Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 211 (2008) 75–85
www.elsevier.com/locate/entcs

Towards Testing the Implementation of Graph Transformations

Andrea Darabos[1] and András Pataricza[2] and Dániel Varró[3]

Department of Measurement and Information Systems
Budapest University of Technology and Economics
Budapest, Hungary

Abstract

We present a method for testing the implementation of graph transformation specifications focusing on test case generation for graph pattern matching. We propose an extensible fault model for the implementation of transformations based on common programmer faults and the technicalities of graph transformations. We integrate traditional hardware testing (combinational circuits) and software testing techniques (mutant generation) for generating test cases.

Keywords: Graph transformation, testing, test generation, pattern matching

1 Introduction

Due to the growing importance of transformations, a standardized Model Driven Architecture (MDA) based model transformation method has been requested in the *OMG Request for Proposal MOF 2.0 Query / View / Transformations* [17]. Graph transformation, which provides a rule and pattern-based manipulation of graphs, is a promising technology for model transformations as evaluated by a taxonomy presented in [15].

The separation of the design and execution time of model transformations is a recent tendency today (see GreaT, Fujaba, Viatra), by providing both an interpreted engine and compiled transformation plug-ins as platform specific implementations (Figure 1). In case of graph transformations, the implementation can be derived by

[1] E-mail:andrea_darabos@t-online.hu
[2] E-mail:pataric@mit.bme.hu
[3] E-mail:varro@mit.bme.hu
[4] This work was partially supported by the SENSORIA European project (IST-3-016004). The third author was also supported by the Bolyai Scholarship.

doi:10.1016/j.entcs.2008.04.031

hand or generated automatically as described in [3]. However, even if these plug-ins are generated automatically, these implementations can be erroneous.

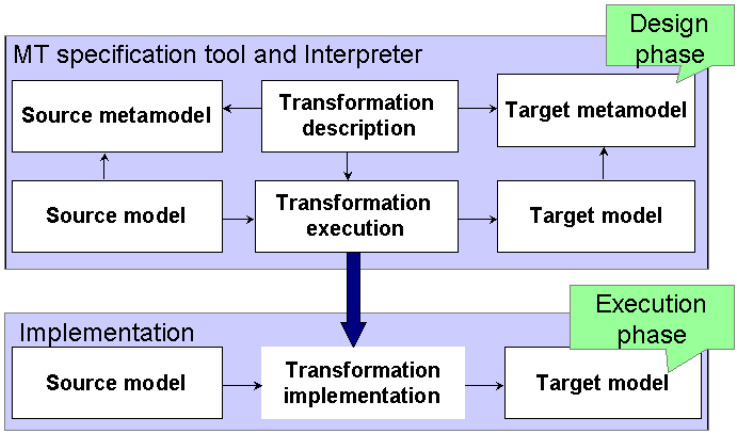

Fig. 1. The meaning of model transformation (MT) implementation

In order to detect conceptual flaws in transformations, typically, either verification (termination, confluence, semantic correctness, etc.) and/or testing techniques are applied. In general, verification is mainly used in the design phase of transformations, while testing is appropriate in the implementation phase, when a stand-alone transformation plug-in has been created for the corresponding specification. Testing has typically two main advantages: *(i)* it can be used for large models without combinatorial explosion, *(ii)* tests are executed directly on the implementation, which in case of model checking often cannot be guaranteed.

Our aim is to test stand-alone graph transformation implementations by generating test cases from graph transformation specifications. In this paper, we focus on the graph pattern matching phase, which is considered to be the most problematic phase of graph transformations.

We propose a fault model to incorporate potential flaws in the implementation. Test generation is performed by using a combinational circuit representation derived from the preconditions of graph transformation (further: GT) rules. Possible faults are mapped to stuck-at-faults (a signal lines is assumed to be stuck at a fixed logic value, regardless of the inputs), as there are various hardware testing methods for the combinational circuit and this fault model. With the help of systematic fault injection, single binary (stuck-at-faults) faults are inserted into the circuit and test vectors are calculated. The exact test cases are generated by mutation rules in the form of test graphs.

2 Graph transformations in Modeling Languages

2.1 Metamodels and models

The abstract syntax of a modeling language is defined by a metamodel (MM). It can be represented formally as a type graph. The instance model or instance graph (M)

is a well-formed instance of the metamodel and describes concrete systems defined in the modeling language. The finite automaton will serve as a running example throughout the paper. To demonstrate our steps, the very simple domain of the finite automaton and belonging instance models are depicted in Figure 2.

