RZ 3621 (# 99631) 07/18/2005
Computer Science 10 pages

Research Report

A Systematic Approach to Designing Model Transformations

Jochen M. Küster, Ksenia Ryndina and Rainer Hauser

IBM Research GmbH
Zurich Research Laboratory
8803 Rüschlikon
Switzerland

{jku,ryn,rfh}@zurich.ibm.com

LIMITED DISTRIBUTION NOTICE

This report will be distributed outside of IBM up to one year after the IBM publication date.
Some reports are available at http://domino.watson.ibm.com/library/Cyberdig.nsf/home.

A Systematic Approach to Designing Model Transformations

Jochen M. Küster, Ksenia Ryndina and Rainer Hauser
IBM Zurich Research Laboratory
8803 Rüschlikon
Switzerland
{jku,ryn,rfh}@zurich.ibm.com

Abstract

Design and implementation of model transformations is one key prerequisite for the vision of model driven development to become true. However, model transformations are quite different from other software artifacts since a model transformation design typically consists of a set of transformation rules instead of object-oriented models such as class diagrams and statecharts. Therefore traditional software engineering approaches are not directly applicable for the development of model transformations. Nevertheless, in the line of existing software engineering practice, model transformations must be developed in a systematic way in order to ensure their quality. In this paper, we present a systematic method for designing model transformations, based on a case study experience. In our approach, the developer initially produces high-level transformation rules and then refines these to a complete low-level transformation design according to detailed guidelines. Following our method allows systematic and iterative development of model transformations, resulting in quality implementations.

1. Introduction

The objective of the Model Driven Architecture (MDA) initiative [1] is to make development of large and complex software systems more efficient. This objective is achieved through increased use of modelling during all software development activities. One important aspect of MDA is the widespread use of model transformations between models constructed during system development.

Existing examples of model transformations include refactoring [23, 25], transforming business process models into Business Process Execution Language (BPEL) [10] and establishing consistency of Unified Modeling Language (UML) models by transforming them into a formal language [9]. Currently, a great deal of research is focused on the expression of model transformation designs and appropriate tool support for model transformation developers. This has led to the Query/Views/Transformation (QVT) proposal by the Object Management Group (OMG) [21], which aims to provide a standardized framework for model transformation development.

Many existing model transformation approaches favor that transformations are captured in a rule-based way by a set of transformation rules (see VIATRA [5], GReAT [13], UMLX [26], BOTL [4] and work by Milicev [17]). Although the actual form of rules and rule application varies, the general idea is that a rule consisting of a left side and right side describes how a model is supposed to be transformed. The left side is to be matched in the source model and then this part is replaced by the right side of the rule. Defining rules and in which order they should be applied can be considered as designing a model transformations.

In practice, software developers need more than a notation and tool support for capturing model transformation designs. A systematic process guiding the developer from the initial phases of model transformation development all the way through to implementation is required. Traditional software engineering approaches are not directly applicable, because model transformations differ significantly from other software artifacts. While conventional software designs usually comprises class and behavioral diagrams, a model transformation design typically consists of a set of transformation rules. The primary challenge for the developer is concerned with how to systematically design transformation rules, ensuring their completeness and correctness. Completeness ensures that all valid input models are considered by the transformation. Correctness includes syntactic correctness of the transformed models and also the preservation of semantic properties during a transformation. Further, systematic testing of the final transformation implementation is also required. However, despite the demand for software engineering approaches tailored to the development of model transformations, almost no research work has been reported in this area.

In this paper, we present a solution to the systematic design of model transformations. We distinguish between high-level, low-level design and design validation of model transformations. Within high-level design, the main ingredients of the transforma-

tion rules are captured informally and different cases for each rule are discovered. These cases are divided into supported and unsupported cases. In low-level design, the supported cases are grouped to low-level design rules which are subsequently expressed in a model transformation language. Design validation focuses on validation of syntactic and semantic correctness of the model transformation developed. The evaluation of our approach has been done using a significant model transformation that removes unstructured cycles from business process models as a case study.

The paper is organized as follows. First, we briefly describe the case study that we use for illustrating our technique. Then, we provide an overview of the development process for model transformations. Thereafter we elaborate both on high-level and low-level design of model transformations.

2. Case Study: Transforming Business Process Models

Business processes are usually represented with graphical models using notations such as UML 2.0 *activity diagrams* [20]. Figure 1 shows a simple example of a business process. The business process shows simple action nodes $A - G$, a structured activity node (SAN) K containing actions E and F. Further, the process contains the usual start and end nodes, fork, join and decision nodes. Nodes within a business process are connected by edges via pins, which are organized into pin sets (also known as parameter sets). This business process also makes use of advanced features of UML activity diagrams such as a broadcasting, shown with the broadcasting node sending a *Cycle ended* signal and the corresponding accept event node labelled with *Receive Cycle ended.*

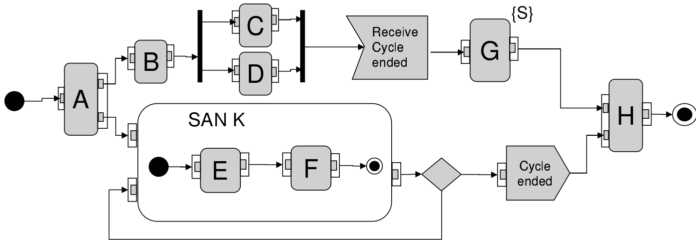

Figure 1. Example of a business process

The semantics of activity diagrams is based on token flow, the basic idea of which is that an action receives tokens on its input pins and it starts to execute once all the pins in a pin set have tokens available. During the execution of an action, the tokens from the complete input pin set are removed, an instance of the action is created and executed, after which the results are placed on the pins of one of the output pin sets. Tokens are then released to the edges connected to that output pin set and the flow continues through the rest of the process in this manner. In the example shown in Figure 1, the business process begins execution at the start node that releases a token to action A. An instance of this action is created, executed and then releases two tokens to its two output pins. Afterwards, both an instance of action B and SAN K are executed. Execution of the SAN leads to creation of instances of E and F. After execution of the SAN K, a cycle may take place depending on evaluation of the decision condition. Once the cycle has ended, a broadcast action sends a *Cycle ended* signal to the other parts of the business process. After receipt of this signal at the accept event node, the token flow is passed to the action node G. Action G has the special feature that it has a streaming output pin, i.e. tokens are released already while it is executing. The business process terminates once a token reaches the global termination node.

