Validation of Model Transformations - First Experiences using a White Box Approach

Jochen M. Küster[1] and Mohamed Abd-El-Razik[23*]

[1] IBM Zurich Research Laboratory, Säumerstr. 4, 8803 Rüschlikon, Switzerland
jku@zurich.ibm.com
[2] IBM Cairo Technology Development Center, El-Ahram, Giza, Egypt
[3] Department of Computer Science, American University in Cairo, Egypt
mohrazik@aucegypt.edu

Abstract. Validation of model transformations is important for ensuring their quality. Successful validation must take into account the characteristics of model transformations and develop a suitable fault model on which test case generation can be based. In this paper, we report our experiences in validating a number of model transformations and propose three techniques that can be used for constructing test cases.

1 Introduction

The success of model-driven engineering generates a strong need for techniques and methodologies for developing model transformations. How to express model transformations and build appropriate tool support is a widely discussed research topic and has led to a number of model transformation languages and tool environments.

For practical use in model-driven engineering, the quality of model transformations is a key issue. If models are supposed to be semi-automatically derived using model transformations, then the quality of these models will depend on the quality of model transformations. Proving correctness of model transformations formally is difficult and requires formal verification techniques. An alternative approach widely applied in the industry is validation by testing. Today, it is common practice to apply large-scale testing for object-oriented programs using tools such as JUnit.

Model transformations can either be implemented as programs (e.g. in Java) or using one of the available transformation languages (e.g. [7, 15, 5, 16]). In both cases, they require a special treatment within testing. One of the key challenges for testing model transformations is the construction of 'interesting' test cases, i.e. those test cases that show the presence of errors. For black box testing of model transformations, the meta model of the input language of the transformation can be used to systematically generate a large set of test cases [11, 10]. If the result of the model transformation is supposed to be executable, a possible testing approach is to take the output of a model transformation and to test whether it is executable [9]. By contrast, a white box approach to testing takes into account design and implementation of the model transformation for constructing test cases. Compared with the extensive work on model-based testing of reactive systems (see Utting et al. [26] for a taxonomy and tool overview), testing of model transformations can still be considered to be in its early stages.

[*] Part of this research was conducted while at the IBM Zurich Research Lab.

In this paper, we present our first experiences with a white box model-based approach to testing of model transformations. Our techniques have been developed while implementing a set of five model transformations for business-driven development [22, 17] which are used in a model-driven engineering approach for business process modeling. We propose three techniques for constructing test cases and show how we have used them to find errors in our model transformations.

The paper is structured as follows: We first introduce the idea of business-driven development and discuss the motivation for designing our model transformations in Section 2. Then we elaborate on our approach to design and implementation of these transformations in Section 3. In Section 4, we introduce three techniques for constructing test cases and explain how we apply them to validate our transformations. We conclude with a discussion of related work and conclusions drawn from our experience.

2 Model Transformations for Business Process Models

The field of business process modeling has a long standing tradition. Business-driven development is a methodology for developing IT solutions that directly satisfy business requirements. The idea includes that business process models are iteratively refined and transformed using model transformations, to move semi-automatically from a higher to a lower abstraction level.

We present business process models in the notation of IBM's WebSphere Business Modeler [1], which is based on UML 2.0 activity diagrams [24]. The language supported by the WebSphere Business Modeler makes some extensions to UML and can be considered as a domain-specific language for business process modeling. In these models, we distinguish *task* and *subprocess* elements. While tasks capture the atomic, not further dividable activities in the business process models, subprocesses can be further refined into more subprocesses and tasks. *Control and data flow* edges connect tasks and subprocesses. The control and data flow can be split or merged using *control actions* such as *decision*, *fork*, *merge*, and *join*. Process start and end points are depicted by *start* and *end nodes*. In addition, the language also contains a number of specific actions such as *broadcast* for broadcasting signals and *accept signal* for receiving signals or *maps* for mapping input data to output data.

In the language supported by the WebSphere Business Modeler, pin sets (based on parameter sets in UML2) are used for expressing implicit forks and joins as well as decisions and merges. Although these constructs leave a lot of freedom to the developer, they are problematic for complex transformations. As a consequence, we distinguish between models that only use control actions and those that only use pin sets. A model in the Control Action Normal Form (CANF) requires that an action only has at most one pin set with exactly one pin in it [2]. A model in the Pinset Normal Form (PNF) requires that all forks, joins, decisions and merges are expressed implicitly using pin sets [2].

To support the idea of business-driven development for business process models, we have designed and implemented a number of model transformations for business process models (see Koehler et al. [17] for a detailed overview). The goal of these transformations is to enable a model-driven approach within business process modeling:

- the *Control Flow Extraction* transformation transforms a business process model with control and data flow into a process model with control flow only,
- the *Data Container Assignment* transformation transforms a business process model without data flow into a process model with data flow,
- the *Cycle Removal* transformation transforms a business process model with unstructured cycles into a process model with structured cycles only [13],
- the *Control Action to Pinset* transformation [2] transforms a business process model into the Pinset Normal Form, and
- the *Pinset to Control Action* transformation [2] transforms a business process model into the Control Action Normal Form.

