Automated Program Generation *for* and *by* Model Transformation Systems[⋆]

Dániel Varró

Budapest University of Technology and Economics
Dept. of Measurement and Information Systems
H-1521 Budapest Magyar tudósok körútja 2.
varro@mit.bme.hu

Abstract. Model transformation systems are graph transformation systems that perform translations between languages defined by a corresponding metamodel as the type graph. The current paper proposes a reflective method for the automatic generation of the implementation for such transformation systems derived from a high–level specification consisting of a set of graph transformation rules and a control flow graph. The program generator takes a UML profile tailored to model transformation systems as the input, and produces the output Prolog program by successive model transformation steps. In this respect, only the core of the program generator is implemented by hand, and afterwards, this core provides automation for additional features of the VIATRA model transformation system.
Keywords: model transformation, graph transformation systems, automated program generation

1 Introduction

1.1 Model transformations in system design

Although the Unified Modeling Language (UML) has become the de facto standard visual modeling language of object–oriented design, both academic investigations and engineering experiments have revealed several shortcomings regarding, especially, its imprecise semantics and the lack of flexibility in domain specific applications [9]. Recently, the UML 2.0 Request For Proposal issued by the OMG has addressed to re-architecture the single and monolith language into a family of languages with individually defined semantics based on a kernel metamodeling language.

However, as the formal semantics of different views of the system (i.e., separate diagrams like class diagrams, statecharts, sequence diagrams, etc.) might be defined in different semantic notations (e.g., by Petri nets, SOS rules, graph transformation systems etc.), the integration of such local views into a consistent global view of the system requires a precise specification of transformations *within* and *between* UML models.

In practice, transformations are necessitated for several purposes: (i) *model transformations within a language* should control the correctness of consecutive refinement

[⋆] This work was supported by the Hungarian National Scientific Foundation Grant (OTKA 030804)

steps, (ii) *model transformations between different languages* should provide precise means to project the semantic content of a diagram into another one, which is indispensable for a consistent global view of the system under design, and (iii) a visual UML model (i.e., a sentence of a language in the UML family) should be transformed into its semantic domain, which process is called *model interpretation*.

As the abstract syntax of UML models is defined visually by a corresponding metamodel, a straightforward representation of such models can rely on the use of directed, typed, and attributed graphs for the underlying semantic domain. Therefore, the use of graph transformation for capturing the semantics of model transformations is a natural choice which also fits well to engineering practices as a consequence of its visual expressiveness [20].

However, even if the formal specification of a transformation is precise (and formally verified), *its implementation is still highly error prone due to a huge abstraction gap between visual UML models and formal mathematical descriptions*. For this reason, the automated generation of a program that implements the transformation is also a major requirement for model transformation systems.

1.2 VIATRA: Visual Automated Model Transformations

VIATRA (VIsual Automated model TRAnsformations) is a prototype tool being developed at the Budapest University of Technology and Economics, that provides a general means to specify and implement various transformations between models defined by their corresponding metamodels by the paradigm of graph transformation.

VIATRA interface The user interface of VIATRA is a set of XMI [1] files, which includes descriptions of metamodels, models and their transformations in all phases of model transformations.

1. **Metamodels:** Both UML and mathematical languages are defined by a corresponding metamodel (e.g., a metamodel for Petri nets, or state transition systems, etc.), which is constructed in a commercial UML CASE tool having XMI export facilities.
2. **Transformations:** The elementary model transformation steps are defined by specially structured graph transformation rules while the entire computation is specified by a control flow graph. In practice, both graph transformation rules and control flow graphs are described in a high-level UML notation using a transformation specific profile based on UML Class diagrams.
3. **Models:** For obtaining a flexible and general interface, the input and output models of VIATRA (thus both UML and mathematical models) are XMI files conforming to their metamodel.

VIATRA features The current version of VIATRA supports the following (main) features.

[1] XMI (XML Metadata Interchange) is the standard XML-based description format for systems based on MOF metamodeling.