If business processes are to be deployed, their design models have to be transformed into an executable specification in a notation such as BPEL [18]. Edges in activity diagrams can be used to create so-called unstructured cycles or arbitrary jumps of control flow, such as in the example. However, BPEL only supports structured flow constructs such as while-loops. As a solution to this problem, a model transformation removing all unstructured cycles from activity diagrams needs to be performed before a BPEL specification can be generated. For the basis of this transformation, a traditional compiler theory technique of *T1-T2 analysis* can be adapted to such normalization of activity diagrams [14].

Originally, T1-T2 analysis was introduced in order to detect whether a directed graph is reducible [11]. It is based on two reduction rules [2]:

T1: If n is a node with a loop, i.e. there exists an edge $n \rightarrow n$, delete that edge.

T2: If there is a node n that is not the initial node and that has a unique predecessor m, then m may consume n by deleting n and making all successors of n (including m, possibly) be successors of m.

These rules can be applied in any order until the so-called "limit flow graph" has been reached and none of the rules are applicable anymore. If the limit graph consists of a single node, the original graph is reducible, and otherwise the graph is irreducible.

The task of developing the required cycle-removal transformation is to make the T1-T2 analysis applicable to activity diagrams. This means that the rules have to be adapted to preserve the behavior captured in the activity diagram and must be described in terms of the syntax of activity diagrams, defined by the metamodel shown in Figure 2. Note that this figure shows a UML activity diagrams metamodel that was abridged and adapted for our purposes. The most essential elements of activity diagrams are ActivityNodes and ActivityEdges that connect the nodes. We distinguish between three different types of ActivityNodes, ControlNodes, Actions and Pins. Actions are either simple such as CallAction or structured actions called SANs. A BroadcastAction and an AcceptAction can be used to broadcast and receive signals, respectively. Pins are connection points for ActivityEdges and are grouped into PinSets (instead of ParameterSets as in the original metamodel).

The abstract syntax of activity diagrams described by the metamodel is further restricted by means of constraints. For example, source and target ActivityNodes of an ActivityEdge must be contained in the same SAN as the edge. A given model is syntactically correct only if it conforms to the metamodel and satisfies all these constraints.

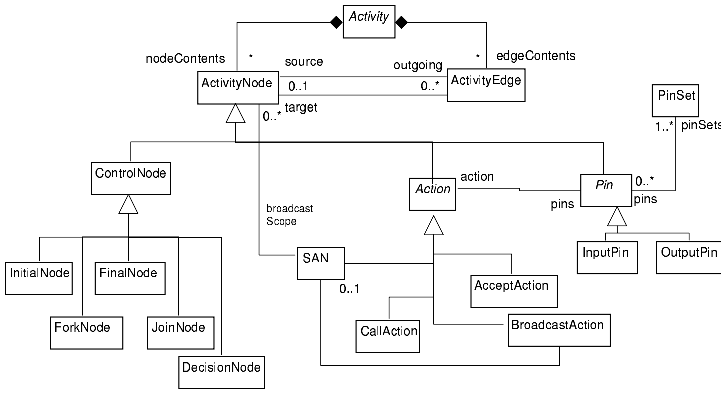

Figure 2. An abridged metamodel for activity diagrams

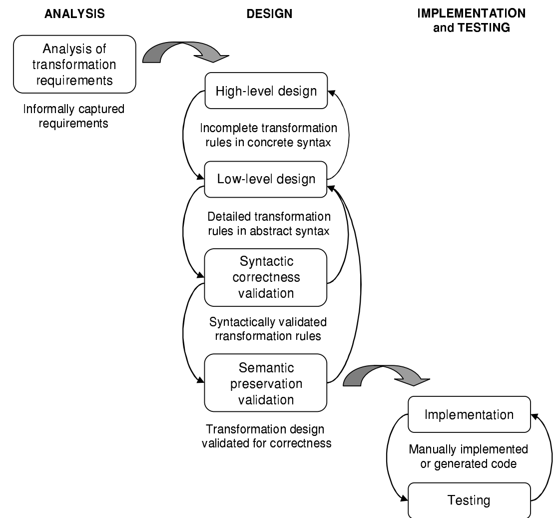

Figure 3. Process for model transformation development

3. Development Process Overview

In this section, we first define the requirements for a development process for model transformations and then provide an overview of this process. We assume that a model transformation design consists of a set of transformation rules.

Requirements for a development process for model transformations can be derived from existing requirements for software processes [7]. Without going into detail, the process should have clearly defined phases with clearly defined outputs and it should allow for incremental and iterative development. Another set of requirements arises from the artifacts that are to be produced by the model transformation development process. These requirements include that a complete and correct transformation implementation should be produced, satisfying the initial transformation requirements.

Similar to foundational ideas from software engineering [22], the development process for model transformation rules can be split into the phases of requirements analysis, design, implementation and testing as shown in Figure 3.

Requirements analysis is needed to derive the key requirements for a model transformation. Such requirements include the source and target language of the model transformation. We suggest that requirements are captured informally. Using the requirements, a high-level design consisting of transformation rules is created as the first step of the design phase. Each transformation rule is modelled semi-formally using diagrams in the concrete syntax of the transformed models. At this stage, transformation rules do not explicitly consider all the possible arrangements of elements in the source model but rather focus on the main idea of each rule, abstracting from its details. Thereafter, each high-level rule is refined to produce detailed transformation rules in the abstract syntax based on the metamodels of the underlying modeling languages. Such low-level rules capture each possible scenario that the transformation is meant to handle. Syntactic and semantic validation of design rules is necessary before they are implemented. Validated rules can be either used as a basis for manual implementation or automated generation of transformation code. Testing the generated or manually implemented transformation code is required to ensure that it meets the original requirements. Note that later activities may require the reiteration of earlier activities, e.g. syntactic validation may require changes in the low-level design.

In comparison with traditional object-oriented software development, the development process for model transformations has specific characteristics. This is especially true for the design phases of development, where a design for a model transformation consists of transformation rules rather than object-oriented diagrams. On the contrary to the existing design processes such as the Rational Unified Process [12] that elaborate on how to for example derive statecharts from sequence diagrams, a method is needed which allows for systematic and complete design of transformation rules. Further, testing of a model transformation has similarities with compiler testing [3], because a model transformation consisting of transformation rules must fulfill similar requirements. It should transform all valid input models to valid output models as a compiler should transform all valid programs into a machine-readable form. On the other hand, requirements analysis seems to be considerably simpler than for traditional software systems. Requirements for a transformation always include the same type of information such as the source and target modelling languages, goals of the transformation and technology choices with respect to technology to be used for implementation.

