellenbeschreibung für entsprechende Transformationen.

### 2.1 Darstellung von Use-Case-Hierarchien

Eines unserer Ziele ist es, grafische Use-Case Darstellungen aus den Use-Case Beschreibung abzuleiten. Ein Diagramm für eine komplette Spezifikation wird schnell unübersichtlich, vor allem, wenn es automatisch generiert wird. Darum bietet es sich an, für komplexe Systeme mehrere Diagramme zu bilden. Im Weiteren wird untersucht, nach welchem Prinzip Systeme im Diagramm gebildet werden und wie eine Gruppe von Templates sich einer Gruppe von Diagrammen zuordnen lässt. Unsere Lösung basiert auf der Ermittlung eines gemeinsamen Nenners – eines Ziels.

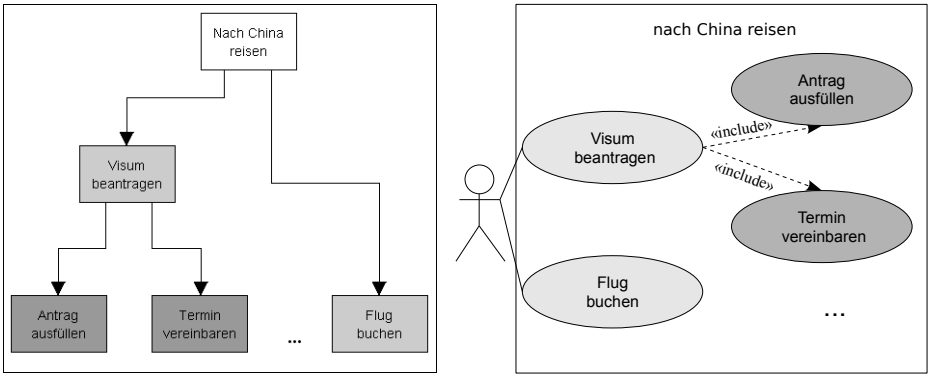


Abbildung 6. Hierarchische Erstellung der Diagramme

Das System in einem Diagramm ist diejenige Einheit, die das Verhalten, das durch die Use-Cases beschrieben wird, realisiert und anbietet [JRH+03]. Es umfasst mehrere Use-Cases. Ein Use-Case wird als Abkommen definiert, das das Systemverhalten beschreibt, mit der Absicht, ein bestimmtes fachliches Ziel (*Goal*) zu erreichen [Coc01]. Dies impliziert, dass eine Gruppe der in einem System vereinigten Use-Cases ein höheres fachliches Ziel (auch als *Geschäftsziel, Business Goal,* bezeichnet) auf dem Weg zu einem erwünschten Systemverhalten beschreibt. Die Teilsysteme werden zu einem ganzen System vereinigt, das das Hauptziel des Systemverhaltens repräsentiert. Die Idee der Zuordnung der Ziele gemäß ihrer Abstraktionsgrade ähnelt dem Konzept der Ebenen nach Cockburn [Coc01]. Neu ist jedoch die Abbildung übergeordneteter Ziele auf die Systemgrenzen ihrer Unterziele.

Basierend auf dem Konzept der Zielstrukturierung kann ein Projekt mit allen Unterzielen, die durch Use-Cases ausgedrückt werden, als eine Hierarchie betrachtet werden. Ein Use-Case kann im Diagramm als ein System dargestellt werden. Seine eingeschlossenen Use-Cases, deren erweiternde und eingeschlossene Use-Cases sowie Generalisierungen bilden den Inhalt des Systems. Solange ein Use-Case-Szenario auf einen eingeschlossenen Use-Case verweist, kann der einschließende Use-Case als ein System im Diagramm abgebildet werden. Ein Use-Case der untersten Abstraktionsebene, der über keine weiteren verlinkten Schritte verfügt, ist als System nicht mehr darstellbar. Das Projekt selbst wird im Diagramm ebenfalls als ein System dargestellt, das eine Übersicht über sämtliche Projektziele liefert. Abbildung 6 zeigt hierzu ein Beispiel: Eines der Ziele des Projektes *Urlaub* sei *nach China reisen*. Dieses Ziel kann durch das Gelingen der Unterziele *Flug buchen*, *Visum beantragen*, *Hotel buchen* und anderen erreicht werden. Im Diagramm wird das Ziel *nach China reisen* als System dargestellt. Die verlinkten Schritte des Use-Case-Szenarios *nach China reisen* werden zu den Use-Cases des Systems. Der Use-Case *Visum beantragen* kann ebenfalls in Form eines Systems dargestellt werden.

## 3 Transformation

Das Basismetamodell aus Abschnitt 2 zeigt die gemeinsamen Konzepte des UML-Use-Case-Metamodells und des Use-Case-Metamodells. Damit liefert es die Grundlage für unsere Transformationen und dokumentiert, welche Konzepte aufeinander abgebildet werden sollen. Dieser Abschnitt erläutert unser Transformationskonzept. In Abschnitt 4 wird ein Beispiel für den Einsatz der Transformationen gegeben.

Templates, wie bereits in Abbildung 2 gezeigt, beinhalten wesentlich mehr Information als Diagramme. Bei der Transformation dürfen diese zusätzlichen Informationen nicht überschrieben werden. Transformationen zwischen zwei Modellen sollten also in beide Richtungen (bidirektional) stattfinden, sowie relationale Beziehungen zwischen Modellen unterstützen. Unser Ansatz sieht einen Zwischenspeicher für die Transformation vor,

um die gemeinsamen Daten synchronisieren und die individuellen Daten wiederherstellen zu können.

Um diese Punkte gewährleisten zu können, wurde beschlossen als Transformationsspeicher ein vereinigtes Modell des Use-Case-Modells und des UML Use-Case-Diagramm-Modells zu bilden. Das vereinigte Modell beinhaltet alle Elemente beider Modelle [Pil07].

Transformationen finden zwischen dem vereinigten Modell und Teilmodellen statt, wie Abbildung 7 zeigt. Dabei werden die Elemente jeweils mit dem Zeitpunkt ihrer letzten Änderungen übermittelt. Neuere Elemente überschreiben ältere und der Zwischenspeicher verwaltet keine Historie. Damit wird auch nicht die Synchronisation konkurrierender Änderungen unterstützt, da wir davon ausgehen, das Benutzer entweder in der Diagramm- oder in der Template-Sicht arbeiten. Änderungen in einer Sicht sollen dann direkt in die andere übernommen werden.

Die Transformation kann in drei Schritte aufgeteilt werden: Im ersten Schritt werden die Daten der beiden Teilmodelle in das vereinigte Modell eingetragen. Dabei müssen die Daten untereinander synchronisiert werden. Anschließend werden im zweiten Schritt neue Diagramme im vereinigten Modell erstellt, bzw. die bestehenden aktualisiert. Im letzten Schritt wird aus dem vereinigten Modell wieder das Use-Case-Modell und das Use-Case-Diagramm-Modell generiert.

Die Verwendung eines vereinigten Modells als Transformationsspeicher bietet den Vorteil, dass die Beziehung zwischen vereinigtem Modell und den Teilmodellen (Use-Case-Modell, bzw. UML-Use-Case-Diagramm-Modell) jeweils eindeutig ist, da jedes Element aus einem Teilmodell einen Repräsentanten im vereinigten Modell hat. Bei einer Transformation zwischen den Teilmodellen wäre dies nicht der Fall (vergleiche Abschnitt 2).

Bei der Synchronisation werden die beiden Teilmodelle in das vereinigte Modell eingetragen und dabei synchronisiert. Die zu transformierenden Elemente sowie die Transformationsvorschriften lassen sich aus dem gemeinsamen Basismetamodell (Abschnitt 2) ableiten. Die folgenden Regeln werden bei der Transformation eingesetzt:

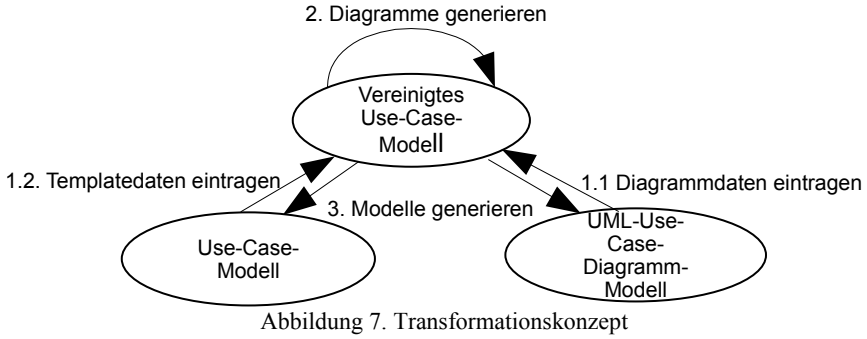


Abbildung 7. Transformationskonzept

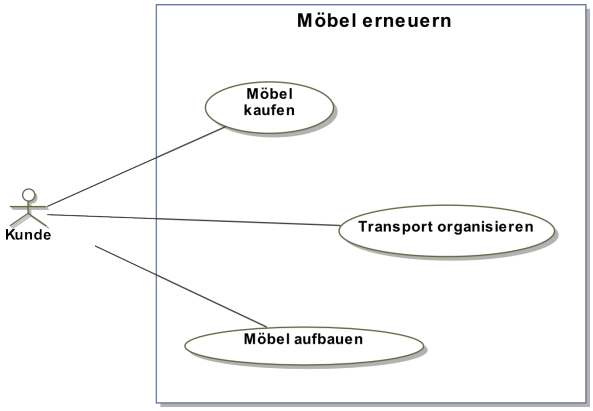


Abbildung 8. Use-Case-Diagramm: Möbel erneuern

1. Alle Elemente aus dem Use-Case-Diagramm-Modell werden in das vereinigte Modell eingetragen. Dies ist möglich, weil alle Elemente des Use-Case-Diagramm-Modells in dem vereinigten Modell zusammenhängend sind. Dadurch entsteht im vereinigten Modell ein vollständiges Bild des UML Use-Case-Diagramm-Modells. Die individuellen Elemente des Modells können als *sychronized* markiert werden.

