Technical Report

Automated Algorithm Generation for Visual Control Structures

by
Szilvia Gyapay
Dániel Varró

December, 2000

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

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

Contents

Chapter 1

Introduction

1.1 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.

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.

UML represents a collection of best engineering practices that have proven to be successful in the modelling of large and complex systems, and recently, UML has been regarded as the standard object–oriented modelling language.

1.1.1 Formal Methods in System Design

Formal methods provide a rigorous and effective way to model, 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).

System verification is carried out on mathematical models, e.g. Petri nets (e.g. modelling parameters of system performance), Data Flow Networks (used for modelling fault propagation), temporal logic (for verifying safety critical criteria). A common property of these methods is that they should traverse extremely large state spaces, which may easily lead to combinatorial explosion.

However, the use of formal verification tools (like model checkers SPIN [10] or PVS [21]) in IT system design is hindered by a gap between practice–oriented CASE tools and sophisticated mathematical tools. This gap is a result of the traditionally system verification which was carried out by running hand–written (by mathematical experts

and not by the designers), ad–hoc implementation of mathematical models on the system description.

- 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.

1.1.2 Mathematical Model Transformation

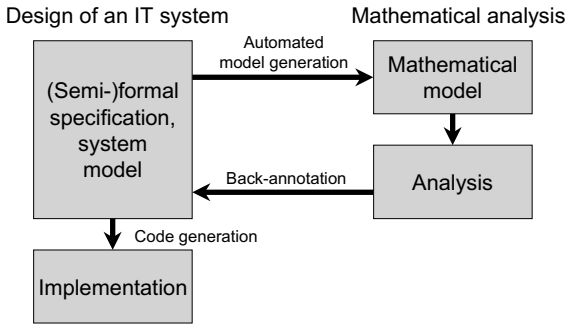


Figure 1.1: A novel approach

According to novel approaches in IT system design [4, 25], the distinct phases of textual specification and system model are integrated into a (semi-)formal UML based description of the system. The mathematical model is planned to be generated automatically from the system model, and the results of the analysis are back-annotated to the same model. As a result, the time spent on re-designing the system is decreased, moreover, the implementation is also supported by automated code generation.

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 [15]

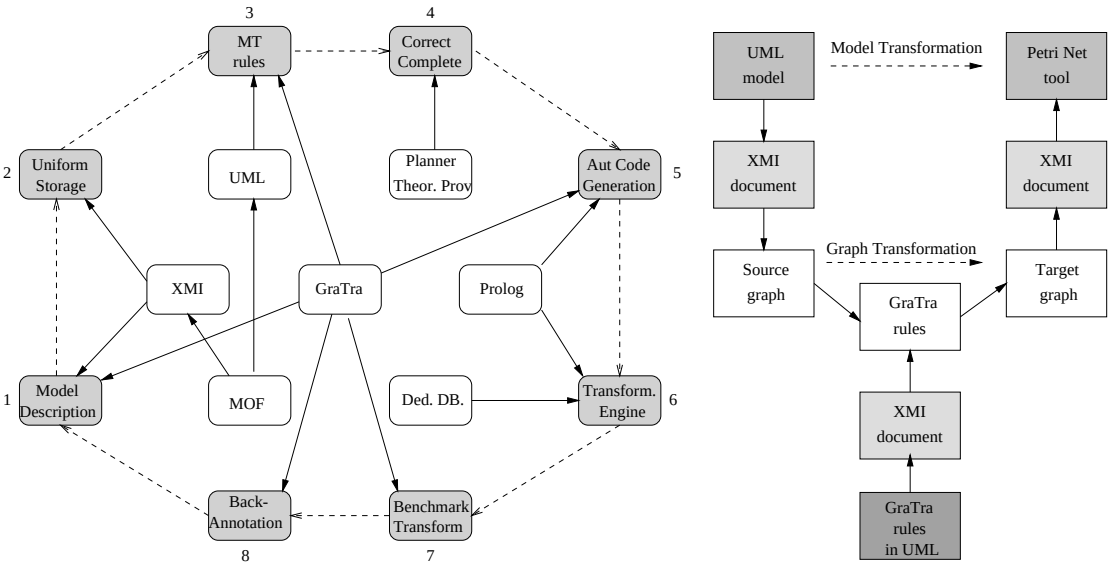
- quantitative analysis of dependability attributes [5, 6]).

A first formal specification method for such model transformations (called VIATRA) was given in [27, 25, 28] (as a result of an ongoing research at the Department of Measurement and Information Systems (DMIS), Budapest University of Technology and Economics) aiming to provide a general framework for a *visual, automated model transformation system* with the facilities of an *automatically generated transformation algorithm of a proven quality*.

The process of model transformation is characterized by a model analysis round-trip illustrated in Fig. 1.2. Typically, a system designer and a transformation designer participates in such a round-trip with the following roles.

- A transformation designer specifies model transformations from UML to various mathematical models. 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.

1.2 A Visual Automated Model Transformation System



(a) Architectural concepts (b) Technological summary

Figure 1.2: An overview of model transformation

Model description A well–defined transformation necessitates a uniform and precise description of source and target models, and should follow the main standards of the industry (as having industrial relevance), therefore, a formal underlying formalism is needed. For this reason, the **Unified Modelling Language (UML)** is used as the front–end of

model transformations, and 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 [19], which allows the definition of meta–objects for similarly behaving instances (more details can be read in chapter 3).

Uniform description 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)** [18], which is a special dialect of XML, the approving novel standard of the Web.

Due to the fact that UML models can be exported in an XMI format (the export process is supported by most UML CASE tools) an open, tool–independent architecture is obtained.

Designing model transformation rules The visual specification of these transformations is supported by **graph transformation**, which combines the advantages of graphs and rules into an efficient 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 during its application (discussed in more details see Chapter 2).

The model transformation rules are aimed to be specified by using the visual notation of UML [26]. However, for obtaining a tool–independent transformation specification, the transformation rules will also be exported in an XMI based format, conforming to the approving standard of graph transformation systems [1].

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 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* could be based on graph models and visual transformation rules. (The current paper mainly contributes to that specific phase of the model transformation round–trip).

The transformation engine As being a logical programming language based on pattern matching methods, Prolog seems to be the most suitable language for the implementation of the transformation engine (a short description of Prolog is discussed in Section 6.1). Therefore, the uniform, XMI based models and rules are translated into a Prolog graph notation serving as the input data and the algorithm to be executed by the transformation engine. An attributed and typed 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.

Benchmark transformations The 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 [25] for further details) has already been designed and implemented. The model transformation method of the current report provide a further benchmark of both academic and industrial relevance.

Back–annotation of analysis results As the results of the mathematical model transformation 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.

1.3 Automatic Transformation of Control Structures

1.3.1 Objectives

In the current report, some major questions of automatic code generation for model transformation systems are discussed. A general automatic code generation approach for such systems must deal with at least the following.

- Generating code for model transformation rules including the graph patterns specified by the left hand side of rules and required modifications (addition, deletion).
- Generating code for the control structure specified by control conditions in transformation units.

The aim of the current report is to extend the model transformation environment by developing a transformation supporting the automated generation of an appropriate algorithm in Prolog from a UML based control specification of transformation units.

In fact, several CASE tools provide the facility of some automated code generation into object–oriented programming languages (like Java or C++). However, code generation in this case means to project the static information contained by UML class diagrams (e.g. in Rational Rose) or function calls described by sequence diagrams (as in a product of TogetherSoft), which is much more simple, than decoding dynamic behaviour.

Existing solutions for implementing UML statecharts (e.g UML DynaCode), which statecharts are used for defining the internal behaviour of objects, lack a general (and verifiable) methodology for the automation.

1.3.2 Conceptional Overview

The control flow in model transformation units is specified in a high abstraction level visual modelling language (i.e. UML statecharts) to avoid the use of an entirely new mathematical method or a purely textual (and thus lower level) programming language, which may be difficult to handle by transformation designers.

As the transformation engine in VIATRA is Prolog due to its powerful unification method which can be exploited in an efficient graph pattern matching for model transformation rules, Prolog was chosen for the target language of implementation. Although the logic programming language Prolog is considered to be one of the highest abstraction level programming languages, it still does not reach the abstraction level of a 4GL[1] visual language like UML.

As a result, an automated algorithm generation has to bridge a huge abstraction gap between a visual and a textual language as usual. As a consequence, our method divides the entire transformation from UML statechart specification of the control structure of model transformation units to executable Prolog code into three distinct phases.

- At first, an abstract model called **Control Flow Graph (CFG)** is generated from the UML based visual specification which represents the control flow in a general way.
- Secondly, a **Prolog code specific (PC) graph model** is derived from the CFG model that is closely related to the final executable Prolog code.
- Finally, a simple algorithm traverses the Prolog code graph and prints the syntactic elements attached to nodes and edges into the target file.

Each transformation step is defined visually by means of well–formed model transformation rules.

All these rather theoretical concepts have been implemented in Prolog, and investigated on benchmark applications in order to assess its run–time performance, and informally verify the correctness of the transformation.

Please note, that during this implementation phase, function (or rather predicate in case of Prolog) calls for model transformation rules and units were treated as atomic (without revealing their internal content), since the current paper does not deal with the automatic code generation for model transformation rules but control structures.

1.3.3 Summary

The report extends the state–of–the–art in the following topics.

- As its main achievement, the current report extends the existing model transformation environment at DMIS with the *facility of an automatic algorithm generation for implementing control structures.*
- As the automated code generation method (which is based on model transformation rules) is *completely implemented in Prolog* (and tested with respect to its run–time performance, in addition) it *serves as a benchmark* of the model transformation approach.
- It provides an *automated implementation for control conditions of transformation units*, which units are widely used in GRACE [13], a future environment in the graph transformation community.
- Due to its model independent constructs and the underlying mathematical framework (i.e. model transformation), it may serve as *a basis for arbitrary future code generation problems.*

[1] Fourth Generation Language

1.4 The Structure of the Report

The structure of the current paper can be divided into three main parts.

The first part (from Chapter 2 to 4) provides a short introduction to both theoretical and practical background of automated algorithm generation.

- **Chapter 2** gives a short introduction to the **theoretical foundations of graph and model transformation** including formal definitions, the main properties of graph and model transformation, and the **concept of transformation units** focusing on its **control description by control conditions** which will serve as the basis of the paper.

- **In Chapter 3**, the basic notation of UML and major concepts of **MOF metamodelling** are discussed. In addition, the **overloaded visual notation of UML statecharts** for supporting visual specification of initial control flow description is described in details.

- **In Chapter 4**, an **overview** is provided discussing the process of the transformation **from UML statecharts to executable Prolog code** of control structures such as **Function calls**, **sequential execution**, **loops**, **if–then–else structure** and **parallelly execution** corresponding to control conditions.

In the second part, Chapters from 5 to 7 are dominated by my own contributions, together with illustrations of a sample control flow at the end of each phase.

- The uniform generation of an abstract model — **Control Flow Graphs (CFG)** representing the control flow — from UML statechart graphs is introduced in **Chapter 5** and specified by **transformation rules**.

- **Chapter 6** contains a brief **introduction** to the logic programming language **Prolog** and the transformation rules **from CFG model to a Prolog Code specific model**.

- **In Chapter 7**, the **Prolog notation of model descriptions** used by the generation codes are introduced together with the **graph traversal algorithm**, which generates the code of the control flow from descriptions in PC model.

.

In **Chapter 8** our transformation method is tested for **time behaviour**, and the intermediate files are assessed with respect to their size on two UML benchmarks, which chapter also concludes our work.

The three appendices contain the complete **source code of transformations** implemented in Prolog;

- the code described in **Appendix A.1** generates a CFG model description from a UML statechart description, according to the CFG Model chapter,

- **Appendix A.2** contains source code to generate a Prolog Code model description from the CFG model description, according to Chapter 6.

- Finally, the Prolog algorithm (discussed in Chapter 7) of transformation unit control is obtained by execution of the Prolog source code (in Appendix A.3) on Prolog Code model description of control flow.

In the following chapter, the a short theoretical foundations of graph and model transformation is discussed (focused on control of transformation units).

Chapter 2

Theoretical Basis of Model Transformation

Graphs are well–known and frequently used means to represent complex objects, diagrams and networks, like flowcharts, entity-relationship diagrams, Petri Nets, and many more. Rules have proved to be extremely useful for describing local transformations; areas like e.g. language definition, logic and functional programming, theorem proving.

Graph transformation combines the advantages of both graphs and rules into a single computational paradigm, used for generation, manipulation, recognition, and evaluation of graphs [3].

The theoretical foundations of model transformation are built upon the concepts of graph transformation. Both source and target models are described by a special typed and directed graph called model graph, and the construction of the target model from a given source model is described by special graph transformation rules called model transformation rules. (For more details see [28].) The current chapter briefly introduces the theoretical basis of both graph and model transformation systems.

2.1 Graph Models and Graph Transformation

2.1.1 Graph Models

Definition 2.1.1 *A **directed graph*** $\mathsf{G} = (\mathsf{NODES}, \mathsf{EDGES}, \mathsf{source}, \mathsf{target}, \mathsf{label})$ *consists of a finite set* NODES *of **nodes** , a finite set* EDGES *of **edges**, two **mappings** assigning the source and the* target *node to each edge, and a mapping* label*, assigning a labelling symbol from a given alphabet to each node and each edge.*

An edge e *in* G *goes from the source* source(E) *to the target* target(e) *and is **incident** to* source(E) *and* target(E)*.*

Definition 2.1.2 *A graph* L *is a **subgraph** of* G*, denoted by* $\mathsf{L} \subseteq \mathsf{G}$*, if the node and the edge sets of* L *are subsets of the respective sets of* G*, and the source, the target and label mappings of* L *coincide with the respective mappings of* G *restricted to* L*.*

Definition 2.1.3 L *has an **occurrence** in* G*, denoted by* $\mathsf{L} \to \mathsf{G}$*, if there is a mapping* occ *which maps the nodes and the edges of* L *to the nodes and the edges of* G*, respectively, and preserves sources, targets, and labellings.*

Definition 2.1.4 *Labels in* ***typed graphs*** *are divided into classes, called types, and that edges of a certain type are restricted to be incident only to certain types of source and target nodes. Typed graphs can be specified by so–called* ***graph schemata***

Definition 2.1.5 ***Attributed graphs*** *are equipped with attributes. An attribute can be a number, a text, an expression, a list or even a graph. Attributes can be of different types and attribute operations compatible with these types are available to manipulate the attributes.*

2.1.2 Graph Transformation

Graph transformation *consists of applying a rule and iterating this process. Each* ***rule application*** *transforms a graph by replacing a part of it by a graph.*

Each rule r contains a left–hand side (LHS) graph L and a right–hand side (RHS) graph R. The application of r to a graph G replaces an occurrence of the LHS L in G by the RHS R. This is done by

1. finding an occurrence of L in G
2. removing a part of the occurrence of L from G,
3. gluing R and the remaining graph D,
4. connecting R with D via the insertion of new edges between the nodes of R and those of D.

The gluing of R and D is specified in the gluing component of a rule and the connection of R with D in the embedding component. Since the replacement shall often be done in a controlled way, rules may also contain application conditions.

Definition 2.1.6 *A* ***graph transformation rule*** $\mathsf{r} = (\mathsf{L}, \mathsf{R}, \mathsf{K}, \mathsf{glue}, \mathsf{emb}, \mathsf{appl})$ *consists of two graphs* L *and* R*, called the* ***left–hand side*** *and the* ***right–hand side*** *of* r*, respectively, a subgraph* K *of* L *called* ***interface graph****, an* ***occurrence*** glue *of* K *in* R*, relating the interface graph with the right–hand side, an* ***embedding relation*** emb*, relating nodes of* L *to* R*, and a set* appl *specifying the* ***application conditions*** *for the rule.*

Definition 2.1.7 *An* ***application*** *of a rule* $\mathsf{r} = (\mathsf{L}, \mathsf{R}, \mathsf{K}, \mathsf{glue}, \mathsf{emb}, \mathsf{appl})$ *to a given graph* G *yields a resulting graph* H*, provided that* H *can be obtained from* G *in the following five steps.*

1. CHOOSE *an occurrence of the LHS* L *in* G*.*
2. CHECK *the application conditions according to* appl*.*
3. REMOVE *the occurrence of* L *up to the occurrence of* K *from* G *as well as all* ***dangling edges****, i.e. all edges incident to a removed node. This yields the* ***context graph*** D *of* L *which still contains an occurrence of* K*.*
4. GLUE *the context graph* D *and the RHS* R *according to the occurrences of* K *in* D *and* R*. That is,construct the disjoint union of* D *and* R *and, for every item in* K*, identify the corresponding item in* D *with the corresponding item in* R*. This yields the* ***gluing graph*** E*.*

5. EMBED *the RHS* R *into the context graph* D *according to the embedding relation* emb. *For each removed dangling edge incident with a node* v *in* D *and the image of a node* v′ *of* L *in* G, *and each node* v″ *in* R, *a new edge (with the same label) incident with* v *and the node* v″ *is established in* E *provided that* v′, v″ *belongs to* emb.