1. **Automated DTD generation:** A corresponding DTD (Document Type Definition) can be generated automatically from metamodels.
2. **Automated program generation:** When VIATRA is supplied with the input files of the transformation and the metamodels, it automatically generates the declaration of the transformation (in Prolog) including the implementation of the control flow graph and graph transformation rules.
3. **Automated transformation:** This transformation program is then executed on an arbitrary source model, and the target model is generated. Although VIATRA *currently* uses Prolog as the underlying transformation engine (i.e., the concrete transformation of models is performed in Prolog), it is hidden from the user and it still provides a more efficient solution than by using an XSLT [2] engine.

The current paper provides *an overview of the underlying program generation method* of VIATRA (a more detailed description can be found in [19]). The main characteristics of our solution are (i) the use of *intermediate transformation steps* to avoid re-implementing code generation for executable programs and input descriptions for model checking tools, (ii) the *reflective specification method* embedded in the code generation process (the implementation of model transformation systems is specified by model transformation systems), (iii) and a (future) *bootstrapping* step to improve the quality of the code generation (when a previous version of the program generator is used for generating the next version of the program generator, in analogy with the well–known bootstrapping techniques of compiler design).

2 Automated Program Generation for Model Transformation Systems

2.1 Theoretical background of model transformations

We provide a brief overview of the main concepts of model transformations, namely, model graphs, model transformation rules and model transition systems.

Definition 1. *A **model graph** G is a directed, typed and attributed graph. The type graph of a model graph is called the **metamodel**, which is related to a model graph by a typing homomorphism. The **metametamodel** is the common language (in other words, the top-most type graph) for describing metamodels, which is reflective (i.e., its type graph is itself).*

In model transformation systems, the source and target models are related by a reference graph, which is, in fact, an ordinary model graph. In general, a reference graph is a common abstraction of the source and target models relating the corresponding nodes and edges to each other.

Definition 2. *A **model transformation rule** $r = (L, N, R, M)$ is a special graph transformation rule, where all graphs L, N and R are model graphs applied in the specified mode M.*

[2] XSTL (eXtensible Stylesheet Language Transformation) is an XML technology used for describing (mainly syntactic) transformations between XML files.

The **application** of r to a **host graph** G replaces an occurrence of L (left-hand side) in G by an image of R (right-hand side) yielding the derived graph H. This is performed by

1. *finding an occurrence* of L in G, which is either an isomorphic or non–isomorphic image according to M
2. *checking the negative application condition* N, which prohibit the presence of certain nodes and edges
3. *removing* those nodes and edges of the graph G that are present in L but not in R yielding the **context graph** D (all dangling edges are removed at this point)
4. *adding* those nodes and edges of the graph G that are present in R but not in L attaining the **derived** graph H.

Currently, the behavior of VIATRA follows conceptually the single pushout approach [6] (i.e., removing dangling edges and allowing non–isomorphic images in graph pattern matching), however, the concrete graph manipulations are defined and implemented by logics based rewriting showing closer correspondence to the techniques of [16].

The entire model transformation process is defined by an initial graph manipulated by a set of model transformation rules (micro steps) executed in a specific mode in accordance with the semantics (macro steps) of a hierarchical control flow graph.

Definition 3. *A **model transition system** $MTS = (Init, R, CFG)$ with respect to (one or more) type graph TG is a triple, where $Init$ defines the **initial graph**, R is a set of **model transformation rules** (both compatible with TG), and CFG is a set of a **control flow graphs** defined as follows.*

– *There are six types of nodes of the CFG, each associated with a rule $r \in R$: **Start**, **End**, **Try**, **Forall**, **Loop** and **Call**.*
– *There are two types of edges: **succeed** and **fail**.*

The control flow graph is evaluated by a virtual machine which traverses the graph according to the edges and applies the rules associated to each node.

1. The execution starts in the **Start** and finishes in the **End** node. Neither types of nodes have rules associated to them.
2. When a **Try** node is reached, its corresponding rule is tried to be executed. If the rule was applied successfully then the next node is determined by the **succeed** edge, while in case the execution failed, the **fail** edge is followed.
3. At a **Loop** node, the associated rule is applied as long as possible (which may cause non-termination in the macro step).
4. When a **Forall** node is reached, the related rule is executed parallelly for all distinct (possible none) occurrences in the current host graph.
5. Finally, at a **Call** node (which has an associated CFG and not a rule) the state of the CFG virtual machine is saved and the execution of the associated CFG is started (in analogy with function calls in programming languages). When the sub CFG machine is terminated, the saved state is restored, and the execution is continued in accordance with the outgoing edge (**succeed** or **fail**).