2. Die Daten der Use-Case-Templates werden mit den Daten des vereinigten Modells abgeglichen. Elemente, die im vereinigten Modell nicht existieren, werden eingetragen und gegebenenfalls als *deleted* markiert. Die individuellen Elemente des Use-Case-Modells können in das vereinigte Modell direkt übernommen und als *sychronized* markiert werden.

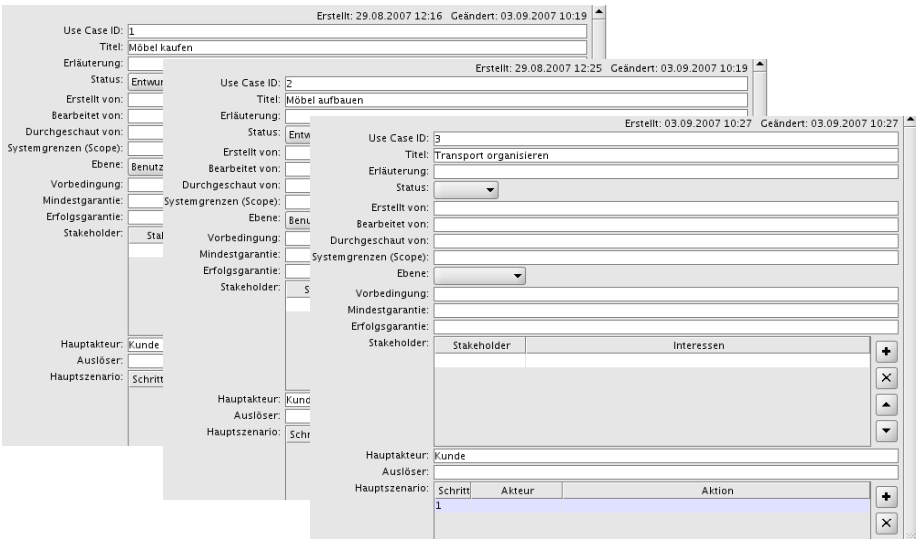


Abbildung 9. Initiale Use-Case-Definition

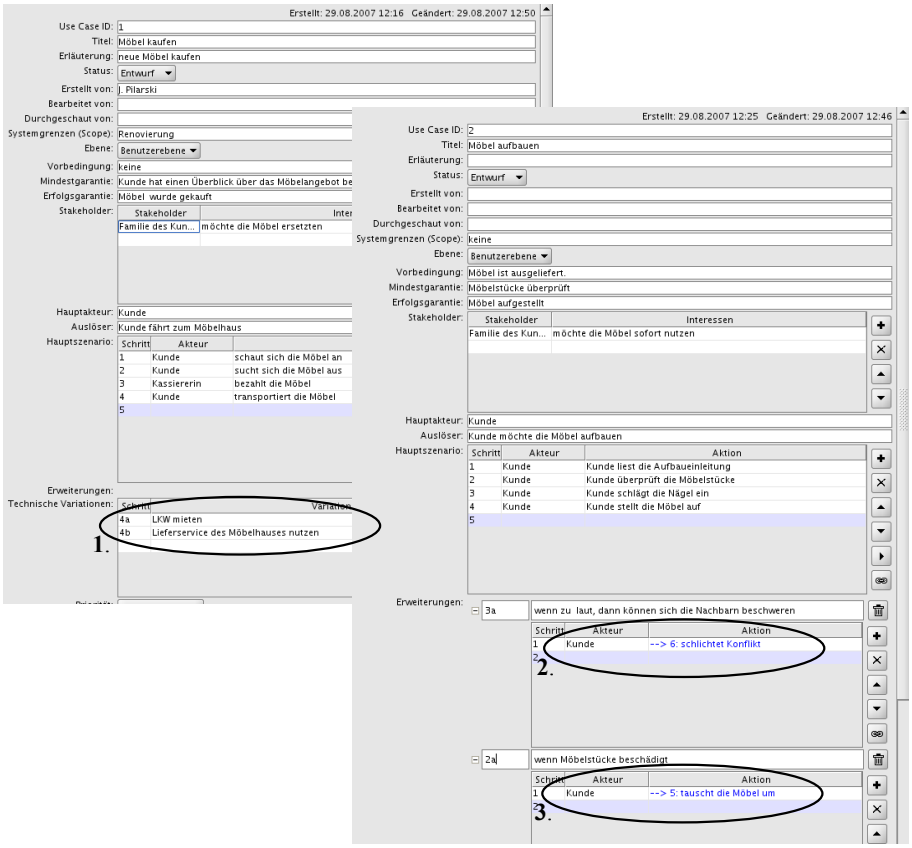


Abbildung 10. Ausschnitt einer detaillierten Use-Case Definition

3. Alle Elemente, die als *sychronized* markiert sind und deren Erstellungsdatum vor der letzten Transformation liegt, werden als *deleted* markiert. Anschließend werden alle *deleted*-Elemente gelöscht.

Damit ist die Synchronisation der Daten abgeschlossen.

Nachdem die Daten beider Modelle synchronisiert sind, müssen unter Umständen neue Diagramme erstellt bzw. aktualisiert werden. Anschließend können die Daten zurück transformiert werden. Dabei werden die Daten direkt aus dem vereinigten Modell in das Use-Case-Modell sowie Use-Case-Diagramm-Modell überführt. Dies ist ein einfacher Schritt, da eine eindeutige Zuordnung zwischen den Elementen des vereinigten Modells und den jeweiligen Teilmodellen existiert.

Dadurch, das wir immer nur wenige Use-Cases in einem Diagramm anzeigen, wird auch die Positionierung erleichtert. Neu generierte Use-Cases können in der Regel einfach an der ersten freien Stelle eingefügt werden: Auf der linken Seite, wenn sie einen Hauptak-

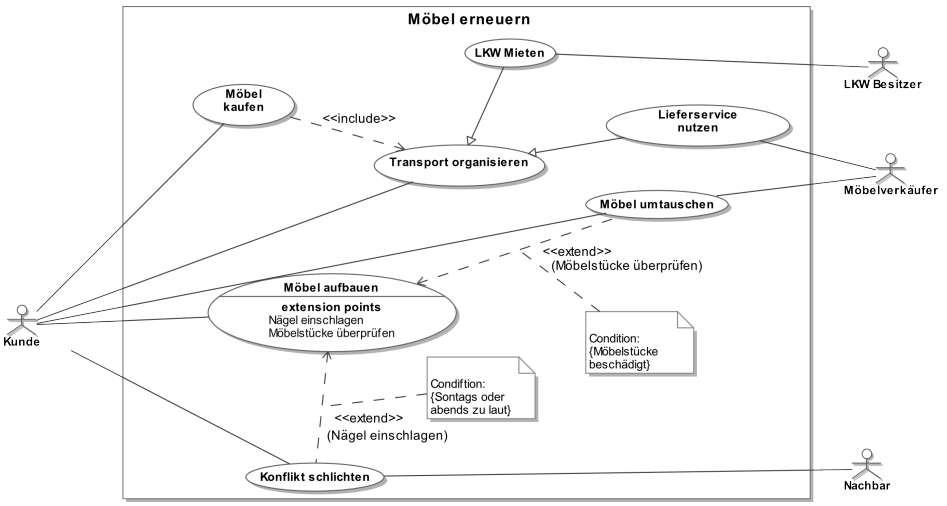


Abbildung 11. Use-Case-Diagramm: Möbel erneuern (vollständig)

teur besitzen, sonst rechts neben dem Use-Case durch den sie inkludiert sind. Benutzerdefinierte Positionen auf der Zeichenfläche gehören zu den Elementen, die von der Transformation nicht betroffen sind und bei einer Aktualisierung bestehen bleiben.

## 4 Beispiel

Die Vorzüge der Kombination von tabellarischen Use-Case-Notationen und einer graphischen Übersicht sowie ihre simultane Verwendung wurden im Allgemeinen in Abschnitt 1 beschrieben. Folgendes Beispiel erläutert den Kreislauf aus Abbildung 1 nach [Bir06] genauer.

Zu Beginn der Anforderungserhebung soll der Verlauf eines Möbeleinkaufs dokumentiert werden. Ein potentieller Kunde möchte die Möbel in seinem Haus erneuern. Sein Erfolgsszenario umfasst folgende Aktionen: *Transport* der Möbelstücke organisieren, *Möbel* in einem Einkaufszentrum *kaufen, Möbel aufbauen.* Diese Schritte sind im ersten Übersichtsdiagramm (Abbildung 8) dokumentiert.

Nachfolgend wird das Diagramm in Templates transformiert (siehe Abbildung 9). Für jeden Use-Case wird ein neues Template erstellt. Der Titel des jeweiligen Use-Case sowie der Name seines Hauptakteurs werden ins Template übernommen.

Im nächsten Schritt (Abbildung 10) wird der Inhalt der Templates verfeinert bzw. um weitere Informationen ergänzt:

Der Transport kann auf zwei Weisen stattfinden: entweder kümmert sich der Kunde um einen LKW, oder überlässt den Transport dem Möbelhaus (Ellipse 1). Diese Informationen sind als technische Variationen festgehalten. Der Use-Case *Möbel aufbauen* kann unter Bedingung *Möbelstücke beschädigt* durch den Use-Case *Möbel umtauschen* erweitert werden (Ellipse 3). Eine hohe Lautstärke beim Möbelaufbau kann in manchen Fällen Konflikte mit den Nachbarn auslösen. Das Szenario, wie diese beigeräumt werden können, wird im Use-Case *Konflikt schlichten* beschrieben (Ellipse 2).

