UML Specification of Mathematical Model Transformation [⋆]

Dániel Varró, Péter Domokos, and András Pataricza

Budapest University of Technology and Economics
Department of Measurement and Information Systems
1111 Budapest, Hungary,
Pázmány P. st. 1/D. IT building B.420.
Contact person: András Pataricza
Phone: +36-1-463-3595
Fax:+36-1-463-4112
pataric@mit.bme.hu

Abstract The design of complex, dependable systems requires a precise formal verification of design decisions during the system modelling phase. For that reason, the mathematical models of various formal verification tools are planned to be automatically derived from the system model usually described by UML–diagrams. In the current paper, a general framework for an automated model transformation system is introduced providing a uniform formal description method of such transformations by applying the powerful computational paradigm of graph transformation. Model transformation rules are constructed in a modular way using a visual UML notation as representation in order to provide a closer correspondence with industrial techniques.

Keywords: system verification, model transformation, graph transformation, transformation unit, UML.

1 Introduction

1.1 Formal Methods in System Verification

Due to the immense complexity of dependable, real–time systems, an early conceptual and architectural validation based on precise formal verification techniques is essential aiming to identify critical bottlenecks to which the system is highly sensitive for obtaining a guaranteed design quality. In order to avoid costly re–design cycles such a system verification must preceed the implementation phase.

The increasing need for effective design has necessitated the development of standardized design languages and methods allowing system developers to work on a common platform of design tools.

[⋆] This work was supported by the Hungarian National Scientific Foundation Grant OTKA T030804

The *Unified Modelling Language (UML)* is a visual specification language (providing a collection of best engineering practises of the several decades) that has been adopted as the standard object–oriented modelling language for a large scale of IT systems ranging from pure software systems to embedded systems (systems reactively interacting with their environment) recently.

Formal methods provide a rigorous and effective way to model, design and analyze computer systems on a strict mathematical platform. For many years, they have been a topic of research with valuable academic results. However, their industrial utilization is still limited to specialized development sites, despite their vital necessity originating in the complexity of IT products and increasing requirements for dependability and Quality of Service (QoS).

The use of formal verification tools (like model checkers SPIN [7] or PVS [14]) in IT system design is hindered by a gap between practice–oriented CASE tools and sophisticated mathematical tools.

– On the one hand, system engineers usually show no proper mathematical skills required for applying formal verification techniques in the software design process.
– On the other hand, even if a formal analysis is carried out, the consistency of the manually created mathematical model and the original system is not assured, moreover, the interpretation of analysis results, thus, the re–projection of the mathematical analysis results to the designated system is problematical.

Moreover, from the engineering point of view, a dependability analysis is a composite one necessitating the assessment of multiple mathematical properties by using different verification tools.

1.2 Mathematical Model Transformation

The step generating the description of the target design on the input language of mathematical tools from the UML model of the system is called *mathematical model transformation*.

The inverse direction of model transformation (referred as *back–annotation*) is of immense practical importance as well when some problems (e.g. a deadlock) are detected during the mathematical analysis. After an automated back–annotation these problems can be visualized in the the same UML system model allowing the designer to fix conceptual bugs within his well–known UML environment.

Several semi-formal transformation algorithms have already been designed and implemented for different purposes.

– formal verification of functional properties [10]
– quantitative analysis of dependability attributes [3, 4]).

Unfortunately, this conventional way of model transformation lacked a uniform and precise description of transformation algorithms resulting in hand–written and rather ad hoc implementations (inconvenient for implementing complex transformations).

Moreover, any formal proof of correctness and completeness aiming to verify these transformation scripts is almost impossible, thus their uncertain quality remains a quality bottleneck of the entire transformation based verification approach.

The aim of our ongoing research is to provide a general framework of a *visual, automated model transformation system* with the facilities of an *automatically generated transformation algorithm of a proven quality* (derived from a visual description of the transformation).

Such a model transformation system must fulfil at least the following user requirements.