2.2 A case study: Semantics of Message Sequence Charts

To provide a more deeper insight into the expressiveness of model transformation systems, below we consider a semantic interpretation of Message Sequence Charts (abbreviated as MSCs in the sequel). The semantics of MSCs that define a partial order on events (following the semi-formal description in [14]) is captured by a corresponding model transformation system.

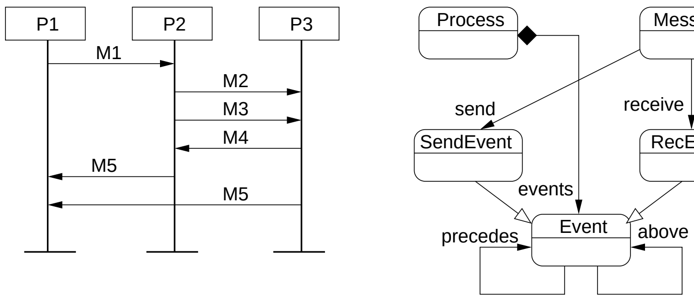


Fig. 1. Message Sequence Charts: visual syntax and metamodel

Message Sequence Charts (a sample MSC model is depicted in Fig. 1 together with a metamodel) are constructed from **Processes** (depicted as *rectangles*) that communicate by sending and receiving **Messages** (shown as *arrows*). The fact that a message is sent or received is represented by a corresponding **Event** (**SendEvent** or **RecEvent**) on the process line (depicted as *vertical lines*). If a message M_i is (supposed to be) sent before another message M_j then the arrow representing M_i appears above the arrow of M_j.

For the semantic interpretation, we define a partial order on MSC events and formalize it by the model transformation system of Fig. 2 as follows.

- **Causality.** If p is the send event and q is the receive event of the same message then p precedes q.
- **Controlability.** If p appears above q on the same process line, and q is a send event then p precedes q. This order reflects the fact that a send event can wait for other events to occur. On the other hand, we typically have less control on the order in which receive events occur.
- **FIFO order.** For any send events p' and q' on the same process line where p' is above q', p precedes q for the corresponding receive events p and q.
- **Transitivity.** The *precedes* relation is transitive, i.e., if p precedes q and q precedes r then p precedes r.

Example 1. We select rule *transClosureR* for deeper investigations. According to the control flow graph, this rule is applied in *loop* mode in the very end of the semantic transformation process, and generates the transitive closure of the *precedes* relation. If