Letztlich werden die tabellarischen Use-Cases in das Diagramm überführt. Abbildung 11 zeigt das Ergebnis dieser Transformation.

Dieses iterative Vorgehen hilft nicht nur eine Übersicht der Projektziele zu gewinnen, sondern auch alle beteiligte Akteure für weitere Gespräche zu ermitteln und ihre Zuordnung zu den entsprechenden Use-Cases zu dokumentieren.

## 5 Fazit

Dieser Beitrag präsentiert unseren Ansatz, die jeweiligen Stärken von UML-Use-Case-Diagrammen und natürlich-sprachlichen Use-Case-Beschreibungen in tabellarischen Templates miteinander zu verbinden. Dadurch wird es möglich, ständig beide Sichten auf die funktionalen Anforderungen zu haben. Details sieht man in den ausführlichen Beschreibungen, die Übersicht erhält man in den UML-Use-Case Diagrammen. Iterative Vorgehen, bei denen mehr als einmal zwischen beiden Darstellungen gewechselt wird (z.B. nach [Bir06]), werden so erleichtert: Aus einem initialen UML-Use-Case Diagramm können die Rümpfe von Use-Cases in Templates nach [Coc01] generiert werden. Änderungen an diesen textuellen Use-Case Beschreibungen können automatisch in die Use-Case-Diagramme zurücktransformiert werden. Der Vorteil dabei ist, das Änderungen immer in der jeweils angemesseneren Modellierungsart vorgenommen werden können: Struktur und Rollenzuteilung im UML-Use-Case Diagramm, lokale Details textuell über die Templates. Auf diese Weise werden zwei domänen-spezifische Sprachen für funktionale Anforderungen miteinander kombiniert.

Um dies zu erreichen, haben wir in Abschnitt 2 ein Basismetamodell verwendet, um die gemeinsamen Konzepte zu identifizieren. Am Beispiel der *Include*-Beziehung haben wir gezeigt, wie wir dabei vorgegangen sind. Auf Basis dieser konzeptionellen Schnittmenge von UML-Use-Cases und natürlich-sprachlichen Use-Cases haben wir Transformationen entwickelt. Uns war dabei wichtig, die beiden unterschiedlichen Sichten auf Use-Cases herauszuarbeiten und dabei die einschlägigen Metamodelle beizubehalten. Verfechter beider Ansätze können so von unserer Arbeit profitieren.

Abschnitt 3 und Abschnitt 4 geben den aktuellen Stand unserer Arbeit wieder. Zur Zeit evaluieren wir einen Prototypen dieser Transformationen. Dabei untersuchen wir

beispielsweise, welcher Ausschnitt des Use-Case-Modells je nach Menge und Abstraktion der Use Cases als UML-Diagramm die beste Übersichtlichkeit bietet.

Eine Evaluation im Einsatz (während einer Systemanalyse) steht noch aus. Dazu halten wir auch noch einige technische Erweiterungen für erforderlich. Als nächste Schritte wollen wir die bereits vorhandenen Transformationen kontinuierlich einsetzen, um UML-Diagramme und Use-Case-Tabellen synchron zu halten. Außerdem halten wir die Einführung von Transformationssprachen für sinnvoll, um unseren Ansatz flexibel zu halten.

Mit diesem Beitrag geben wir ein Beispiel für den Einsatz von Modellen und Transformationen in den frühen Phasen eines Softwareprojekts. Gerade das Spannungsfeld zwischen Verifizierbarkeit der Modelle und deren Lesbarkeit als Grundlage für die Validierung durch einen Kunden sind in diesen Phasen reizvoll. Wir laden Andere ein, unseren Ansatz in diesem Kontext zu diskutieren.

## Literaturverzeichnis


[Bir06] Birk, A.: Erfahrungen mit Anwendungsfallspezifikation bei der Entwicklung großer IT-Systeme. Jahrestreffen der GI-Fachgruppe Requirements Engineering, München (2006) http://www.gi-ev.de/fachbereiche/softwaretechnik/re/pages/fg_treffen/2006/birk.pdf

[BSM+00] Bronstein, I.N., Semendjajew, K.A., Musiol, G., Mühlig, H.: Taschenbuch der Mathematik. Verla Harri Deutsch (2000).

[CH03] Krzysztof Czarnecki, Simon Helsen: Classification of Model Transformation Approaches. In online proceedings of the 2nd OOPSLA'03 Workshop on Generative Techniques in the Context of MDA. Anaheim, (October 2003).

[Coc01] Cockburn, A.:Writing Effective Use Cases. Addison Wesley (2001).

[Cri06] Crisp, C.: Konzept und Werkzeug zur erfahrungsbasierten Erstellung von Use Cases., Masterarbeit, Hannover (2006).

[FKM+01] Fliedl, G., Kop, C., Mayerthaler, W., Mayr, H., Guidelines for NL-Based Requirements Specifications in NIBA, M. Bouzeghoub et al. (Eds.): NLDB 2000, LNCS 1959, pp. 251-264, 2001.Springer-Verlag Berlin Heidelberg (2001)

[JRH+03] Jeckle, M., Rupp, Ch., Hahn, J., Zengler, B., Queins, S.: UML 2 glasklar. Hanser Wissenschaft Muenchen (2003) .

[Kna07] Knauss, E.: Einsatz computergestützter Kritiken für Anforderungen. GI Softwaretechnik-Trends, Band 27, Heft 1, (2007)

[Lüb06] Lübke, D.: Transformation of Use Cases to EPC Models. Workshop EPK (2006)

[MCV05] Mens, T., Czarnecki, K., Van Gorp, P.: A Taxonomy of Model Transformations, http://drops.dagstuhl.de/opus/volltexte/2005/11/ (2005) .

[OMG07] Unified Modeling Language: Superstructure , Version 2.1.1 . Object Management Group, http://www.omg.org, 15.05.07 (2001).

[Pil07] Pilarski, J.: Konzept und Implementierung von Transformationen zwischen Use Case Diagrammen und tabellarischen Darstellungen, Bachelorarbeit, Hannover (2007).

[Ros99] Rosenberg, D.: Use Case Driven Object Modeling with UML : Addison Wesley Longman, Inc (1999).

[Rup06] Chris Rupp: Requirements-Engineering und Management, 4. Auflage. Hanser (2007)

[Stö05] Störrle, H. : UML 2 für Studenten. PEARSON Studium (2005).

[SV05] Stahl, Th., Völter, M.:Modellgetriebene Softwareentwicklung. dpunkt.verlag (2005).

# Entwicklung und Evaluierung einer domänenspezifischen Sprache für SPS-Schrittketten

Bastian Cramer Dennis Klassen Uwe Kastens
University of Paderborn
Department of Computer Science
Fürstenallee 11, 33102 Paderborn, Germany
{bcramer,dennis.klassen,uwe}@uni-paderborn.de

**Abstract:** Domänenspezifische Sprachen mit passenden Entwurfs- und Transformationswerkzeugen unterstützen Anwender in speziellen Gebieten ihre Entwürfe in Implementierungen umzusetzen. Sind solche Sprachen visuell, so können auch graphische Notationen aus dem Anwendungsgebiet übernommen werden, um die Akzeptanz der Sprache zu verbessern. In diesem Artikel berichten wir über den Entwurf, die Implementierung und Evaluation einer visuellen Sprache, die im industriellen Umfeld zur Steuerung von Robotern einer Produktionsstrecke eingesetzt wird. Der Einsatz eines Werkzeugsystems zur Sprachimplementierung (DEViL [Sch06a]) ermöglichte, dass mit akzeptablem Aufwand Prototypen generiert und Sprachvarianten erprobt wurden.

## 1 Einleitung

Domänenspezifische Sprachen eignen sich sehr gut, um Entwürfe in speziellen Anwendungsgebieten treffend zu formulieren und mit passenden Werkzeugen in Programme oder andere formale Strukturen zu transformieren. Visuelle Sprachen sind für solche Zwecke besonders nützlich, da Strukturen und Zusammenhänge sehr anschaulich dargestellt werden können. Außerdem können sie gut an etablierte Notationen des Anwendungsgebietes angepasst werden und verkleinern dann die Distanz zwischen den Vorstellungen des Anwenders und den Sprachelementen.

In diesem Artikel berichten wir über eine visuelle Sprache, die wir für die Firma Robert Bosch GmbH entwickelt haben, um damit die Steuerung von Robotern zu formulieren, die Elektromotoren montieren. Mit Entwürfen in dieser Sprache sollen Entwickler, die eine Motoren-Produktionsstrecke konzipieren, ihre Vorstellungen zum Ablauf der Montage an Programmierer vermitteln, welche die Steuerung der Roboter implementieren. Die Randbedingungen für die Entwicklung der Sprache werden in Abschnitt 2 dargestellt.

Die von uns entwickelte Sprache soll eine informelle Notation ablösen, die bisher für diesen Zweck eingesetzt wurde. Damit soll erreicht werden, dass Entwürfe treffender beschrieben und Entwicklungszyklen abgekürzt werden. Der Entwurf der Sprache musste Konzepte der bisher verwendeten Notation gut aufnehmen und Erweiterungen und Präzisierungen sensibel einfügen, damit die Sprache von beiden Entwicklergruppen akzeptiert wurde. Im Abschnitt 3 stellen wir Sprachkonstrukte exemplarisch vor und berichten über die Evaluation der Sprache als akzeptanzsteigernde Maßnahme.