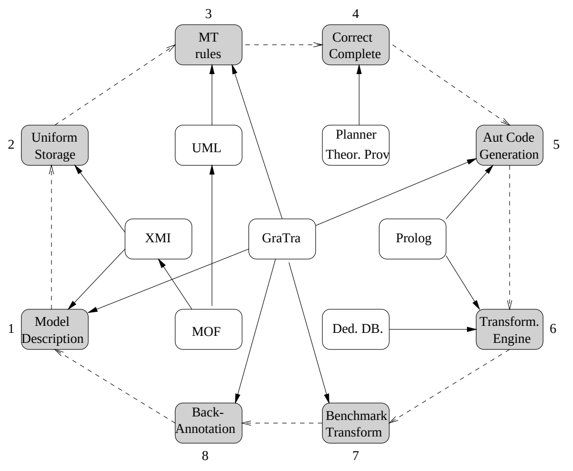
– A large number of model transformations are planned to be designed to perform dependability analysis in various application domains ranging from early evaluation methods based on Petri nets to model checking techniques using temporal logic as underlying mathematical model.
– "Mathematical" model transformations are not only designed by mathematicians but system designers as well. Thus, these transformations must be defined by a visual, easy to understood formalism.
– The specification of a model transformation should be given in mathematically precise, unambiguous form.

The current paper extends our basic concepts introduced in [19, 20] by a UML based specification method of model transformation systems, providing a general introduction to our designated system at first in Sec. 2. Section 3. summarizes the basic theoretical concepts of model transformation. In Sec. 4, a visual specification of model transformation (serving as a general description language for various type of model transformations) is given using UML as a visual modelling language. Finally, Section 5 concludes our paper.

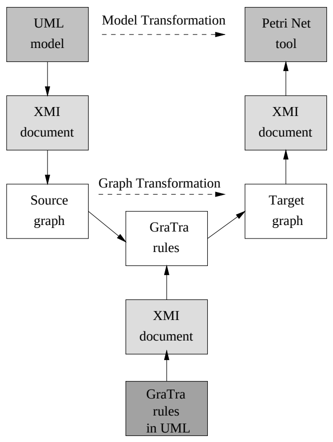
2 A Visual Automated Model Transformation System

The process of model transformation is characterized by a model analysis roundtrip illustrated in Fig. 1. Typically, a system designer and a transformation designer participates in such a roundtrip with the following roles.

– A transformation designer specifies model transformations from UML to various mathematical models (like e.g. Petri nets, temporal logic). From his specification, a transformation algorithm is generated at compile time.
– A system analyst designs complex systems using UML as modelling language. During the software life cycles, he needs several verification steps to be performed running the previously generated model transformation programs.



(a) Architectural concepts



(b) Technological summary

Figure 1. An overview of model transformation

Model description The overall aim of a transformation designer is to generate the input language of a specific tool handling some mathematical models from a generally accepted system models. (Let us take Petri nets as the target model during this following introductory description in order to avoid speaking too generally.)

A well–defined transformation necessitates a uniform and precise description of source and target models, therefore, a formal underlying formalism is needed. On the other hand, these models represent some aspects of IT systems with industrial relevance, thus, model descriptions should follow the main standards of the industry.

For this reason, the **Unified Modeling Language (UML)** is used as the front–end of model transformations, additionally, the user–end specification language of model transformation rules is UML as well. UML conceptually follows the four–layer **Meta Object Facility (MOF)** meta-modelling architecture [11], which allows the definition of meta–objects for similarly behaving instances.

Uniform storage of models The front–end and back–end of a transformation (UML as the source model and a formal verification tool as the target model) is defined by a uniform, standardized description language of system modelling, that is, **XMI (XML Metadata Interchange)** [13], which is a special dialect of XML, the approving novel standard of the Web.

Due to the fact that UML models are exported in an XMI format (the export process is supported by most UML CASE tools) an open, tool–independent architecture is obtained. In order to avoid to write separate transformation algorithms for different tools of Petri Nets, a model transformation algorithm generates an XMI based description of the target model, from which the input languages of different tools can be generated by simple syntactical translators (based on general XML parsers).