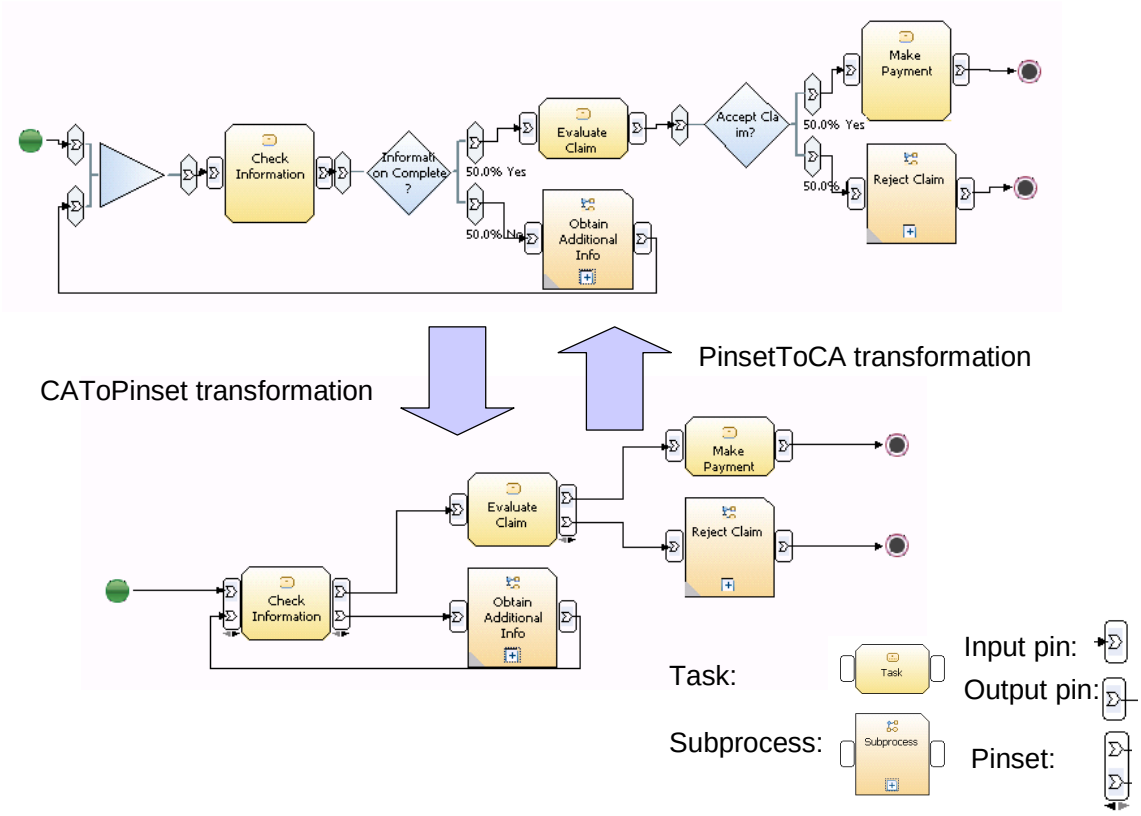


Fig. 1. Example of a process model in both forms

Figure 1 shows an example of a process model in the Pinset Normal Form (lower model) and in the Control Action Normal Form (upper model). All previously mentioned transformations have been implemented as Eclipse plugins to the WebSphere Business Modeler. In the following, we will concentrate on the Control Action to Pinset (CAToPinset) transformation, although we have applied similar techniques to other transformations. First, we will discuss our approach to design and implementation, and then elaborate on testing the transformations.

3 Design and Implementation of Model Transformations

For design and implementation of the model transformations introduced, we apply an iterative approach [20] that consists of producing a high-level design which is then used as a basis for the implementation.

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. As such, it can be considered as an early design in the elaboration phase, following the terms of the Rational Unified Process [18]. This high-level design provides an 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 then be used for discussions among the developers.

A model transformation within high-level design is specified with a set of *conceptual transformation rules* $r : L \rightarrow R$, each consisting of a left and right side. 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 modeling 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. In addition to elements from concrete syntax, those elements that are considered to be abstract are represented using additional abstract elements. These elements will typically be refined in later design phases or during implementation.