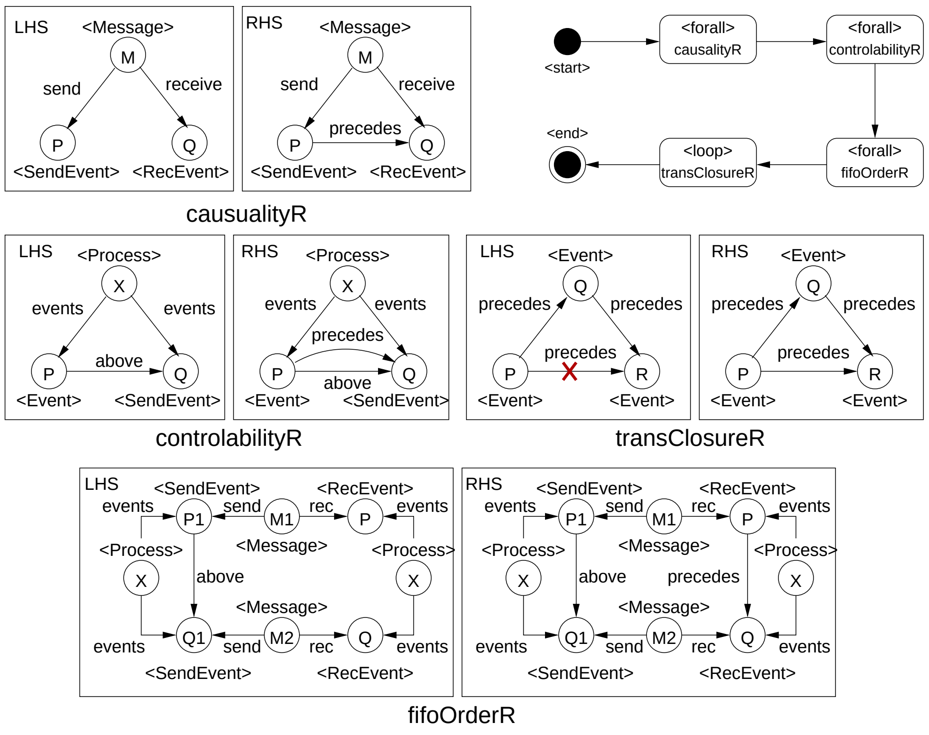


Fig. 2. Defining semantics for Message Sequence Charts

event P is already connected to Q by a `precedes` edge and Q is already connected to R (according to the left-hand side of the rule), and if P is not connected yet to R by a `pre-cedes` edge (negative condition), then a new `precedes` edge is added leading from P to R (as defined by the right-hand side). *Loop* mode prescribes that rule *transClosureR* should be applied as long as possible.

The rules of the corresponding model transformation system are rather straightforward with respect to the metamodel and the informal semantics of MSCs, which clearly demonstrates the descriptive power of our approach. Even though the control flow graph of our case study is relatively simple, the use of CFGs for restricting the valid computations still helps reduce the complexity of rule preconditions (thus resulting in more efficient pattern matching). In our case, we can safely omit the negative application conditions from rules that can be executed in *forall* mode (such as *causalityR*, *controlabilityR* and *fifoOrderR*) without duplicating *precedes* edges since those rules are applied parallelly in a single (deterministic) transformation step.

2.3 Program generation for graph transformation rules

The automated program generation of VIATRA allows the transformation designers to focus on the *design* of a model transformation rather than the *implementation*. Previous experiments (in project HIDE [3]) demonstrated that the quality of an automatically generated executable transformation program is much higher than a manually written target program. Moreover, once the automated program generator is completed, the time and workload related to the design of a single transformation is drastically decreased.

In VIATRA, the program generation process of model transformation rules is divided into several intermediate steps.

1. Model transition systems are specified in a UML CASE tool (Rational Rose was used for our experiments), and exported in the standard XMI format.
2. This UML model is transformed into a GraTra model conforming to a metamodel of graph transformation systems.
3. In VIATRA, model graphs are represented as predicates in a fact database. For this reason, the previous GraTra model is projected into a Logic model containing a sequence of terms for each rule.
4. The bridge between visual (graph based) and the textual language of Prolog is provided by the parse tree of the Prolog code. In this sense, the Logic model is transformed into a corresponding Prolog parse tree, and the target Prolog code is printed out by traversing this tree in an in-order way.

The importance of these intermediate steps is threefold.

– At first, there is a huge abstraction gap between a visual UML-based specification of the transformation and even such a high-level programming language as Prolog. Thus splitting the transformation into several subtransformations decreases the complexity of the individual steps, which eases not only the implementation but also the verification of the automated program generation.
– Secondly, the use of intermediate models increases reusability. For instance, when generating the input language of a model checker for the verification of a model transformation (an ongoing research activity), only the final step needs to be altered.
– Finally, the intermediate GraTra model provides the right basis for generating the upcoming standard XML description of graph transformation systems [18] by a simple transformation from the GraTra XMI format to GXL/GTXL.