We next concentrate on the design activities for model transformations, using our case study for illustrative purposes. Note that our overall design process is not restricted to a specific type of model transformation. However, the detailed description of high and low-level design assumes an *updating* unidirectional transformation [6].

We assume that during requirements analysis for the case study transformation the following requirements have been identified. The model transformation should use the concepts of the T1-T2 analysis to transform activity diagrams with unstructured cycles

into equivalent activity diagrams without unstructured cycles. In the following, we will concentrate on the design of T2 reduction rule, leading to a number of transformation rules.

4. High-level Design

The high-level design of a model transformation aims at producing a semi-formal description of a transformation, abstracting from its details such as all possible cases to be supported. More specifically, the high-level design of a model transformation provides a partially incomplete description and is not executable. The main objective of this activity is to capture the fundamentals of the transformation graphically to produce a description that can be used for discussions among the developers.

A model transformation within high-level design is specified with a set of transformation rules $r : L \to R$, each consisting of a left and right sides and informally specified *application conditions*. The left side L and right side R show subsets of the source and target models for the transformation respectively. Concrete syntax of the underlying modelling languages is used, depicting how a part of the source model resembling the left side L is replaced by the part of the model described by R. The application conditions are used to specify any restrictions on the rule application.

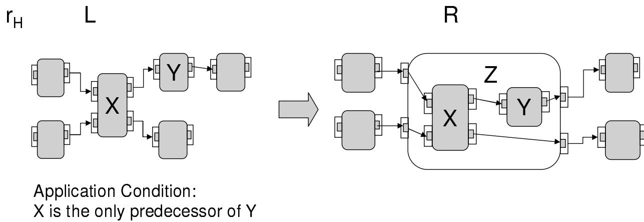

Figure 4. High-level transformation rule r_H

In Figure 4, one of the rules for the high-level design of the T2 transformation is shown. The left and right sides are drawn using concrete syntax of activity diagrams, omitting certain syntactical details. For example, this rule abstracts from the details such as the number of pins in a pin set. Nevertheless, the main idea of the transformation rule is captured: if the only predecessor of a node Y is X, then these two nodes are placed into a new SAN during the transformation.

Identification of all the possible cases that a transformation needs to handle can be facilitated with establishing *features* of a high-level transformation rule. Each feature essentially groups related cases that arise from different arrangement of elements in L and R. For instance, one feature of the rule r_H is "Node Types", which acknowledges that nodes X and Y can be instances of different types of model elements. Another feature, "Pin Set Structures" assists us in establishing different cases of pin set structures used in L. For example, simple and overlapping pin sets may need to be handled differently by the transformation.

Features of a transformation rule can be detected by examining the modelling language of the transformed models. We propose that by constructing different possible instantiations of the rule, the distinguishing features can be identified. Once a feature is identified, different cases for that feature are established. For each such *feature case*, it must be decided whether or not the transformation rule should support it. Initially, unsupported cases may arise from the requirements for the transformation, but during development this set may be expanded. Further, non-applicable cases that are prohibited by the underlying modelling language must also be identified. Unsupported and non-applicable cases form the set of *negative cases*.

X → Y	SAN	Call	Broad	Acpt	Init	Decis	Fork	Join	Final
SAN	■	■	■	■	n/a	■	n/s	n/a	n/s
Call	■	■	■	■	n/a	■	n/s	n/a	n/s
Broad	■	■	■	■	n/a	■	n/s	n/a	n/s
Accpt	■	■	■	■	n/a	■	n/s	n/a	n/s
Init	n/s	n/s	n/s	n/s	n/a	n/s	n/s	n/a	n/s
Decis	■	■	■	■	n/a	■	n/s	n/a	n/s
Fork	n/s	n/s	n/s	n/s	n/a	n/s	n/s	n/a	n/s
Join	n/s	n/s	n/s	n/s	n/a	n/s	n/s	n/a	n/s
Final	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a

Figure 5. Different cases of the feature "Node Types" for the rule r_H

We now discuss several examples of features and their respective cases for the rule r_H. We start with the feature of "Node Types". The different feature cases are shown in Figure 5. Combinations marked with a black square represent supported cases, "n/s" and "n/a" indicate unsupported and non-applicable cases respectively. Note that in the first iteration of our design process we do not support ForkNodes and JoinNodes.

In addition, we investigate the feature of "Pin Set Structures". Here, the different cases of overlapping, disjoint, concurrent and streaming pinsets are identified, as illustrated in Figure 6.

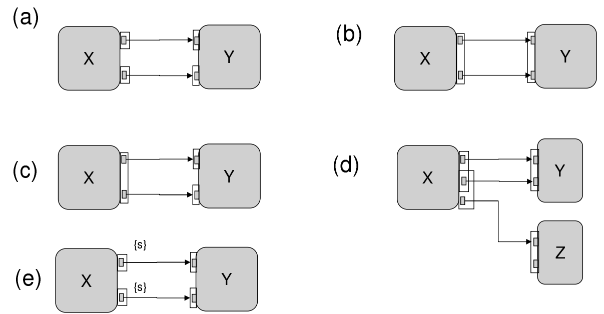

Figure 6. Different cases of the feature "Pin Set Structures"

Frequently, cases from different features occur simultaneously in a given source model and sometimes such scenarios need to be handled differently by the transformation. As a consequence, considering features in isolation is not sufficient for a complete transformation design and *feature interaction* [1] also needs to be handled. Examining feature interaction can be assisted by instantiating the high-level design rules. Figure 7 depicts several cases

[1] similar to feature interaction [27] known in telecommunications

where the features “Node Types” and “Pin Set Structures” interact. Figure 7 (a)-(d) show different instantiations of the left side of r_H for different node types of X. In (e)-(h) the instantiation with X as a CallAction is mixed with the different cases of the “Pin Set Structure” feature.