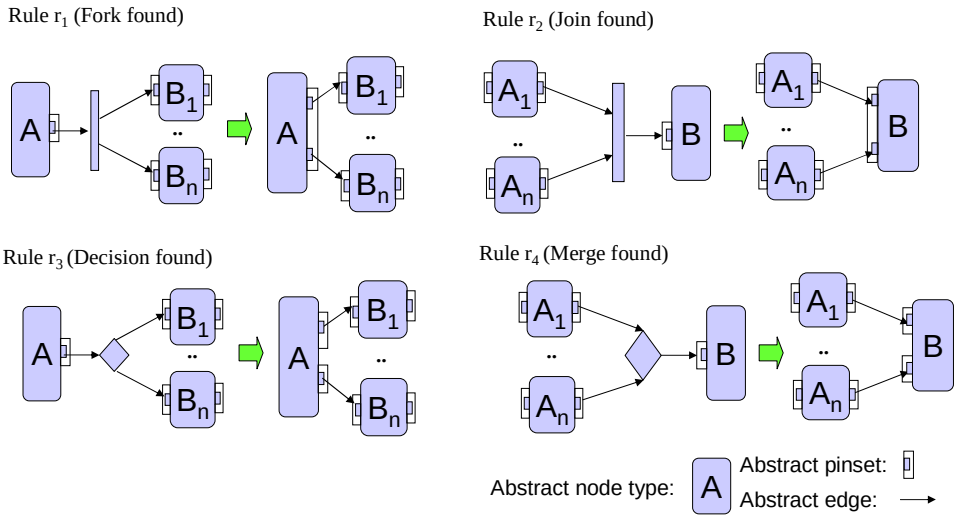


Fig. 2. Rules of the Control Action to Pinset transformation

In Figure 2, rules of the CAToPinset transformation are shown. In addition to concrete syntax elements such as the *fork*, abstract elements are used, such as an abstraction for the node type. Overall, the rules abstract from the details such as the number of pins in a pin set, the number of outgoing or incoming edges, the type of the nodes and the type of the edge (control or data flow). Nevertheless, the main idea of each transformation rule is captured. For example, rule r_1 removes a fork, creates a new pin within the pin set of A, and connects the edges outgoing from the fork directly to the pins of A.

In general, different ways of implementing a model transformation exist. A pure model-driven approach consists of using one of the existing transformation engines, e.g. supporting the language QVT [25]. This has the advantage that the developer is given an environment that allows a definition of the transformation in a transformation language. In our case, we decided to implement the transformations directly in Java. This target implementation was then packaged as an Eclipse plugin and executed in the WebSphere Business Modeler.

In both cases, the conceptual rules of the transformation have to be refined by identifying the different cases they have abstracted from and defining how to handle them. For example, rule r_1 in Figure 2 has to be specified in more detail to take into account the possibility of data flow along the edges, the possibility of having multiple edges and special cases where parts of the fork are unconnected. In addition, the rule has to be refined with regards to the different possible node types for A and B_1 to B_n. In our case, possible node types include *start* and *end nodes*, *task*, *subprocess*, loop nodes such as a *while* loop, all control action nodes, and a number of specific action nodes such as the *broadcast* node. It is because of this number of model elements together with attached constraints that the transformation, which might look trivial at the conceptual level, requires some effort during implementation as well as thorough testing.

4 Systematic Testing of Transformations

Along the line of general principles of software engineering [12], we can distinguish between testing in the small and in the large. Testing in the small applied to model transformations can be considered as testing each transformation rule whereas testing in the large requires testing of each transformation.

For both types of testing, challenges of testing specialized to model transformations can be expressed as follows (adapted from [14]):

- the generation of test cases from model transformation specifications according to a given coverage criterion,
- the generation of test oracles to determine the expected result of a test, and
- the execution of tests in suitable test environments.

In our approach to model transformation development, the third challenge is easy to overcome because we can execute tests directly in our development environment. The main challenges are the first and second ones because the model transformation specification in our case is based on the informal conceptual rules introduced above. In the following, we will show how we can partially overcome these challenges. First, we will discuss common types of errors that we have encountered when implementing the transformations. Then we discuss three techniques for test case generation and discuss cases where the test oracle problem is easy to overcome.

4.1 Fault model for model transformations

A fault model describes the assumptions where errors are likely to be found [4]. Given our approach to model transformation development, we can apply a model-based testing

approach that takes into account the conceptual transformation rules as models. Based on our experience, the following errors can occur when coding a conceptual transformation rule:

1. *Meta model coverage*: the conceptual transformation rule has been coded without complete coverage of the meta model elements, leading to the problem that some input models cannot be transformed (e.g. the rule only works for certain node types, only for control flow edges, or only for one edge between two tasks but not for two edges).
2. *Creation of syntactically incorrect models*: the updating part of the transformation rule has not been implemented correctly. This can lead to models that do not conform to the meta model or that violate constraints specified in the meta model of the modeling language.
3. *Creation of semantically incorrect models*: the transformation rule has been applied to a source model for which it is not suitable, i.e. the result model is syntactically correct but it is not a semantically correct transformation of the source model.
4. *Confluence*: The transformation produces different outputs on the same model because the transformation is not confluent. This also includes the possibility that the transformation leads to intermediate models that cannot be transformed any further because non-confluence of the transformation has not been detected and treated.
5. *Correctness of transformation semantics*: the transformation does not preserve a desired property that has been specified for the transformation. Possible properties include syntactic and semantic correctness (see above) but also refinement or behavioral properties such as deadlock freedom.
6. *Errors due to incorrect coding*: there are also errors possible that cannot be directly related to one of the other categories. These errors can be classical coding errors.

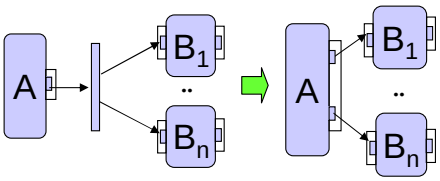
Often, there is an interplay between meta model coverage and syntactic correctness. A meta model coverage error can lead to a syntactically incorrect model. The challenge in all cases is how to systematically generate test cases and how to create the appropriate test oracles. Errors due to incorrect coding are indirectly found when testing for the first four types of errors. In addition, further techniques such as code walk-throughs can be applied. In the following, we introduce three techniques and discuss how they can be applied to find different types of errors. The last two types of errors are not explicitly dealt with in this paper and are left to future work.