Um mit einer domänenspezifischen Sprache praktischen Nutzen zu erzielen, werden sprachspezifische Werkzeuge benötigt. Wir haben für diese Sprache u.a. einen visuellen Strukturediteor, Übersetzer für SPS nach XML und einen Codegenerator für SPS implementiert und in den Entwicklungsprozess für die Produktionsstrecken eingebettet. Abschnitt 4 vermittelt einen Eindruck dieser Aufgaben jenseits der Sprachentwicklung im engeren Sinne.

Neben den grundsätzlichen Aufgaben zur Implementierung von Sprachen erfordert eine visuelle Sprache in erheblichem Maße zusätzliches technisches und konzeptionelles Wissen sowie weiteren Entwicklungsaufwand. Wenn man trotzdem noch Entwurfsvarianten erproben will, um die Akzeptanz zu verbessern, ist der Aufwand für eine manuelle Implementierung nicht akzeptabel. Wir haben für die Realisierung dieser Sprache das von uns entwickelte Werkzeugsystem DEViL [Sch06a] eingesetzt. Es generiert vollständige Sprachimplementierungen mit visuellen Strukturediteoren als Frontend aus Spezifikationen hohen Abstraktionsniveaus [Sch06b]. Im Abschnitt 5 vermitteln wir einen Eindruck von dem Einsatz und der Wirksamkeit des Systems.

Einige Hinweise auf verwandte Arbeiten und eine Zusammenfassung beschließen den Artikel.

## 2 Vorstellung der Domäne

Die Firma Robert Bosch GmbH produziert an ihrem Standort in Bühl kleine Elektromotoren in großen Stückzahlen für vielfältige Einsätze in Kraftfahrzeugen. Auf langen Produktionsstrecken mit vielen einzelnen Stationen bauen Roboter die Motoren schrittweise zusammen und prüfen sie. Für jeden neuen Motortyp muss die Produktionsstrecke neu eingerichtet werden. Dazu wird für jede Station der Aufbau des Roboters, die Zuführung des Materials und die Weiterleitung der Produkte entworfen. Darauf abgestimmt werden die Arbeitsschritte des Roboters geplant und in speicherprogrammierbaren Steuerungen (SPS [Aue89]) programmiert.

Bei der Entwicklung der motorspezifischen Produktionsstrecke kooperieren zwei Gruppen von Entwicklern mit unterschiedlichen Aufgaben: Die hier „Projektleiter“ genannte Gruppe entwirft den mechanischen Aufbau jeder Station und den Einsatz der Roboter, Greifarme, Schub- und Dreheinrichtungen, Taster und Sensoren. Die zweite Gruppe, hier als „Programmierer“ bezeichnet, entwickelt Programme zur Steuerung der Komponenten mit speicherprogrammierbaren Steuerungen. Die beiden Entwicklergruppen kommunizieren über den geplanten Ablauf der Produktionsschritte der Stationen. In mehreren Iterationen werden die Pläne verfeinert, in Programme umgesetzt, an den Geräten getestet und verbessert. Die Pläne der Abläufe, sogenannte „Schrittketten“ werden in einer speziellen Notation beschrieben, zwischen den Entwicklergruppen ausgetauscht, fortgeschrieben und dokumentiert. Die Notation enthält graphische Elemente zur Skizzierung von Ablaufschritten und textuelle Annotationen für technische Angaben. Abbildung 2 und 3 zeigen Beispiele dazu.

Der Entwicklungsprozess einer Produktionsstrecke wird durch die Projektleiter mit der Entwicklung der mechanischen Konstruktion begonnen (Abbildung 1, Position 1). Aus den dabei entstehenden Dokumenten wird mit einem Generator ein Grundprojekt erstellt

(Abbildung 1, Position 2). Dabei wird für jedes aktive Element der Produktionsstrecke eine sogenannte Schrittkette festgelegt, die den Ablauf des Elements beschreiben soll. Die Programmierer extrahieren die einzelnen Schrittketten (Abbildung 1, Position 3). Mit dem Schrittketten-Dokumentationssystem, das in der Abteilung verwendet wird, können die vorhandenen Schrittketten in festgelegter Form als Schrittketten-Ablaufzettel ausgedruckt werden (Abbildung 1, Position 4). Abbildung 3 stellt beispielhaft Bearbeitungszustände eines Ablaufzettels dar und soll einen Eindruck über die bei Bosch verwendeten Dokumente zur Beschreibung der Schrittketten verschaffen. Eine genauere Ansicht für die Schritte und Konditionen ist in der Abbildung 2 zu finden. Diese Vorlagen werden durch Projekleiter, gemäß den geplanten Bewegungsabläufe, händisch bearbeitet (Abbildung 3) und an die Programmierer übergeben (Abbildung 1, Position 5). Programmierer setzen die Modelle in SPS-Code um und können daraus SPS-Programme erzeugen oder aktuelle Ablaufzettel zur Kontrolle erneut ausdrucken, bis der gewünschte Ablauf der Maschine erreicht wurde und das SPS-Program vollständig ist.

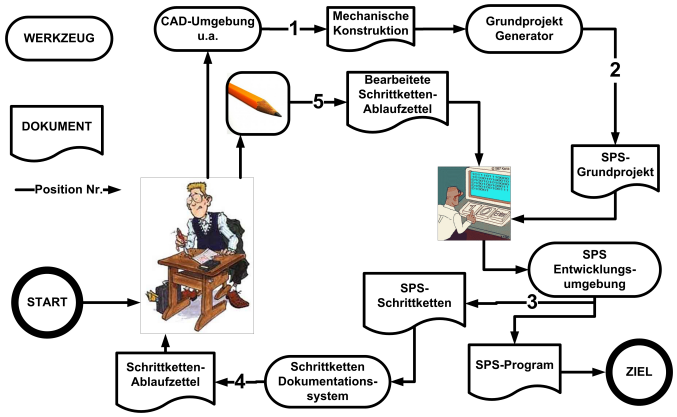


Abbildung 1: Entwicklungsprozess vor der Einführung der DSL

In der Abbildung 2 sind drei Schritte genauer dargestellt. Als Beispiel betrachten wir den Schritt N16, in dem die Aktion „Parkpos.anfahren“ durchgeführt wird. Das grau unterlegte Kästchen stellt den Schritt dar. Darüber sind zwei vertikale Linien angeordnet, diese repräsentieren eine ODER-Verknüpfung zwischen drei Konditionen, unter denen dieser Schritt geschaltet wird. Die untereinander aufgelisteten Variablennamen sind UND-verknüpft. Unter dem Schritt können mehrere Aktionen aufgeführt sein, die in dem Schritt durchzuführen sind, bevor die Konditionen für den nächsten Schritt geprüft werden. Eine Schrittkette kann über hundert Schritte besitzen und erstreckt sich dann über mehrere Seiten.

Die Schrittketten-Ablaufzettel sind während des gesamten Entwicklungsprozesses im Umlauf (Abbildung 1, Punkt 7,9). Alle Änderungen an dem Ablauf oder den Maschinen fordern stets den Einsatz des Schrittketten-Ablaufzettels. Häufig sind die Änderungen nur gering, wie z.B. das Entfallen einiger Schritte, die Änderung der Schrittreihenfolge oder eine einfache Namensänderung der Schritte. Die Abbildung 3 zeigt einen Auszug aus einem laufenden Projekt mit durchgeführten Korrekturen.

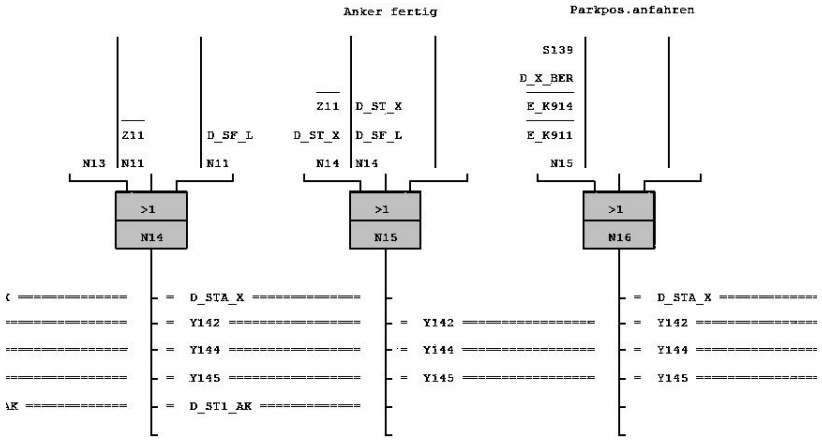


Abbildung 2: Schritte, dargestellt mit dem Schrittketten-Dokumentationssystem

Die größten Engpässe sind die Kommunikation zwischen den Entwicklergruppen und die dabei entstehenden Dokumente. Die Beschreibungsmöglichkeiten sind in dieser Notation sehr begrenzt und passen nicht mehr zum aktuellen SPS-Standard. Das führt immer wieder zu ungenauen oder missverständlichen Beschreibungen und erfordert zusätzliche Iterationen (Abbildung 1, Punkt 4-9), bis das erstellte SPS-Programm dem gewünschten Bewegungsablauf entspricht.

Diese Situation sollte im Rahmen des hier beschriebenen Projektes durch eine domänenspezifische Sprache verbessert werden.

# 3 Sprachentwurf

Wie aus dem vorhergehenden Abschnitt hervorgeht, wird eine domänenspezifische Sprache benötigt, welche den Aufgaben angemessen ist. In diesem Abschnitt wird die Vorgehensweise für einen solchen Sprachentwurf vorgestellt. Die SPS-Programmiersprachen spielen dabei eine wesentliche Rolle, da die DSL Programme für SPS beschreiben soll. Dafür wird zu Beginn eine Übersicht über die SPS-Sprachen gegeben. Anschließend wird der Sprachentwurf und die dazu verwendeten Elemente aus der vorhandenen Notation beschrieben. Im Anschluss daran werden die Evaluierung und die daraus resultierende DSL beschrieben.