The application of r *to* G *yielding* H *is called* ***direct derivation*** *from* G *to* H *through* r *and is denoted by* $G \Rightarrow_r H$ *or simply by* $G \Rightarrow H$.

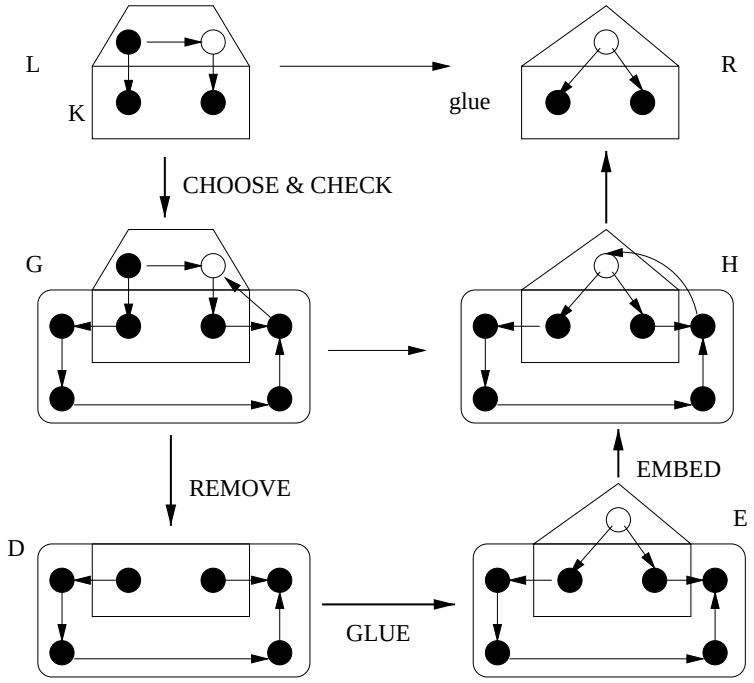


Figure 2.1: Illustration of a graph transformation step

Figure 2.1 illustrates these steps on a sample graph where appl requires the occurrence of L in G to be isomorphic to L and emb relates the unfilled nodes of L and R.

At first sight, the definition of a graph transformation step looks unnecessarily complicated but it has the main advantage that all the relevant graph transformation approaches (e.g. [8, 7, 16, 24]) can be defined as a special case of it.

2.1.3 Graph Transformation Systems

Definition 2.1.8 *A set* P *of rules is the simplest form of a* ***graph transformation system****.*

Definition 2.1.9 *A set* P *of rules together with an* ***initial graph*** S *and a set* T *of* ***terminal labels*** *form a* ***graph grammar****.*

Definition 2.1.10 *Given a set* P *of rules and a graph* G_0, *a sequence of successive direct derivations* $G_0 \Rightarrow G_1 \Rightarrow \ldots \Rightarrow G_n$ *is a* ***derivation*** *from* G_0 *to* G_n *by rules of* P, *provided that all used rules belong to* P. *The graph* G_n *is said to be* ***derived*** *from* G_0 *by rules of* P.

Definition 2.1.11 *The set of all graphs labelled with symbols of* T *only that can be derived from the initial graph* S *by rules of* P*, is the* ***language*** *generated by* P*,* S*, and* T*.*

Please note the non–determinism occurring in graph transformation: We first have to choose one of the rules applicable to a given graph. Furthermore, the chosen rule may be applicable at several occurrences of its LHS. The result of a graph transformation depends on these choices which are still completely arbitrary.

The non–determinism may be restricted by control conditions in several ways:

1. by determining the next rule in dependence on the previous ones, or,
2. by applying a rule according to priority.

2.2 Model Transformation

2.2.1 Model graph transformation

Definition 2.2.1 *A* model graph *G is a directed, typed and attributed graph with the following structure.*

- *A graph node is associated with an identifier Id, and a type T.*
- *An edge has an own Id, a reference to a source Id_S and a target Id_T identifier, with a type T_edge.*
- *Both nodes and edges may be related to attributes (represented as special graph nodes) with an Id identifier an data value V.*

A first version of the graph model can be found in [28] together with an algorithm building the appropriate graph structure from an XMI document.

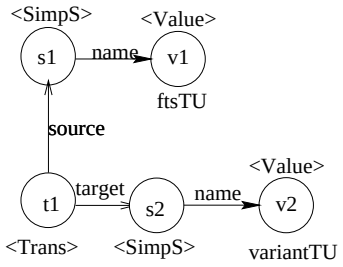


Figure 2.2: A sample model graph

Example 2.2.2 *In Figure 2.2, a sample model graph can be observed containing*

- model graph **nodes** such as
  - `SimpS` typed `s1` and `s2`
  - `Trans` typed `t1`,
- model graph **edges** (their own ID are not shown in the illustration)

  - a `source` typed from `t1` to `s1`,
  - a `target` typed edge from `t1` to `s2`. The meaning of `Trans` node is that there exists a transition between `s1` and `s2` and `source` denotes `s1` as the starting node,
  - two `name` typed edges from `s1` to `v1` and from `s2` to `v2`,

- `v1` and `v2` nodes are **attributes** attached to `SimpS` typed nodes, indicating their names (data) as values.

.

Definition 2.2.3 *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*), connected only by reference edges and nodes.*

A sample model transformation rule `RuleSS` is depicted in Figure 2.3. As can be seen, `SimpS` typed nodes of source model in the left–hand side (indicated by Left–Source), while Left–Target is empty, i.e. the target model before the application of the rule has no corresponding (already transformed) nodes, are transformed to `CFG` nodes in the Right–Target by the application of `RuleSS`, while the source model is left unchanged.

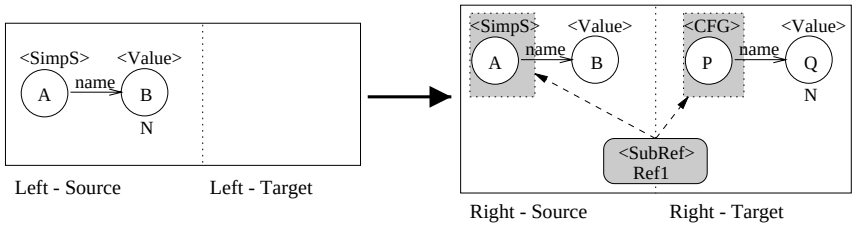


Figure 2.3: A sample model transformation rule RuleSS

A model transformation, which applies to model graphs RuleSS *as long as possible*, transforms the previous (Fig. 2.2) sample model graph of a given source model to the depicted one in Figure 2.4 of a target model.

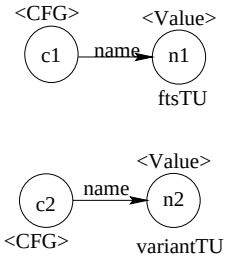


Figure 2.4: The transformed sample model graph to a target model

The process can be followed in Figure 2.5 as seeing both model graphs during the transformation.

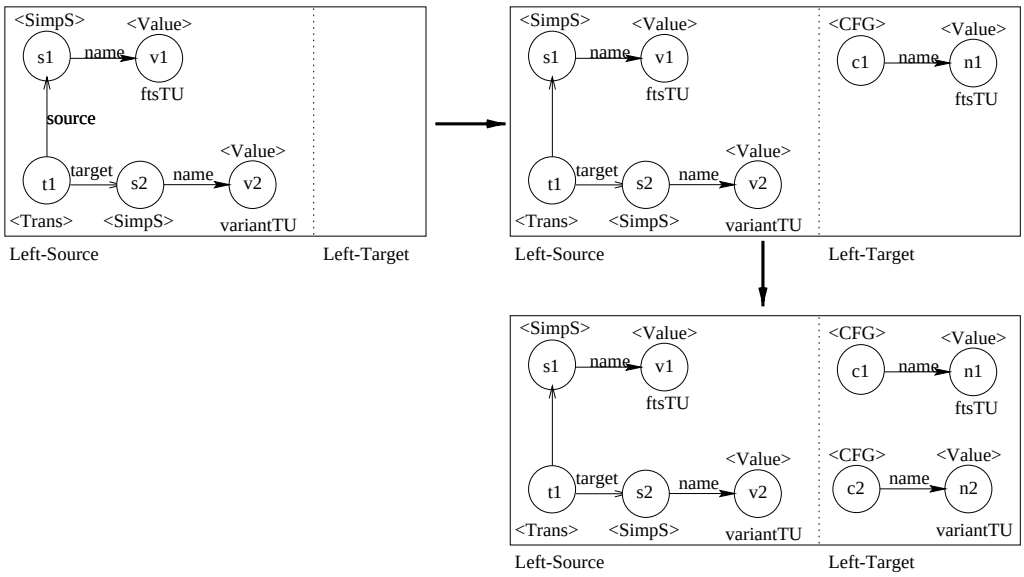


Figure 2.5: Result of the model transformation

2.3 Graph Transformation Unit

Real applications of graph transformation surely consist of hundreds of rules thus the management requires some sophisticated mechanisms. Structuring principles are needed to eliminate the feature that different graph transformation approaches are suitable for different applications. One of the several modularization concepts for graph transformation systems is the transformation unit which allows to decompose large graph transformation systems into small reusable units. These units are independent of a specific graph transformation approach, namely, they are suitable to manipulate graphs in all existing or user-defined graph transformation approaches.

The graph transformation is a sequence of rule applications starting with initial graphs, hence it can be regarded in such a way that it obtains an input-output relation on graphs where each pair comprise an initial graph and its transformed graph. In particular, transformation units provide a functional abstraction meaning that a complex graph transformation is encapsulated as a binary relation of graphs.

A transformation unit consists of five basic elements:

- **Initial and terminal graphs.** The initial and terminal graphs define the possible start and the final states of the transformation, the description of initial graphs can be interpreted as a precondition for the graph transformations performed by the transformation unit and the terminal graphs can be regarded as postconditions.

- **Rules.** Rules is a finite set of arbitrarily rules (graph transformation rules in our case).

- **Uses.** This is a set of imported transformation units providing a modular structuring mechanism. One of the main feature of transformation units is unity which allows to view an imported unit as atomic to use its functionality in the current transformation unit.

- **Control conditions.** The application of the transformation rules can usually proceed in a non-deterministic way for two reasons. The first reason of the non-deterministic is the applicability of more than one rule to a graph. Finding an occurrence of the LHS of the rule in the graph is the second one (it emerged in the introduced sample, too). Although this non-determinism is useful during the designing and analysing phase, it has to be eliminated for the final algorithm generation phase.

  In order to ensure the correctness of the generated model transformation code or algorithm, its specification must be completely deterministic and be able to be regulated by control mechanisms. With the help of the transformation units, the control flow of the transformation algorithm can be described and given a sequence of steps by the extended regular expressions of the control conditions.

In the following, transformation units will be introduced formally, following [3, 14].

Definition 2.3.1 *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).*

In order to be independent from a particular graph model, a specific type of rules and the transformation units are defined over some graph transformation approaches.

Since control conditions may contain identifiers (usually referred to imported transformation units or rules), the semantics of control conditions depends on its environment.

Definition 2.3.2 Environment *is a mapping, which associates each identifier with a binary relation.*

A *graph transformation approach* $\mathcal{A}$ consists of

- a class $\mathcal{G}$ of *graphs*
- a class $\mathcal{R}$ of *rules*
- a *rule application operator* $\Rightarrow$ specifying a binary relation $\Rightarrow_r \subseteq \mathcal{G} \times \mathcal{G}$ for each $r \in \mathcal{R}$, i.e. $\Rightarrow_r$ yields all pairs $(G, G'$ of graphs where G' is obtained from G by applying r
- (in order to specify the initial and terminal graphs of a transformation unit) a class $\mathcal{E}$ of *graph class expressions*, such that each $e \in \mathcal{E}$ specifies a subclass $SEM(e) \subseteq \mathcal{G}$
- and a class $\mathcal{C}$ of *control conditions* over some set ID of identifiers (since control conditions may contain identifiers usually for the imported transformation units or local rules, their semantics depends on their *environment*) such that each $c \in \mathcal{C}$ specifies a binary relation $SEM_E(c) \subseteq \mathcal{G} \times \mathcal{G}$ for each every mapping $E : ID \rightarrow \mathcal{P}(\mathcal{G} \times \mathcal{G})$ where ID is a set of names and E is the related environment.

A transformation unit over a graph approach $\mathcal{A}$ means that $I, T \in \mathcal{E}$, $R \subseteq \mathcal{R}$ and U is a set of (already defined) transformation unit over (A).

Definition 2.3.3 *The* ***interleaving semantics*** *of transformation unit contains a pair* G, G' *of graphs if* G *is an initial graph and* G' *is a terminal graph,* G *can be transformed into* G' *using the rules and the imported transformation units, and* (G, G') *is allowed by the control condition.*

As the current paper is concerned with the transformation of control conditions, the following section provides a brief overview of most commonly used control conditions, those described by a set of extended regular expressions (discussed in [28, 14] in details).

2.3.1 Control Conditions

Control conditions can regulate and restrict the process of the transformation in order to be completely deterministic. Any binary relation on graphs may be used as control conditions.

For example every string language L over ID (or an extended arbitrary set of control conditions) can serve as a control condition: for an environment E each string $x_1 \cdots x_n$ in L specifies the binary relation obtained by the sequential composition of the meaning of x_i in E, i.e. $SEM_E(x_1 \cdots x_n) = E(x_1) \circ \cdots \circ E(x_n)$ and the empty string specifies the identity relation on $\mathcal{G}$. For instance, if $x_1, \ldots, x_n (n \geq 1)$ are graph transformation rules and $E(x_i)$ represents all pairs (G_i, G_i') of graphs such G_i' is obtained by applying x_i to G_i, then $x_1 \cdots x_n$ specifies all pairs (G, G') where G' is the result of the application of $x_1, \ldots, x_n$ in this order to G. L specifies the union of the semantic relations given by elements of L.

In the following, a selection of control conditions used to specify and program with graph transformation will be discussed:

- The class of *regular expressions* REG over is recursively given by $\varepsilon, \emptyset \in REG$, $ID \subseteq REG$, and $(e_1; e_2)$, $(e_1|e_2)$, $(e*) \in REG$ if $e, e_1, e_2 \in REG$, where
  - the expression ε requires that no rule or imported transformation unit is applied
  - $\emptyset$ specifies the empty set
  - $id \in ID$ applies the appropriate rule or transformation unit exactly once
  - $e_1; e_2$ stands for the **concatenation** of e_1 and e_2, i.e. the applying of e_2 right after e_1
  - $e_1|e_2$ means the union of e_1 and e_2, i.e. they can be applied parallelly (non–deterministically) (**fork**)
  - e^* represents the transitive closure of e, i.e. arbitrarily often iterating e.

  Formally, for each environment E:
  - $SEM_E(\varepsilon)$ means the identity relation on $\mathcal{G}$
  - $SEM_E(\emptyset) = \emptyset$
  - $SEM_E(id) = E(id)$
  - $SEM_E(e_1; e_2) = SEM_E(e_1) \circ SEM_E(e_2)$ where $\circ$ means the sequential composition
  - $SEM_E(e_1|e_2) = SEM_E(e_1) \cup SEM_E(e_2)$ and
  - $SEM_E(e^*) = SEM_E(e)^*$.

- The control condition $once(id)$, where id denotes a rule or an imported transformation unit, allows only interleaving sequences in which id is applied **exactly once** while other rules or imported transformation units can be applied arbitrarily often.

  Formally, for each environment E, $(G, G') \in SEM_E(once(id))$ if there exist $G_0, \ldots, G_n \in \mathcal{G}$ and $id_1 \ldots id_n \in ID^*$ such that

1. $G_0 = G$ and $G_n = G'$
2. $(G_{i-1}, G_i) \in E(id_i)$ for $i = 1, \ldots, n$, and
3. there exists exactly one $i \in \{1, \ldots, n\}$ with $id_i = id$.

- A control condition c specifies a binary relation on graphs. The control condition $c!$ serves as an induced control condition which applies c **as long as possible**.

  Formally, for each environment E the condition $(c!)$ specifies that a pair (G, G') is in $SEM_E(c!)$ if $(G, G') \in SEM_E(c)^*$ and there is no graph G'' with $(G', G'') \in SEM_E(c)$.

- A **conditional** is of the form **if** a **then** c_1 **else** c_2 serves as a **branch** of the control flow (depending on the evaluation of a), where c_1, c_2 are already defined control conditions.

  Formally, for each environment E it allows all pairs $(G, G'$ of graphs such that $G \in SEM_E(a)$ and $(G, G') \in SEM_E(c_1)$ or $G \notin SEM_E(a)$ and $(G, G') \in SEM_E(c_2)$.