4.2 Meta model coverage testing

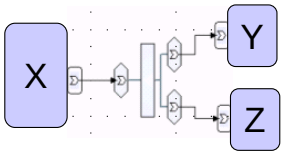
In our approach to model transformation development, a given conceptual rule can be transformed into a *meta model template*. The idea of a meta model template is to be able to create automatically template instances that represent suitable test cases.

In the transition from a conceptual rule to a meta model template, abstract elements must either be made concrete or must be resolved by parameters together with a parameter set. To identify for each parameter in the conceptual rule the possible parameter values, the meta model of the underlying modeling language must be taken into account.

a) Conceptual rule:



b) Template(X,Y,Z):



X={StartNode, Fork, Join, Decision, Merge, Task, Subprocess, LoopNode, Broadcast, AcceptSignal}

Y={FinalNode, Fork, Join, Decision, Merge, Task, Subprocess, LoopNode, Broadcast, AcceptSignal, Map}

Z={FinalNode, Fork, Join, Decision, Merge, Task, Subprocess, LoopNode, Broadcast, AcceptSignal, Map}

c) Template instances:

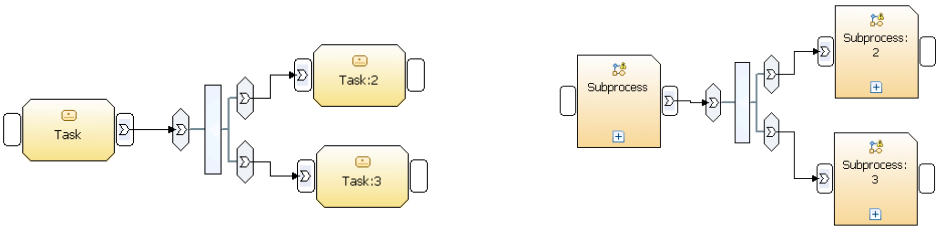


Fig. 3. Conceptual rule, meta model template and possible instances

Figure 3 b) shows a meta model template derived from rule r_1 shown in Figure 3 a). We make concrete the number of available nodes B_1, .. ,B_n and fix it to be $n = 2$. Further, we also fix the pin set structure of the nodes.

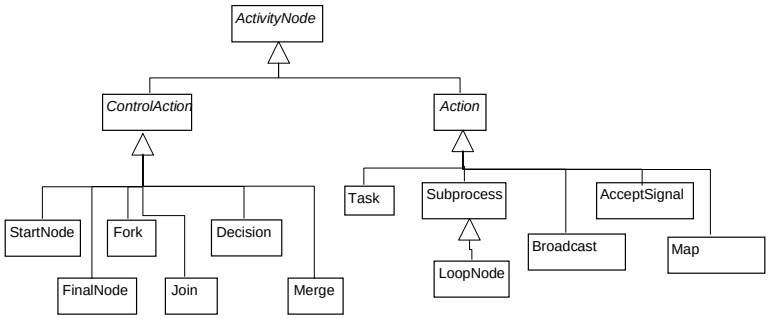


Fig. 4. Abridged metamodel extract

The remaining abstraction of the nodes is parameterized by the possible meta model classes. These can be identified when looking at the meta model (shown in Figure 4) and must be captured for each parameter in the meta model template. Figure 3 c) shows two template instances derived from the template.

Due to the abstraction process, one conceptual transformation rule can give rise to a number of different meta model templates. Figure 5 shows another template and possible instances. Here, we assume a different edge structure between the nodes. Note that when specifying the parameters for X, Y, Z one has to take into account well-formedness constraints of the language which e.g. do not allow that X includes the *StartNode*.

It is important to realize that meta model coverage testing is a classical case where white box testing is very powerful. This is because from each rule a number of templates can be derived that together can ensure a high degree of meta model coverage (per rule). If we obtain meta model coverage for each rule, we can deduce meta model coverage for the entire transformation.

After meta model templates have been defined, automatic generation of template instances yields a set of test cases for the transformation rule for which the template has been defined. Both the systematic instantiation of the templates and the testing can be automated. In the context of our work, a straightforward generation of templates has been implemented [2] that requires specification of the template and the suitable parameters. Based on this, a number of test cases is then generated automatically.