Example 2.1 According to the metamodel, a well-formed instance of a finite automaton is composed of *states* and *transitions*. A transition is leading between its *from* state and *to* state. The initial states of the automaton are marked with *init*, the active states are marked with *current* edges. Special, e.g. colored states, are definable by inheritance.

A sample automaton `a1` consisting of three states (`s1`, `s2`, `s3`) and three transitions between them t1 (leading between `s1` and `s2`), `t2` and `t3` is depicted as an instance model. We can notice that the initial state of `a1` is `s1`.

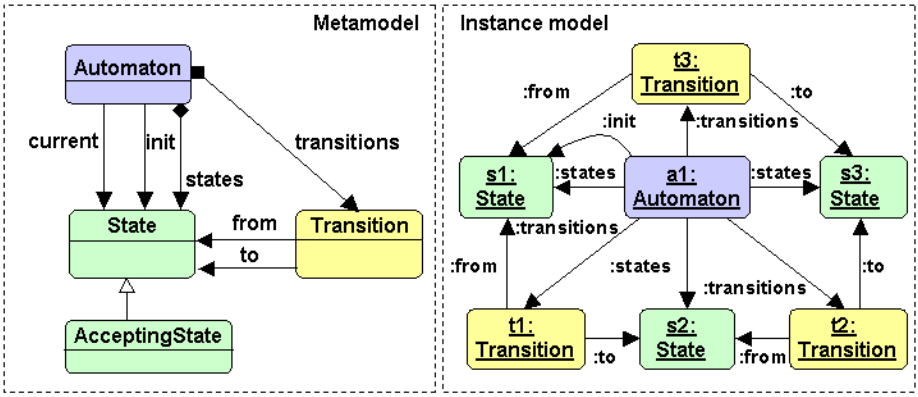

Fig. 2. Metamodel and instance model of finite automata

2.2 Graph transformations

Graph transformation [20] is a pattern and rule based formalism for the manipulation of graph models. On rule application, a graph is transformed by replacing a part of it by another graph. With the definition of a metamodel and a set of rules over that metamodel the dynamic changes of an initial model can be described. On rule application, a graph is transformed by replacing a part of it by another graph.

A graph transformation rule R contains a left-hand side graph LHS, a right-hand side graph RHS, and negative application condition graphs NACs. The LHS and the NAC graphs are together called as the precondition of the rule R.

The application of a rule to a instance model M (which is instance model of the metamodel) replaces a matching of the LHS in M by an image of the RHS (formally there is a graph morphism between the LHS and the instance model M). This is performed by *(i)* finding a matching of LHS in M, *(ii)* checking the negative application conditions NACs (which prohibit the presence of certain objects and links) *(iii)* removing a part of the instance model (that can be mapped to LHS but not to RHS) yielding the context model, and *(iv)* gluing the context model with an image of the RHS together by adding new objects and links (that can be mapped

to the RHS but not to the LHS) and obtaining the derived model M'. A graph transformation is a sequence of rule applications from an initial model M_i.

Typically, the most critical phase of a graph transformation step is graph pattern matching, i.e. to find a single (or all) occurrence(s) of a given graph in a instance model M.

Example 2.2 The dynamic semantics of finite automatons can be described with the help of graph transformation rules. The example rule depicted in Figure 3 shows the firing of a transition. If the `S1` state of the `A1` automaton is active (there exists a `C1` edge between them), and there exists a transition, which leads from `S1` to `S2`, then the rule is applicable, and the current state of the automaton will be `S2`.

The process of pattern matching can also be illustrated. If we regard the instance model in Figure 2 as an instance model, and we assume, that there is an additional `current` edge from `a1` to `s1` in it, then the example GT rule can be applied onto this instance graph: with variable instantiation `A1-a1`, `S1-s1`, `T1-t1`, `S1-s2`, `C1-c1`, `St1-st1`, `St2-st2`, etc. The rule can be applied here, and as a result, the `current` edge will be leading from `a1` to `s2` in the instance graph.