Designing model transformation rules The formal description of these transformations are supported by **graph transformation**, which combines the advantages of graphs and rules into an powerful computational paradigm. Model transformation rules are defined in a special form of graph grammar rules.

A **graph transformation rule** is a special pair of pattern graphs where the instance defined by the left hand side is substituted with the instance defined by the right hand side when applying such a rule (similarly to the well–known grammar rules of Chomsky in computational linguistics).

The model transformation rules are aimed to be specified by using the visual notation of UML. However, for obtaining a tool–independent transformation specification, the transformation rules will also be exported in an XMI (or XML) based format, conforming to the approving standard of graph transformation systems [6].

Correctness and completeness of transformations After having specified a set of transformation rules, the **correctness** and **completeness** of the transformation has to be verified aiming to prove that the resulted Petri net model is semantically equivalent to the source UML model. These questions will be verified by **planner algorithms** and **theorem proving** techniques of artificial intelligence operating on user defined basic equivalent source and target structures.

Automated code generation Even if the description of the transformation is theoretically correct and complete, additionally, the source and target models are also mathematically precise, the implementation of these transformations has a high risk in the overall quality of a transformation system. As a possible solution, **automatic transformation code generation** is aimed based on XMI documents and visual transformation rules.

The transformation engine As the transformation engine is implemented in Prolog, the uniform, XMI based models and rules are translated into a Prolog notation supplying the input and the program to be executed, respectively, for the transformation engine. An attributed and labelled graph representation is generated for the source model, and a similar graph is to be obtained as a result of the transformation.

The transformation process specified by visual model transformation units is executed in the form of a Prolog program manipulating the previous graph based models by the powerful backtracking and unification method of Prolog. However, the algorithmic skeletons extracted from the specification are rather simple, thus allowing to substitute Prolog with a more powerful but lower abstraction level language (like C or Java) after a successful prototyping phase.

Benchmark transformations Our model transformation system is supposed to be used in real industrial applications. A benchmark transformation (transforming the static aspects of UML models into timed Petri Nets for dependability analysis in an early phase of system design; see [18] for further details) has already been designed and implemented. Further benchmarks of industrial relevance (dependability evaluation with a special emphasis on the safety of an artificial kidney controller; and a railway interlock system) are soon to come.

Back–annotation of analysis results As the results of the mathematical are automatically back–annotated to the UML based system model, the system analyst are reported from conceptual bugs in their well–known UML notation. After certain modifications and corrections on the system model are performed, the system verification process might step into a consecutive phase (using for instance temporal logic instead of Petri nets as the target model).

3 Theoretical Foundations of Model Transformation

In this section, basic concepts of graph transformation systems (such as graphs, graph transformation rules, transformation units, etc.) are adapted to the special needs of model transformation in order to provide its precise mathematical background. For this purpose, we will basicly follow the directions of [1]. Further details on the theoretical foundations of model transformation can be found in [19].

3.1 Basic Definitions

Definition 1. *A* model graph *G is a directed, typed and attributed graph with the following constraints.*

- There are two types of nodes:
  - `xmi_element` (representing an XMI element) with two associated attributes: `ID` as a unique identifier, and `Type` for storing the XMI name of the element.
  - `value` (for basic data values) with three attributes: `ID` as a unique identifier, `Type` for representing the XMI name of the data element, and `Value` for storing its value.

  However, in order to provide a uniform description of graphs, these two types of nodes are represented by a single type `g_node`, with three attributes (`ID`, `Type`, and `Value`) with a constraint requiring the `ID` and `Value` to be equal in case of XMI elements.
- There is a single type of edge, `g_edge`, with two attributes attached (without considering `FromNode` and `ToNode` used for referring to graph nodes), namely, `Type` (which describes the name of an XMI attribute or reference) and `LinkType` (which provides additional information for the generation of an XMI document).

Definition 2. *A* graph transformation rule $r = (L, R, Emb, App)$ *contains a left–hand side (LHS) graph L, a right–hand side (RHS) graph R, some embedding mechanisms Emb and application conditions App.*

The application of r to a graph G replaces an occurrence of the LHS L in G by the RHS R. This is performed by

1. finding an occurrence of L in G,
2. removing a part of the graph G determined by the occurrence of L yielding the *context* graph D,
3. gluing R and the context graph D by using the embedding mechanism Emb, and obtaining the *derived* graph H.