According to the introduced extended regular expressions (all of described control conditions), the definition of model transformation unit is the following:

Definition 2.3.4 *A* model transformation unit *is a transformation unit where*

- the **graph model** is the one described in Section 2.1.1
- the R set of **rules** are well–formed **model transformation rules**,
- the class of **control conditions** (containing control flow information) are composed of concatenation, once, as long as possible, branch and fork control conditions.

One may easily observed, that the control conditions allowed in model transformation units provide the most general control structures used in traditional programming languages. Therefore the automatic generation of corresponding transformation algorithm in one of these languages is natural goal.

A sample model transformation unit with some introduced control condition is depicted in Fig. 2.6 (taken from [26]) for illustrating the concept of transformation units and control conditions.

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

Figure 2.6: Sample model transformation unit *"uml2imTU"*

Figure 2.6 shows a model transformation unit (`uml2imTU`), 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,

- `uml2imTU` 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.

2.4 Conclusion

In the current chapter, the basic foundations of graph and model transformation were discussed. The paradigm of graph transformation was introduced to provide a rule–based manipulation method for arbitrary graphs commonly used in system modelling. Model transformation systems serving as a framework for automatic transformations between system and mathematical models were introduced on the basis of graph transformation by means of model graphs, model transformation rules and units.

To achieve an automatically generated model transformation code or algorithm, the final specification (created by the transformation designer) should be completely deterministic otherwise the verification of the generated transformation code against the specification of the transformation is almost impossible. With the help of transformation units, the control flow of the transformation algorithm can be described by control conditions in such a way where the gap between the control specification and a declarative (or rather logic) programming language (Prolog in our case) is minimal.

In the following, the visual representation of the control flow of model transformation units will be described using the notation of the Unified Modelling Language.

Chapter 3

Visual Control Structure in UML

The **Unified Modelling Language (UML)** is a language for specifying, visualizing, constructing, and documenting the artifacts of software systems, as well as for business modelling and other non–software systems. The UML represents a collection of best engineering practises that have proven successful in the modelling of large and complex systems.

3.1 Introduction

The development of IT systems and software products are described from different aspects represented by various sort of UML diagrams such as e.g.

- The all of **use case diagrams** is to identify system level relations at the top level of hierarchy. They show the relationships among **actors** and and **use cases** (the specification of a sequence of actions, including variants, that a system can perform, interacting with actors of the system) within a system.
- **Class diagrams** provide a collection of declarative (static) model elements, such as classes, types, and their contents and relationships.
- **Sequence diagrams** show object interactions arranged in time sequence. In particular, it shows the objects participating in the interaction and the sequence of messages exchanged.
- **Collaboration diagrams** show interactions organized around instances and their links to each other.
- **Statechart diagrams** implement a state machine. They consist of hierarchy of states and transitions between states both used to describe the internal behaviour of system object. Statechart diagrams are widely used in the design and specification of embedded systems (like ABS in cars, lift controllers, further machines with built–in computers).

  For defining the control of model transformation systems, statechart diagrams will be discussed in details in the following. (See [17] to get a more detailed description on the previous diagrams.)

State machines in UML statechart diagrams differ from classical statecharts (introduced by Harel in [9]) in several ways [17].

- At first, classical statecharts are meant to specify behaviours of processes, while UML state machines primarily come to represent behaviour of a type.
- Harel's statecharts have an elaborated set of predefined actions, conditions and events which are not mandated by UML state machines.
- The notion of activities (processes) does not exist in UML state machines. Therefore all predefined actions actions and events that deal with activities are not supported, as well as the relationships between states and activities.
- Classical statecharts are based on the zero-time assumption, meaning transitions take zero time to execute. The whole system execution is based on synchronous steps where each step produces new events that will be processed at the next step.

3.1.1 UML Specification of Model Transformation

As discussed in Chapter 2, 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, in VIATRA the visual notation of UML has been overloaded [26] (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.

As a result, the transformation designers may use their well–known UML CASE tools. Therefore, during the specification of model transformation, only the visual syntax of UML statecharts is used while its original semantics is omitted, i.e. the UML notation and its original software modelling concepts are overloaded for the description of model transformation systems. As a result, the visual construction of transformation units, rules and control flow structures can be defined in an easy–to–understand UML notation.

There are several similarities between UML notation and control conditions in model transformation units:

- In transformation units, control conditions are usually defined by extended regular expressions (see Section 2.3.1). As regular expressions can easily be transformed into a 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).

- The dynamic internal behaviour of a class is described by UML statecharts, which encapsulate the events and actions to which the class is sensitive. The control and internal operation of transformation units are characterized by a similar encapsulation, i.e. representing the control flow by transitions between states.

As the structure of UML statecharts is usually defined by the standard UML metamodel released by the Object Management Group on the basis of their Meta Object Facility (MOF) standard, the next section contains a short introduction on the concepts of MOF metamodelling (following basicly [25]).

3.2 Meta Object Facility

The concepts of metamodelling originate in the need for an effective design process of **formal specification and modelling languages**. The large number of similar languages — often supported nowadays by visual diagrams — necessitates a common description language (called **metameta–model** later) that is able to describe the instances of these languages as **sentences**. Traditionally, such a description is based upon a set of production rules called a **grammar**.

However, the sentences of this top–level modelling language (called later as a **model**) can be used for designing a lower level grammar for generating lower level languages hence a **model hierarchy** is available in this sense with several **meta–layers** where the sentences of a higher level language can be used for specifying a lower level language.

This hierarchy can be observed in Figure 3.1, and later demonstrated also in Table 3.1.

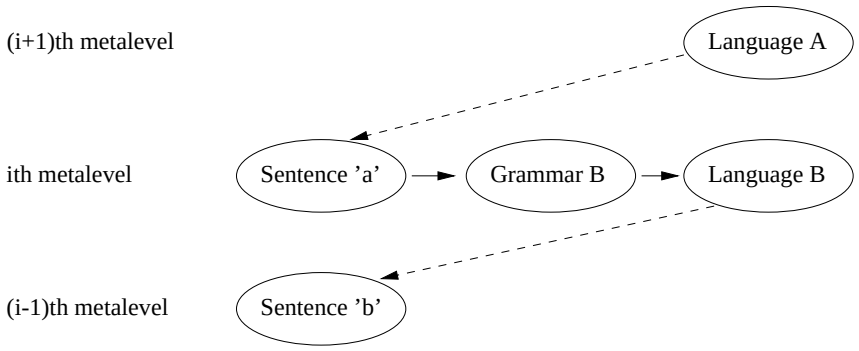


Figure 3.1: Meta–layers in language specification.

3.2.1 Basic MOF Notation

Metadata is a general term for data which in some sense describes a language, and usually defined in the terms of the **Meta Object Facility (MOF)** standard [19].

In this MOF context, the term **model** has a broader meaning than in its general sense (namely, description of something in the real world). Here, *a **model** is a collection of metadata* that is related in the following ways:

- The collection of metadata describes **information**.
- All the metadata conforms to rules controlling its structure and consistency, i.e. it has a common **abstract syntax**.
- The metadata has a **meaning** in a common semantic context.

Metadata itself is a kind of information, hence it can be described by metadata as well. In the MOF terminology, *metadata that describes metadata is called **meta–metadata***, and *the model that consists of a meta–metadata is called a **metamodel***. Metamodels are integrated into a *topmost level meta–metamodel, which is the **MOF Model***, by defining a common syntax for the definition of several metamodel types.

The MOF metadata framework is typically depicted as a four layer architecture that can be observed in Table 3.1.

Meta–level	MOF terms	Examples
M3	meta–metamodel	The MOF Model
M2	meta–metadata metamodel	GraTra metamodel (interchange format)
M1	metadata model	GraTra models, e.g. a graph grammar
M0	data	modelled systems, e.g. a graph

Table 3.1: MOF Metadata Architecture

3.2.2 The MOF Model

*The **MOF Model** is an abstract language for defining MOF metamodels.* Originally, it was developed to provide a general means for describing the language constructs of UML.

Although MOF and UML was designed for different purpose (i.e. metadata versus system modelling), the MOF Model and the core of the UML metamodel are closely related in their modelling concepts (classes for objects of similar structure, associations as relations between these classes, generalization, etc.); therefore, the corresponding UML notation is commonly used for MOF–based metamodels as well. Nevertheless, in order to distinguish between the metamodel elements of UML and the basic constructs of the MOF Model, latter ones printed with capitals.

The main metadata modelling constructs provided by the MOF are the following:

- ***Classes** are type descriptions of "first class instance" MOF meta–objects.* Classes defined at the M2 meta–level have their instances at the M1 level. The structural features of Classes can be described at both object and class level by **Attributes** (value holders in an instance of the class), **Operations** (specifying the name and type signature by which the behaviour is invoked). Classes may also inherit their structure and behaviour from other Classes by **Generalization**.

- ***Associations*** *support binary relations between Class instances.* Each Association has two **AssociationEnds** that may specify aggregation semantics and structural constraints on cardinality and uniqueness. When a Class is the type of an AssociationEnd, the Class may contain a **Reference** that allows navigability of the Association's links (i.e. Classes) from a Class instance.

- ***Packages*** *are collections of related Classes and Associations.* Packages can be composed by importing other Packages by inheriting from them. They can also be nested, which provides a form of information hiding.

- ***DataTypes*** *allow the use of non–object types* for Operation parameters and Attributes.

- ***Constraints*** *are used to describe semantic restrictions on elements in a MOF metamodel by defining well–formedness rules for the metadata described by a metamodel.* The **Object Constraint Language (OCL)** [20] is often used as a formal language for expressing constraints.

A semi–formal description of the UML metamodel [17] (containing each of the more than 120 pictorial objects) was released by the OMG using the constructs of the MOF Model as a convincing demonstration that such a rich language as UML can be described by a small subset of its own modelling language.

This section illustrated that the MOF Model provides a natural method to specify and design various metamodels even in different levels of hierarchy. This visual language is a well–defined subset of UML, the widely–known standard of object–oriented software design, thus its modelling concepts can easily be understood in academic research as well as in industrial applications.

In the current paper, UML statechart graphs serve as the basis of model transformation. UML statechart graphs are simplified statecharts represented by model graph (see the definition of model graphs in Section 2.1) with the following notation according to the MOF metamodel of UML statechart (depicted in Fig. 3.3):

- **classes** are denoted as model graph nodes (having an identifier and a type),

- **attributes** are denoted as special model graph nodes, they are typed, have an identifier and serve as valueholders in the model,

- **associations** as graph edges typed by the corresponding reference names.

In the following, the features and components of *UML statecharts* will be described by means of the standard MOF–based UML metamodel.

3.3 The MOF Metamodel of UML Statecharts

A **UML statechart diagram** shows the sequence of states that an object goes through during its life in response to events from the surrounding, where states are represented by state symbols and transitions are represented by arrows connecting these state symbols. As hierarchy of states is available in statecharts, states may also contain subdiagrams by physical containment and tiling.

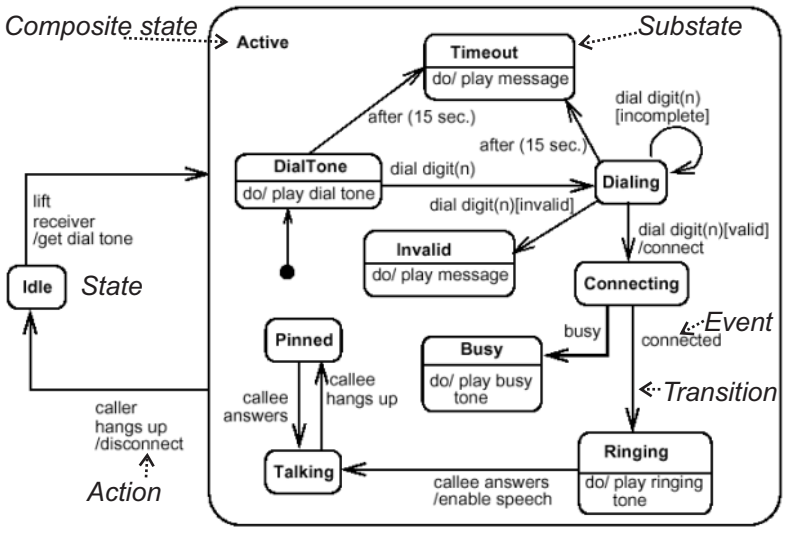


Figure 3.2: A UML Statechart

A sample diagram modelling the process of a telephone call is depicted in Figure 3.2. The telephone is in an `Idle` state until one `lifts the receiver`. When the receiver is lifted the telephone enters its `Active` state and plays a `DialTone` in the beginning. For instance a telephone may get into a `Dialing` state from the `DialTone` state as a result of an event called `dial digit`. This fact is indicated by a transition between the two states labelled with the event.

The basic components of statecharts defined in the standard UML Metamodel in a MOF notation [19] (in Fig. 3.3) are the following:

- A **state machine** is a behaviour that specifies the sequences of states that an object goes through during its life in response to events, together with its response and actions. The behaviour is specified as a traversal of a graph of a state nodes interconnected by one or more joined transition arcs. The transitions are triggered by series of event instances.

  In the metamodel a `StateMachine` is composed of `States` and `Transitions`.

  Assocations

  - The `context` association links a `StateMachine` and its owning `ModelElement`, whose behaviour is specified by the `StateMachine`. The `ModelElement` may contain multiple `StateMachines` while each `StateMachine` is owned by one `ModelElement`.
  - The `top` association denotes the top level `State` (exactly one as depicted by multiplicity 1) directly owned by `StateMachine`. Other `States` are owned by parent composite states. The rest of the `StateMachine` is an expansion of this `CompositeState`.
  - The `transitions` associates the `StateMachine` with its `Transitions`. All `Transitions` which are basicly relationships between `States` are owned by `StateMachine`.

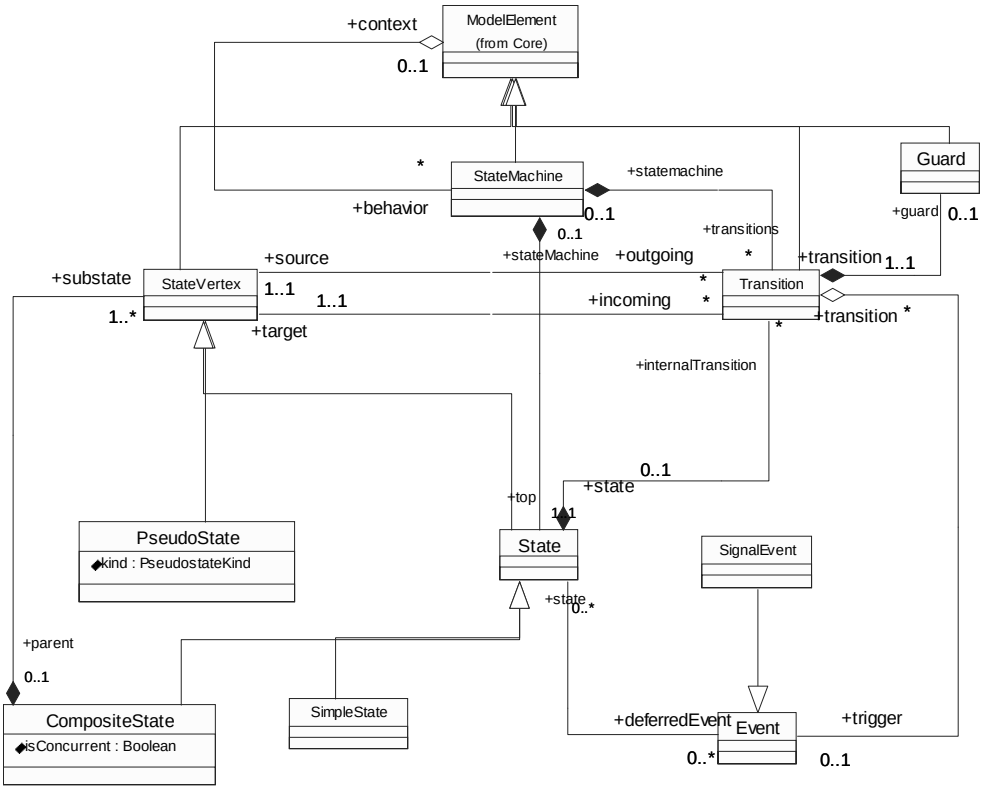


Figure 3.3: The MOF metamodel of UML statecharts

- A **state vertex** is an abstraction of a node in a statechart graph. In general, it can be the source or destination of any number of transitions.

  In the metamodel a `StateVertex` is a subclass of `ModelElement`.

  Associations

  - `incoming` specifies the transitions entering the vertex
  - `outgoing` specifies the transitions having the vertex.

- A **state** is a condition or situation during the life of an object during which it satisfies some condition, performs some action or waits for some events. A state models a dynamic situation in which one or more conditions hold.

  In the metamodel `State` is a subclass of `StateVertex`, thereby inheriting the fundamental features of incoming and outgoing transitions associated with state vertices.

  Assocations

  - `internalTransition` is an associated set of `Transitions` that occur entirely within the `State`.