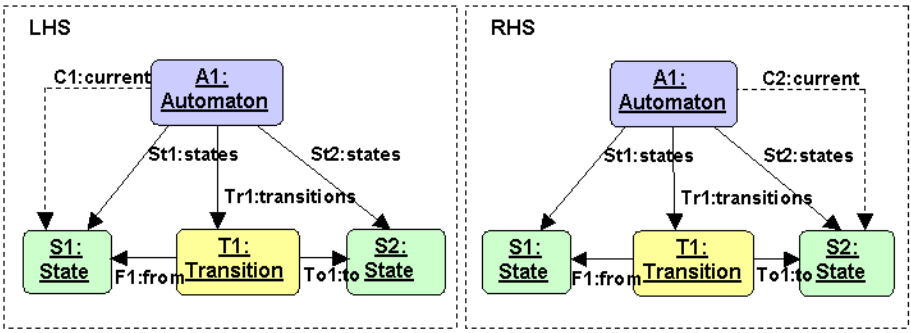

Fig. 3. The fire GT rule of a finite automaton

3 Fault Model

For the testing of graph transformation implementations, (in fact, for any kind of testing), a formal fault model has to be defined for the possible fault types in the implementations. This was inspired by object-oriented testing and hardware-based testing techniques, assuming similarities between traditional software developers and transformation developers and these were adopted for graph transformations. Thus the following fault types were declared for the pattern matching phase (the fault model is extensible, additional faults can be defined via the same method):
General implementation faults (based on programmer's experience):

- *Omission fault.* In a graph transformation implementation, the omission fault means, that some elements are missing from the implementation of the pattern matching criteria, described originally in the specification. This can lead to situations, when graph transformation rules can be matched to a smaller subset of

graph elements than it was specified (e.g. node `s1` or `states` edge is missing from the implementation of pattern matching).

- *Interchange fault.* An interchange fault means, that the criteria was implemented with incorrect type definitions (e.g. too specific type used, `AcceptingState` instead of `State`). Most commonly, the programmer makes such a fault in the generalization hierarchy.
- *Side effect fault.* This fault means unnecessary, redundant elements in the implementation, having more criteria defined for pattern matching than those specified (e.g. additional nodes or edges were implemented in the criteria).

GT specific faults of the pattern matching phase:

- *Dangling edge production fault.* The production of dangling edges is not allowed in the *DPO Double-Pushout* approach [5], therefore this criterion must be investigated.
- *Violation of injectivity fault.* If only injective matchings are allowed, the non-injective matching of elements (different nodes in a GT rule have the same image in the match) is a violation of injectivity fault.

  In the future we also plan to consider non-injective matchings, where the violation of identification condition needs to be investigated.

The pattern matching criteria for each rule are defined by the LHS of rules in the specification. It is assumed, that the graph pattern is well-typed (syntactically correct), therefore only implementation (semantic) errors are aimed to be detected.

4 Test Case Generation for Graph Pattern Matching

4.1 High level Overview

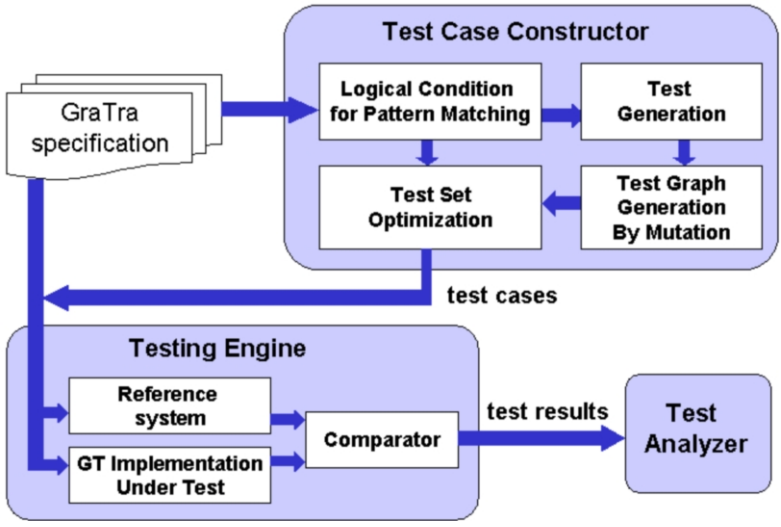

Fig. 4. Testing workflow

Figure 4 provides a brief overview of our test case generation approach. There are three primary components of the envisaged framework: test case constructor, testing engine and test analyzer. For the current paper, we only focus on pattern matching, but the conceptual elements of the framework are more general, like the test case constructor and the testing engine. The input of the test case constructor is a set of GT rules as specification, and the output is a set of test cases in the form of test graphs for the testing of the implementation of the transformation.