Definition 3. *A* model transformation rule *(r_{mt}) is a special graph transformation rule, where both graphs L and R are partitioned into two disjoint parts (*source *and* target*). Graph objects of the disjoint parts are connected only to reference nodes by special reference edges.*

A sample model transformation rule, which generates an intermediate hypergraph (IM) as target model from UML as source model, is depicted in Figure 2. Please note that in order to improve the clarity of the illustrations, different types of graph nodes are depicted in various graphical notation (e.g. resembling to the original notation in case of UML constructs).

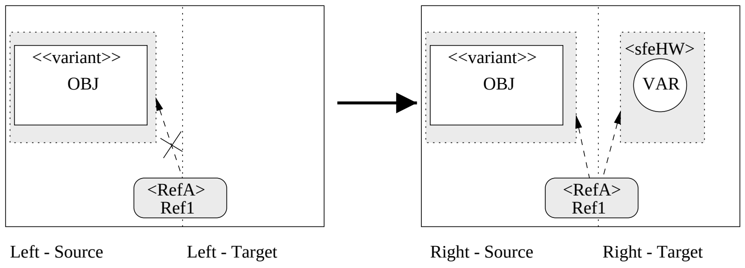


Figure 2. A sample model transformation rule (`variantR`)

The LHS of this rule requires a UML object with the stereotype `variant` to be present on the source side without a reference (negative application condition; a dashed line with a cross), while there are no restrictions for the target side. According to the RHS, a novel IM node of type `sfeHW` and a reference node `Ref1` are inserted and connected to the UML object dashed reference edges.

The LHS of a model transformation rule is frequently a subgraph of the RHS (hence no deletion is needed). In such a case, model transformation rules yield a larger graph as a result of the transformation step.

3.2 Transformation Units

As possible industrial applications of model transformation surely consist of very large and complex models containing hundreds of rules, model transformation rules must be extended by a sophisticated structuring mechanisms that allow to compose them in a modular way. In the graph transformation community, the concepts of *transformation units* were introduced for the very same purpose (e.g. [1,8]).

Definition 4. *A* transformation unit $tu = (I, U, R, C, T)$ *is a system where I and T are graph class expressions (describing initial and terminal graphs), R is a finite set of rules and C is a control condition, and U is the set of imported transformation units (which is empty, initially).*

Definition 5. *A* model transformation unit *is a transformation unit where*

– the **graph model** is the one described in Section 3.1
– the R set of **rules** are well–formed *model transformation rules*,
– the class of **control conditions** (containing control flow information) are composed of *extended regular expressions* (discussed in details e.g. in [9]. Each c_i is either a transformation unit or rule identifier or a previously defined control condition.
  - $c_1; c_2$ stands for the control flow in which c_2 is applied right after c_1 (**concatenation**)
  - $c_1 | c_2$ represents such a control flow where c_1 and c_2 can be applied parallelly (**fork**).
  - **if** a **then** c_1 **else** c_2 serves as a **branch** of the control flow (depending on the evaluation of a)
  - $c!$ applies c **as long as possible**

Initial and terminal class expressions serve as preconditions and postconditions on graphs transformed by the unit. Model transformation rules are nested into transformation units, which units themselves can be imported by further transformation units (circular import is usually forbidden). In this sense, the entire transformation is defined in a hierarchical way; similarly to the process of IT system design.

Several ways of non–determinism are embedded in the application of graph transformation rules, for instance choosing an appropriate rule to be applied or finding an occurrence of the LHS of the rule in the graph. Although non–determinism is often useful in the phase of a mathematical analysis, it has to be eliminated in practical applications before the implementation phase. Control conditions provide a natural mechanism to restrict the control flow of model transformation.