In fact, each intermediate transformation step is specified (and implemented) as model transformation, which means that *the entire code generation process is captured by graph transformation*. In this sense, the implementation of model transformation rules is specified by model transformation rules. This approach is similar to the bootstrapping process of compiler design, where, for instance, a C compiler is written in C and compiled by an existing C compiler, and recompiled by itself afterwards to provide a more efficient and reliable target code. In VIATRA, the current version of the program generator is implemented manually, while future versions (with additional features, and more efficient / reliable target code) are generated by using the existing version of the program generator.

```
causalityR:-
  node(msc:message(M)),
  edge(msc:send(E1,M,P)),
  node(msc:sendEvent(P)),
  edge(msc:receive(E2,M,Q)),
  node(msc:recEvent(Q)),
  add(edge(msc:precedes(E3,P,Q))).
```

```
controlabilityR:-
  node(msc:process(X)),
  edge(msc:events(E1,X,P)),
  node(msc:event(P)),
  edge(msc:events(E2,X,Q)),
  node(msc:recEvent(Q)),
  edge(msc:above(E3,P,Q)),
  add(edge(msc:precedes(E4,P,Q))).
```

```
fifoOrderR:-
  node(msc:process(X)),
  edge(msc:events(E1,X,P1)),
  node(msc:sendEvent(P1)),
  edge(msc:events(E2,X,Q1)),
  node(msc:sendEvent(Q1)),
  edge(msc:above(E3,P1,Q1)),
  edge(msc:send(E4,M1,P1)),
  node(msc:message(M1)),
  edge(msc:receive(E5,M1,P)),
  node(msc:recEvent(P)),
  edge(msc:send(E6,M2,Q1)),
  node(msc:message(M2)),
  edge(msc:receive(E7,M,Q)),
  node(msc:recEvent(Q)),
  add(edge(msc:precedes(E8,P,Q))).
```

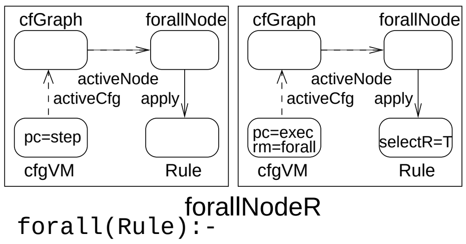
```
transClosureR:-
  node(msc:event(P)),
  edge(msc:precedes(E1,P,Q)),
  node(msc:event(Q)),
  edge(msc:precedes(E2,Q,R)),
  node(msc:state(R)),
  ( edge(msc:precedes(E3,P,R) -
>
    fail ; true),
  add(edge(msc:precedes(E4,P,R))).


msc:-
  forall(causabilityR),
  forall(controlabilitR),
  forall(fifoOrderR),
  loop(transClosureR).
```

Fig. 3. Program generation for transformation rules

Example 2. We continue our case study with a brief insight into the structure of the generated Prolog code (see Fig. 3). Again, we discuss only the behavior of rule *transClosureR* in details. The Prolog code of the rules implements the graph pattern matching by consecutive unifications during which the variables `P, Q, R, E1, E2` are instantiated. The outermost terms (`node` and `edge`) are responsible, for instance, for the hierarchical matching of patterns (i.e., a node of type *SendEvent* should also be matched by the *Event* pattern in case of MSCs). The negative part (within parenthesis) causes failure for the current matching if and only if `R` is already a substate of `P`, and then steps to the next matching to be examined by backtracking. Finally, after a successful pattern matching, a `precedes` edge is added between `P` and `R`.

The example also demonstrates that the generated program partially contains transformation dependent (translated) Prolog code (i.e., the sequence of terms representing queries on the model graph) and (interpreted) calls to built-in routines from a VIATRA library (like `node()`, `edge()`, and routines implementing different modes of rule application discussed in the upcoming section).



```
 forall(Rule):-
call(Rule),  fail.
 forall(Rule).
```

Fig. 4. Executing a step of the virtual machine

2.4 Implementing the virtual machine

The implementation of the virtual machine that executes the control flow graph (CFG) of a model transition system builds upon the reflective property of program generation in VIATRA as the operational semantics of this virtual machine is also defined by model transition systems. In this respect, the program corresponding to a rule is generated automatically, and the implementation of the “meta” CFG (i.e., the CFG that defines the behavior of the virtual machine) is very simple. The entire semantics of the virtual machine consists of 11 rules, from which the handling of *forall* nodes is depicted in Fig. 4.

Example 3. The semantics of rule *forallNodeR* is as follows. When the virtual machine is to execute a step (`pc = step`), then the active CFG node in the active graph should be found and the `selectR` attribute of the associated graph transformation rule is set to true in order to select the rule for execution. Moreover, the program counter `pc` is set to `exec` indicating that now the virtual machine should execute the rule before making the next step and attribute `rm` (which represents the execution mode of a rule) is set to `forall`.

In the rule execution phase, the piece of code in the right region of Fig. 4 is called, which is responsible for executing the rule for all possible matches in the current state. This is obtained by causing an artificial backtracking whenever the rule application is succeeded. Finally, if all possible matches are processed, the application of a rule in *forall* mode is also successfully completed (including the case when there are no possible matches of the LHS).