The main steps of the test case generation are the following:

- *Pattern Matching Criteria* The logical criteria for the successful matching of each rule is extracted from the transformation specification in the form of a Boolean expression. The formula is satisfied, when a successful matching is found for the belonging GT rule. The idea is to reuse existing techniques for hardware testing, therefore the Boolean formula is depicted in form of a combinational circuit, for which traditional test generation algorithms can be applied.
- *Test Generation* With systematic fault injection, single faults are injected into the inputs of the circuit. For its simplicity, the method of Boolean differences [16,21] is applied here which generates binary test vectors for stuck-at-faults in the combinational circuit representing the pattern matching. The method of Boolean differences guarantees that with the generated test vectors the fault is observable on the output of the circuit. If a variable in the generated test vector is one, then the corresponding condition is satisfied, else it is not satisfied. For further details see Section 4.2.
- *Test Graph Generation* After the test vectors are calculated with binary values, the corresponding test graphs have to be produced. The LHS copy of the tested GT rule is created, and with mutation rules the specified faults are injected into the LHS copy graph. The calculated test vectors control the process of mutation rule application. The resulting test graphs are the possible realizations of the calculated logical test vectors, created according to the fault model introduced in Section 3. For instance, a binary test vector expressing that some node has a wrong type can have multiple realizations, e.g. a more general type can be one case (e.g. `AutomatonElement` instead of `State`), or a more specific type (`AcceptingState` instead of `State`) in the generalization hierarchy can be implemented. Thus, for each test vector, multiple test graphs can be created. For further details see Section 4.3.
- *Test Set Optimization* The set of produced test graphs should be examined for test optimization, in order to create a more compact set of test cases, it should be optimized. Naturally, it has to be decided, whether the aim is only fault detection or diagnosis as well. In the latter case, test compaction can only be carefully applied, not to loose information for diagnosis.

After the test set is created, it is passed to the *Testing Engine*. The testing engine is responsible for executing the transformation specification on the given test graph both in the reference system (a GT Interpreter tool) and the transformation implementation. The comparator compares the results of pattern matching of

both components, and collects the results for each test graph. Here an important restriction has to be made: only those GT rules are suitable for testing, for which the difference of RHS and LHS is nonzero. It means, that in order to being able to compare the results of pattern matching, rule application must make visible changes on the test graph. The test analyzer collects and visualizes the results of the test engine.

Due to space restrictions, we discuss in further details only the test generation and test graph generation phases. The interested reader can find more about this topic in [6].

4.2 Details of representation and test generation

The general, formal criteria for a match are presented in [23]. The idea is to describe these criteria for each GT rule in the rule set of the specification, and to create a combinational circuit representation of this Boolean formula, which will supply us with the usability of traditional testing methods. The formula evaluates to 1, if a match fulfills the defined criteria meaning that the pattern matching was successful.

The construction of the formula follows the upcoming scheme:
Existence of images of the LHS elements in the instance graph $\wedge$
$\wedge$ Correct type of elements in the match $\wedge$
$\wedge$ Attribute conditions satisfied by the matched attributes $\wedge$
$\wedge$ Isomorphism/homomorphism condition $\wedge$
$\wedge$ Fulfillment of dangling edge conditions $\wedge$
$\wedge$ No violation of NACs

For the example rule showed in Figure 3, the criteria is the following:
Automaton(a1) $\wedge$ State(s1) $\wedge$ Transition(t1) $\wedge$ State(s2) $\wedge$
$\wedge$ current(c1, a1, s1) $\wedge$ states(st1, a1, s1) $\wedge$ states(st2, a1, s2) $\wedge$
$\wedge$ transitions(tr1, a1, t1) $\wedge$ from(f1, t1, s1) $\wedge$ to(to1, t1, s2) $\wedge$
$\wedge$ s1 $\neq$ s2

For the presented example, no dangling edge or NAC condition was present, therefore we briefly summarize the construction of these criteria.

- *The dangling edge condition*: All nodes and edges in LHS of a GT rule R but not in RHS are deleted when the rule is applied. When applying this rule R on a instance graph, all the edges *to* and *from* these nodes which are not part of the match are the dangling edges. The dangling edge condition is fulfilled, if no dangling edges will be produced on rule application.
- *The NAC condition*: If only single, non-hierarchical NAC graphs are used, the NAC condition is satisfied, if its elements cannot be found in the match. The Boolean formula can be written for the NAC graph as above, and it is inverted before connecting it to the pattern matching criteria. In case of hierarchical NAC conditions, the Boolean formula of a lower level NAC is inverted before connecting it to the higher level condition. More on this topic can be found in [19].