## 3.1 SPS-Konzepte als Grundlage der DSL

Die DSL soll speziell für die Entwicklung der Schrittketten eingesetzt werden, wofür nur bestimmte Teile der SPS-Sprachen verwendet werden. Der Standard IEC 61131 [Inf03] vereinigt fünf Sprachen für SPS. Die Ablaufsprache (AS) dient der Gliederung und Orga-

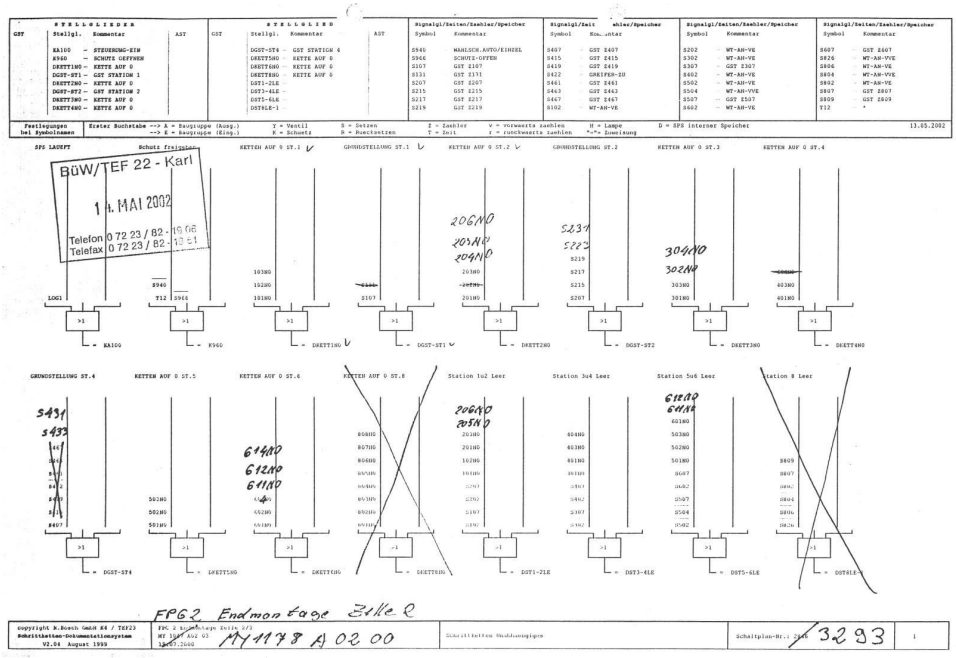


Abbildung 3: Schrittketten-Ablaufzettel mit Änderungen

nisation einer Schrittkette und bildet einen gerichteten Graph aus einer Menge von Schritten und Transitionen (Abbildung 2). Die Schrittaktionen und die Transitionsbedingungen können in den weiteren SPS-Sprachen beschrieben werden. Zu ihnen gehören zwei textuelle Sprachen (Tabelle 1), Anweisungsliste (AWL), Strukturierter Text (ST), und zwei Sprachen mit visuellen Elementen (Tabelle 1), Kontaktplan (KOP), Funktionsbaustein (FBS).

Die AWL fasst mehrere Anweisungen zusammen. Operanden können Werte zugeordnet werden, z.B. bei der Deklaration von Variablen. Die ST besteht aus Ausdrücken, die einen Wert liefern. Dazu gehören z.B. mathematische Ausdrücke oder boolesche Funktionen. Außerdem ist das Beschreiben verschiedener Konstrukte wie Auswahl- oder Wiederholungsanweisungen möglich.

Mit den Sprachen KOP und FBS wird ein Netzwerk in Form von Linien und Blöcken dargestellt. Diese Notation ist den früher benutzten Schaltplänen aus der Elektrotechnik nachempfunden und beschreibt Stromfluss-Pläne.

FBS stellt eine Menge von Funktionsbausteinen dar. Durch Verknüpfung mehrerer Funktionsbausteine miteinander können komplexe Funktionen realisiert werden. Dazu gehören diverse mathematische Funktionen oder vollständige Motorsteuerungen.

Einen Eindruck der Sprachen vermittelt Tabelle 1. In allen vier Beispielen wird eine UND-Funktion mit Eingängen % X2,5 und % X2,3 und dem Ausgang TRANS78 dargestellt.

| **ST** | **AWL** |
|---|---|
| `:= %IX2.4 & %IX2.3` | `LD %IX2.4`<br>`AND %IX2.3` |
| **FBS** | **KOP** |
| `+-----+`<br>`%IX2.4---\| & \|-- TRAN78`<br>`%IX2.3---\| \|`<br>`+-----+` | `\| %IX2.4 %IX2.3 TRAN78 \|`<br>`+---\|\|-----\|\|------()---+`<br>`\|  \|` |

Tabelle 1: SPS-Sprachen für verschiedene Zwecke: ST, AWL, FBS und KOP

Mit dem Expertenwissen über SPS kann mit der Entwicklung der domänenspezifischen Sprache begonnen werden. Dazu wird ermittelt, was genau die neue DSL beschreiben soll. Im Vorfeld dieser Arbeit ist eine Skizze (Abbildung 4) entstanden, welche die grundlegenden Strukturen zeigt und die Darstellungswünsche der Benutzer aufgenommen hat. Daraus ist ersichtlich, dass die neue Sprache alle Sprachelemente einer Schrittkette in einer grafischen Übersicht darstellen sollte. Ein weiterer Anhaltspunkt für die visuelle Sprache sind die Schrittketten-Ablaufzettel (Abbildung 3).

Bereits an dieser Stelle sind Gemeinsamkeiten und Gegensätze zwischen vorhandenen Werkzeugen und den Wünschen der Entwickler erkennbar. Z.B. sollte der Fluss der Schrittketten von links nach rechts beibehalten werden, entsprechend der bereits vorhandenen Notation in den Schrittkettenablaufzetteln (Abbildung 3, 4). Geändert werden sollte die Beschreibung der Transitionen und Bedingungen: Auf den Schrittkettenablaufzetteln (Abbildung 3) werden Bedingungen modelliert, unter welchen der dazugehörige Schritt geschaltet wird; in der neuen Sprache (Abbildung 4) werden Transitionen modelliert, die zum Verlassen des Schrittes führen.

Als erstes werden die zu modellierenden Elemente der SPS-Sprachen ermittelt. Dafür wird das Expertenwissen aus dem Umfeld benötigt, unterstützend dazu werden die vorhandenen Projektdokumente untersucht. Sobald der Satz der zu modellierenden Elemente feststeht, kann dieser in einer abstrakten Struktur für DEViL [Sch06a] implementiert werden. Aus dieser abstrakten Struktur ist DEViL bereits in der Lage einen Strukturediteor mit einer simplen Baumdarstellung zu generieren. Bereits hier sind mit den generierten Strukturediteoren Evaluierungstests möglich. Dabei kann untersucht werden, ob sich bereits vorhandene Projekte mit den Strukturbäumen ausdrücken lassen. Nachdem die abstrakte Struktur feststeht, kann mit der Modellierung der visuellen Komponenten der DSL begonnen werden.

### 3.2 Evaluierung der DSL

In diesem Projekt haben wir der Evaluierung besondere Beachtung geschenkt. Dazu wurden verschiedene Methoden und Vorgehensweisen angewendet, mit denen unterschiedliche Effekte evaluiert wurden, z.B. Kommunikation zwischen den Benutzern, Verständlichkeit der Modellierungssprache usw.. Die dabei erzielten Ergebnisse sind z.T. benutzer-

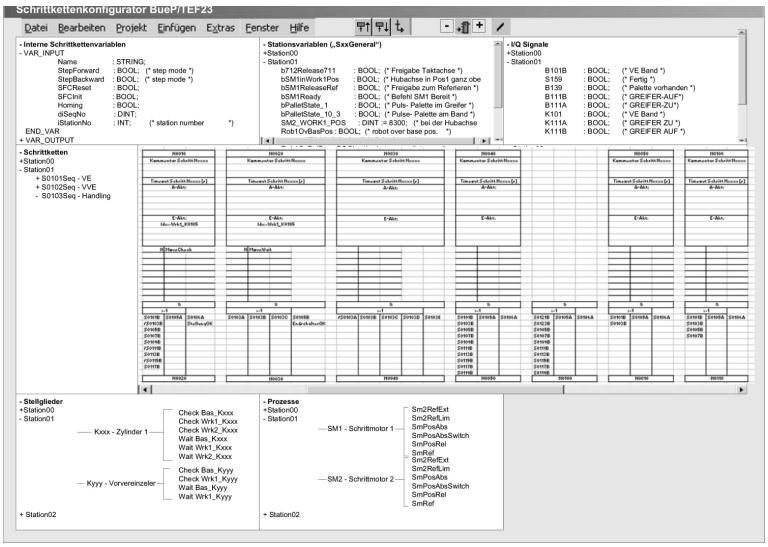


Abbildung 4: Erste Skizze des Schrittkettenkonfigurators

abhängig und müssen richtig interpretiert werden. Außerdem sollte beachtet werden, dass sich die Wünsche und Meinungen der Entwickler im Verlauf der Evaluierung ändern, so ist z.B. in der Abbildung 4 zu sehen, dass die Benutzer alle Elemente zum Konfigurieren einer Schrittkette in einer einzigen Sicht dargestellt wünschen. Dieser Wunsch hat sich im Verlauf der Evaluierung geändert, da realistische SPS-Projekte in einer Sicht umfangreich und zu unübersichtlich werden.