  - Another association is `deferredEvent` that specifies the `Events` to be deferred if received within the `State`. Multiplicities '0..*' indicates that a `State` can can defer multiple `Events`, and an `Event` can be deferred by multiple `States`.

- A **simple state** is a state that does not have subvertices.

  In the metamodel a `SimpleState` is a subclass of `State` that does not have any additional features.

- A **composite state** is a state that consists of subvertices.

  In the metamodel a `CompositeState` is a subclass of `State` that contains one or more subvertices that are subtypes of `StateVertex`.

  Assocations and attributes

  - Its `subvertex` association denotes a set of `States` that constitute the subvertices of a `CompositeState`. Each subvertex is uniquely owned by its parent `CompositeState`.
  - The `isConcurrent` attribute is a boolean value that specifies the decomposition semantics: if this attribute is true, then the composite state is decomposed directly into two or more orthogonal conjunctive components (usually associated with concurrent execution). If its value is false, then there are no direct orthogonal components in the composite state. This means that exactly one of the subvertices can be active at a given instant (i.e. sequential execution).

- A **pseudostate** is an abstraction of different types of nodes in the state machine graph which represent transient points on transition paths from one state to another. Pseudo states are usually used to construct complex transitions from simple transitions.

  In the metamodel a `PseudoState` is a subclass of `StateVertex`, which generalizes all statechart nodes.

  Attributes

  - It possess the `kind` attribute that can be (e.g. **initial**, **fork** or **branch** to determine the kind of the pseudostate.

- **FinalState** is the point where the described flow must stop.

  In the metamodel a `FinalState` is a subclass of `State` with the mentioned feature.

- An **event** is the specification of an occurrence that has a location in time and space. An instance of an event can lead to the activation of a behavioural feature in an object.

  In the metamodel an `Event` is a subclass of `ModelElement` and is a part of a `Transition` that represents its `trigger`.

- A **signal event** represents events that result from the reception of a signal.

  In the metamodel `SignalEvent` is a subclass of `Event`.

- A **guard** condition is a boolean expression that may be attached to a transition in order to determine whether that transition is enabled or not.

  In the metamodel `Guard` is a `ModelElement` so it can substituted in refined state machines. Its `expression` attribute is a boolean expression which specifies the guard condition.

- A **transition** is a relationship between a `source` state vertex and a `target` state vertex.

  In the metamodel`Transition` is a subclass of `ModelElement` that participates in various relationships with other state machine metaclasses (by associations):

  Assocations

  - `trigger` specifies the single `Event` which activates it
  - `guard` is a predicate that must evaluate to true at the instant the transition is triggered
  - `source` denotes the `StateVertex` affected by firing the `Transition`.
  - `target` denotes the `StateVertex` that results from a firing of the `Transition` when the `StateMachine` was originally in the source `State`. After the firing the `StateMachine` is in the target `State`.

After having discussed the metamodel of UML statecharts we turn to the UML–based visual control flow representation of transformation units.

3.4 Control Flow Description by UML Statecharts

Due to the similarities between UML notation and control conditions in model transformation units (see Section 3.1.1), the visual notation of control flow structures will be described in this section by means of the syntax of UML statecharts [26].

Due to the fact that UML statecharts are a generalization of finite automaton (supporting e.g. hierarchical behaviour), the control structures could be encoded into statecharts by extended regular expressions (discussed in Section 2.3.1) according to the assignment described in Table 3.2.

To demonstrate the encoding, a sample about the controlling of the model transformation (i.e. the control flow) is depicted in Fig. 3.4. This figure represents a simplified UML statechart (see Section 3.3 about the features of the UML Statecharts) which describes the control condition of the transformation unit `uml2imTU` (shown in Fig. 2.6) by means of the statecharts.

In the control flow statechart,

- 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.
- **Transitions** without trigger denote the *concatenation* control condition, the self–transition with a trigger "!" identifies the *as long as possible* semantics of a rule or transformation unit.

Regular expression	Statechart notation
concatenation (;)	transitions connecting their source and target states
as long as possible (!)	self-transitions with "!" as SignalEvent
fork (|)	synchronization bar (Pseudostate of kind "Fork")
if–then–else	decision cube (Pseudostate of kind "Branch"))
logical condition	guarded transition (conditions are contained by Guards)

Table 3.2: From regular expressions to statecharts

- **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), and
- 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.

The operation of the depicted statechart is equivalent to the control of the transformation unit `uml2imTU`. For a short reminder, the rules, the imported transformation units and the control of the transformation unit was the following:

- **rules:** variantR, rule_A, rule_B
- **uses:** ftsTU,linkTU
- **control:** ftsTU; (((variantR!); linkTU) | **if** c **then** rule_A **else** rule_B)

The first step is the application of `ftsTU` as well as in the control of the transformation unit `uml2imTU`. Considering the correspondence between the statechart notation and the semantics of extended regular expressions in the Table 3.2 according to the transformation unit `uml2imTU` and the statechart in Fig. 3.4, their equivalence is easy to be verified.

According to the technological process of model transformation (discussed in Chapter 1), the UML statecharts created by a specific CASE tool are exported into an XMI model format. This XMI description is later processed by a parser in order to automatically generate the corresponding graph model (the algorithm is defined in [28]).

This graph model (called UML statechart graph) serves as the input of the model transformation described in Chapter 5.

As an example, the corresponding statechart graph of Figure 3.4 is given in Figure 3.5 (later in Chapter 5 this graph will be transformed into the CFG graph of Fig. 5.23).

3.5 Conclusion

This chapter provided an introduction to several method widely used in the design process of complex IT systems. We summarized the basic notation of UML statecharts and the

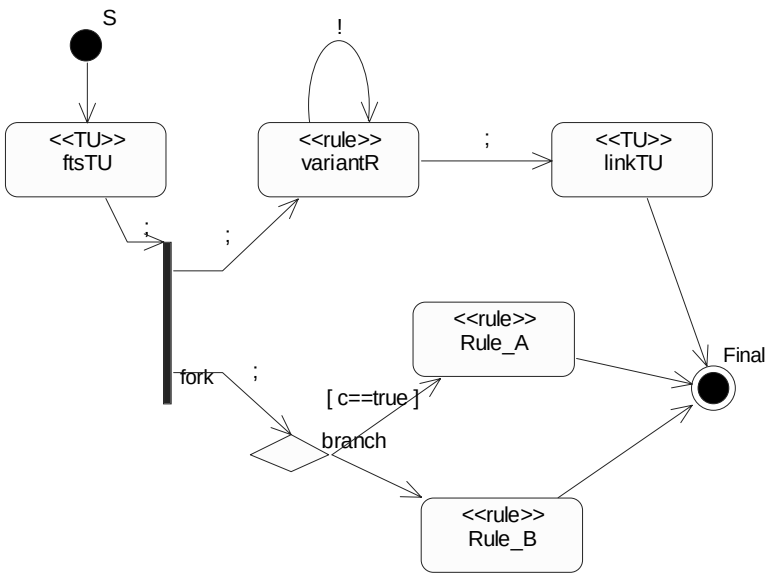


Figure 3.4: Statechart of the control flow

major concepts of MOF metamodelling. By overloading and restricting the visual notation of UML statecharts, an easy to understand but mathematically precise formalism was discussed as visual control specification method for model transformation systems.

The following chapters are concerned with the specification of a complex model transformation from this UML statechart control specification to an appropriate algorithm in Prolog. Each specification start from the description of the source and target metamodels (in a MOF notation) while the process of transformation will be described by well–formed model transformation rules. With this respect, components (i.e. the automated transformation of control specifications) of a complex model transformation system are also defined by using the method of model transformation.

To generate the algorithm of the control flow, the first step is to describe the control flow by UML statechart, then it has to be transform to other models in order to get the final, language specific graph model.

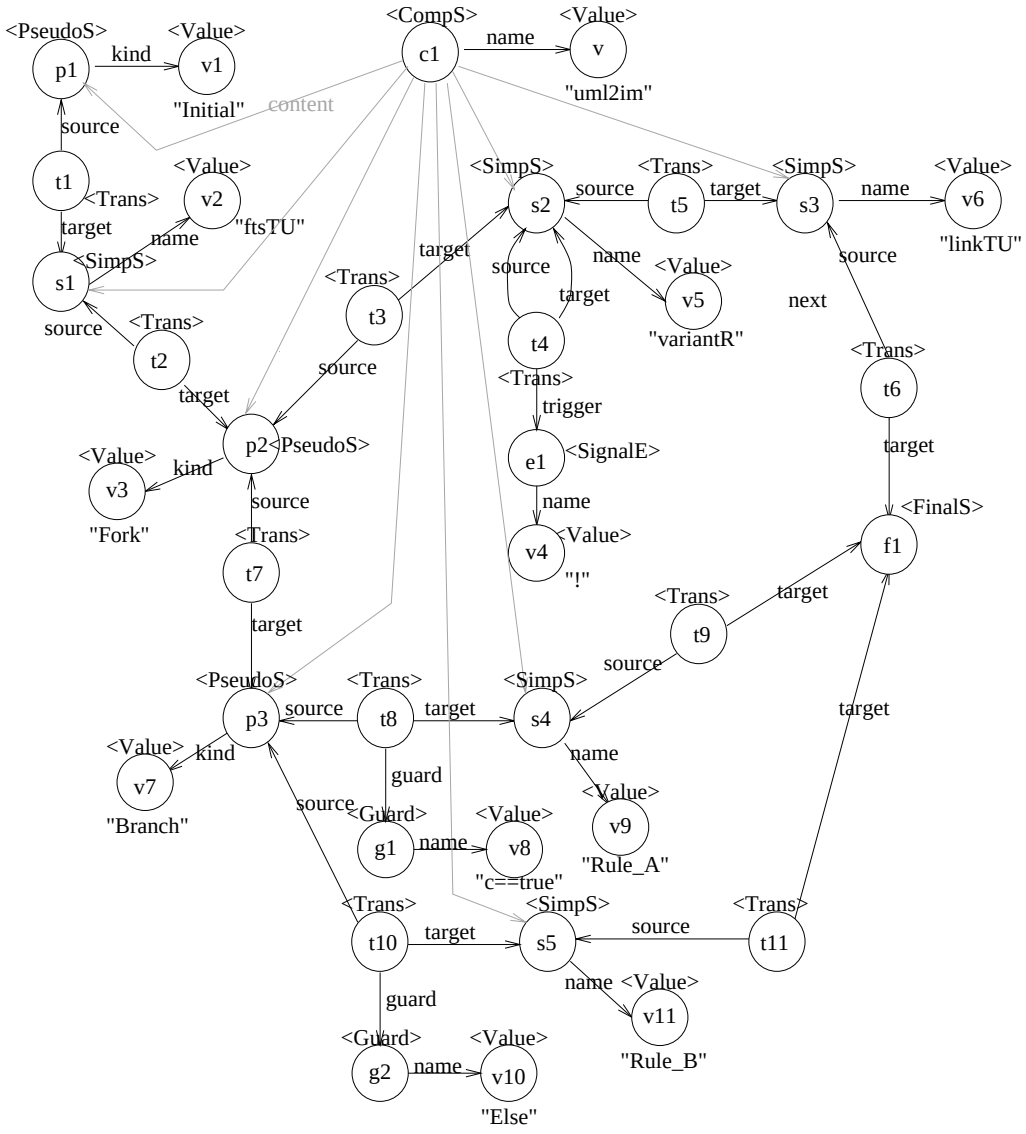


Figure 3.5: The UML statechart graph of Control Flow

Chapter 4

From UML Statecharts to Prolog Code: An Overview

As described in Chapter 3, the control flow in model transformation units is specified in a high abstraction level visual modelling language (i.e UML statecharts) to avoid the use of an entirely new mathematical method or a purely textual (and thus lower level) programming language.

We also discussed in Chapter 1 that Prolog was chosen for the target language of implementation due to its powerful unification method exploited in an efficient graph pattern matching for model transformation rules. Although Prolog is considered to be one of the highest abstraction level programming languages (as using logic for programming), it still does not reach the abstraction level of a 4GL[1] visual language like UML.

As a result, an automated algorithm generation has to bridge a huge abstraction gap between a visual and a textual language. As a consequence, we have chosen to divide the entire transformation from UML statecharts to executable Prolog code into three distinct phases.

- At first, an abstract model representing the control flow in a general way is generated from UML statechart graphs. This abstract model called **Control Flow Graph (CFG)** is a well–known and commonly used construct to describe the flow of control (e.g. widely used by compilers for code optimization). In the current paper, an own version of CFGs is adapted, described in Chapter 5 together with its projection from UML statechart graphs.

- At a second step (see Chapter 6), a **Prolog code specific (PC) graph model** is derived from the CFG model that is closely related to the final executable Prolog code. The reason for this second intermediate model is a farsighted design decision aiming to use a general code generation algorithm for different programming languages.

- Finally (in Chapter 7), a simple algorithm traverses the Prolog code graph and prints the syntactic elements attached to nodes and edges.

We would like to emphasize that although the current paper only discusses the generation of an executable Prolog code, the method applied for the transformation process is

[1] Fourth Generation Language

so general that arbitrary target programming languages could have been chosen including object–oriented (e.g. Java or C++) or functional languages (e.g. Standard ML) as well.

4.1 Basic Assignments

As we discussed previously in Chapter 1, the main task of the current paper is to generate an appropriate algorithm implementing the control flow of transformation units. Thus, function calls to transformation rules or units are handled as *atomic*, disregarding from the implementation of these constructs.

The basic concepts of the overall transformation are summarized in figures 4.1 – 4.5.

- **Function calls**: in our case, function calls represent a call for a rule or for a transformation unit.

  UML Function calls refer to transformation rules and transformation units **as simple states** in UML statecharts. As depicted in the Figure 4.1, the state of the UML statechart calls a transformation rule called `rule_A`.

  CFG Atomic constructs are represented in CFG model as `CFG nodes` coupled with their name by which the referred transformation rule or transformation unit can be executed like a function call.

  Prolog In Prolog code, function (or rather predicate) calls are related to the connected functions by its name: the execution of a function is called in Prolog by its name properly placed in the code (`rule_A`).

- **Sequential execution** as the default control flow element.

  UML A transition from one state to another state is depicted in Figure 4.2 and represents the execution of these states in the given order (indicated by the direction of the arrow between the two states): at first `rule_A`, then `rule_B`.

  CFG In CFG model, `next` edges determine the flow of the process calling functions one after the other.

  Prolog The sequential execution of function calls are separated by commas (`','`) in Prolog, e.g. `rule_A, rule_B`.

- **Loop**: repeat the current step (in our case the execution of a rule or a transformation unit) as long as possible.

  UML UML statecharts denote the as long as possible execution of a states (in Figure 4.3) `rule_A` by `self-transitions triggered by '!'` (the notion of the as long as possible execution in regular expressions) connected to the current state.

  CFG A node with a `Loop type` and a `loop edge`, which has the same source and target node: this `Loop` node, refers to a loop in order to execute the call of the corresponding function (`rule_A`) until it is executable (e.g. applying a transformation rule to a source graph until it has a matching occurrence determined by `rule_A`).

Prolog The corresponding Prolog code consists of two functions. They perform the as long as possible execution of `rule_A` by means of `'fail'` built–in syntactical predicate in Prolog, and the repeating call of the containing function `fun_A` (an artifical function to process these five structures, which is neither a rule nor a transformation unit) (for details see Section 6.1).

- **Branch**: if–then–else structures

  UML Branch constructs are represented in UML statecharts by `decision cube` — holding the `conditions` for conditional threads, i.e. `'Else'` for another thread showing the direction of the process if the evaluation of the condition is true, i.e. false.

  CFG In the intermediate model, branches are shown as Branch nodes having with `Conditional` and `Else` types as their subsequent nodes, to which the appropriate `conditions` (in Figure 4.4: cond) are assigned.

  Prolog Another syntactical built–in symbol is `'!'`, the cut symbol in Prolog. In our sample, `fun_X` calls `fun_A` function (after the execution of `rule_A`). `Fun_A` (neither a rule nor a transformation unit) represents a conditional thread with a cut at the choice points. In case of unsuccessful pattern matching of `cond` the execution leaves this thread and continues at the function with the same name (`fun_A` indicated the `else` thread of the branch).

- **Fork**: the subsequent instructions, operations, etc. can be executed parallelly (or non–deterministically).

  UML This non–deterministic choice is represented in UML statecharts by `synchronization bars` (in Figure 4.5): after the execution of `rule_A`, the process can be continued by calling either `rule_B` or `rule_C`.

  CFG The node with the `Fork` type denotes this parallelly execution in CFG model. The `next edges` show the possible parallelly execution of rules `rule_B` and `rule_C` there are no differences between threads.

  Prolog When Prolog calls `fun_A` both predicates `rule_B` and `rule_C` can be executed by backtracking through the choice point.

In the following chapter, the concepts of the transformation from UML statechart graphs to Control Flow Graph (CFG) will be discussed.