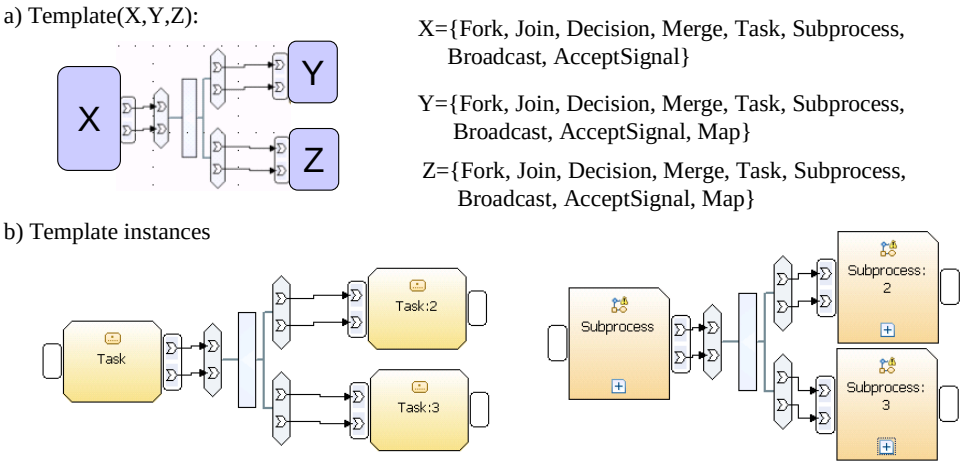


Fig. 5. Meta model template and possible instances

Beyond finding meta model coverage errors, meta model coverage testing can also be applied to find both syntactic and semantic correctness errors as well as errors due to incorrect coding. For syntactic correctness, the test oracle is the tool environment which in our case can detect whether the transformation result is syntactically correct. With regards to semantic correctness, each result must be manually compared and evaluated.

A remaining question is whether each rule needs to have complete coverage of the meta model or whether sometimes partial coverage can be tolerated if it is stated by suitable precondition constraints. An extension of the plain meta model coverage approach can take precondition constraints into account and give rise only to those test cases that fulfill the precondition constraints.

Meta model coverage testing is a powerful mechanism and can also be used for partially ensuring that constraints hold for the model that is created by the transformation.

However, as test cases for meta model coverage are derived directly from a transformation rule, this technique has its limitations for those cases in which constraints are formulated for a number of model elements: If these model elements are not part of a certain rule, no test case generated using meta model coverage testing will be a suitable test case. This is why in the next section we present a technique that, given a constraint, aims at construction of test cases for this particular constraint.

4.3 Using constraints for construction of test cases

Typically, the meta model of a language also specifies well-formedness constraints. These constraints can be expressed using the Object Constraint Language (OCL) or in natural language. Violations of constraints give rise to syntactic correctness errors. As constraints can be violated by the interplay of several transformation rules, they cannot be completely detected by meta model coverage testing.

As a consequence, we believe that existing constraints specified in the language specification should be used to construct interesting test cases that aim at discovering errors due to the violation of constraints. As a transformation changes model elements, it needs to be tested that all constraints that may be violated due to the change hold after applying a transformation. We can test constraints both on the rule and transformation level.

After identification of the changed model elements, we take those constraints into consideration that are dependent on the model elements changed. A constraint is independent of a model element if the existence or value of the model element instance does not influence the value of the constraint, otherwise it is dependent.

The idea to construct test cases to ensure constraints after application of the transformation is then as follows:

- Identify model elements changed by the transformation.
- Identify constraints that are dependent on these model elements.
- For each constraint, construct a test case that checks validity of the constraint under the transformation.

The test oracle for these tests is again the execution environment which in our case checks the constraints after application of the transformation.

An important issue is how we can detect which model elements are changed by the transformation, in the absence of a complete specification of the transformation rules. Partially, these elements can be detected when regarding the conceptual rule. At the same time, one can also obtain this information directly from the programmer.

With regards to the CAToPinset transformation, the model elements changed by r_1 are the pin set of A, because r_1 extends the pin set by adding an additional pin. Furthermore, edges are affected because r_1 changes their source or target nodes. In a similar way, we can find model elements changed by the other rules.

In our example of business process models, some of the constraints that are dependent on the changed model elements are:

- C1: A final node has one incoming edge.

– C2: An initial node has one outgoing edge of type control flow.
– C3: A Loopnode has one regular output pin set.
– C4: A Map has at least one output object pin.

All constraints are concerned with edges or with pin sets and are thus dependent on the changed model elements.

Given a constraint, we construct a test case for it as follows: Constraints can be divided into positive constraints requiring the existence of model elements and negative ones requiring the non-existence of model elements. In both cases, we try to create test cases that, after the transformation has been applied, can result into a violation of the constraint.