Figure 3 shows a sample model transformation unit (`variantTU`), which derives the graph $\mathcal{G}'$ from input graph $\mathcal{G}$. The transformation unit states that

– the initial and terminal graph must be a well–formed model graph,

– `variantTU` has a rule called `variantR`,
– two further units (not discussed here in details) are imported, namely, `ftsTU` and `linkTU`
– the control condition prescribes that
  1. ftsTU is executed first;
  2. the control flow forks afterwards;
     (a) in one thread one should apply `variantR` as long as possible followed by the transformation described in `linkTU`;
     (b) in the other thread, if condition c_1 evaluates to true then `rule_A` is applied otherwise `rule_B` is executed.

uml2imTU$(\mathcal{G}, \mathcal{G}')$*:*
  initial: model_graph$(\mathcal{G})$
  terminal: model_graph$(\mathcal{G}')$
  rules: variantR (Figure 2.), rule_A, rule_B
  uses: ftsTU,linkTU
  control: ftsTU; (((variantR!); linkTU) | (**if** c **then** rule_A **else** rule_B))

Figure 3. Model transformation unit "variantTU"

When using graph transformation rules and transformation units for the purpose of mathematical model transformations, certain properties are needed to be assessed.

– **Locality.** The rule–based character of graph transformation ensures a certain degree of locality of action as it manipulates mainly a small piece of the model.
– **Complexity.** One central problem of graph transformation is the efficient matching of the LHS of a rule to a subgraph of the current working graph. However, graph isomorphism (isomorphism between the LHS of the rule and a subgraph of the instance graph) is an NP–hard problem, specially typed graphs and restricted LHSs may reduce complexity to polynomial average for pattern matching algorithms. Similar problems can also occur when testing the application conditions.
– **Termination.** A graph transformation system is called *terminating*, if infinite derivations are impossible. In our case, termination is ensured if all transformation rules increases the number of source elements that has a related target element.

4 Model Transformation Design in UML

In this section, our aim is to provide a framework for a general specification method for model transformation systems supporting the visual construction of transformation units, rules and control flow structures.

Although several sophisticated visual tools and environments exist in the graph transformation community for a similar purpose — supporting the creation of diagram editors (GenGEd, DiaGen [2]) or the manipulation of graphs (AGG [17], Progres [16]) — they do not provide a suitable solution for model transformation systems due to the following disadvantages.

- The construction of such a rich visual language as UML in terms of graphs from scratch is time consuming (please remember, UML is the source language in most model transformations).
- Control flow restrictions are only considered in Progres [15] (and probably in the future system GRACE) while transformation units are not supported.
- In possible industrial applications, transformation designers should learn to handle a completely new environment and visual notation.

In order to avoid these drawbacks, model transformation rules and units are described by the rich visual language of UML. "Overloading" UML (i.e. using purely its visual notation and disregarding from its original software modelling concepts) is not a novel idea. For instance, the MOF metamodelling language uses the class diagrams of UML to represent metamodels instead of static structure of an IT system. As a result, any system designer who is familiar with UML understands MOF metamodels (at least partially).

One major reason for applying the same idea for designing model transformations in a UML notation is the fact that UML is the most common source model to be transformed. Our approach allows to use the original UML constructs in model transformation rules, moreover, non–UML objects can also be represented visually by pictorial stereotypes (supported by several UML CASE tools). The UML model of transformation rules are exported into XMI, and this XMI document is processed later on by special parsers.

Stereotypes are used in the original UML notation to group classes, packages, etc. that behave similarly while in our model transformation system, stereotypes are responsible for providing a graphical notation for all the instances of a specific metamodel class.

4.1 Import Structure

The key concept in the UML representation of the importing mechanism of model transformation units is the embedding mechanism described by packages. A UML package may contain further packages in itself, providing a natural mechanism for denoting tree–like hierarchy of transformation units on package diagrams.

Whenever a package serves as a transformation unit, this fact is indicated by the corresponding stereotype `TU` (as indicated in Fig. 4(a).). As UML support multiple namespaces (i.e. packages and classed with identical names can appear in different packages without a name clash), a general package structure was introduced for transformation units, strictly following its textual description (compare Fig. 3. and Fig. 4(b).).