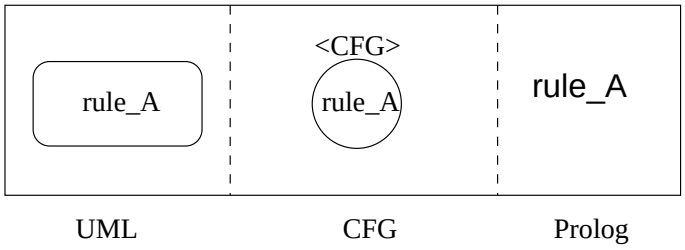


Figure 4.1: Transformation of Function calls

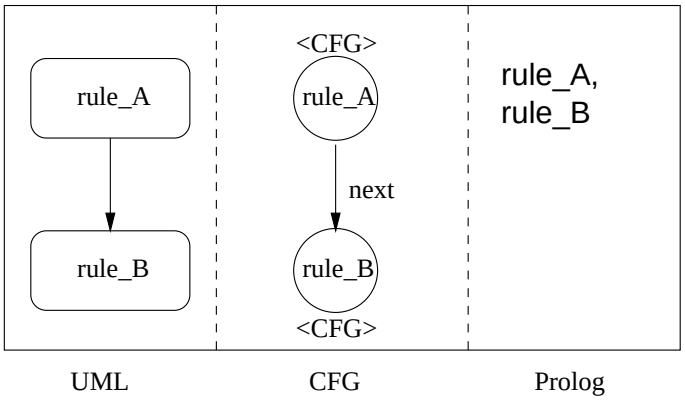


Figure 4.2: Transformation of rule sequences

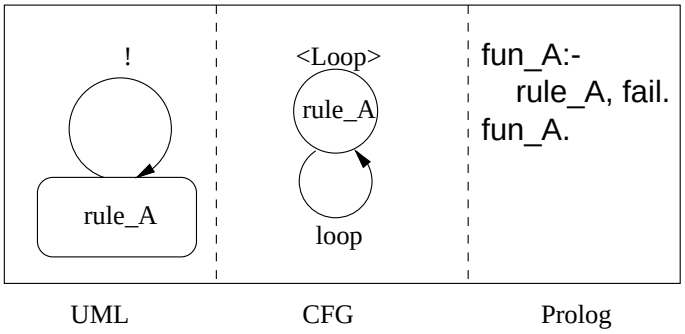


Figure 4.3: Transformation of Loop structure

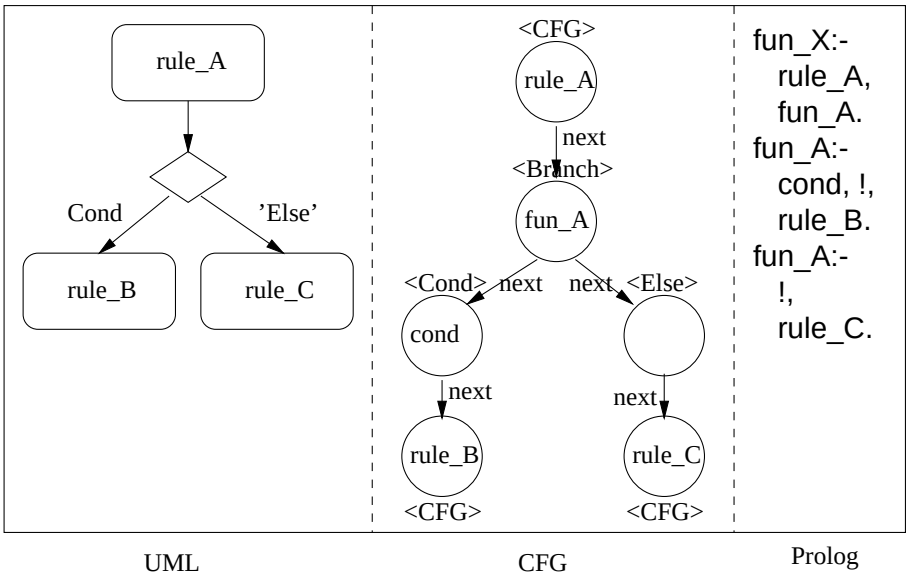


Figure 4.4: Transformation of Branch structure

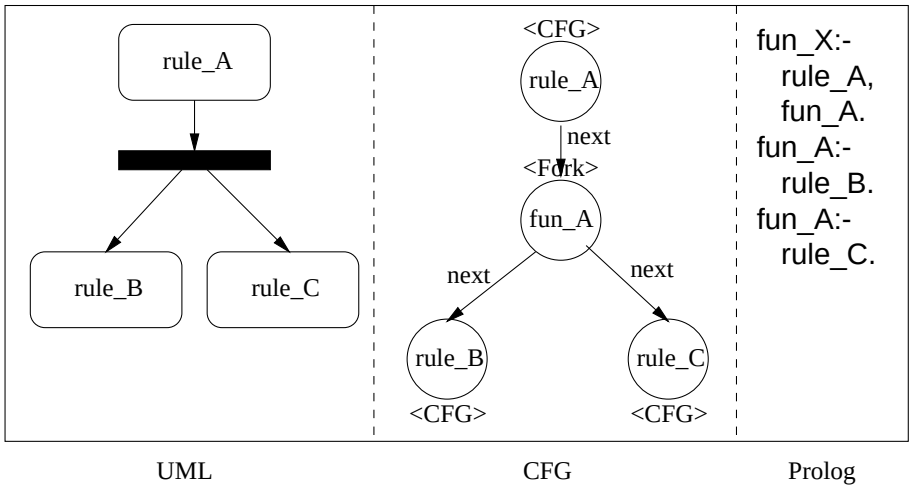


Figure 4.5: Transformation of Fork structure

Chapter 5

From UML Statechart Graphs to Control Flow Models

Control flow graphs are widely used in several fields of computer science. Several nodes in the control flow graph represent computations and the edges represent the flow of control. Thus providing an internal abstraction level between high– and low–level specifications. For instance, in case of compilation [2] is control flow graphs represents the conditional and unconditional jumps between basic blocks of statements. A similar flow graph is used for performing structural tests on programs.

In a model transformation sense, control flow graphs are typed an directed graphs representing the following types of programming elements:

- **Function calls** represent a call for a rule or for a transformation unit.
- **Branch**: if–then–else structures
- **Fork**: the subsequent instructions, operations, etc. can be executed parallelly (or non–deterministically)
- **Loop**: repeat the current step (operation) as long as possible;
- **Sequential execution** as the default control flow element.

As discussed in Chapter 4, the generation of the algorithm for control of model transformation is carried out in two steps from its UML statechart specification to the final language specific model graph. The first transformation is carried out from UML statecharts to a Control Flow Graph (CFG model) model which provides an internal abstraction level of control by means of basic programming elements.

5.1 The MOF Metamodel of Control Flow Graph

The previous concepts are defined by the MOF metamodel of Control Flow Graphs (depicted in Fig. 5.1) in the following way.

- `Fun` node denotes a function (referring to the application of a rule or a transformation unit), and each of them has a `name` (connecting to a `Value` node with a `name` edge) and may have several `ContentNode`s linked by `content` edges (for collecting nodes that belong to a specific function).

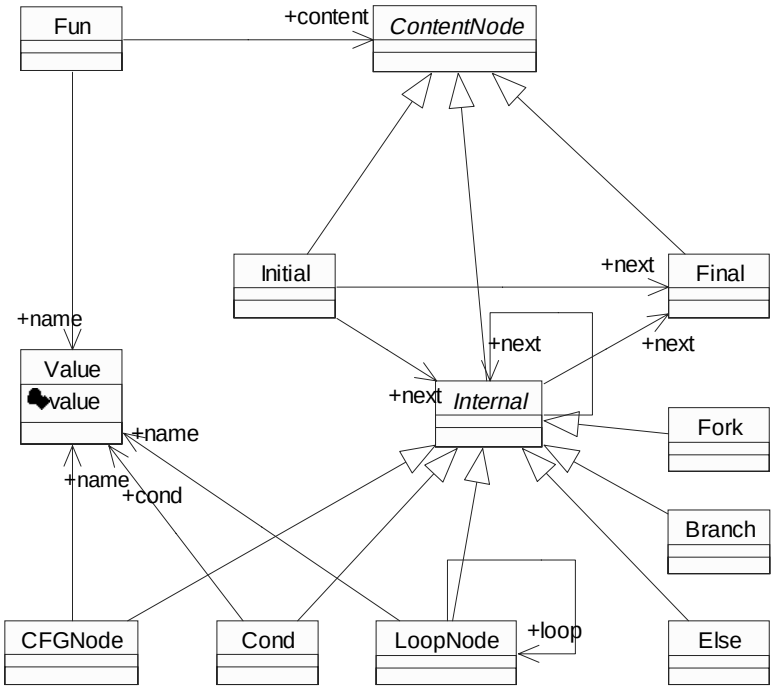


Figure 5.1: The MOF metamodel of the CFG intermediate model

- `ContentNode` is abstract, it is instantiated into either an `Initial`, `Final` or `Internal` node. `Initial` node represents the starting point while `Final` node refers to the final point of the execution for the function which contains them.
- The abstract `Internal` node can be either a `CFGNode`, `Condition`, `Branch`, `Fork` or `Else` node.
  - Each `CFGNode` refers to a rule or transformation unit in the original model (see Section 3.4 and the corresponding rule in Fig. 5.4), and has a `name` as an attribute.
  - `LoopNodes` refer to a rule or transformation unit by `name`, and the connected `loop` edge requires its application as long as possible.
  - `Branch` nodes represent a conditional branching of execution, and may have several `next` nodes: `Cond` and `Else` nodes. If the corresponding `value` of a `Cond` node evaluates to true, the next step is defined by the node subsequent to `Cond`. If there does not exist any true conditions related to a `Branch`, the next step is defined by an appropriate `Else` node.
  - A `Condition` node contains a logical condition (connected to a `Value` node with a `cond` edge).
  - `Fork` nodes have more than one `next` nodes originated from them, specifying that its subsequent nodes can be applied parallelly (non–deterministically).
- `Value` nodes have an attribute serving as a value holder.
- `next` edges can lead in the denoted way between Fun and content nodes with some restrictions in multiplicities.

Multiplication restrictions of edges:

- Each `Fun` node must have at least two `content` nodes: an `Initial` and a `Final` node (exactly one of each).
- More than one `next` edge can leave from `Fork` and `Branch` nodes.
- Each `Initial` node must have one outgoing `next` edge (and has no incoming `next` edge).
- Each `Final` node has only incoming `next` edges (at least one).
- A `Branch` node may have at most one `Else` and at least one `Condition` node connected by a `next` edge; `Condition` and `Else` nodes may only have their incoming `next` edges from `Branch` nodes. These thread nodes have exactly one outgoing `next` edge.
- From each `Fork` node, at least two `next` edges leave.
- `Value` nodes have no outgoing edges.
- Only `LoopNodes` may have a `next` edge to themselves.

The described features, restrictions and properties of the CFG intermediate model are straight consequences of the basic programming concepts. In the next section, the model transformation rules of the transformation from UML statechart graphs to the CFG model will be introduced in details. The correspondence between UML statecharts and UML statechart graphs is discussed in details in Section 3.2.2, and shown in Fig. 3.4 and 5.23. In the following, the notion of UML statechart will always refer to a UML statechart graph.

5.2 Transformation to the CFG Model

The model graph transformation from UML statechart graphs to the corresponding CFG model consists of the application of several model transformation rules. These rules fulfil the notation and execution concepts introduced in Section 2.2 and will be applied to the UML statechart of described in Fig. 3.4 in order to obtain a CFG model (Figure 5.23 shows the resulted CFG model).

Some observation are listed in order to understand the notation of an instance of CFG model:

- At the beginning the CFG model is empty.
- Each elements (nodes) in both models have an **ID** (identifier) **depicted in the centre of the node**.
- Some abbreviations are used for the notation of the nodes and edges in both models.
- The **type of the nodes** (the name of the comprising class) is printed in the rules in <> and begins with a capital letter.
- The **type of the edges** is printed non–capital letters **above the arrows**.
- The data of a **value** to its <Value> node is place **outside the node**.
- The notation of the UML statechart are written with `typewriter letters`, the constructs of the target model (CFG) are written in ***bold italics***, and types of reference nodes between the two models are typed with sans serif fonts.

The flow of the transformation The transformation between the two model uses all of the described rules. The steps of the transformation are carried out in the introduced order of the rules (each rule must fulfil the "as long as possible" semantics of rule applications).

In order to help tracing the process of the transformation, an instance CFG graph (transformed from a UML statechart graph, shown in Fig. 3.5) is provided after each transformation rule illustrating the current state of the transformation emphasizing the effects of the latest rule by gray scale nodes and bold edges. The Prolog implementation of these transformation rules are listed in Appendix A.1.

5.2.1 State Rules

In the following rules, some special correspondences used to couple source and target objects are represented by the reference relation SubRef. When a rule has already been applied to a sub–statechart this fact is denoted by reference edges connected to the appropriate reference nodes.

RuleCompositeState This rule (Fig. 5.2) transforms a `CompositeState` with a `name` to a ***Fun*** node with the similar name (connected by a ***name*** edge to a ***Value*** node). As a result, the ***Fun*** node indicated by gray scale is generated in Fig. 5.3.

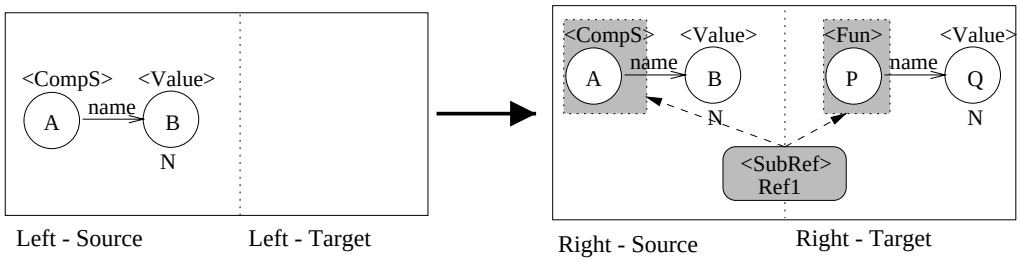


Figure 5.2: Rule of CompositeState nodes (RuleCS)

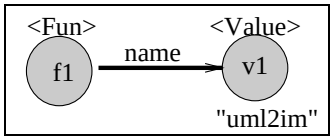


Figure 5.3: RuleCS results

RuleSimpleState By applying rule **RuleSS** (Fig. 5.4), a `SimpleState` and its `name` is transformed to a corresponding ***CFG*** node by `SubRef` reference node and a ***Value*** node which holds the name.

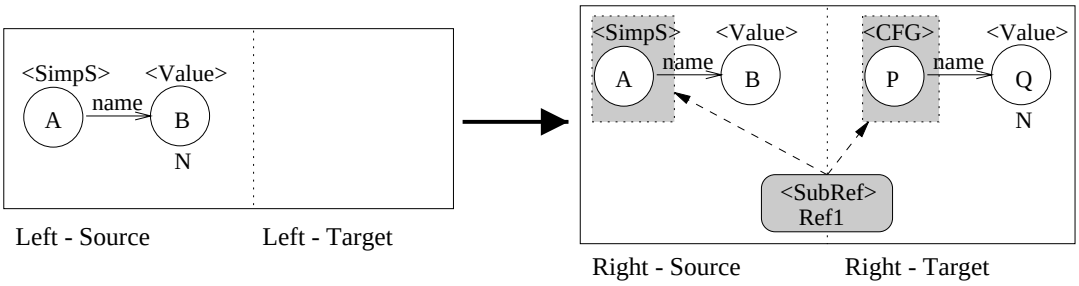


Figure 5.4: Rule of SimpleState (RuleSS)

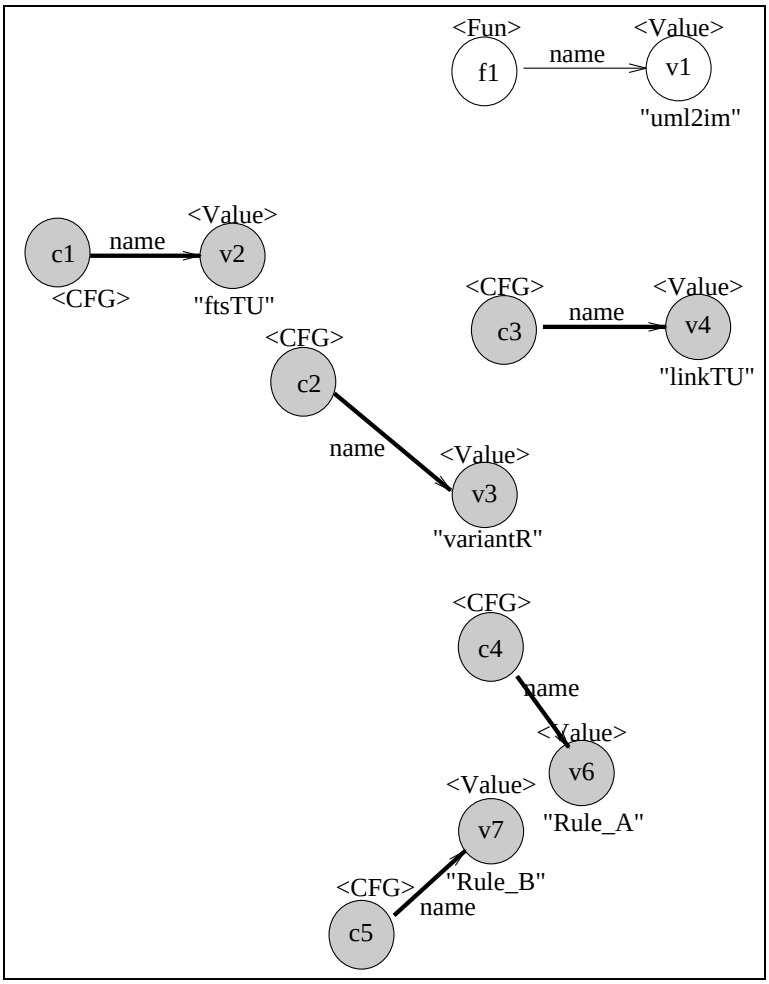


Figure 5.5: RuleSS results

Rules of PseudoStates `PseudoStates` `Branch`, `Fork`, and `Initial` in UML Statechart must be transformed (the rules **RuleBranch**, **RuleFork** and **RuleInitial** are depicted in Fig. 5.6, 5.8, 5.9) to typed nodes ***Branch***, ***Fork*** and ***Initial*** nodes respectively, i.e. these rules simplify the notation (in the corresponding figures showing the results (Fig. 5.7 and 5.10) the process of rule application can be followed).