Finally, the current node of the control flow graph is updated according to the success of rule application, and we proceed with the new current node.

3 Conclusion

In the current paper, we presented an automatic program generation framework *for* the implementation of model transformation systems, where the process of program generation is specified *by* consecutive model transformations. Our approach (implemented in the VIATRA tool) is reflective in the sense that the next versions of the program generator can be derived by the previous version similarly to the bootstrapping techniques of compiler design. In fact, the virtual machine of VIATRA that executes a control flow graph has been implemented by using the automated program generator of graph transformation rules.

VIATRA has already been applied successfully to provide an automated implementation of transformations specified by means of model transition systems.

– The specification (and implementation) of the VIATRA virtual machine required 11 graph transformation rules and a simple automata.
– In [11], a transformation from UML Statecharts to Extended Hierarchical Automata (EHA) has been carried out that provides formal operational semantics for statecharts. In [19], we formalized the entire transformation by model transitions systems with over 40 rules.
– The original paper defined the EHA semantics as a Kripke automata. We also provided visual semantics for EHA by model transition systems having approximately 20 rules (with simple LHS graphs).
– The completeness of UML statechart specifications in a dependable environment has been investigated in [13]. Currently, an automated verification program is under implementation using VIATRA.
– A transformation from UML statecharts to Stochastic Reward Nets [4] is also under implementation (having currently 25 rules for a well–separated subproblem) to provide access to Petri Net based analysis techniques.

3.1 VIATRA as a graph transformation tool

As state-of-the-art tools of graph transformation systems (such as GenGEd (with AGG) [1] , DiaGen [10], Progres [17], and FUJABA [12]) have been evolving for more than a decade, VIATRA is naturally not the only tool that is capable of performing model transformations. However, we believe that VIATRA is tailored to the special needs of model transformations between UML and semantic domains to such an extent that makes our tool more flexible and powerful in this specific application domain of graph transformation systems than sophisticated general-purpose graph transformation tools. Therefore, it is rather (i) the underlying model transformation methodology, (ii) its openness and compliance with leading industrial standards, and (iii) its (ongoing) integration with model checking tools that makes VIATRA unique rather than the core graph transformation engine itself.

– **Openness, Compliance with standards:** VIATRA is an open environment built around XMI technology, which is the de facto standard in UML-based modeling environments. XMI DTDs for non-UML models are generated automatically from metamodels, which is a more flexible solution in domain specific applications than tools forcing to use a fixed set of XML elements. Moreover, similarly to the story diagram-based rule descriptions [8] of FUJABA, VIATRA uses a UML profile based on class diagrams as the formal specification language of model transformations.

– **Model transformations:** Transformations of UML models necessitate to manipulate complex data structures with *a large number of rules* (see our benchmark transformations or [7], where a Java implementation of UML models is also specified by graph transformation rules), which makes *graph transformation tools without control condition impractical* for such applications due to the increased level of nondeterminism. In addition, *a typical model transformation rule is executed parallelly* (in *forall* mode) for each independent matching. However, *forall* type rule applications are not directly supported by general purpose graph transformation tools. Moreover, as in most cases more than a single language is involved in transformations, the *simultaneous handling of multiple metamodels is not flexible* in these tools.
– **Verifying model transformation systems:** An ongoing research activity *integrates model transformation systems with existing model checking tools* for formal verification purposes which requires that the Kripke automata of the system is derived from the same (intermediate) semantic representation as the automatically generated target program. As a benchmark application, we generate SAL [2] specifications from UML Statecharts, where statecharts semantics are captured by model transformation systems.

Acknowledgment

The author would like to thank András Pataricza for suggesting many improvements in the early versions of the paper, and the anonymous referees for their valueable comments.