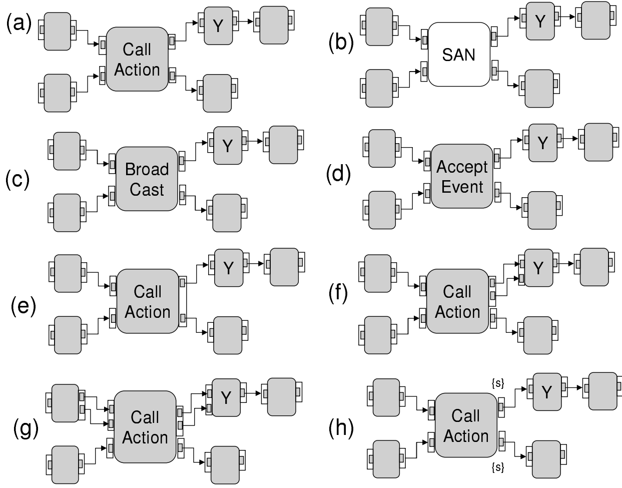

Figure 7. Feature interaction examples

From all the different cases of the left side such as those illustrated in Figure 7, we need to check whether or not they should be transformed as specified by the main high-level rule r_H. When applying r_H to different feature interaction cases, we realize that some cases must be transformed differently. For example, with regards to Figure 7 (b) the CallAction should be moved into the existing SAN instead of creating a new SAN, in order not to introduce unnecessary hierarchy. Repeating this refinement process for other feature interaction cases leads to refinements of r_H, which are shown in Figure 8 (shown here in instantiated form for the feature “Node Types”).

The result of this refinement process will be that all feature interaction cases are assigned to a high-level rule or they are categorized as unsupported or non-applicable. It is important to realize that a correct implementation ultimately has to deal with all the possible feature interaction cases by either transforming it or producing suitable error messages.

The complexity induced by the number of cases to be considered can be dealt with as follows. A number of feature interaction cases grouped by the same case in one main feature, such as all feature interaction cases where X and Y are CallActions, can be assigned to the same high-level rule without considering each individual case. This reduces the cases to be considered in isolation but can also lead to errors if a particular feature interaction case must be treated differently, then to be fixed during low-level design. Nevertheless, our approach supports the software engineer in systematic identification of all possible cases, a prerequisite for constructing a correct and complete implementation.

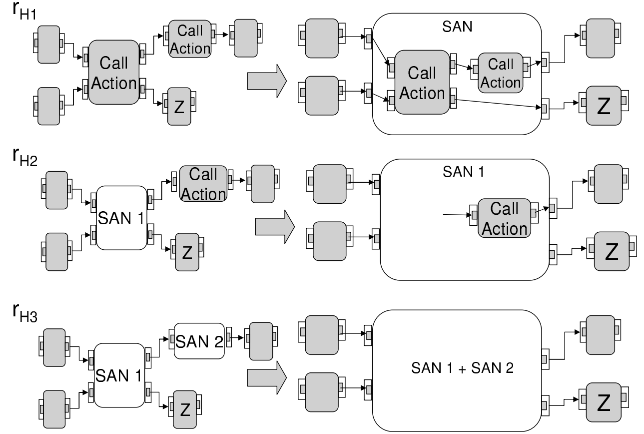

Figure 8. High level rules

5. Low-level Design

High-level design transformation rules need to be refined to create a more detailed transformation specification. One should be able to generate transformation code from such a low-level specification or use it as a basis for manual implementation. This means that the high-level transformation rules need to be refined into a complete and unambiguous description of the transformation. We next describe a notation suitable to capture a transformation at low-level.

5.1 A Transformation Language for Low-Level Design

Several languages for specification of transformation rules have already been proposed, including VIATRA [5], GReAT [13], UMLX [26], BOTL [4] and work by Milicev [17]. All of them propose languages that describe transformations at the abstract syntax or metamodel level. This allows for generic application of a transformation language, as in this way it is not dependant on a concrete syntax of any modeling notation. Additionally, an abstract syntax description is much closer to a final transformation implementation that deals with internal model representations. In the following, we informally describe our transformation language that reuses ideas from various approaches but adds some features that prove beneficial for low-level design. Note that the focus of this paper is not to define yet another transformation language, but lies on the design process for model transformations.

Similarly to high-level design, a low-level transformation rule in our language consists of left and right sides. Both sides of a rule are composed of graphical *patterns* that can be matched against a given source model and used to indicate a transformation update. The graphical notation used for these patterns is based on UML *object diagrams*, where objects represent distinct instances of classes from the underlying metamodel. During pattern-matching, objects are matched by type and attribute values, while associations are matched by role names. On the left side of a rule, patterns are used to show what elements and relations “must occur” in a source model in order for the rule to be applicable. Furthermore, negative cases are also captured using left side patterns, they are then called *antipatterns*. These antipatterns describe

structures that “must not exist”. On the right side, *update patterns* show how the target model relates to the source model by indicating the elements that are removed from or inserted into the source model when applying a transformation rule.

Figure 9 contains a simple example of a transformation rule, which demonstrates all the main concepts of our transformation language. The depicted rule r can be used to remove every OutputPin in a source model, which belongs to more than one PinSet and is not connected to an ActivityEdge.

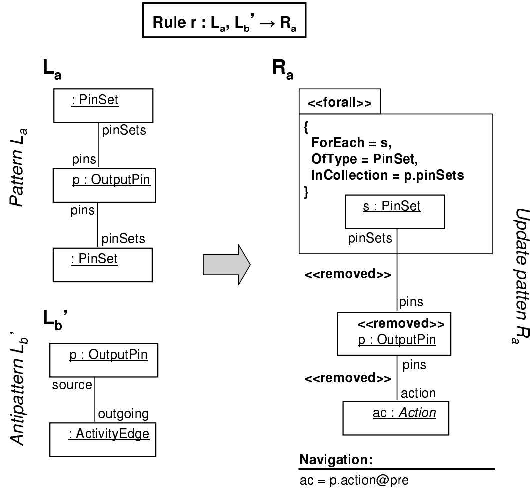

Figure 9. An example of a low-level transformation rule

As can be seen from the diagram, both left and right sides of a rule comprise a collection of uniquely named patterns. The box at the top of the diagram gives the textual rule definition, which contains references to the pattern diagrams shown below. Names of antipatterns are marked apostrophes to distinguish them from other patterns, as illustrated by the antipattern L_b’ in Figure 9.

The left side pattern L_a in Figure 9 specifies that this rule applies to such a source model fragment that contains an OutputPin p that is associated with “at least two” PinSets. On the other hand, the antipattern L_b’ indicates that the OutputPin p must not be associated with an ActivityEdge object.

On the right side of a rule, three stereotypes are available to specify the update: *new*, *removed* and *modified.* The *new* and *removed* stereotypes can be used to indicate creation and removal of objects and associations respectively. The *modified* stereotype can only be applied to objects to show that their attribute change during the transformation.