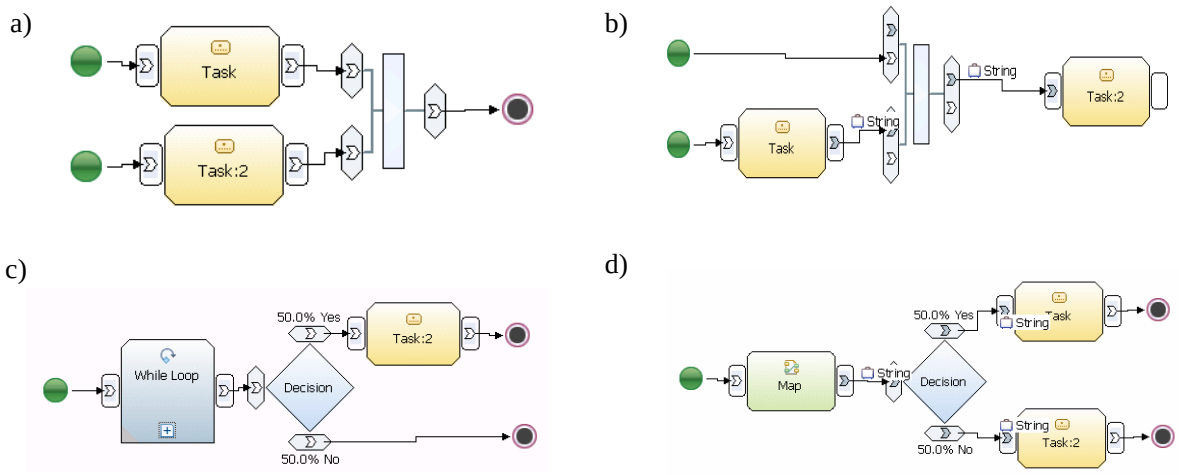


Fig. 6. Test cases for constraints

For example, with regards to constraint C_1, which requires that a final node has one incoming edge, we try to create a test case that after transformation results in the situation that the final node has two incoming edges. Figure 6 a) shows such a test case. An incorrect implementation will simply remove the *join* node and try to reconnect the incoming edges to the final node, which of course results into a syntactically incorrect model. Figure 6 b) shows a test case for C_2 (removal of the *join* node can lead to the creation of a *String* data flow edge from the start node, if incorrectly coded). In Figure 6 c)d) we present similar test cases for the constraints C3 and C4. All of these test cases have revealed errors in the implementation of the model transformation CAToPinset.

4.4 Using rule pairs for testing

Another source of errors arises from the interplay of rules: The application of one rule at some model element in the model might inhibit the application of another rule at the same model element. The property of confluence requires that the application of transformation rules on the same or an equivalent model yields the same result. As

stated in [19], confluence of transformations need not always be ensured. However, it is important to detect whether the overall transformation is confluent because this can cause very subtle errors that are difficult to detect and reproduce. Confluence errors can give rise to syntactic as well as semantic errors.

In theory, the concept of parallel independence [6] of two rules has been developed which requires that all possible applications of the two rules do not inhibit each other i.e. it is always the case that if one rule r_1 was applicable before applying r_2 it is also applicable afterwards.

If two rules are not parallel independent, they might give rise to confluence errors. To detect such errors at design time, we have discussed in [19] a set of criteria which are based on the construction of critical pairs. The idea of a critical pair is to capture the conflicting transformation steps in a minimal context and analyze whether a common successor model can be derived. For exact calculation of critical pairs, a complete specification of the rules is required, e.g. in one of the model transformation languages.

In testing, the challenge is to construct test cases systematically that lead to the detection of confluence errors. In our approach, a complete specification of the transformation rules is not available. We can still use the conceptual rules for construction of test cases as follows: Based on the idea of critical pairs, we argue that it is useful to construct systematically all possible overlapping models of two rules. These overlapping models can represent a critical pair and can thus be used to *test* for the existence of a confluence error.

The overlapping models can be constructed systematically. The idea is to take the left sides of two rules and then calculate all possible overlaps of model elements. Based on an overlap, a model is constructed which joins the two models at the overlapping model elements. If the overlapping model is syntactically incorrect, it is discarded. Otherwise, it is taken as a test case.

For example, for rules r_1 and r_3 in Figure 2 one possible overlap is to identify the node B_1 of r_1 with node A of rule r_3. The result is shown in Figure 7 a), assuming $n = 2$, a *task* node type for all nodes and a simple edge structure. Figure 7 c) shows another test case constructed from overlapping the rules. This test case gave rise to a confluence error because removing the *fork* leads to the construction of a pin set with two pins at the *decision* which is invalid and leads to an execution error because in our environment the construction of invalid intermediate models is not possible. If the fork is removed first, then no invalid model is constructed. Note that in a different execution environment supporting invalid intermediate models, the test case would not lead to an execution error.

Figure 7 b) and d) show further test cases constructed from overlapping rules r_2 and r_4, and r_1 and r_3, respectively.

The idea of templates introduced above can also be used for rule pairs: If instead of two rules two template rules are used for constructing the overlapping rule, then the overlapping rule will be a template and can be instantiated automatically. This leads to an increased number of test cases that can be constructed automatically. To detect confluence errors automatically, the execution of test cases can require human intervention if no ability to compare results of test case execution automatically is available.

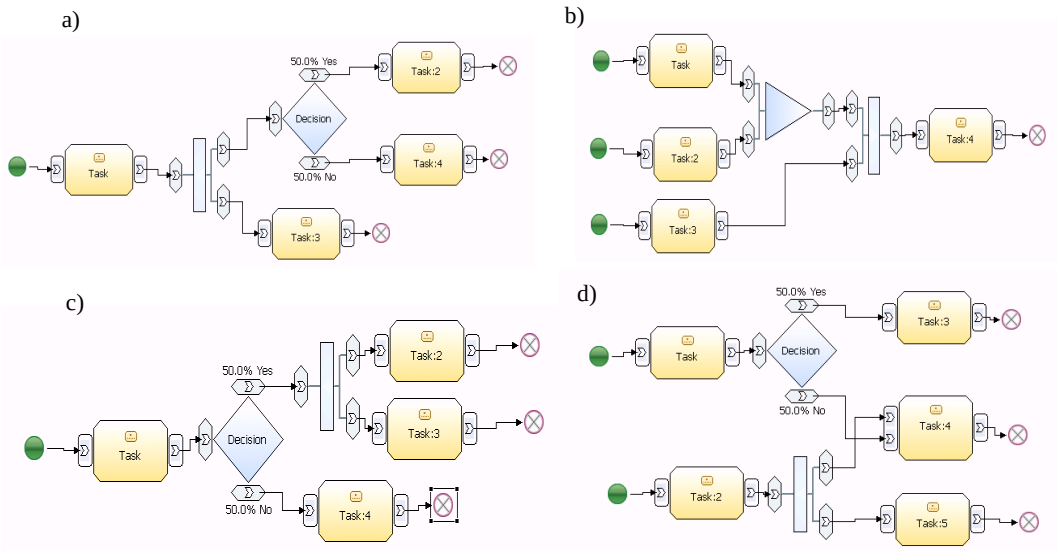


Fig. 7. Test cases for confluence (adapted from [2])