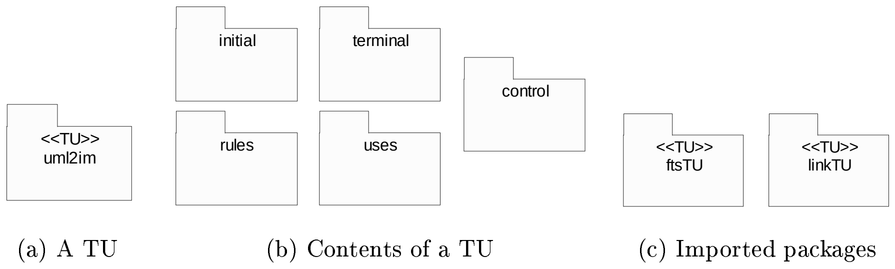


Figure 4. Transformation units (TU) represented in UML

Transformation units can be imported by placing the necessitated units into the `uses` package of the importing unit. Figure 4(c). shows the content of such a `uses` package.

4.2 Rule Representation

A model transformation rule is represented by a UML package with a stereotype `rule` (see Fig. 5(a).) and placed into the appropriate `rules` package of its owner transformation unit (`uml2im` in our example). A rule is divided into a LHS and a RHS, also denoted as packages (as depicted in Fig. 5(b)) placed into the corresponding rule package (`variantR` in our case). Please note that no name clashes should occur due to the namespace concepts of UML.

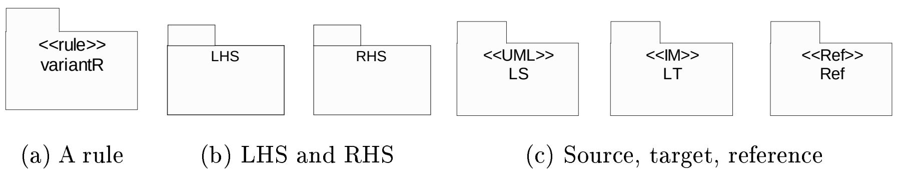
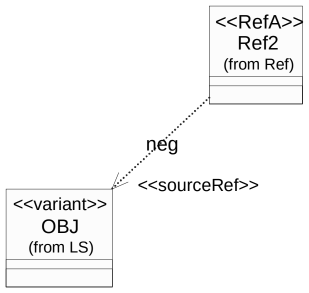
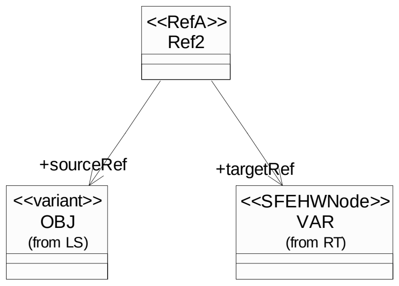


Figure 5. Rule structure in UML

On both sides (LHS and RHS), one must distinguish between the group of source and target objects, additionally, the references (defining elementary equivalence between source and target objects) are also needed to be displayed. Figure 5(c). illustrates how these concepts are denoted when appearing on the LHS. In each case, an appropriate stereotype defines the name of the corresponding metamodel (in our example, `UML` as source, an Intermediate Model (`IM`) as target, and Ref as the referential metamodel).



(a) LHS of rule `variantR`



(b) RHS of rule `variantR`

Figure 6. Contents of model transformation (MT) rules in UML

Source and target objects in model transformation rules are represented visually as classes on class diagrams. When a UML model is transformed, its constructs can also be given using their original notation (i.e. using a state as a state, a class as a class). On the other hand, in case of mathematical models (Petri Nets, computational tree logic, etc.), their constructs are represented by special stereotypes corresponding to a metamodel class. References, which are of immense importance when formally verifying transformations, are denoted as classes with associations to source and target objects.

In Fig. 6. the visual notation of the model transformation rule `variantR` (see Fig. 2) is illustrated. The LHS prescribes (as source pattern) that a UML class cannot be transformed if it has a reference of type `RefA` indicated by a negative application condition `neg` depicted as a dependency.