In the example illustrated in Figure 9, essentially three update steps are captured. Firstly, the actual OuputPin object matched on the left side of the rule is removed from the source model. In addition to this, all the associations that existed for that object in the source model also need to be removed. Thus the second step of the update is the removal of the association between Action ac and the OuputPin p. A navigation clause at the bottom of the pattern diagram shows navigation to this Action object ac using the *Object Constraint Language (OCL)* [19]. Note that in the metamodel for activity diagrams, Action is an abstract class and cannot have direct instances. According to our transformation language, an object “instantiating” an abstract class that appears in a pattern can be matched against an instance of any of the subclasses of that abstract class. Hence, the bject a in Figure 9 can for example match a CallAction or an AcceptAction instance.

The third and the last update step that is shown in the update pattern R_a in Figure 9 removes the associations between the OutputPin p and all the PinSets to which it belonged in the source model. A *forall* construct proposed by Milicev [17] is used here to deal with a collection of objects during a transformation. The package stereotyped *forall* in Figure 9 shows the scope of the forall construct. Tagged values in curly brackets are used to specify the collection to which the contents of the construct apply. The tags ForEach and OfType provide the reference name and type of the iterator element in the collection under consideration. The InCollection tag uses OCL to navigate to the actual collection in the source model. In the pattern R_a, the collection consists of all the PinSets associated with OutputPin p. The contents of the *forall* construct show that for each object s in this collection, the association to p is removed.

In addition to the specification of transformation rules, the transformation language also contains a means of specifying a rule application strategy for defining in which order the rules should be applied. Given a set of rules $\{r_1, .., r_n\}$, a rule application strategy can be specified by defining for each rule how many times it should be applied and then arranging the rules in a sequence. For example, the strategy $\langle r_1, r_2 \downarrow, r_3 \rangle$ specifies that first r_1 should be applied once, then r_2 should be applied as long as possible and then r_3 should be applied once.

5.2 From High-level to Low-level Design

The low-level design aims at providing a complete specification of the model transformation. As a consequence, it needs to provide a complete set of rules expressed in the syntax of the transformation language introduced in the previous sub-section. Any informally specified application conditions must be formally expressed using OCL constraints.

The result of the high-level design is a set of high-level transformation rules and a set of supported feature interaction cases, where for each case it is known along which rule it should be transformed. We propose an iterative process for systematic refinement of the high-level rules to low-level rules, as described below and depicted in Figure 10.

1. Choose a high-level rule r_{Hi} associated with a set of feature interaction cases.
2. Select one feature interaction case from the set of cases supported by the r_{Hi}.
3. Using r_{Hi} as a reference, design the low-level rule r_{i1} for the selected feature interaction case. The left and right sides of the low-level rule should be described in the transformation language, refining the level of detail contained in r_{Hi}.

4. Traverse the remaining cases in the set of the feature interaction cases associated with r_{Hi} and design low-level rules for them as follows:

   (a) Select one focus feature and keeping it fixed to one case, move through the feature interactions cases by varying the other features. Then change the case for the focus feature and once again move through all the feature interaction cases where that feature case is present. Continue until all the cases associated with the high-level rule r_{Hi} are considered.

   (b) For each feature interaction case during the traversal, determine whether it can be handled by the already created low-level rules r_{i1} to r_{in}. If no rule can handle the current case, design a new low-level rule $r_{i(n+1)}$. It may also be possible to generalise existing rules, making them applicable to more cases.

5. Return to step 1 and iterate over the remaining steps until all the high-level rules have been refined.

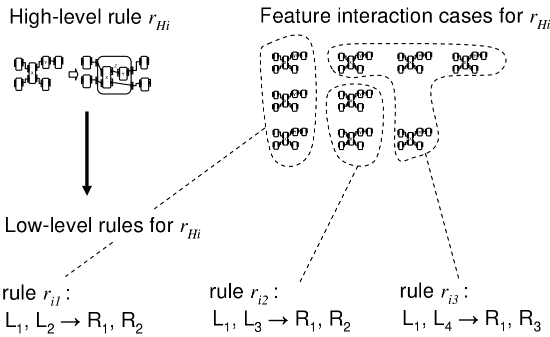

Figure 10. Systematic refinement of high-level transformation rules

The low-level rules that are derived from the same high-level rule often reuse some left side and update patterns. This is illustrated in Figure 10, where the left side pattern L_1 and update pattern R_1 appear in each of the low-level rules r_{i1}, r_{i2} and r_{i3}.

The proposed process for refining high-level rules is now illustrated with an example from the case study. We begin the process by choosing one of the informally captured rules - the rule r_{H1} shown in Figure 8 (a). As suggested in step 2 of the process, we next consider one of the feature interaction cases that is handled by this rule - the case depicted in Figure 7 (a). This is an interaction between the case where both X and Y are CallActions and the case of the simplest pin set structures shown in Figure 6 (a). According to step 3, we then design the left and right sides of the corresponding low-level design rule, which we call r_{11}.

In order to construct the left side of the low-level rule r_{11} from the informal diagram in Figure 8 (a), we have to extract enough detail about the model fragment that distinguish it from other fragments where the transformation rule should not apply. Most importantly, a fitting source model fragment should contain two connected CallAction elements X and Y. This condition is captured in the pattern L_1 in Figure 11 (c). This pattern states that the source model must contain two CallAction objects connected by an ActivityEdge, using an OutputPin and an InputPin as connection points. The elements in the concrete model fragment that match this pattern are shown in bold in Figure 11 (a).

In addition to finding two connected CallActions X and Y in the source model, we also need to check that X is the only predecessor of Y. The informal diagram depicting a negative case for this is shown in Figure 11 (b) and Figure 11 (d) shows the corresponding antipattern L_2' for the low-level design rule.