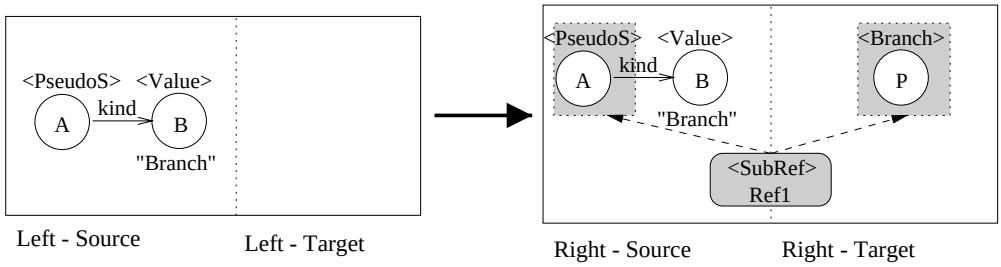


Figure 5.6: Rule of PseudoState Branch (RuleBranch)

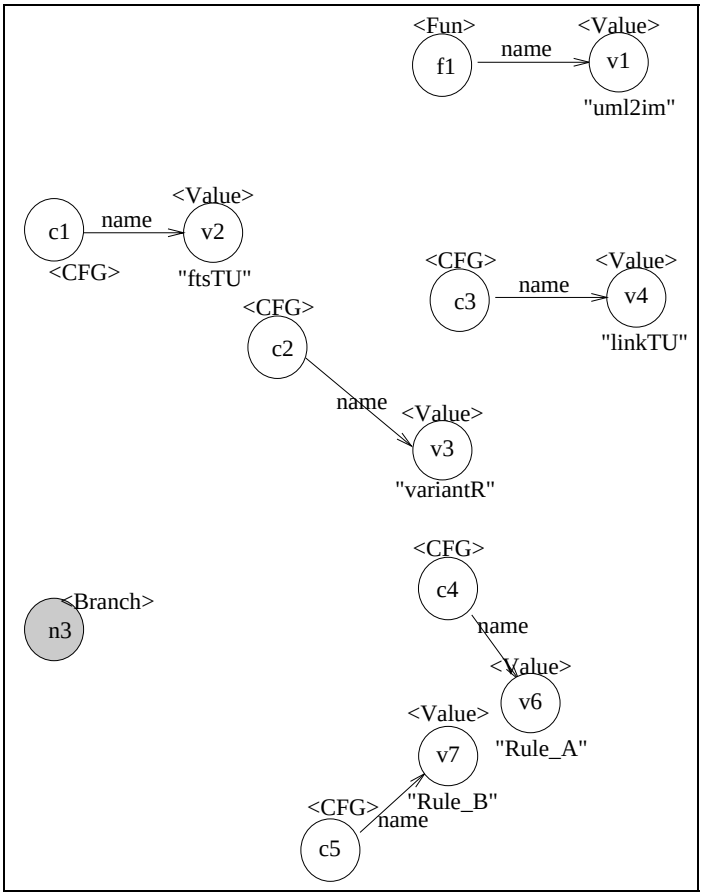


Figure 5.7: RuleBranch results

RuleInitial The rule handling the ***Initial*** node is depicted below.

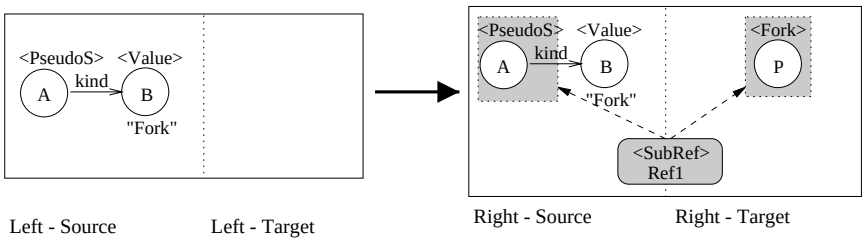


Figure 5.8: Rule of PseudoState Fork (RuleFork)

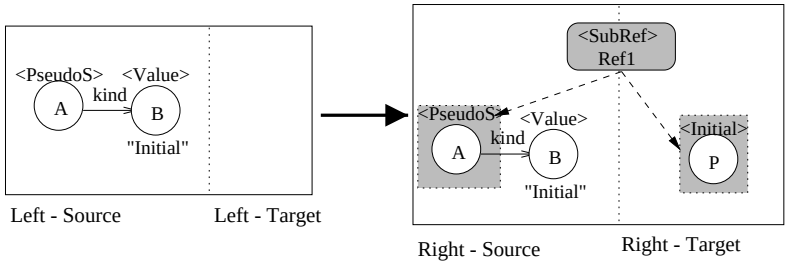


Figure 5.9: Rule of PseudoState Initial (RuleInitial)

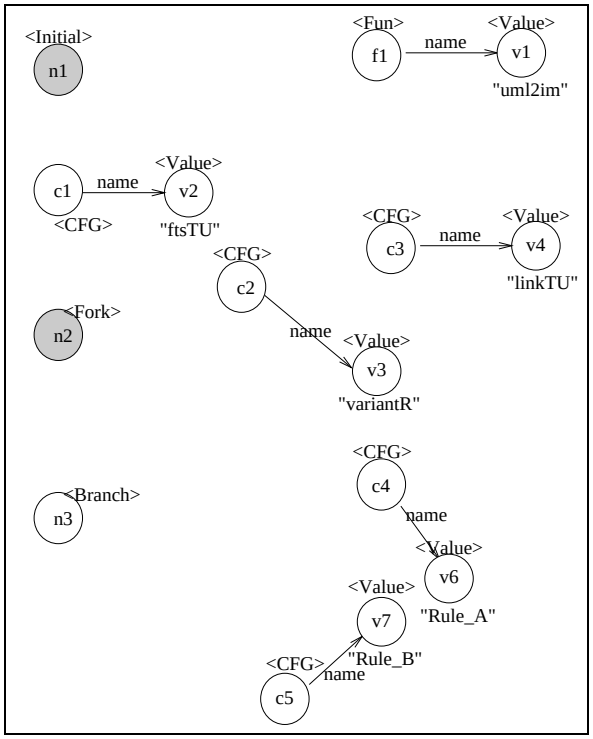


Figure 5.10: RuleInitial results

RuleFinal `FinalStates` are transformed into ***Final*** nodes keeping its functionality, i.e. this state denotes the end of the process (see Fig. 5.11).

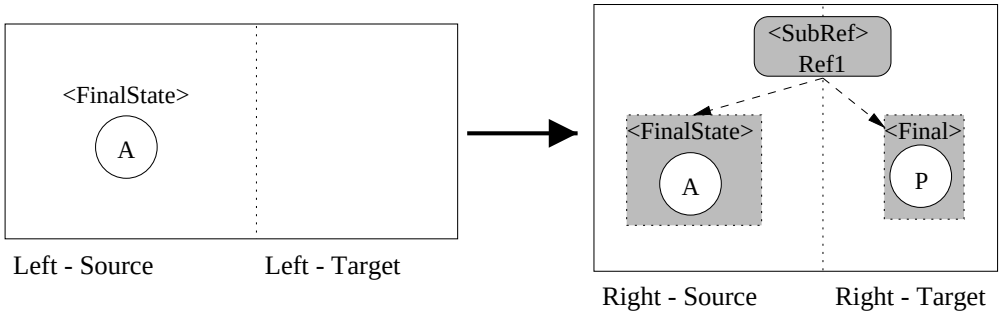


Figure 5.11: Rule of FinalState (RuleFinal)

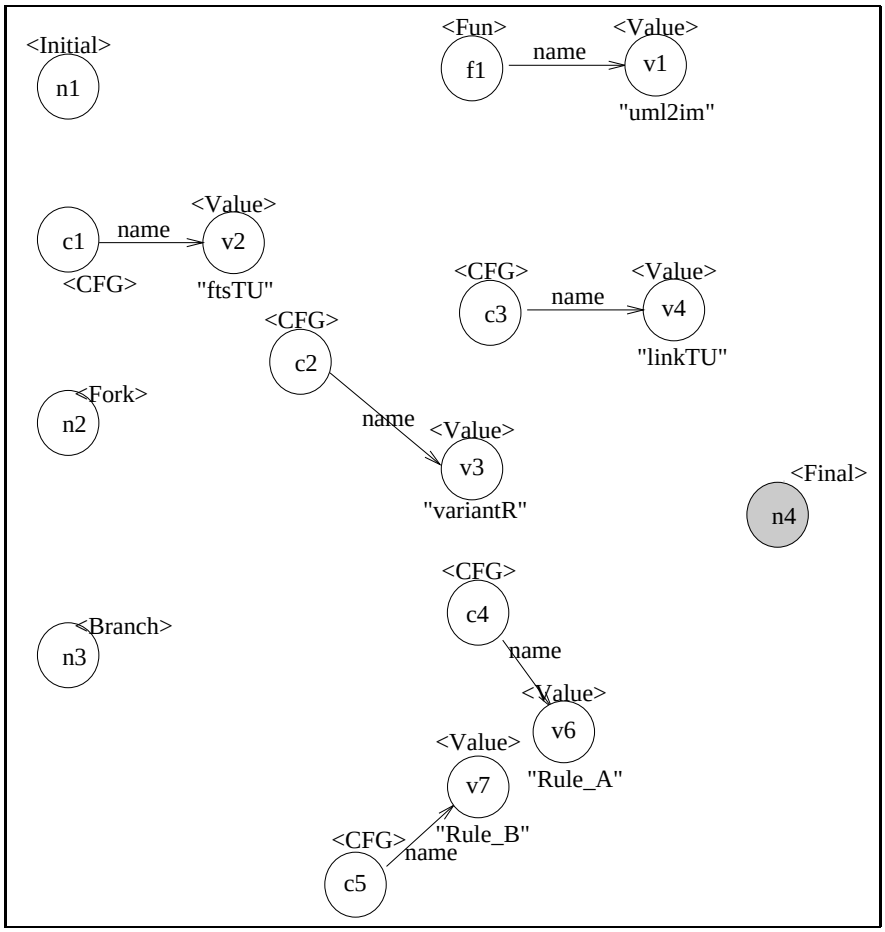


Figure 5.12: RuleFinal results

5.2.2 Edge Rules

RuleSubvertex As can be seen in the MOF metamodel of UML statecharts (in Fig. 3.3), `CompositeStates` have `subvertices`. The appropriate notion (depicted in Fig. 5.13) in CFG model is that ***Fun*** nodes connect to their "subnodes" by ***content*** edges.

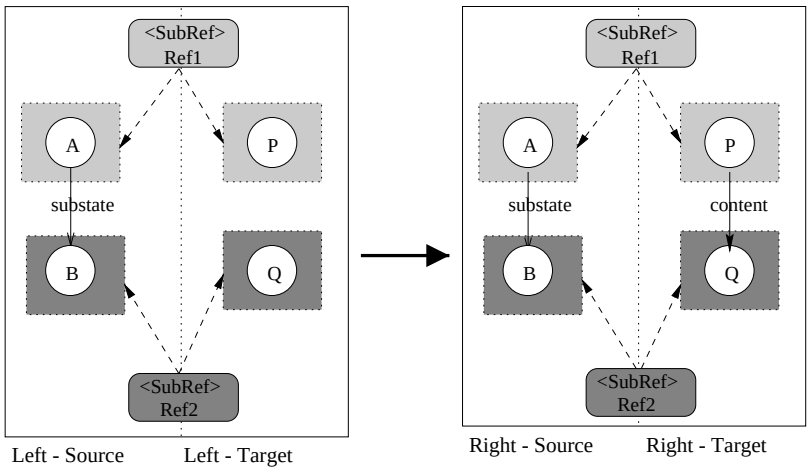


Figure 5.13: Rule of subvertex edge (RuleSubvertex)

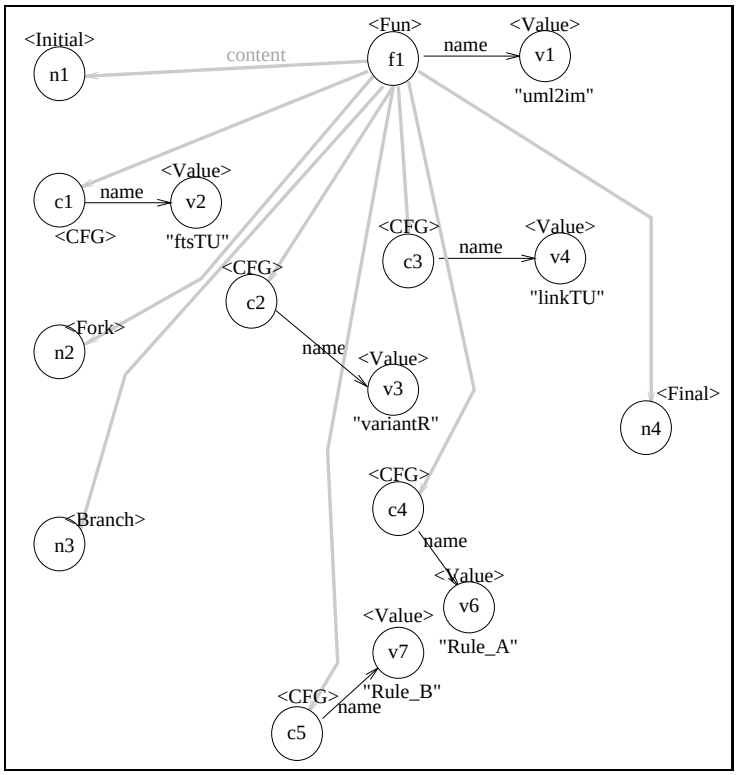


Figure 5.14: RuleSubvertex results

RuleTrans The transition from a `source` vertex to a `target` vertex is represented in UML statechart graphs by a `Transition` statechart node which connects the source vertex to a `source` edge and to the target vertex to a `target` edge. **RuleTrans** transforms this structure to a ***next*** edge between the two corresponding (referred by SubRef references) nodes in the target model.