- *The injectivity condition* is formalized as follows: ∀ X, Y nodes ∈ LHS, and p, q images of X and Y in the instance graph: $X \neq Y \Rightarrow p \neq q$ which means, that two different elements of the LHS of a given GT rule cannot be mapped to the same element in the match. In our example, this was the `s1` ≠ `s2` condition for states `S1` and `S2`.

The combinational circuit generated from the criteria is depicted in Figure 5.

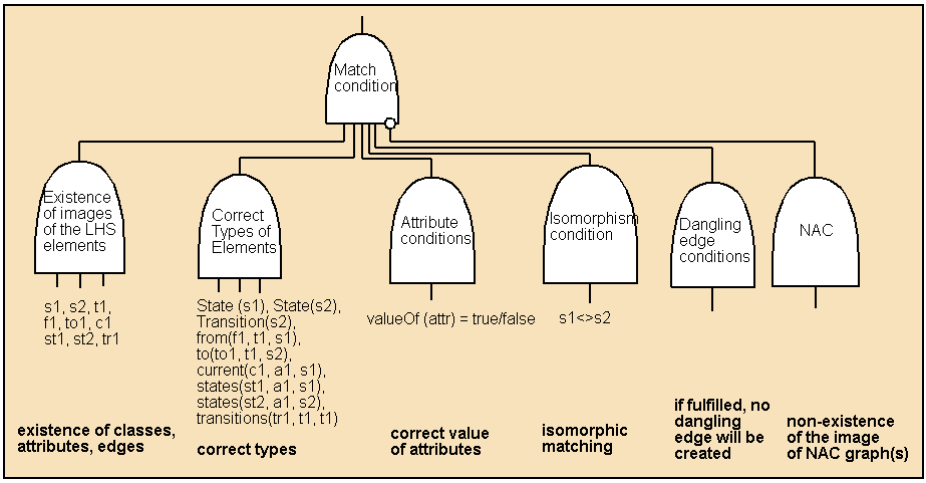

Fig. 5. Combinational circuit representation with details of the example

The test vector generation is performed on the combinational circuit with systematic fault injection, with the help of the method of Boole differences resulting in binary test vectors for each GT rule. In our example, a test vector for the omission fault of the `Automaton(a1)` element from the criteria is the following: (0,1,1,1,1,1,1,1,1,1,1) for (a1,s1,t1,s2,c1,st1,st2,tr1,f1,to1,s1≠s2).

Some details of the processing of these test vectors and the test graph generation are discussed in the next section.

4.3 Details of test graph generation

The generation of test graphs is based on a set of mutation GT rules, all of which define the injection of faults of fault types defined in the fault model (Section 3). These rules describe the possible realizations of a given fault type: e.g. in case of the interchange fault type (where we supposed faults inside the generalization hierarchy), the fault can originate from a too specific or a too general type realized by the implementation. The mutation rules are metatransformation rules, which are applied on the LHS copy of a GT rule from the specification as instance graph.

Returning to our example of the finite automaton, after applying the mutation rule depicted in Figure 6 on the `State` entity of the GT rule LHS copy (Fig. 3), we would gain an `AcceptingState` entity, which is of a more specific type. A test graph including this fault would test, whether the implementation regards the correct type of element when pattern matching or not. The original GT rule (Fig. 3) is applied