5 Related Work

With the advance of model-driven engineering, the idea of applying a model-driven approach to testing has also received increasing attention. Heckel and Lohmann [14] describe how Web applications can be tested in a model-driven approach. One key idea is to use models as the test oracle for specifying the output of a test. A similar approach could be applicable for model transformations: If each transformation rule is completely specified in a transformation language, testing the implementation of the transformation can use the specification as a test oracle.

Mottu et al. [23] describe mutation analysis testing for model transformations. They identify four abstract operations within model transformations: Navigation within the model, filtering of elements, output model creation and input model modification. Based on these operations, they define mutation operators. For example, a possible mutation operator for navigation changes the association used within the navigation. A mutation operator for model creation is to replace a creation of an object with a parent class. Their mutation analysis can be used to ensure the quality of the test case set and has therefore a different focus compared to our work.

Fleurey et al. [11] describe an approach to generate test models. They first calculate the effective meta model for the transformation and then determine a coverage criterion based on this effective meta model which is similar to our concept of a meta model template for a rule. The coverage criterion is used for generating test models. Overall, their approach can be considered a black box approach for model transformation testing because they do not explicitly take different rules into account. Such an approach can be seen as complementing our work which is based on white box testing. We believe that if white box testing has succeeded (e.g. applying the techniques presented in this paper), it can be followed by large-scale black box testing.

Markovic and Baar [21] study how common refactoring operations on class diagrams such as moving an attribute or an operation can be expressed in the transformation language QVT and how OCL constraints are influenced by these operations. They describe a means how OCL constraints can be automatically refactored in these circumstances. On the contrary to their work, we use OCL constraints to construct test cases.

Recent work by Baudry et al. [3] summarizes model transformation testing challenges. They first discuss current limitations of black box testing by generating arbitrary input models. They also introduce the idea of testing the output of a transformation e.g. for a UML to Java transformation the Java output program can be checked by executing it. Another idea is to use patterns for specifying input or output of a transformation. This last idea is somewhat related to our approach, where the conceptual rule abstracts from the details of a transformation.

In the area of graph transformation which can be used as a formal basis for model transformation, there have been several approaches that deal with verification of graph transformations. With regards to testing, Darabos et al. [8] describe a way to generate test cases for graph pattern matching. They first extract logical criteria for matching a rule in form of a Boolean expression and then transform the Boolean expression into a combinatorial circuit. Using fault injection into the circuit, mutations of the left side of the rule are created and form a set of test graphs. Our work can be seen as complementary to their work because their fault model is different from ours and we do not assume a graph-transformation-based implementation.

6 Conclusions

Validation of model transformations is a key issue to ensure their quality and thereby enables the vision of model-driven architecture become reality. In the context of business-driven development, model transformations are used for transforming more abstract models into more concrete ones and to move between different representations of models. In this paper, we have reported our first experiences with testing a set of model transformations for business process models systematically.

We have proposed three techniques which follow a white box testing approach. Using this approach, we have been able to significantly improve the quality of the model transformations under development. Both the meta model coverage technique as well as the construction of test cases driven by constraints has shown the existence of a number of errors. Rule pairs have indicated fewer errors, possibly due to the low number of rules.

If model transformation rules are completely specified already at the model level, using one of the model transformation languages, our techniques can also be applied but may require modifications. With regards to meta model coverage testing, one could manually abstract from a set of related rules and construct a conceptual rule which can be transformed into a template. The technique for using constraints for construction of test cases can make use of the transformation rules for automatically identifying the elements changed and can then be applied in the same way as described above.

Further, a complete specification of rules also enhances the ability to use rule pairs for construction of test cases for confluence.

In our environment, we make the assumption that intermediate models must be correct with regards to the language specification. Sometimes, it is rather the case that either the model is correct only after application of the entire transformation or at certain checkpoints during the transformation. In such a case, the meta model coverage technique must apply the entire transformation to a generated test case.

There remain further challenges that we have not been able to address yet, for example, the automation of constructing test cases from OCL constraints. Here we see two possible improvements, firstly the automatic detection of constraints that could be violated by providing an algorithm that, given a meta model element, finds all attached constraints. Secondly, the automatic conversion of such a constraint into a possible test case. Future work also includes the elaboration of tool support in order to fully automate testing of transformations.