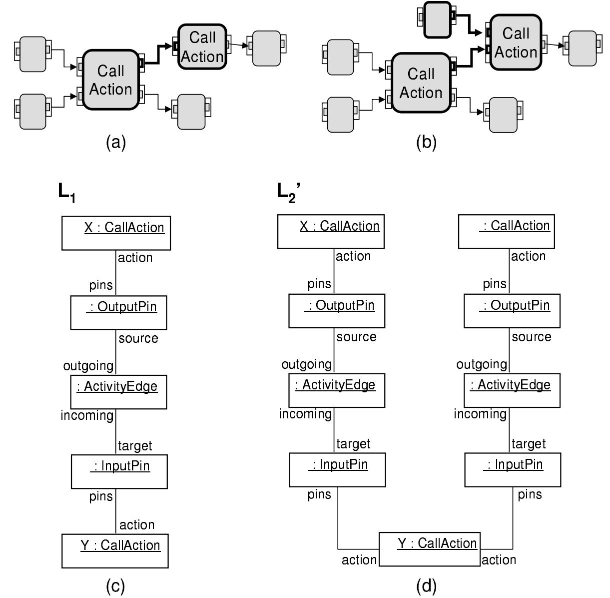

Figure 11. Left side patterns for the low-level rule r_{11}

The right side of the rule r_{11} is obtained by refining the right side of the high-level transformation rule shown in Figure 8 (a) into a number of update patterns. Firstly, a new SAN needs to be created and the two CallActions X and Y are moved inside it. The edges connecting X and Y also need to be explicitly moved inside the new SAN. Furthermore, the old connections and pin sets of X and Y must be copied to the new SAN in such a way that the semantics of the model are preserved. The update pattern R_1 in Figure 12 captures the first part of the update described above. The created SAN is stereotyped *new* and contains a derivation statement for its visibility attribute, which is copied from Y. In addition to the created SAN, the R_1 pattern also contains *new* associations between this object and the CallActions X and Y. This specifies that X and Y are "moved into" the new SAN. Note here that low-level design requires one to consider detail that might be omitted in the high-level design. In the case of the update captured in the pattern R_1, old associations of X and Y to a SAN from the source model are removed.

The update pattern R_2 in Figure 13 "moves" the edges connecting X and Y into the new SAN and removes the associations to the old SAN. As the number of edges between X and Y is not known in advance, a *forall* construct is used here. In this *forall* construct, the underlying collection consists of all the ActivityEdges leading

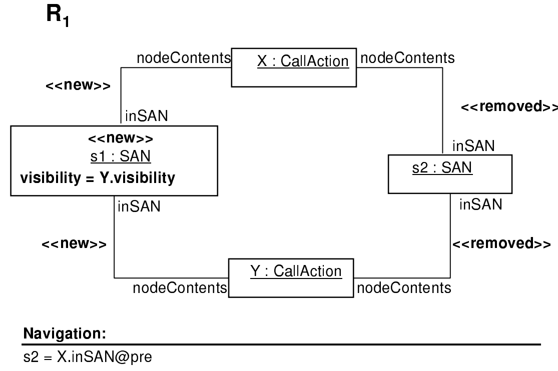

Figure 12. Right side pattern R_1

to CallAction Y. The contents of the *forall* construct show that for each object e in this collection, a new association to the new SAN is created and the association to the old SAN is removed.

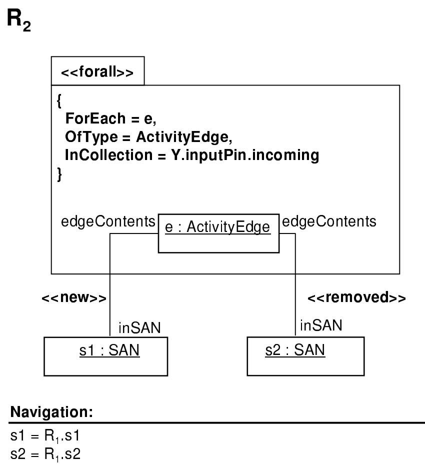

Figure 13. Right side patterns R_2

Further update patterns for the reconnection of X, Y and the new SAN with *ActivityEdges* and the copying of pin set structures are drawn up to complete the r_{11} rule. The completion of this rule definition brings us to the end of step 3 in the refinement process.

During step 4, we choose "Node Types" as the focus feature and keeping it fixed at the case where both X and Y are CallActions, consider the different cases of the "Pin Set Structure" feature. We discover that rule r_{11} handles most of the pin set structure cases shown in Figure 6. The only case that needs adjustments to the rule is the case with streaming pin sets depicted in Figure 6 (h). We create a variation of rule r_{11} to handle this case (see below) and also add an antipattern to the left side of r_{11} itself to exclude its application to streaming pin sets.

On further iterations through step 4 of the process, we consider other cases of our focus feature "Node Types" and their interaction with the possible pin set structures. As we identified during the high-level design phase, the rule r_{H1} applies to the cases where both X and Y can be instances of any of the three Action subclasses: CallAction, BroadcastAction and AcceptAction. Our transformation language allows us to handle all these different combinations with a single rule by using the abstract superclass in the left side and update patterns. Therefore, we adjust the rules derived so far by substituting the CallAction objects by Action objects in all the patterns. This means however, that source model fragments where either of X and Y is a SAN can also be matched by the left side patterns of the rules. This is not desirable, since such cases are handled differently by the transformation as shown in Figure 8 (b) and (c). In order to exclude the cases with SANs, we add the following condition expressed in the OCL to the left side of the rules. OCL_1 : (X.oclIsTypeOf(SAN) = false) and (Y.oclIsTypeOf(SAN) = false).

Once we have completed the low-level design for high-level transformation rule r_{H1}, we proceed to r_{H2}. In Figure 14, we show several patterns for the low-level design of this rule. The difference in the performed update can be seen from the update pattern R_3. In this rule, no new SAN is created since X itself is a SAN. Instead, the Action Y is moved into the SAN.

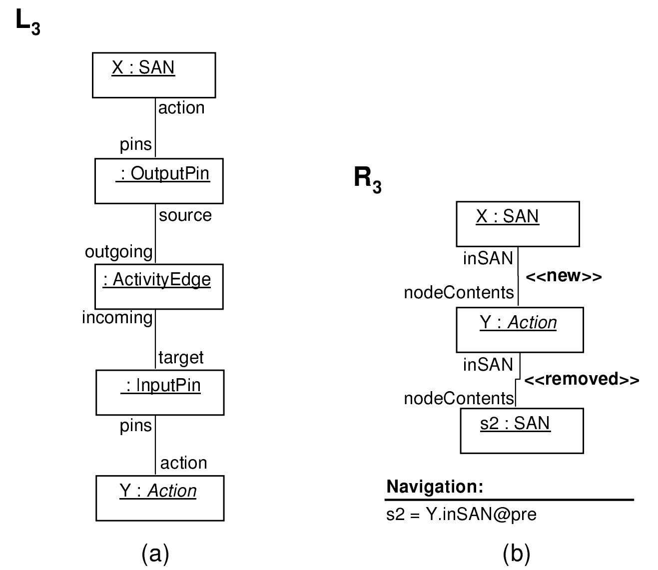

Figure 14. Patterns for r_{H2}

After having designed the low-level transformation rules, one has to decide about the rule application strategy. This rule application strategy also determines whether the model transformation may run into possible termination problems. In general, those rules that may be applied as long as possible may give rise to these kinds of problems [15]. One can avoid termination problems by making sure that each rule to be applied as long as possible can only be applied a finite number of times, for example by constructing the left side accordingly. Similarly, also confluence problems can occur within model transformations. The complete specification and optimization of a rule application strategy are beyond the scope of this paper.