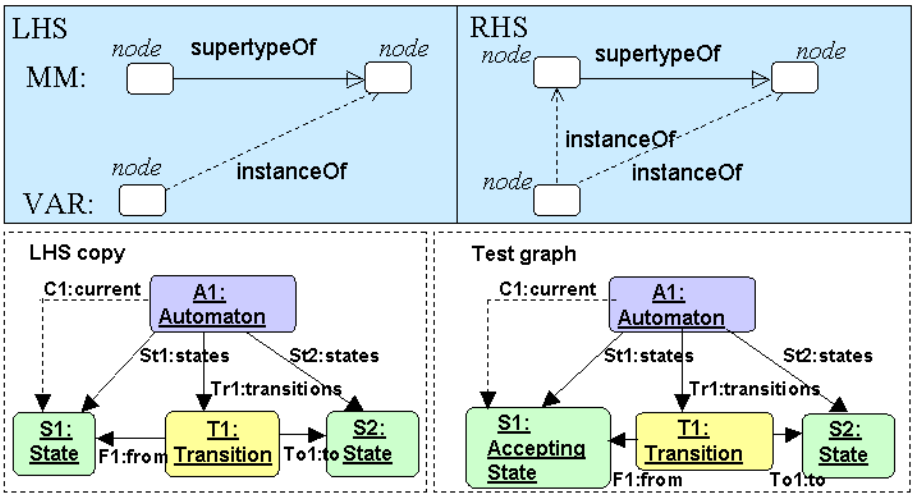

Fig. 6. Mutation rule for one realization of the interchange fault and the generated test graph from the LHS copy of the example

onto this test graph, and the success or failure of pattern matching indicates the correctness of the implementation.

The test vectors calculated on the combinational circuit can be regarded as a control structure for the mutation rules; they define, which test graphs have to be produced with the help of mutation rules. The instance graph is the LHS copy of the GT rule under test, and the result graph is the test graph. For each test vector, a new LHS copy is created, and the according to the possible mutation GT rules, as many test graphs are created as the number of different possible mutation rules were defined for this fault type. Thus, a *test graph set* is generated for each test vector.

5 Related Work

The formal correctness analysis of model transformations has been already investigated in the literature.

Syntactic correctness and completeness was examined in [9] and sufficient conditions guaranteeing the termination and uniqueness of transformations were set up in [12] based on the critical pair analysis [10] technique.

An automated formal verification technique is presented in [18,7,24] based on various model-checking techniques to prove semantic correctness criteria in graph transformation systems starting from a concrete initial graph. A static analysis technique is proposed in [1] to investigate the correctness of graph transformation systems by using a Petri net abstraction. A tool for checking inductive invariants has been presented recently in [4]. To guarantee the preservation of constraints during model transformations, aspect-oriented techniques are proposed in [13].

However, much less results are available for the testing of graph transformations. Jeff Gray underlined the importance of testing model transformations and presented a model transformation testing framework in [14]. This framework fo-

cuses on automating test execution, i.e. to automatically compare test results with the expected behavior, while we focus on automatic test generation.

The testing of code generators specified by graph transformation rules has been addressed in the literature by adapting well-known test strategies such as test case generation by model checking [2] or the classification tree method [22]. In [11] tests are generated for black-box implementations of web services based upon domain partitioning. While the overall goal i.e. to derive test cases directly from GT rules is similar, we assume that implementation is strongly linked to the GT specification, furthermore we use systematic fault injection and combinational circuit testing techniques in the background.

On the tool level, one pioneer is FUJABA which generates JUnit test cases [8] from graph transformations specified by graphical story diagrams. This approach focuses on the correctness of model manipulation steps (based on the right-hand side), which nicely complements the results of our current paper.

6 Conclusion and Future Work

In this paper we presented a method for the test generation for the pattern matching of graph transformation implementations, and a test execution method as well. Our primary goal was to use well-known hardware/software testing techniques and the extendibility of the fault model, therefore we elaborated a method which can be used for any graph transformation implementation and extended on demand with e.g. more fault types or with the injection of multiple faults.

The complexity issues of this problem will be part of measurements of the implementation, but it can be said that the pattern matching criteria - from which the test generation is performed - is comparable, proportional with the LHS size of the GT rules of the specification. Therefore, as the size of GT rules is generally much smaller than the size of instance models, it seems to be surmountable. Another important question is the size of test graphs, which, in the presented version is equal to the size of GT rule LHS graphs, as test graphs are generated with the slight modification of corresponding GT rule LHS graphs.

Our aim is to extend our testing method with the consideration of rule application, the RHS or postcondition of GT rules as well. Secondly, we plan to improve the fault model with control flow faults and design methods for testing the control structure of graph transformations as well. Furthermore, more work has to be done in the area of test set optimization. It is a future goal to examine the usability of our method for not only fault detection, but also for diagnosis as well, and to try out our method on graph transformation implementations.

References

[1] Baldan, P., A. Corradini and B. Koenig, *A static analysis technique for graph transformation systems*, LNCS, **2154** (2001), pp. 381–395.