References

1. IBM WebSphere Business Modeler. http:///www-306.ibm.com/software/integration/ wbimodeler/.
2. M. Abd-El-Razik. *Business Process Normalization using Model Transformation*. Master thesis, The American University in Cairo, in collaboration with IBM, 2006. In preparation.
3. B. Baudry, T. Dinh-Trong, J.-M. Mottu, D. Simmonds, R. France, S. Ghosh, F. Fleurey, and Y. Le Traon. Model Transformation Testing Challenges. In *Proceedings of IMDT workshop in conjunction with ECMDA'06, Bilbao, Spain*, 2006.
4. R. Binder. *Testing Object-Oriented System Models*. Addison Wesley, 1999.
5. P. Braun and F. Marschall. BOTL - The Bidirectional Objekt Oriented Transformation Language. Technical report, Fakultät für Informatik, Technische Universität München, Technical Report TUM-I0307, 2003.
6. A. Corradini, U. Montanari, F. Rossi, H. Ehrig, R. Heckel, and M. Löwe. Algebraic Approaches to Graph Transformation Part I: Basic Concepts and Double Pushout Approach. In G. Rozenberg, editor, *Handbook of Graph Grammars and Computing by Graph Transformation, Volume 1: Foundations*, pages 163–245. World Scientific, 1997.
7. 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.
8. A. Darabos, A. Pataricza, and D. Varró. Towards Testing the Implementation of Graph Transformations. In *Proceedings of the 5th International Workshop on Graph Transformations and Visual Modeling Techniques*, pages 69–80, 2006.
9. T. Dinh-Trong, N. Kawane, S. Ghosh, R. France, and A. Andrews. A Tool-Supported Approach to Testing UML Design Models. In *Proceedings of ICECCS'05, Shanghai, China*, June 2005.
10. K. Ehrig, J. M. Küster, G. Taentzer, and J. Winkelmann. Generating Instance Models from Meta Models. volume 4037 of *LNCS*, pages 156–170. Springer, 2006.
11. F. Fleurey, J. Steel, and B. Baudry. Model-Driven Engineering and Validation: Testing model transformations. In *Proceedings SIVOES-MoDeVa Workshop*, November 2004.
12. C. Ghezzi, M. Jazayeri, and D. Mandrioli. *Fundamentals of Software Engineering*. Prentice-Hall, 1991.

13. R. Hauser and J. Koehler. Compiling Process Graphs into Executable Code. In *Proceedings of the 3rd International Conference on Generative Programming and Component Engineering,*, volume 3286 of *LNCS*, pages 317–336. Springer-Verlag, October 2004.
14. R. Heckel and M. Lohmann. Towards Model-Driven Testing. In *Proceedings of the International Workshop on Test and Analysis of Component-Based Systems (TACoS'03)*, volume 82, 2003.
15. F. Jouault and I. Kurtev. Transforming Models with ATL. In J.-M. Bruel, editor, *Satellite Events at the MoDELS 2005 Conference, MoDELS 2005 International Workshops, Doctoral Symposium, Educators Symposium, Montego Bay, Jamaica, October 2-7, 2005, Revised Selected Papers*, volume 3844 of *LNCS*, pages 128–138, 2005.
16. 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.
17. J. Koehler, R. Hauser, J. Küster, K. Ryndina, J. Vanhatalo, and M. Wahler. The Role of Visual Modeleling and Model Transformations in Business-Driven Development. In *Proceedings of the 5th International Workshop on Graph Transformations and Visual Modeling Techniques*, pages 1–12, 2006.
18. P. Kruchten. *The Rational Unified Process*. Addison Wesley, 2003.
19. J. M. Küster. Definition and validation of model transformations. *Software and Systems Modeling*, 2006. DOI: 10.1007/s10270-006-0018-8, to appear.
20. J. M. Küster, K. Ryndina, and R. Hauser. A Systematic Approach to Designing Model Transformations. Technical report, IBM Research, Research Report RZ 3621, July 2005.
21. S. Markovic and T. Baar. Refactoring OCL Annotated UML Class Diagrams. In Lionel C. Briand and Clay Williams, editors, *Model Driven Engineering Languages and Systems, 8th International Conference, MoDELS 2005, Montego Bay, Jamaica, October 2-7, 2005, Proceedings*, volume 3713 of *LNCS*, pages 280–294. Springer, 2005.
22. T. Mitra. Business-driven development. IBM developerWorks article, http://www.ibm.com/developerworks/webservices/library/ws-bdd, IBM, 2005.
23. J.-M. Mottu, B. Baudry, and Y. Le Traon. Mutation Analysis Testing for Model Transformation. In A. Rensink and J. Warmer, editors, *Model Driven Architecture - Foundations and Applications, Second European Conference, ECMDA-FA 2006, Bilbao, Spain, July 10-13, 2006, Proceedings*, volume 4066 of *LNCS*. Springer, 2006.
24. Object Management Group (OMG). *UML 2.0 Superstructure Final Adopted Specification. OMG document pts/03-08-02*, August 2003.
25. Object Management Group (OMG). *Meta Object Facility (MOF) 2.0 Query/View/Transformation. Final Adopted Specification. OMG document ad/2005-11-01*, November 2005.
26. M. Utting, A. Pretschner, and B. Legeard. A taxonomy of model-based testing. Technical report, Department of Computer Science, The University of Waikato (New Zealand), Technical Report 04/2006, 2006.