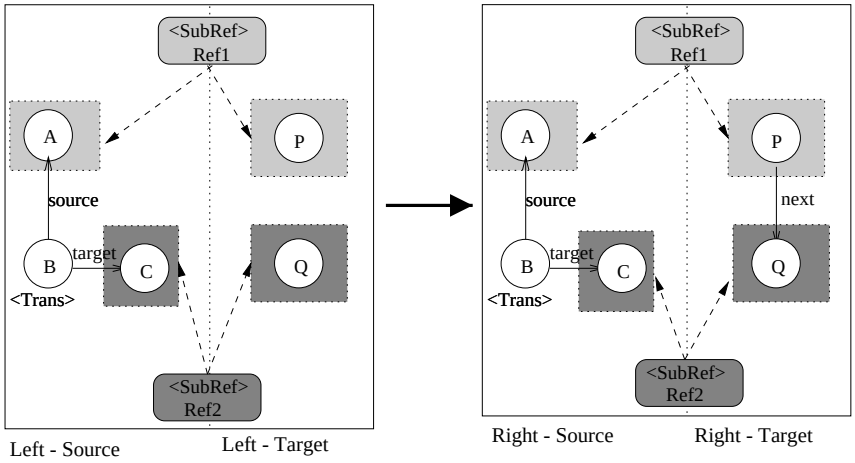


Figure 5.15: Rule of transition (RuleTrans)

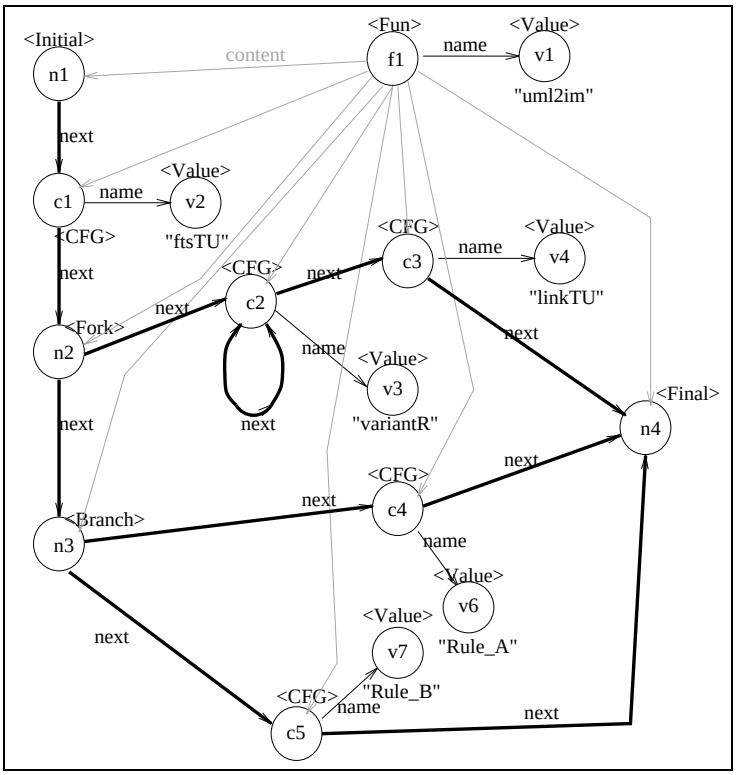


Figure 5.16: RuleTrans results

RuleLoop As shown in Table 3.2, transitions can be triggered by a `SignalEvent`. `Empty trigger` denotes the *sequential* execution of the `source` and `target` vertex, while "`!`" denotes the *as long as possible* execution, which is related to self–transitions.

As the default control flow element is sequential execution, the self–transition is transformed to a ***LoopNode*** notion instead of ***CFGNode*** and ***loop*** edge instead of `next` edge, according to the naming of the programming element (see in Fig. 5.17).

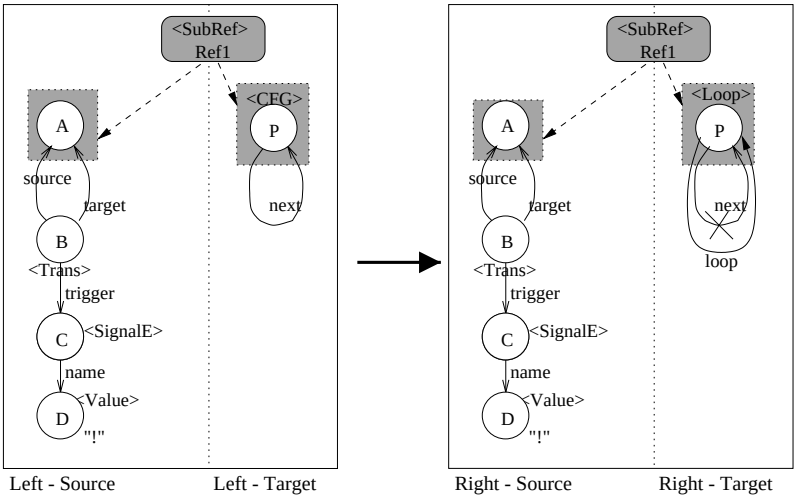


Figure 5.17: Rule of self–transition (RuleLoop)

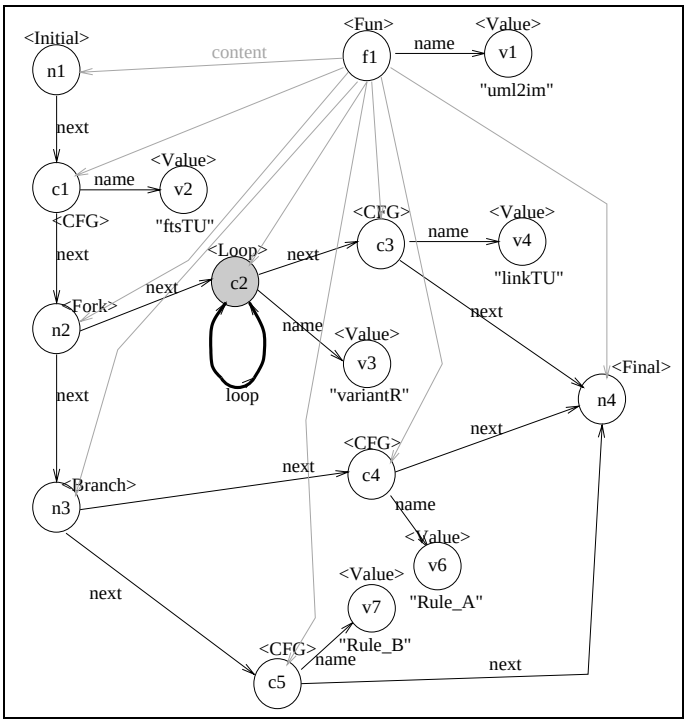


Figure 5.18: RuleLoop results

Transformation of branches The if–then–else structure is represented in UML statecharts graphs by `guarded` transitions, i.e. values are connected to `Transitions` by `Guard` and `Value` nodes. In case of conditional threads, a logical expression `Cond` is held by the Value node, in other case Value node holds `"Else"`.

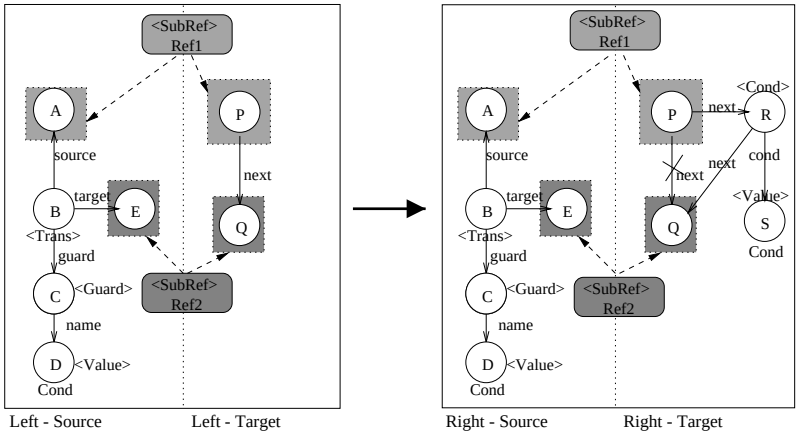


Figure 5.19: Rule of conditions (RuleCond)

Rules of guarded transitions The guarded transitions are already transformed into a ***next*** edge in CFG model between the transformed source and target nodes (denoted by SubRef). During the application of the related rules (represented in Fig. 5.19 and 5.21),

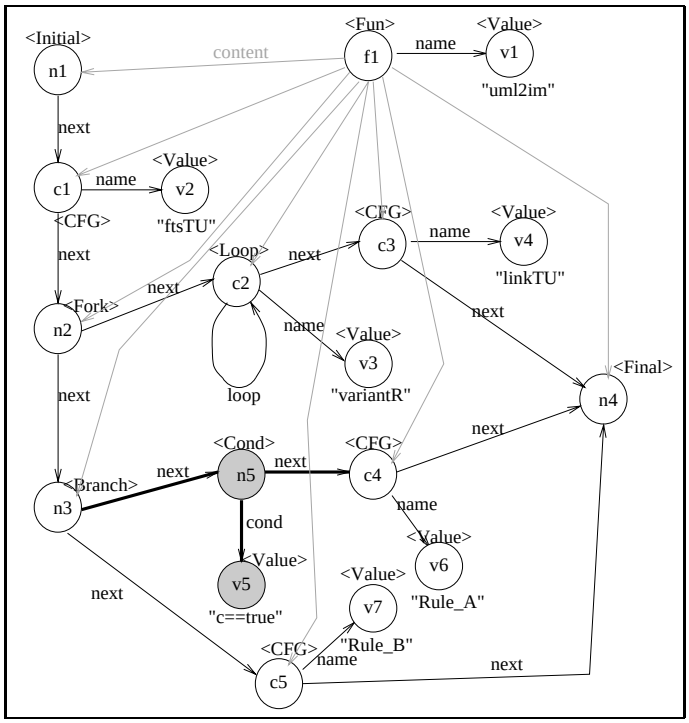


Figure 5.20: RuleCond results

the subsequent node (denoted by `next` in the left target side edge) will be reachable only through a ***Condition*** or an ***Else*** node by ***next*** edges, and ***Cond***ition node get a ***Value*** node which holds the logical expressions ***Cond***.

Please note that a Branch may have several conditional threads (at least one) and at most one Else thread.

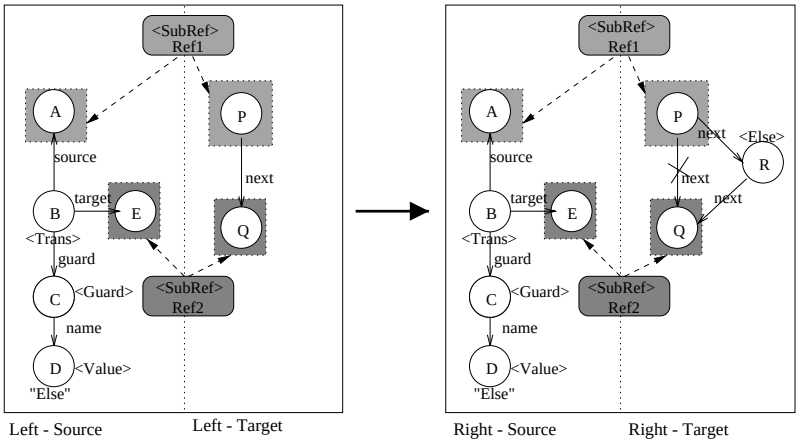


Figure 5.21: Rule of else structure (RuleElse)

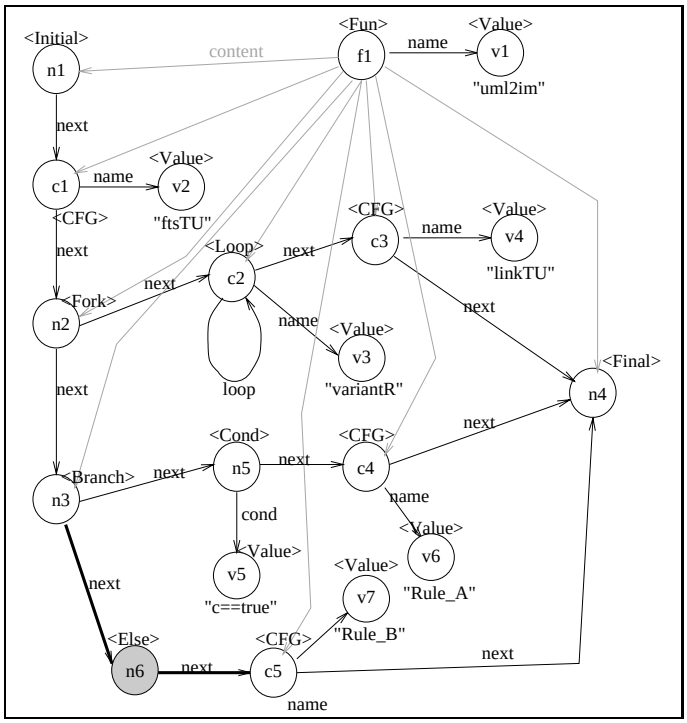


Figure 5.22: RuleElse results

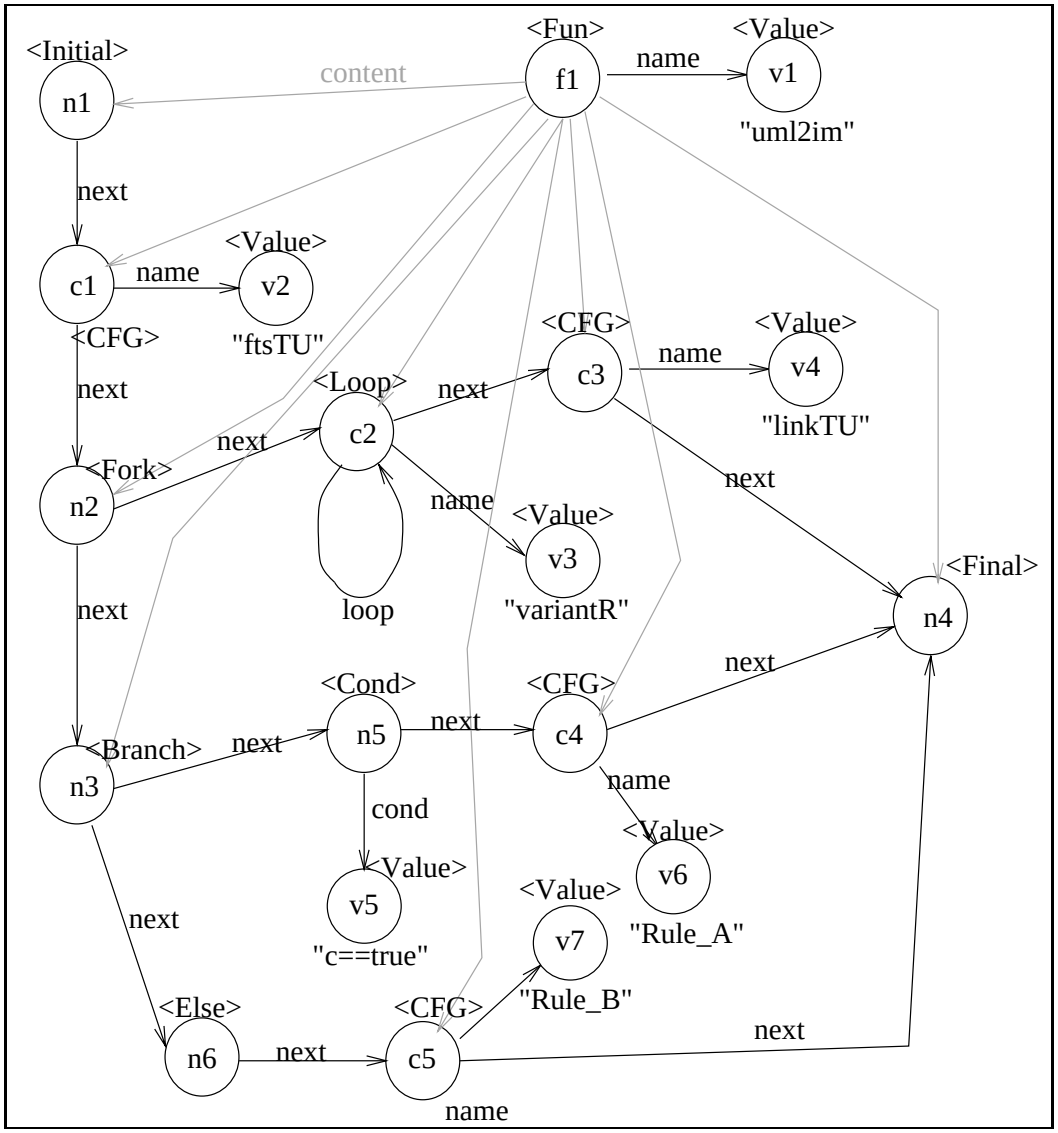


Figure 5.23: The result of the transformation from the statechart graph of Control Flow

Chapter 6

From Control Flow Graphs to Prolog Code Models

The second step of the transformation from UML statecharts to a model of executable Prolog code is performed from CFG models to the Prolog code model. The structure of the intermediate model (CFG model) may serve as a basis for the model of any programming languages as containing all the abstract programming constructs of control flow (see Section 5.1).

This second phase, i.e. the final Prolog graph of the transformation supports the implementation (code–writing) of transformation's control flow in the target programming language. In our case, Prolog was chosen for this target language, which is introduced in the sequel according to [11].

6.1 A Short Introduction to Prolog

Prolog is a simple but powerful programming language developed at the University of Marseilles [23], as a practical tool for programming in logic [12]. From a user's point of view the major attraction of the language is ease of programming. Clear, readable, concise programs can be written quickly with few errors.

6.1.1 Informal Introduction to Prolog Programs

A fundamental unit of a logic program is the goal or procedure call. e.g. `gives(tom, apple, teacher)` `reverse([1,2,3], L)` `X<Y`.

A goal is merely a special kind of term, distinguished only by the context in which it appears in the program. The (principal) functor of a goal identifies what predicate the goal is for. It corresponds roughly to a verb in natural language, or to a procedure name in a conventional programming language.