References

1. R. Bardohl and H. Ehrig. Conceptual model of the graphical editor GENGED for the visual definition of visual languages. In H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg (eds.), *Proc. Theory and Application to Graph Transformations (TAGT'98)*, vol. 1764 of *LNCS*, pp. 252–266. Springer, 2000.
2. S. Bensalem, V. Ganesh, Y. Lakhnech, C. Munoz, S. Owre, H. Rueß, J. Rushby, V. Rusu, H. Saïdi, N. Shankar, E. Singerman, and A. Tiwari. An overview of SAL. In C. M. Holloway (ed.), *LFM 2000: Fifth NASA Langley Formal Methods Workshop*, pp. 187–196. 2000.
3. A. Bondavalli, M. Dal Cin, D. Latella, I. Majzik, A. Pataricza, and G. Savoia. Dependability analysis in the early phases of UML based system design. *International Journal of Computer Systems - Science & Engineering*, vol. 16(5):pp. 265–275, 2001.
4. M. Dal Cin, G. Huszerl, and K. Kosmidis. Evaluation of safety-critical systems based on guarded statecharts. In R. Paul and C. Meadows (eds.), *Proc. of the Fourth IEEE International Symposium on High Assurance Systems Engineering*. IEEE, 1999.
5. H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg (eds.). *Handbook on Graph Grammars and Computing by Graph Transformation*, vol. 2: Applications, Languages and Tools. World Scientific, 1999.
6. H. Ehrig, R. Heckel, M. Korff, M. Löwe, L. Ribeiro, A. Wagner, and A. Corradini. *In [15]*, chap. Algebraic Approaches to Graph Transformation — Part II: Single pushout approach and comparison with double pushout approach, pp. 247–312. World Scientific, 1997.

7. G. Engels, R. Hücking, S. Sauer, and A. Wagner. UML collaboration diagrams and their transformation to Java. In R. France and B. Rumpe (eds.), *Proc. UML'99 - The Unified Modeling Language. Beyond the Standard. Second International Conference*, vol. 1723 of *LNCS*, pp. 473–488. Springer, 1999.
8. T. Fischer, J. Niere, L. Torunski, and A. Zündorf. Story diagrams: A new graph transformation language based on UML and Java. In H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg (eds.), *Proc. Theory and Application to Graph Transformations (TAGT'98)*, vol. 1764 of *LNCS*, pp. 296–309. Springer, 2000.
9. C. Kobryn. UML 2001: A standardization odyssey. *Communications of the ACM*, vol. 42(10), 1999.
10. O. Köth and M. Minas. Generating diagram editors providing free-hand editing as well as syntax-directed editing. In *GRATRA 2000, Joint APPLIGRAPH and GETGRATS Workshop on Graph Transformation Systems*, pp. 32–39. 2000.
11. D. Latella, I. Majzik, and M. Massink. Automatic verification of UML statechart diagrams using the SPIN model-checker. *Formal Aspects of Computing*, vol. 11(6):pp. 637–664, 1999.
12. U. Nickel, J. Niere, and A. Zündorf. Tool demonstration: The FUJABA environment. In *The 22nd International Conference on Software Engineering (ICSE)*. ACM Press, Limerick, Ireland, 2000.
13. Z. Pap, I. Majzik, and A. Pataricza. Checking general safety criteria on UML statecharts. In U. Voges (ed.), *Proc. SAFECOMP 2001, Computer Safety, Reliability and Security, 20th International Conference*, vol. 2187 of *LNCS*, pp. 46–55. Springer, 2001.
14. D. Peled. *Software Reliability Methods*, chap. Message Sequence Charts, pp. 300–302. Springer, 2001.
15. G. Rozenberg (ed.). *Handbook of Graph Grammars and Computing by Graph Transformations*, vol. 1: Foundations. World Scientific, 1997.
16. A. Schürr. *In [15]*, chap. Programmed Graph Replacement Systems, pp. 479–546. World Scientific, 1997.
17. A. Schürr, A. J. Winter, and A. Zündorf. *In [5]*, chap. The PROGRES Approach: Language and Environment, pp. 487–550. World Scientific, 1999.
18. G. Taentzer. Towards common exchange formats for graphs and graph transformation systems. In J. Padberg (ed.), *UNIGRA 2001: Uniform Approaches to Graphical Process Specification Techniques*, vol. 44 of *ENTCS*. 2001.
19. D. Varró and A. Pataricza. Mathematical model transformations for system verification. Tech. rep., Budapest University of Technology and Economics, 2001. `http://www.inf.mit.bme.hu/~varro/publication/publication.htm`.
20. D. Varró, G. Varró, and A. Pataricza. Designing the automatic transformation of visual languages. *Science of Computer Programming*. In print.