According to the RHS, a novel target IM object of type `SFEHWNode` and a reference node (`Ref2` of type `RefA`) is added, the former one to the target model, while the latter one to the reference model. Source and target classes are linked to the reference node by associations with special role names (`sourceRef` and `targetRef`).

4.3 Control Flow Description

In transformation units, control flow is usually defined by extended regular expressions. As regular expressions can easily be transformed into finite automaton, UML statecharts seem to provide the most suitable visual notation for control conditions, due to the fact that they are a generalization of finite automaton (supporting e.g. hierarchical behaviour).

As depicted in Fig. 7 (which describes the control condition of the transformation unit `uml2imTU` shown in Fig. 3), each rule and imported transformation unit are referred in the statechart diagram as a simple state. Initial and final states are used to indicate where the execution should be started and finished.

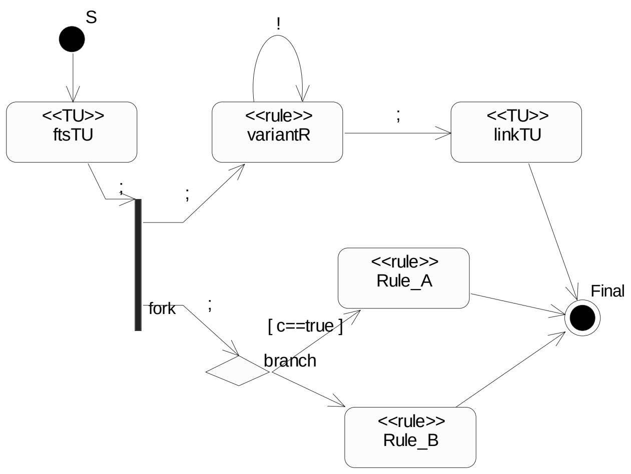


Figure 7. Statechart of the control flow

Transitions triggered by a semicolon (;) denote the *concatenation* control condition, while the self–transition with a trigger "!" identifies the *as long as possible* semantics of a rule or transformation unit.

Synchronization bars (special pseudo states in UML) are used for depicting the *fork* operation in the control flow (the join operation is implemented by the final state), while the *if–then–else* structure (performing a branch in the control flow) is shown by a decision cube (also a pseudo state in UML) and guarded transitions containing a logical condition that has to be fulfilled for firing the given transition.

One can easily verify that the visual description (provided by the statechart in Fig. 7) is equivalent to the control condition of transformation unit `uml2imTU` (shown in Fig. 3).

5 Conclusion

In the current paper, the basic features of a general purpose model transformation system was outlined. Our designated environment will support the automatic generation of the model transformation algorithm of a proven quality by verifying the correctness and completeness of transformations.

As a result, different kind of mathematical models like temporal logic, Petri nets, process algebra, etc. are closely integrated to UML based system models, Thus, a complex verification environment will be provided at hand for system designers without requiring a thorough knowledge of modelling by sophisticated tools.

Moreover, the architecture was designed to fulfil the requirement of tool–independence by using a uniform description of models based on XMI, the novel

XML based model interchange format already used extensively to exchange UML models of different CASE tools.

The method of model transformation follows the paradigm of graph transformation, which has been applied successfully in various (both theoretical and practical) fields.

- Model transformation rules are a special form of graph transformation rules containing reference relation for coupling the source and target objects.
- As complex rule based systems of industrial relevance may contain hundreds of rules, transformation units are adopted that allow the modular construction of a large set of rules.
- The control flow of model transformation is described by extended regular expressions providing a means for all major control flow operations.

A complex model transformation system should support the visual specification of transformations as well. For this purpose, we have chosen to overload the visual notation of UML (forgetting about its original means to model software systems) as existing graph transformation tools are insufficient for a fine–grained integration of MOF metamodels and visual languages.

- The structure of transformation units is denoted by a cluster of UML packages with special stereotypes.
- The objects in model transformation rules are depicted by classes with a stereotype to their metaclass.
- The control conditions of transformation units are represented by UML statecharts, naturally with certain restrictions.