Neben dem Expertenwissen und Kenntnissen über die Domäne muss für die Evaluierung der DSL der Kenntnisstand der Benutzer ermittelt werden. Für diesen Zweck sind Interviews und Beobachtungen vor Ort notwendig. Die Ergebnisse können für die Dialogmodellierung verwendet werden, die beschreibt, wie sich das System zu verhalten hat und welchen Arbeitsfluss die Benutzer beim Erledigen ihrer Aufgaben haben. Außerdem kann mit Hilfe der Interviews herausgearbeitet werden, wie die Benutzungsschnittstelle auszusehen hat. Der Aufbau einer Grundakzeptanz ist für die weitere Entwicklung enorm wichtig. Erst wenn diese erreicht wurde, kann mit der Evaluierung der DSL fortgefahren werden.

Die Evaluierung wurde parallel zum Prototyping und der Implementierung durchgeführt. Bereits eine Skizze auf Papier (Abbildung 4) verkörpert den ersten Prototyp und die ersten Schritte der Evaluierung. Das “Rapid Prototyping” wurde durch den Einsatz des Generators DEViL ermöglicht. Da sich sowohl einzelne Teile der Benutzungsschnittstelle als auch Sprachelemente separiert evaluieren lassen, kann dadurch den Benutzern eine Vielzahl von Alternativen angeboten werden. Ein vielfältiges Angebot ist notwendig, da die Benutzer nicht den Überblick über die verschiedenen Modellierungstechniken besitzen. Das bedeutet, die Benutzer können nicht direkt mitteilen, welche Modellierungsmethode sie bevorzugen, sondern können aus den präsentierten Methoden eine Auswahl treffen und

diese bewerten.

Die Auswahl der Benutzer für die Evaluierung beschränkt sich auf die Mitarbeiter in der TEF-Abteilung bei Bosch in Bühl. Es ist wichtig, dass beide Entwicklergruppen, Programmierer und Projektleiter, bei der Evaluation vertreten sind. Denn sie haben unterschiedlich tiefe Kenntnisse von SPS-Konstrukten, welche die DSL stark prägen.

- Die Programmierer sind im Umgang mit SPS erfahrener, das sie bereits SPS-Software entwickelt und mit grafischen Editoren gearbeitet haben.
- Die Projektleiter programmieren nicht selbst. Ihr Bezug zu der SPS entsteht über den mechanischen Ablauf der Maschinen und die Schrittketten-Ablaufzettel.

Aus den Methoden, die für solche Evaluierungen einschlägig sind [SCK07], haben wir in diesem Projekt folgende eingesetzt:

**Interview** In einem Interview werden Benutzer aus verschiedenen Entwicklergruppen einzeln interviewt. Das Gespräch dient dazu, die Meinungen und die Aussagen der Benutzer zu dem System zu sammeln und zu untersuchen. Vorteilhaft ist, dass auf die Benutzer einzeln eingegangen werden kann und somit unvorhergesehene Fragen und Beanstandungen geklärt werden können.

**Kontrolliertes Experiment** Bei einem kontrollierten Experiment wird den ausgewählten Benutzern eine Aufgabe gestellt, die sie mit dem System bewältigen müssen. Dabei wird beobachtet, welche Entwicklergruppen in welchen Bereichen Schwierigkeiten haben.

**Feld-Beobachtung** Eine der wichtigsten Evaluierungsmethoden für diese Arbeit ist der Feldtest. Dabei werden die Benutzer während der Arbeit mit dem neuen System beobachtet. Im Vergleich zum kontrollierten Experiment ist diese Methode praxisrelevanter und kann unvorhergesehene Ergebnisse liefern.

Als eine wirksame Evaluierungsmethode hat sich eine Mischung aus Gruppeninterview, Feld-Beobachtung und kontrolliertem Experiment herausgestellt. Dabei wurden mehrere Benutzer aus beiden Gruppen zu einer Präsentation eingeladen. Der Ablauf der Versuche wurde dann wie folgt durchgeführt: Eine Schrittkette sollte zwischen mehreren Benutzern diskutiert und gegebenenfalls editiert werden. Die Änderungen an der Schrittkette, die im Verlauf des Tests entstanden, wurden z.B. von einem Projektleiter durchgeführt. Dabei konnten die Testpersonen ihre persönliche Meinung einbringen. Durch solche Experimente konnten die Schwierigkeiten in der Benutzung der Prototypen des Strukturediteors herausgestellt und beseitigt werden. Die eindeutigsten und aussagekräftigsten Beobachtungen konnten bei der Kommunikation zwischen den Entwicklergruppen gemacht werden. Durch diese Experimente konnte die Darstellung der Benutzungsschnittstelle und der Softwarebeschreibungssprache an die Kommunikation zwischen den Entwicklergruppen angepasst werden. Die unten aufgeführten Evaluierungstests wurden sowohl mit einzelnen Personen, als auch, wie hier beschrieben, in einer Gruppe abwechselnd durchgeführt.

Der erste Test sollte die Aufteilung der visuellen Sichten in den verschiedenen Prototypen herausstellen. Dabei galt es herauszufinden, wie gut sich die Benutzer in dem Strukturediteor zurecht finden und welche Informationen in einer Sicht gebündelt werden oder eine eigene Sicht erhalten sollten. Dazu wurden verschiedene Prototypen des Strukturediteors

bereitgestellt, in denen Ausschnitte aus den SPS-Programmen, formuliert in der experimentellen DSL, in gemischten Gruppen aus Programmierern und Projektleitern diskutiert wurden.

Die Ergebnisse dieser Tests wurden fortlaufend analysiert, um neue Prototypen bereitzustellen. Durch eine große Prototypenanzahl konnte eine für die Benutzer optimale Benutzungsschnittstelle geschaffen werden, die sich recht deutlich von den Anfangswünschen der Benutzer (Abbildung 4) unterscheidet. Dabei entstanden fünf Sichten, in die sich der Strukturedi tor aufteilt: Hauptsicht, Schrittketten, Schrittketten-Variablen, Schrittketten- Aktionen und globale Variablen.

Der zweite Test bezog sich auf die Darstellung der Schritte und der Schrittketten. Das Ziel war, herauszufinden, wie verständlich die Schrittketten und Schritte mit der neuen DSL dargestellt werden. Dabei mussten die Testpersonen die Informationen aus einer vorgegebenen Schrittkette notieren. Zusätzlich konnten die Benutzer in einem Interview die Eindrücke über die Darstellungsart und die Aufteilung der Informationen auf dem Bildschirm mitteilen.

Die Programmierer und die Projektleiter verfolgen unterschiedliche Ziele und haben sehr unterschiedliche Blickwinkel auf die Schrittketten. Daher konnten die aus diesem Test erhaltenen Ergebnisse zwischen den verschiedenen Entwicklergruppen nicht verglichen werden. Die Projektleiter hatten, wie erwartet, anfänglich Schwierigkeiten zwischen den Konditionen und Transitionen zu unterscheiden. Nur eine von fünf Testpersonen aus dieser Gruppe hat die Veränderung und die Bedeutung sofort erkannt. Mit dem gleichen Ergebnis konnten die Schrittaktionen von den Eingangs- und Ausgangsaktionen nicht differenziert werden. Dieses Ergebnis konnte für diese Entwicklergruppe als positiv verbucht werden, da sich die Benutzer auf die für sie relevanten Aufgaben konzentrieren konnten. Aus der Gruppe der Programmierer gab es lediglich drei Testpersonen. Ohne große Mühe wurde die Darstellung von dieser Entwicklergruppe sofort erkannt und vollständig interpretiert.

In einem weiteren Test wurde die Darstellung der visuellen DSL untersucht. Nachdem in vorangegangenen Tests die Bedeutung der verschiedenen Aktionen in einem Schritt und die Bedeutung der Transitionen geklärt wurden, konnten weitere Untersuchungen dieser Elemente durchgeführt werden. Dabei konnte speziell auf die Darstellungen der einzelnen Elemente wie Aktionen in einem Schritt, Transitionen usw. eingegangen werden. Den Testpersonen wurden dann z.B. verschiedene Transitionen vorgegeben, die interpretiert und beschrieben werden mussten. Die Darstellung der Transitionen wurde sowohl von den Projektleitern als auch von den Programmierern als intuitiv und leicht verständlich eingestuft.

Abschließend wurden mehrere Tests des gesamten Werkzeugs durchgeführt. Dazu mussten die Benutzer vorhandene Schrittketten öffnen, bearbeiten und anschließend abspeichern. Die Ergebnisse aus diesen Tests sollten Schwächen des Systems aufzeigen. Nach jedem Durchlauf wurden die Beanstandungen ausgebessert und ein neuer Testdurchlauf gestartet.

Für die Konstrukte der unterschiedlichen SPS-Sprachen wurden durch gezielte Evaluierung geeignete Darstellungen gefunden. Dabei können Schrittketten mit sämtlichen beinhalteten Elementen in einer Ansicht dargestellt werden, sodass ein Benutzer eine Schritt-

kette mit demselben Strukturedítor in derselben Ansicht editieren kann. Die Evaluierung ging so weit, dass z.B. selbst die Darstellung der Kommentare untersucht wurde. Diese sind für die Benutzer beim Erledigen ihrer Aufgaben an einigen Stellen elementar wichtig und an anderen Stellen störend und überflüssig.

**Schrittketten, Schritte:** Schritte werden in einer Schrittkette nach wie vor in einem Fluss von links nach rechts dargestellt und bestehen aus drei Teilen. Im oberen Bereich, wie in der Abbildung 5 zu sehen, befindet sich der sogenannte Schrittkopf, dem der Benutzer die auszeichnenden Informationen über einen Schritt entnehmen kann. Die dargestellten Informationen sind: Name des Schrittes, Kommentare und Timer-Attribute. Hinweisend werden ganz oben die Schrittnamen der Vorgängerschritte eingeblendet, wie in der Abbildung 5 sichtbar. Weitere Schrittattribute werden in der Ansicht nicht dargestellt, da sie bei dem Entwurf und der Übersicht für die Benutzer zweitrangig sind. Diese können jedoch durch ein zusätzliches Dialogfenster erreicht werden. Im mittleren Teil sind die Aktionen des Schrittes untergebracht, darunter sind die Transitionen aufgeführt.