[2] Baldan, P., B. König and I. Stürmer, *Generating test cases for code generators by unfolding graph transformation systems*, in: *ICGT'04*, LNCS, **3256**, (2004), pp. 194-209.

[3] Balogh, A., D. Varró, A. Pataricza and G. Varró, *Generation of platform-specific transformation plugins for EJB 3.0*, to appear for SAC'06.

[4] Becker, B., H. Giese and D. Schilling, *A plugin for checking inductive invariants when modeling with class diagrams and story patterns*, in: *Proc. of the 3rd International Fujaba Days*, 2005, pp. 1–4.

[5] Corradini, A., U. Montanari, F. Rossi, H. Ehrig, R. Heckel and M. Loewe, "in [20] Basic Concepts and Double Pushout Approach," World Scientific, 1997 pp. 163-245.

[6] Darabos, A., A. Pataricza, D. Varró, *Testing the implementation of graph transformations, Technical Report*, in:
http://www.inf.mit.bme.hu/varro/publication/TR-2006-01.pdf.

[7] Dotti, F. L., L. Foss, L. Ribeiro and O. M. dos Santos *Verification of Distributed Object-Based Systems*, in: *FMOODS*, 2003, pp. 261-275.

[8] Geiger, L., C. Schneider and C. Reckord, *Template- and modelbased code generation for MDA-Tools*, in: *Proc. of the 3rd International Fujaba Days*, 2005.

[9] Hausmann, J. H., R. Heckel and S. Sauer, *Extended model relations with graphical consistency conditions.*, in: *UML Workshop on Consistency Problems in UML-based Software Development*, 2002, pp. 61-74.

[10] Heckel, R., J. M. Küster and G. Taentzer, *Confluence of typed attributed graph transformation systems*, in: *Proc. 1st Intl Conf. Graph Transformation*, LNCS, **2505** (2002), pp. 161-176.

[11] Heckel, R., L. Mariani, *Automatic Conformance Testing of Web Services*, in: *Proc. Fundamental Approaches to Software Engineering* (2005).

[12] Küster, J. M., R. Heckel and G. Engels, *Defining and validating transformations of UML models*, in: *Proc. IEEE Symposium on Human Centric Computing Languages and Environments (HCC)*, 2003, pp. 145–152.

[13] Lengyel, L., H. C., T. Levendovszky, *Eliminating crosscutting constraints from visual model transformation steps*, in *7th International Workshop on Aspect-Oriented Modeling*, 2005.

[14] Lin, Y., J. Zhang and J. Gray, *Model comparison: A key challenge for transformation testing and version control in model driven software development*, in: *OOPSLA*, 2004.

[15] Mens, T. and P. V. Gorp, *A taxonomy of model transformation and its application to graph transformation*, in *GraMoT*, 2005.

[16] Miron, A., M. A. Breuer and A. D. Friedman, "Digital systems testing and testable design," Computer Sci. Pr., 1990.

[17] OMG, *MOF 2.0 query / views / transformations rfp ad/2002-04-10* (2002), URL: http://www.omg.org/cgi-bin/apps/doc?ad/02-04-10.pdf.

[18] Rensink, A., *The GROOVE simulator: A tool for state space generation*, In M. Nagl and J. Pfalz, editors, *Applications of Graph Transformations with Industrial Relevance (AGTIVE)*, LNCS, **3062** (2004), pp. 479-485.

[19] Rensink, A., *Representing First-Order Logic Using Graphs*, In *Graph Transformations: Second International Conference (ICGT)*, LNCS, **3256** (2004), pp. 319.

[20] Rozenberg, G., editor, "Handbook of Graph Grammars and Computing by Graph Transformations: Volume 1: Foundations," World Scientific, 1997.

[21] Sallay, B., A. Petri, K. Tilly and A. Pataricza, *High level test pattern generation for VHDL circuits*, in: *IEEE European Test Workshop*, 1996, pp. 202-206.

[22] Stürmer, I., M. Conrad, *Test suite design for code generation tools*, in: *ASE'03*, (2003), pp. 286.

[23] Varró, D., "Automated Model Transformations for the Analysis of IT Systems," Phd thesis, Budapest University of Technology and Economics (2003).

[24] Varró, D. and A. Pataricza, *Generic and meta-transformations for model transformation engineering*, in: *Proc. UML 2004: 7th International Conference on the Unified Modeling Language*, LNCS **3273** (2004), pp. 290-304.