Such a UML based specification environment has the main advantage that transformation designers need not get acquainted with a completely new visual environment but within their well–known UML tool. With this respect, its industrial utilization gets much more easier.

However, there are several issues that need further investigations. Research has already started to cover the following most important ones.

- A parser that is able to point out major conceptual errors in the XMI description of a model transformation.
- Generating the Prolog algorithm implementing a visual specification.
- Attaching semantic constraints to source and target objects by means of the Object Constraint Language (OCL) [12].

References

1. M. Andries et al.: Graph Transformation for Specification and Programming. *Science of Computer Programming*, **34** (1999) 1–54.
2. R. Bardohl, G. Taentzer, M. Minas and A. Schürr.: Application of graph transformation to visual languages. In [5].

3. A. Bondavalli, I. Majzik, and I. Mura. Automatic dependability analyses for supporting design decisions in UML. *HASE'99: the 4th IEEE International Symposium on High Assurance Systems Engineering*, 1999.
4. M. Dal Cin, G. Huszerl, K. Kosmidis: Evaluation of safety–critical system based on guarded statecharts. *In Proc. HASE'99 4th IEEE International Symposium on High Assurance Systems Engineering*, (1999).
5. H. Ehrig, G. Engels, H.J. Kreowski and G¿ Rozenberg, editors. *Handbook on Graph Grammars and Computing by Graph Transformation, Vol. 2: Applications, Languages and Tools.* World Scientific, Singapore, 1999.
6. G. Engels, G. Taentzer (organizers): *APPLIGRAPH Subgroup Meeting on Exchange Formats for Graph Transformation Systems*, Paderborn, September 5–6. 2000.
7. G. Holzmann: The model checker SPIN. *IEEE Transactions on Software Engineering*, **23** 279–295, (1997).
8. H.J. Kreowski, S. Kuske. Graph transformation units and modules. In [5]
9. S. Kuske: More about control conditions for transformation units. *In Hartmut Ehrig, Gregor Engels, Hans-Jrg Kreowski, Grzegorz Rozenberg, Proc. Theory and Application of Graph Transformations*, volume 1764 of Lecture Notes in Computer Science, pp. 323–337. 2000.
10. D. Latella, I. Majzik, and M. Massink. Towards a formal operational semantics of UML Statechart Diagrams. In *Proc. IFIP TC6/WG6.1 3rd International Conference on Formal Methods for Open Object-Oriented Distributed Systems*, 1999.
11. Object Management Group. *Meta Object Facility Version 1.3*, September 1999. `http://www.omg.org`.
12. Object Management Group. *Object Constraint Language Specification Version 1.3*, June 1999. `http://www.rational.com/uml`.
13. Object Management Group. *XML Metadata Interchange*, October 1998. `http://www.omg.org`.
14. S. Owre, N. Shankar. *The Formal Semantics of PVS*, Technical Report, Computer Science Laboratory, SRI International, August, 1997.
15. A. Schürr: Programmed graph replacement systems. in G. Rozenberg (ed.): *Handbook on Graph Grammars: Foundations*, Vol. 1, Singapure: World Scientific, 479–546, (1997).
16. A. Schürr. Introduction to PROGRES, an attributed graph grammar based specification language. In M. Nagl, editor, *Graph–Theoretic Concepts in Computer Science*, Lecture Notes in Computer Science, vol. 411, 151–165, , Springer, Berlin, (1990).
17. G. Taentzer, C. Ermel, M. Rudolf. The AGG approach: language and environment. In. [5].
18. D. Varró. Automatic transformation of UML models. Master's thesis, Budapest University of Technology and Economics, 2000.
19. D. Varró, G. Varró, and A. Pataricza. Designing the automatic transformation of visual languages. Submitted to *Science of Computer Programming.* Special issue to appear in December 2000.
20. D. Varró, G. Varró, and A. Pataricza. Designing the automatic transformation of visual languages. In H. Ehrig and G. Taentzer, editors, *GRATRA 2000 Joint APPLIGRAPH and GETGRATS Workshop on Graph Transformation Systems*, pages 14–21. Technical University of Berlin, Germany, March 2000.