**Schritt-Aktionen** In der Abbildung 5 sind einige Aktionen in einem Schritt zu sehen. Auch wenn die Schrittaktionen in AWL anders als die Eingangs- und Ausgangsaktionen in ST sind, wurde die Darstellung annähernd gleich gehalten, um ein einheitliches Bild der logisch zusammen gehörenden Aktionen zu bekommen. Besonders sei auf den Schritt N0025 in der Abbildung 5 hingewiesen. In der Ausgangsaktion dieses Schrittes ist eine bedingte Anweisung dargestellt. Solche Darstellungen sind mit den Schrittketten-Ablaufzetteln nicht möglich.

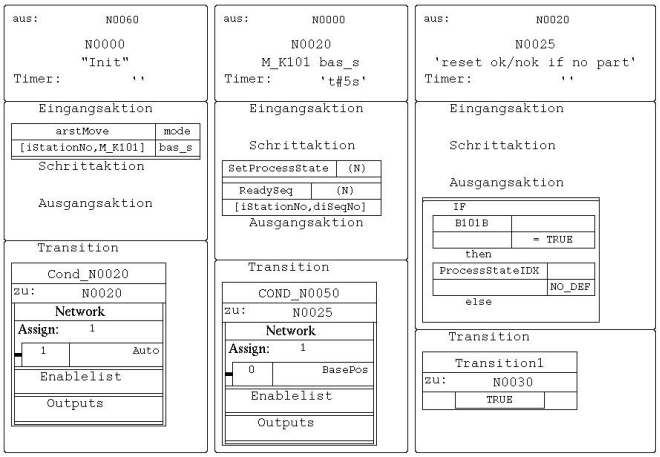


Abbildung 5: Schritte/Aktionen/Transitionen

**Transitionen** Eine Transition beschreibt die Bedingung und den Übergang zum nächsten Schritt. Wenn ein Schritt mehrere Transitionen besitzt, werden diese nebeneinander aufgereiht. Eine Transition ist, wie der Schritt, in drei Teile gegliedert. Als erstes ist der Name der Transition platziert, gefolgt von dem Zielschritt. Unten werden die Transitionsbedingungen dargestellt.

# 4 Werkzeugeinbettung

Die neue domänenspezifische Sprache wurde im Rahmen des Projekts in einem visuellen Strukturedtor (Abbildung 7) realisiert, der mit einem Code Generator ausgestattet wurde. Damit die Eingliederung in die vorhandene Entwicklungsroutine möglich ist, wurde zusätzlich ein Übersetzer für SPS nach XML implementiert. Die Abbildung 6 gibt eine Übersicht des Entwicklungsprozesses mit dem neuen Werkzeugsystem im Vergleich zu dem ursprünglichen Prozess in Abbildung 1. Nach der Einführung des neuen Systems beginnt der Entwicklungsprozess, wie bereits in Abschnitt 2 beschrieben, mit der mechanischen Konstruktion (Abbildung 6, Position 1) der Produktionsstrecke. Sobald die benötigten Schrittketten festgelegt wurden, kann mit ihrer Entwicklung, in dem Schrittkettenkonfigurator, angefangen werden (Abbildung 6, Position 2). In dem Schrittkettenkonfigurator können Benutzer die Schrittketten beliebig mit Schritten, Aktionen und Transitionen vervollständigen (Abbildung 6, Position 3,4).

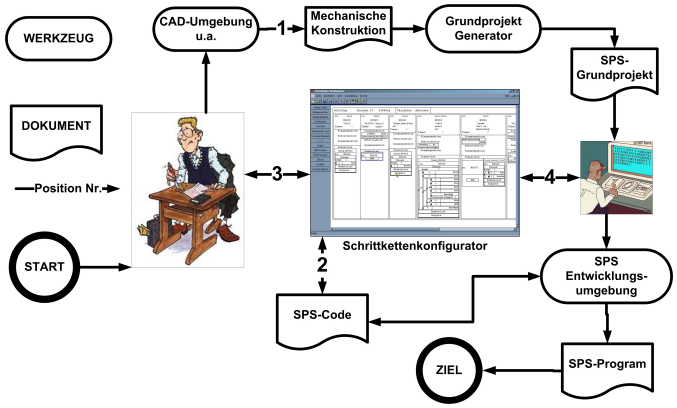


Abbildung 6: Veränderter Entwicklungsprozess mit dem Schrittkettenkonfigurator

Die Projektleiter beschreiben im Wesentlichen, wie viele Schritte benötigt werden und welche Aktionen in diesen Schritten enthalten sein müssen. Die genaue Parametrisierung der Aktionen ist die Aufgabe der Programmierer. Weiterhin beschreiben die Projektleiter einfache Transitionen, die ebenfalls durch Programmierer nachträglich vervollständigt werden. Alle Informationen, die das Wissen der Projektleiter über SPS-Programmierung übersteigen, werden in Form einfacher Kommentare in die betroffenen Module eingetragen. Enormer Vorteil in der Kommunikation ist, dass die Benutzer bei einer gemeinsamen Schnittstelle über die gleichen Elemente und Darstellungen sprechen.

Nachdem ein erster Entwurf durch die Projektleiter erfolgt ist, übernehmen die Programmierer die Beschreibung. Für die Einbettung der Schrittketten in das Grundprojekt, in der Entwicklungsumgebung IndraWorks von Rexroth [Ind], wird aus dem Schrittkettenkonfigurator direkt SPS-Code generiert. Somit kann SPS-Code zwischen dem Schrittkettenkonfigurator und IndraWorks ausgetauscht werden. Der zweite nennenswerte Verbesserungspunkt im Entwicklungsprozess ist, dass die Programmierer die erhaltenen Schrittketten

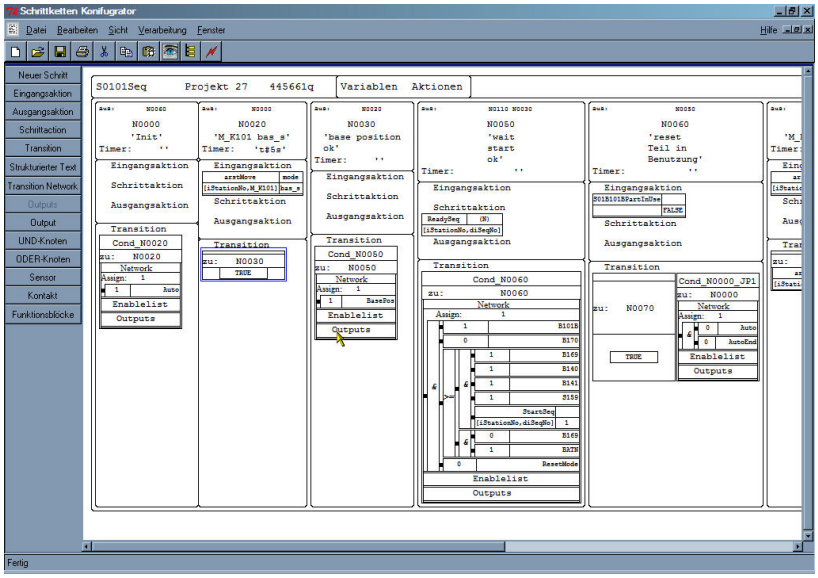


Abbildung 7: Schrittkettenkonfigurator

nicht neu erfassen müssen, sondern dort anknüpfen können, wo die Projektleiter aufgehört haben. Verpackt in einem Strukturedi tor ist die domänenspezifische Sprache perfekt in den Entwicklungsprozess integriert und unterstützt die Entwickler angemessen.

## 5 Generatorsystem

Für die Entwicklung der visuellen DSL und ihrer Implementierung haben wir den Generator DEViL [Sch06a] eingesetzt. DEViL basiert auf den Konzepten des Übersetzergeneratorsystems Eli [Eli] und ist der Nachfolger von VL-Eli [KS02]. DEViL generiert aus Spezifikationen hohen Abstraktionsniveaus vollständige graphische Strukturedi toren, die aus einer Multifensterumgebung bestehen und alle Eigenschaften heutiger visueller Editoren wie Laden und Speichern von Dokumenten im XML-Format, Kopier- und Einfüge- sowie Undo/Redo-Operationen, Drucken von Dokumenten und Suchen bieten.

Zentrale Datenstruktur der generierten Editoren ist ein abstrakter Strukturbaum. Er wird aus einer Spezifikation generiert, die einem klassenbasierten Entwurf ähnelt. In der Spezifikationssprache sind die Definition von Attributen mit vor- oder selbstdefinierten Typen, Referenzen auf Sprachkonstrukte sowie Aggregation von Sprachelementen und Mehrfachvererbung erlaubt. Aus solch einer Sprachspezifikation kann DEViL bereits einen rudimentären Strukturedi tor mit einer Baumansicht generieren. Dadurch konnten wir schon mit unterschiedlichen Strukturen unserer DSL experimentieren, bevor wir die Visuelle Repräsentation der Sprachelemente entworfen hatten.

Um eine visuelle Sprache zu definieren, benutzt DEViL das Konzept der visuellen Muster.