6. Validation of Design

Validation of the design created is an important issue for model transformation development. The low-level design has already been partially validated against the high-level design, by checking whether all supported cases identified have been considered.

Further validation includes syntactic correctness and semantics preservation as well as termination and confluence.

There is a general trade-off on how much validation can be performed before implementing the model transformation. If there is no execution environment available but the rules are implemented in Java, then validation must be done by hand and may be quite some overhead. In all cases, we favor that basic syntactic correctness and semantics preservation are already validated before implementation. The validation of other properties such as confluence can be postponed until a working implementation is available, in particular if it is to be validated by testing.

Syntactic correctness is concerned with ensuring that the models produced by a transformation conform to the syntax of the target language. This can be done by inspecting the low-level design rules, as is done for example in [4]. On the contrary, ensuring that a model transformation does not lead to models that have a different semantics is much more difficult.

Although a complete preservation of semantics is not always required, one often wants to ensure that the result model at least preserves certain semantic properties [24]. With regards to our case study, we even require that the resulting activity diagram models the same behavior as before.

This preservation of semantic properties is difficult to prove formally, given the case that the models do not have a formal semantics as is the case for UML activity diagrams. Nevertheless, in our case at least an informal justification of semantics preservation is needed. In general, one can follow the following hypothesis [8]. A model transformation preserves the semantics of a model if each transformation step preserves the semantics of the model. The hypothesis requires that each transformation step preserves the semantics of the model. Whereas the possible transformation steps are specified by the rules, the set of possible models is typically very large. Given the absence of a formal semantics for these models, we suggest that the rules are checked on a set of test cases and so-called *before/after comparison* are performed.

A before/after comparison informally compares the semantics of the model before and after the rule application. The success of this approach crucially depends on the selection of the "right" test case models and also on the correct discussion of the model semantics before and after the rule application. The cases identified within high-level design and the resulting feature interaction cases are an ideal candidate for such test case models because they provide an abstraction on the one side and capture all the different cases of (partial) models relevant to the transformation rules in a minimal context. As such, they provide a solution to the difficult problem of finding suitable test case models.

The outcome of the before/after comparison gives rise to one of the following. In case of the detection that a rule does not preserve model semantics, one possibility is to adapt the rule in this case. Another option is to include the test case in the set of negative cases and then adapt the precondition of the rule accordingly.

In the following, we illustrate several problems that we found when discussing the issue of semantics preservation for the case of the rule r_{H1}, see Figure 15. If a BroadcastAction is moved into a SAN, then the scope of the action changes to the scope of the SAN. As a consequence, applying r_{H1} in a situation where there exist already AcceptActions within the SAN leads to semantic changes and must be avoided. As a consequence, we introduce a new OCL constraint in the precondition of the rule, requiring that no such actions exist.

Figure 15 (b) shows the problem when r_{H1} is applied to a CallAction with an implicit fork. After the transformation, Y is executed before Z. As a consequence, we include this case into the set of unsupported cases.

Figure 15 (c) illustrates the case of streaming pin sets. Here, the result of applying r_{H1} changes the semantics because before the two nodes following the CallAction were obtaining tokens while CallAction is still running. After the transformation, tokens streamed out from the pin set remain stuck at the SAN. As a consequence, r_{H1} must be changed to copy the streaming also to the pin sets of the SAN.

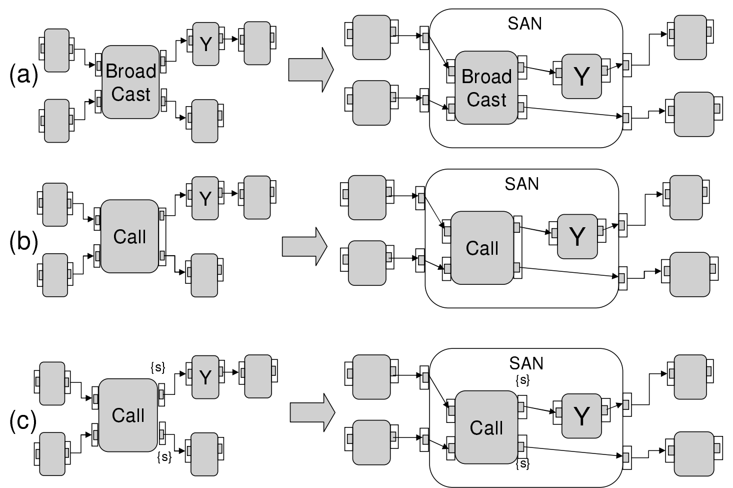

Figure 15. Cases for validating semantic preservation

The previous problems have illustrated the semantic subtleties that must be taken care of when designing the transformation. They show that each case arising from the feature interactions must be carefully investigated in order to decide whether or not an application of r_{H1} is semantics preserving, giving rise to additional unsupported cases or changes in the rule r_{H1} itself.

7. Conclusion

Developing model transformations is an important task for making the vision of the MDA to become reality. As model transformations may become rather complex, a design language as well as a development process are needed. Whereas a model transformation language is subject to extensive research, only little work on development processes for model transformations is known. The closest work we found concerning a development process for model transformations focuses on testing model transformations [16].

In this paper, we have presented our approach to systematic design of model transformations. After an informal capture of the key requirements, first a high-level design is developed which sketches the transformation rules at an abstract level. Using a feature-based approach, different cases for these transformation rules are systematically discovered. Combination of different cases for each feature gives rise to feature interaction cases. Adapting and refining the high-level rules and expressing each case in a model transformation language yields the low-level design. Prior to implementation, the low-level design is validated to ensure syntactic correctness and semantics preservation. Whereas syntactic correctness validation is rather straightforward, a complete validation of semantics preservation is extremely difficult. We have illustrated how semantics preservation can be ensured, by systematically examining feature interaction cases and performing so-called before after comparisons. Following our approach allows the software engineer to systematically deal with all feature interaction cases early within the development process and enables thereby the construction of a correct model transformation implementation.