Visuelle Muster sind in visuellen Sprachen häufig vorkommende Repräsentationen wie Listen, Formulare, Mengen oder Tabellen. Der Sprachentwickler kann aus einer großen Bibliothek von Mustern wählen, diese anpassen und dann einfach auf das Sprachmodell anwenden (Abbildung 8). Alle für die graphische Darstellung der Sprachelemente notwendigen Implementierungen leistet das System dann automatisch. Soo konnten wird mit recht geringem Aufwand die graphischen Darstellungen von Sprachelementen ändern, wenn die Evaluation dazu den Anlass gab. Des Weiteren kann der Sprachentwickler eine Code Erzeugung für den Struktureditor definieren. Hier stehen ihm alle Möglichkeiten des Eli Übersetzergeneratorsystems zur Verfügung, insbesondere die Unparser. Mit diesen Mitteln haben wir die Erzeugung von SPS-Code realisiert. Die von DEViL generierten Struktureditoren erlauben es dem Benutzer, immer syntaktisch korrekte Programme zu erzeugen. Um komplexere semantische Analysen durchzuführen, stehen viele weitere vordefinierte Funktionen bereit und über eine Sprachschnittstelle können C/C++ bzw. Tcl-Funktionen eingebunden werden.

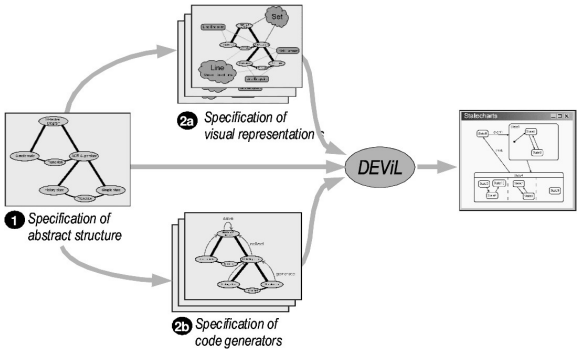


Abbildung 8: Konzept des DEViL-Systems

## 6 Verwandte Arbeiten

Es existiert eine Vielzahl verschiedener Werkzeugsysteme und Struktureditoren für SPS-Software, die unter anderem auch über visuelle Darstellungen, visuelle Sprachen und Code Generatoren verfügen. Auch gibt es verschiedene Werkzeugsysteme, mit denen Struktureditoren aus Spezifikationen generiert werden. Im Rahmen dieses Papiers können wir lediglich eine kleine Übersicht darüber geben.

Wie schon aus dem SPS-Standard hervorgeht, gibt es visuelle Notationen für SPS und umfangreiche Entwicklungsumgebungen für SPS-Software, wie z.B. IndraWorks [Ind]. Diese Programmierumgebung bietet leider keine Möglichkeit die Darstellungen der Sprachen zu verändern. Da diese Umgebung für professionelle Softwareentwickler ausgerichtet ist, ist sie für unerfahrene Benutzer nicht gut geignet. Durch einfachere und besser angepasste Notationen können auch die unerfahrenen Benutzergruppen erreicht werden.

Werkzeugsysteme zur Entwicklung visueller Struktureditoren sind z.B. GenGed [Bar98],

MetaEdit+ [Con02] und das in diesem Projekt verwendete DEViL, so wie Eclipse basierende Plattformen wie oAW oder GMF [BSM+03]. Das System GenGed z.B. basiert auf einer als Graph modellierten abstrakten Struktur. In DEViL dagegen werden Sprachkonstrukte durch attributierte Grammatiken spezifiziert und als attributierte Bäume mit Querreferenzen repräsentiert, was eine entscheidende Rolle bei der Strukturmodellierung und den Sicht-Definitionen spielt. Einen ähnlichen Ansatz der Modellierung findet man beim MetaEdit+. Dieses System hat leider eine eingeschränkte Flexibilität bezüglich grafischer Repräsentationen. DEViL bietet dagegen spezialisiertere und stärker parametrisierbare grafische Fähigkeiten. So gibt es z.B. Spezialmuster für Bäume VPExplorerTree bzw. VPTree [Sch06b] mit Anbindung an das Graphlayout Werkzeug Dot und diverse Listenmuster. Außerdem sind Funktionen eingebettet um z.B. die Überlappungsfreiheit zu kontrollieren. Die Spezifikationen des Modells mit EMF ist grob mit DEViL vergleichbar. GMF z.B. unterscheidet analog zu DEViL zwischen der semantischen und der editierbaren Struktur, DEViL bietet jedoch weitere Spezialsprachen um die Spezifikation noch weiter zu vereinfachen

Die Entwicklung einer visuellen Sprache [SK03] wurde bereits mit dem VL-Eli [KS02] System, dem Vorgänger von DEViL, in anderen Projekten erfolgreich durchgeführt. So wurde in Zusammenarbeit mit Sagem Orga SIMtelligence Designer/J [SPKF02], eine visuelle Sprache zur Anwendungsentwicklung für SIM-Karten, entwickelt.

# 7 Zusammenfassung

Das Ziel dieser Arbeit war die Modernisierung und Verbesserung des Entwicklungsprozesses von SPS-Schrittketten in industrieller Umgebung. Dazu wurde eine visuelle domänenspezifische Sprache mit dem Werkzeugsystem DEViL für zwei, miteinander kommunizierende, Entwicklergruppen entwickelt und speziell an deren Aufgaben angepasst.

Für den zu entwickelnden Strukturedtor wurden anhand eines strukturierten Entwicklungsprozesses erst die Situation bei Bosch analysiert, sowie die Aufgaben der Benutzer und das Umfeld der Aufgaben untersucht. Die Benutzungsschnittstelle und die DSL wurden für genau die durchgeführten Aufgaben ausgelegt. Durch die verwendeten Werkzeuge bestand der Hauptteil dieser Arbeit aus der Evaluierung der visuellen Sprache und der Benutzungsschnittstelle. Durch wiederholte kontrollierte Experimente, Interviews und Gruppenbesprechungen mit den Benutzern wurde eine für beide Entwicklergruppen geeignete visuelle DSL sowie eine übersichtliche und aufgabenkonforme Benutzungsschnittstelle entworfen. Der dabei entwickelte Schrittkettenkonfigurator entspricht den Vorstellungen der Benutzer. Durch die Einführung des Schrittkettenkonfigurators wird der Entwicklungsprozess der Schrittketten beschleunigt und vereinfacht.

Durch den in dieser Arbeit entwickelten Schrittkettenkonfigurator wird dem Werkzeugsystem DEViL und dem Vorantreiben dieser Arbeit bei Bosch sehr großes Interesse entgegen gebracht. Obwohl der Schrittkettenkonfigurator alle zu Beginn dieser Arbeit gestellten Anforderungen erfüllt, wurde erst während der Entwicklung die Ausbaufähigkeit des Systems deutlich. So wurden nachträglich zum konformen Ausdrucken der Schrittketten aus dem Schrittkettenkonfigurator zusätzliche Module in DEViL eingebaut und stehen somit auch

allen weiteren mit DEViL generierten Editoren zur Verfügung. Über die systematische Evaluierung des DEViL-Systems und mit ihm erzeugter Sprachimplementierungen wird in [Sch06b] und [SCK07] berichtet. Der Struktureditor soll in Zukunft weitere Funktionen bekommen und z.B. für Analysen und diverse Simulationszwecke erweitert werden. Einige solcher Erweiterungen sind bereits geplant, z.B. eine visuelle Simulationssicht für Schritte und Aktionen. Das gesamte Konzept hat bei Bosch großen Zuspruch gefunden.

## Literatur


[Aue89] A. Auer. *SPS - Aufbau und Programmierung 2., überarb. Aufl.* Huethig, 1989.

[Bar98] Roswitha Bardohl. GenGed: A Generic Graphical Editor for Visual Languages based on Algebraic Graph Grammars. In *1998 IEEE Symp. on Visual Lang.*, Seiten 48–55, September 1998.

[BSM+03] Frank Budinsky, David Steinberg, Ed Merks, Ray Ellersick und Timothy Grose. *Eclipse Modeling Framework*. Addison Wesley, aug 2003.

[Con02] MetaCase Consulting. *MetaEdit+ User's Guide*, 2002. http://www.metacase.com/.

[Eli] *Eli Website*. http://www.uni-paderborn.de/fachbereich/AG/agkastens/eli_home.html.

[Ind] *IndraWorks von Rexroth*. http://www.boschrexroth.com/.

[Inf03] Deutsche Kommission Elektrotechnik Elektronik Informatik. *IEC 61131-3 Speicherprogrammierbare Steuerungen Teil 3*, 2003. Programmiersprachen Deutsche Fassung EN 61131-3:2003.

[KS02] Uwe Kastens und Carsten Schmidt. VL-Eli: A Generator for Visual Languages. In *Proceedings of Second Workshop on Language Descriptions, Tools and Applications (LDTA'02)*, number 2027 in Electronic Notes in Theoretical Computer Science, Grenoble, France, 2002. Band 65, Elsevier Science Publishers.

[Sch06a] Carsten Schmidt. DEViL Homepage, 2006. http://ag-kastens.uni-paderborn.de/forschung/devil/.

[Sch06b] Carsten Schmidt. *Generierung von Struktureditoren für anspruchsvolle visuelle Sprachen*. Dissertation, 2006.

[SCK07] Carsten Schmidt, Bastian Cramer und Uwe Kastens. Usability Evaluation of a System for Implementation of Visual Languages. *In Symposium on Visual Languages and Human-Centric Computing, IEEE Computer Society Press*, Seiten S. 231– 238, September 2007.

[SK03] Carsten Schmidt und Uwe Kastens. Implementation of visual languages using pattern-based specifications. *Software - Practice and Experience*, 33:1471–1505, Dezember 2003.

[SPKF02] Carsten Schmidt, Peter Pfahler, Uwe Kastens und Carsten Fischer. SIMtelligence Designer/J: A Visual Language to Specify SIM Toolkit Applications. In *Proceedings of Second Workshop on Domain Specific Visual Languages (OOPSLA 2002)*, Seattle, WA, USA 2002, 2002.