Future work includes the systematic derivation of test cases from low-level design rules, the elaboration of our model transformation language and the development of appropriate tool support. Appropriate tool support will enable the software engineer to keep track of feature interaction cases already supported and will also help to deal with the large number of feature interaction cases. This can be achieved by systematically generating all cases internally and keeping track of those ones supported by which low-level rule.

Acknowledgement: The authors thank Thomas Gschwind and Jana Koehler for their valuable comments on an earlier version of this paper.

8. References

[1] The Model-Driven Architecture, Guide Version 1.0.1, omg/2003-06-01. OMG Document, April 2002.

[2] A. Aho, R. Sethi, and J. Ullman. *Compilers–Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[3] A. S. Boujarwah and K. Saleh. Compiler test case generation methods: a survey and assessment. *Information and Software Technology*, 39(9):617–625, 1997.

[4] P. Braun and F. Marschall. BOTL - The Bidirectional Objekt Oriented Transformation Language. Technical report, Fakultät fr Informatik, Technische Universität München, Technical Report TUM-I0307, 2003.

[5] G. Csertán, G. Huszerl, I. Majzik, Z. Pap, A. Pataricza, and D. Varró. VIATRA: Visual Automated Transformations for Formal Verification and Validation of UML Models . In *Proceedings 17th IEEE International Conference on Automated Software Engineering (ASE 2002)*, pages 267–270, Edinburgh, UK, September 2002.

[6] K. Czarnecki and S. Helsen. Classification of Model Transformation Approaches. In *Proceedings OOPSLA'03 Workshop on Generative Techniques in the Context of Model-Driven Architecture*, 2003.

[7] J. C. Derniame, B. A. Kaba, and D. Wastell, editors. *Software Process: Principles, Methodology, and Technology*, volume 1500 of *LNCS*. Springer-Verlag, 1999.

[8] G. Engels, R. Heckel, and J. M. Küster. Consistency Preserving Model Evolution. In J.-M. Jezequel, H. Hussmann, and S. Cook, editors, *UML 2002 - The Unified Modeling Language. 5th International Conference, Dresden, Germany, September 30 - October 4, 2002, Proceedings*, volume 2460 of *LNCS*, pages 212–226. Springer-Verlag, 2002.

[9] G. Engels, R. Heckel, and J. M. Küster. The Consistency Workbench - A Tool for Consistency Management in UML-based Development. In P. Stevens, J. Whittle, and G. Booch, editors, *UML 2003 - The Unified Modeling Language. Modeling Languages and Applications. 6th International Conference, San Francisco, October 20 -24, USA, Proceedings*, volume 2863 of *LNCS*, pages 356–359. Springer-Verlag, 2003.

[10] R. Hauser and J. Koehler. Compiling process graphs into executable code. In G. Karsai and E. Visser, editors, *Proceedings of the 3rd International Conference on Generative Programming and Component Engineering*, LNCS 3286, pages 317–336, Vancouver, Canada, October 2004. Springer-Verlag.

[11] M. Hecht and J. Ullman. Flow graph reducibility. *SIAM Journal of Computing*, 1(2):188–202, 1972.

[12] I. Jacobsen, G. Booch, and J. Rumbaugh. *The Unified Software Development Process*. Addison Wesley, 1999.

[13] G. Karsai, A. Agrawal, F. Shi, and J. Sprinkle. On the Use of Graph Transformation in the Formal Specification of Model Interpreters. *Journal of Universal Computer Science*, 9(11):1296–1321, 2003.

[14] J. Koehler, R. Hauser, S. Sendall, and M. Wahler. Declarative techniques for model-driven business process integration. *IBM Systems Journal*, 44(1), 2005.

[15] J. M. Küster. Definition and validation of model transformations using a graph transformation-based approach. *Software and Systems Modeling*, 2005. submitted.

[16] Y. Lin, J. Zhang, and J. Gray. A Testing Framework for Model Transformations. In *Model-driven Software Development - Research and Practice in Software Engineering*. Springer-Verlag, 2005. to appear.

[17] D. Milicev. Domain Mapping Using Extended UML Object Diagrams. *IEEE Software*, 19(2):90–97, March/April 2002.

[18] OASIS. *Business Process Execution Language for Web Services (BPEL4WS) 1.1.*, May 2003.

[19] Object Management Group (OMG). *Response to the UML 2.0 OCL RfP (ad/2000-09-03), version 1.6, OMG document ad/2003-01-07*, January 2003.

[20] Object Management Group (OMG). *UML 2.0 Superstructure Final Adopted Specification. OMG document pts/03-08-02*, August 2003.

[21] Object Management Group (OMG). *QVT-Merge Group. MOF 2.0 Query/Views/Transformations, Revised Submission. OMG document ad/2004-04-01*, April 2004.

[22] W. W. Royce. Managing the Development of Large Software Systems: Concepts and Techniques. In *Proceedings of IEEE WESCON, Los Angeles, CA*, pages 1–9, 1970.

[23] P. v. Gorp, H. Stenten, T. Mens, and S. Demeyer. Towards Automating Source-Consistent UML Refactorings. In P. Stevens, J. Whittle, and G. Booch, editors, *UML 2003 - The Unified Modeling Language. Modeling Languages and Applications. 6th International Conference, San Francisco, October 20 -24, USA, Proceedings*, volume 2863 of *LNCS*, pages 144–158. Springer-Verlag, 2003.

[24] D. Varró and A. Pataricza. Automated formal verification of model transformations. In J. J. et al., editor, *CSDUML 2003: Critical Systems Development in UML; Proceedings of the UML'03 Workshop*, number TUM-I0323 in Technical Report, pages 63–78. Technische Universität München, September 2003.

[25] J. Whittle. Transformations and Software Modeling Languages: Automating Transformations in UML. In J.-M. Jezequel, H. Hussmann, and S. Cook, editors, *UML 2002 - The Unified Modeling Language. 5th International Conference, Dresden, Germany, September 30 - October 4, 2002, Proceedings*, volume 2460 of *LNCS*, pages 227–242. Springer-Verlag, 2002.

[26] E. Willink. UMLX: A graphical transformation language for MDA. In A. Rensink, editor, *Proceedings of the Workshop Model Driven Architecture, CTIT Technical Report TR-CTIT-03-27*, pages 13–24. University of Twente, 2003.

[27] P. Zave. Feature interactions and formal specifications in telecommunications. *IEEE Computer*, 26(8):20–29, Aug. 1993.