A logic program consists simply of a sequence of statements called sentences, which are analogous to sentences of natural language. A sentence comprises a head and a body. The head either consists of a single goal or is empty. The body consists of a sequence of zero or more goals (i.e. it too may be empty). If the head is non–empty, the sentence is called a clause.

If the body of a clause is empty, the clause is called a unit clause, and is written in the form

```
P.
```

where P is the head goal. We interpret this declaratively as *"Goals matching P are true"* and procedurally as *"Goals matching P are satisfied."*

If the body of a clause is non-empty, the clause is called a non-unit clause, and is written in the form

```
P :- Q, R, S.
```

where `P` is the head goal and `Q`, `R` and `S` are the goals which make up the body. We can read such a clause either declaratively as *"P is true if Q and R and S are true"*, or procedurally as *"To satisfy goal P, satisfy goals Q, R and S."*

A sentence with an empty head is called a directive , of which the most important kind is called a query and is written in the form

```
?- P, Q.
```

where P and Q are the goals of the body. Such a query is read declaratively as *"Are P and Q true?"* and procedurally as *"Satisfy goals P and Q."*

Sentences generally contain variables. Note that *variables in different sentences are completely independent*, even if they have the same name, i.e. the lexical scope of a variable is limited to a single sentence. Each distinct variable in a sentence should be interpreted as standing for an arbitrary entity, or value. To illustrate this, here are some examples of sentences containing variables, with possible declarative and procedural readings:

1. `employed(X) :- employs(Y,X).`
   - *"Any X is employed if any Y employs X."*
   - *"To find whether a person X is employed, find whether any Y employs X."*
2. `derivative(X,X,1).`
   - *"For any X, the derivative of X with respect to X is 1."*
   - *"The goal of finding a derivative for the expression X with respect to X itself is satisfied by the result 1."*
3. `?- ungulate(X), aquatic(X).`
   - *"Is it true, for any X, that X is an ungulate and X is aquatic?"*
   - *"Find an X which is both an ungulate and aquatic."*

In any program, the predicate for a particular (principal) functor is the sequence of clauses in the program whose head goals have that principal functor. For example, the predicate for a 3-ary functor `concatenate/3` might well consist of the two clauses

```
concatenate([], L, L).
concatenate([X|L1], L2, [X|L3]) :- concatenate(L1, L2, L3).
```

where `concatenate(L1,L2,L3)` means *"the list `L1` concatenated with the list `L2` is the list `L3`"*. Note that for predicates with clauses corresponding to a base case and a recursive case, the preferred style is to write the base case clause first.

In Prolog, several predicates may have the same name but different arities. Therefore, when it is important to specify a predicate unambiguously, the form name/arity is used; e.g. `concatenate/3`.

6.1.2 Procedural Semantics of Prolog

The procedural semantics of Prolog defines exactly how the Prolog system will execute a goal, and the sequencing information is the means by which the Prolog programmer directs the system to execute the program in a sensible way. The effect of executing a goal is to enumerate, one by one, its true instances. Here then is an informal definition of the procedural semantics. The semantics is illustrated by the simple query

```
?- concatenate(X, Y, [a,b]).
```

We find that *it matches the head of the first clause* for `concatenate/3`, with `X` instantiated to `[a|X1]`. The new variable `X1` is constrained by the new query produced, which contains a single recursive procedure call:

```
?- concatenate(X1, Y, [b]).
```

Again this goal matches the first clause, instantiating `X1` to `[b|X2]`, and yielding the new query:

```
?- concatenate(X2, Y, [])
```

Now the single goal will only match the second clause, instantiating both `X2` and `Y` to `[]`. Since there are no further goals to be executed, we have a solution

```
X = [a,b]
Y = []
```

i.e. a true instance of the original goal is

```
concatenate([a,b], [], [a,b])
```

If this solution is rejected, backtracking will generate the further solutions

```
X = [a] Y = [b]
X = [] Y = [a,b]
```

in that order, by re-matching, against the second clause for concatenate, goals already solved once using the first clause.

Thus, in the procedural semantics, the set of clauses

```
H :- B1, ..., Bm.
H' :- B1', ..., Bm'.
...
```

are regarded as a procedure definition for some predicate `H`, and in a query

```
?- G1, ..., Gn.
```

each `Gi` is regarded as a procedure call. To execute a query, the system selects a goal (e.g. `Gj`) by its computation rule, and searches a clause whose head matches `Gj` by its search rule. Matching is done by the unification algorithm (see [22]) which computes the **most general unifier (mgu)**, of `Gj` and `H`. The mgu is unique if it exists. If a match is found, the current query is reduced to a new query

```
?- (G1, ..., Gj-1, B1, ..., Bm, Gj+1, ..., Gn)mgu.
```

and a new cycle is started. The execution terminates when the empty query has been produced.

If there is no matching head for a goal, the execution backtracks to the most recent successful match in an attempt to find an alternative match. If such a match is found, an alternative new query is produced, and a new cycle is started.

6.1.3 Control Restrictions in Prolog

This section summarizes some basic control restrictions in Prolog, which is widely used during the automatic Prolog code generation for model transformation systems.

Non–determinism in Prolog (Fork structures) Non–deterministic choices (i.e. disjunction) are implemented by extra predicates with the same head name but distinct bodies. Each predicate can be executed sequentially by backtracking or parallelly in special Prolog systems.

Example 6.1.1 *The following example can be read as* "For any X, Y and Z, X has Z as a grandfather if Y is the parent of X (either a father or a mother), and the father of Y is Z."

```
grandfather(X,Z) :-
  parent(X,Y),
  father(Y,Z).
parent(X,Y) :-
  mother(X,Y).
parent(X,Y) :-
  father(X,Y).
```

The Cut Symbol Besides the sequencing of goals and clauses, Prolog provides another important facility for specifying control information, which is the cut symbol, written `!`.

The effect of the cut symbol is as follows. When first encountered as a goal, cut succeeds immediately. If backtracking should later return to the cut, the effect is to fail the parent goal, i.e. that goal which matched the head of the clause containing the cut, and caused the clause to be activated. In other words, the cut operation commits the system to all choices made since the parent goal was invoked, and causes other alternatives to be discarded.

The goals handled deterministically are the parent goal itself, any goals occurring before the cut in the clause containing the cut, and any subgoals which were executed during the execution of those preceding goals.

Example 6.1.2 *In this example (which computes the absolute value Y of X), the call for* `abs/2` *will be executed exactly once.*

```
abs(X, Y):-
  X >= 0, !,
  Y is X.
abs(X, Y):-
  X < 0, !,
  Y is -1*X.
```

If–Then–Else If–then–else statements take the form of

```
Head :- If1, !, Then1.
...
Head :- Ifm, !, Thenm.
Head :- Elsen.
```

i.e. they contain cuts and disjunctions. In this sense, if an `Ifi` goal succeeds, the non–deterministic choice point placed on the `Head` predicate is removed.

As long as possible (Loops) In exhaustive searches through an entire state space or in generate–and–trial type algorithms, all the data elements fulfilling special requirements have to be enumerated one by one (and e.g printed as a side effect).

This can be achieved in Prolog by a call of the built–in predicate `fail`, which never succeeds when set up as a goal. As a result, the control flow backtracks to the last choice point to choose another match of predicates and instantiation of variables.

Example 6.1.3 *This `young_women` predicate succeeds when all the women in the database, whose age is less than 25 are printed (supposing that each woman has exactly one age).*

```
young_women:-
  woman(X),
  age(X,Age),
  Age < 25,
  write(X),
  fail.
young_women.
```

6.2 Model Transformation to the Prolog Code Model

The metamodel of the Prolog code model is resulted from the implementation of programming elements in Prolog. The Prolog skeleton of the control elements (discussed in Section 6.1) serves as a basis for understanding the main structure of the MOF metamodel.

6.2.1 The MOF Metamodel of the Prolog Code Model

These previous implementation structures are defined by the MOF metamodel of the Prolog code model (depicted in Figure 6.1) in the following way.

The concept of the Prolog Code model The transformation from CFG model must result in an instance of the target Prolog Code model, from which the description the corresponding Prolog code is easy to be generated. Such Prolog programs can be described by sequences of predicate calls referred by a name (which can be in our case either names of rules and transformation units, as well as generated predicate names referring to control structures) and built–in syntactical constructs represented by pure textual elements (between quotation marks), like ':-', '!', 'fail', 'nl' (newline), ',', and '.'.

These elements in the metamodel are implemented in the following way.

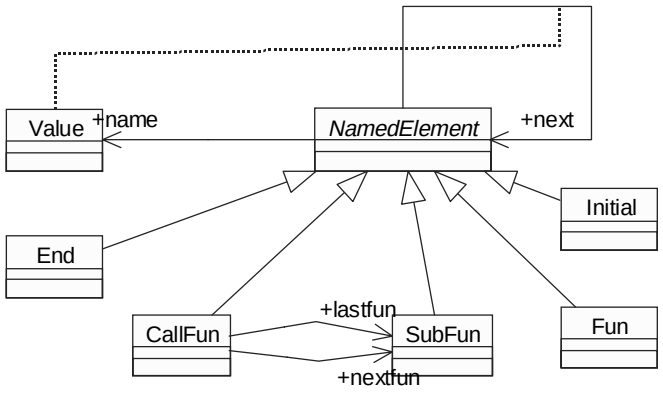


Figure 6.1: The MOF metamodel of the Prolog code model

- `NamedElement` can be `Fun`, `Initial`, `CallFun`, `SubFun` and `End`. Each NamedElement has a name connecting to a `Value` node with a `name` edge. These names can be predicates, clauses or textual elements.
- `Fun` nodes denote the functions, i.e. the heads of predicates.
- `Value` node is valueholder. It can be either names of `NamedElements` (connected by `name` edges) or a textual element (connected by `next` edges) of Prolog code.
- A `next` edge may have a `value`, which represents the concatenation of textual elements ',' (comma) and 'nl' (newline) to separate predicates and clauses in the sequential execution.
- `End` node denotes the final point of functions. This meaning is not denoted in the class diagram: `End` nodes have only incoming `next` edges.
- Each element of model CFG is transformed to a `CallFun` node (except for `Value` nodes) to emphasize the structure of Prolog programs that all the names refers to a function call.
- In order to generate a code automatically, the representation of branches, forks and loops requires the help of some additional nodes, such as `SubFun` and `nextfun`, `lastfun` edges, which coordinate the execution order among the predicates with the same head (like in branches, forks and loops structure — see Section 6.1.3.
- The named elements are connected to each other by `next` edges.

The following notations are used in the transformation (the notation of CFG model instances can be seen in Section 5.2):

- The instance (target graph) of Prolog code model is empty at the beginning.
- Each node has an **ID placed in the centre of the node**.
- Some abbreviation are used for the notation in the target model.

- The **type of nodes** (the name of the comprising class) is printed in <> beginning with capital letters.
- The **type of edges** is printed in non–capital letters placed **above the arrows**.
- The connected **value**s to **next** edges are placed below the arrows.
- The **value** (usually a string or an integer) of a <Value> node is placed **outside**.
- The notation of the CFG model are written with `typewriter letters`, the letters of notation of the target Prolog code model are referred by ***bold italic letters***, and types of reference nodes between the two models are typed with sans serif fonts.

In order to get a correct Prolog Code from model CFG, RefIn and RefOut references between the models are used to assign the corresponding nodes to denote the direction of the execution. RefIn refers to the incoming next edges, while RefOut nodes determine the place of the outgoing next edges.

6.3 The Process of the Transformation

All the described rules between the two models are used by the transformation. The steps of the transformation are carried out in the introduced order of rules (each rule must fulfil the "as long as possible" semantics of rule applications).

In order to help tracing the process of the transformation, an instance Prolog Code model (transformed from an instance of CFG model, depicted in Figure 5.22) graph is provided after each transformation rule illustrating the current state of the transformation emphasizing the effects of the latest rule by gray scale nodes and bold edges. In these graphs, the connected values to the nodes are next to the node for the easier reading and to reduce the complexity of the illustration. The Prolog implementation of these transformation rules are listed in Appendix A.2.

6.3.1 Control Structure Rules

The nodes rules are the following, applied step by step to the instance of CFG model in Figure 5.23 .

PCRuleFun (depicted in Fig. 6.2) The meaning of `Fun` node representation in CFG model is equivalent to the concept of ***Fun*** node in the final model, therefore the application of the rule results in the same notation (the rule transforms `Fun` node and its `name` to a ***Fun*** node, a ***name*** edge and ***Value*** node with the same name value).

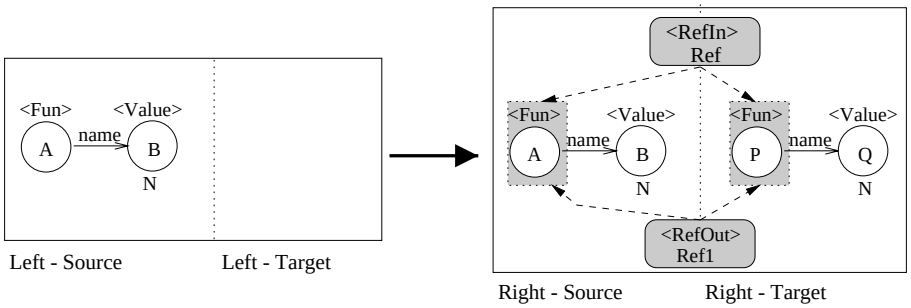


Figure 6.2: Rule on Fun nodes (PCRuleFun)

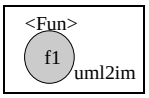


Figure 6.3: PCRuleFun results

PCRuleCFG As the concept of functions does not change, the rule of CFG nodes (depicted in Fig. 6.4) retains CFG node and its name in an unaltered form.

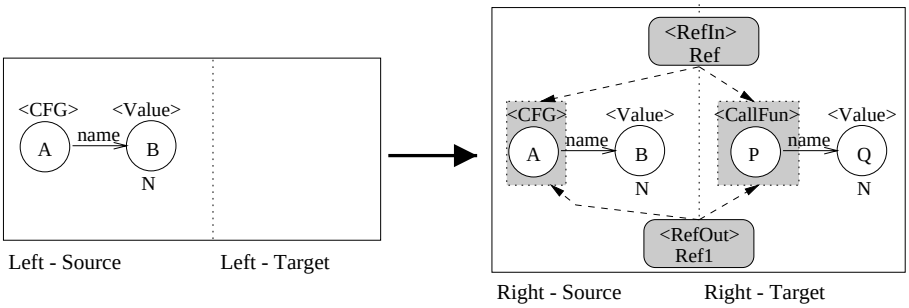


Figure 6.4: Rule of CFG node (PCRuleCFG)

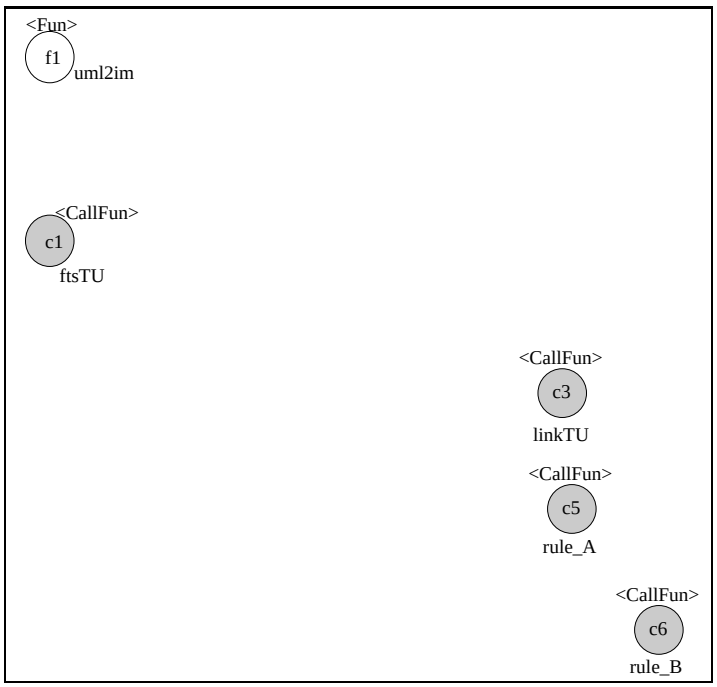


Figure 6.5: PCRuleCFG results

PCRuleBranchIn A `Branch` node may have several conditional and else threads. According to our control flow graph model, these threads can continue through completely different nodes, i.e. the execution determined by conditions must branch.

In order to get a correct code of branches in Prolog, the described structure of if–then–else is created (see Section 6.1.2) by the introduction of novel nodes, ***CallFun*** with a generated name ***genname(N)*** of the comprising `Fun`. This correspondence between Branch node and ***CallFun*** determine a RefIn reference, i.e. the node in model CFG, which has the Branch node as its subsequent node, its corresponding node in Prolog Code model must have ***CallFun*** as its subsequent node. As `Branch` may have several threads, following the described structure of if–then–else in Prolog, the execution is ended by the call of the generated function (represented with the ***End*** node).

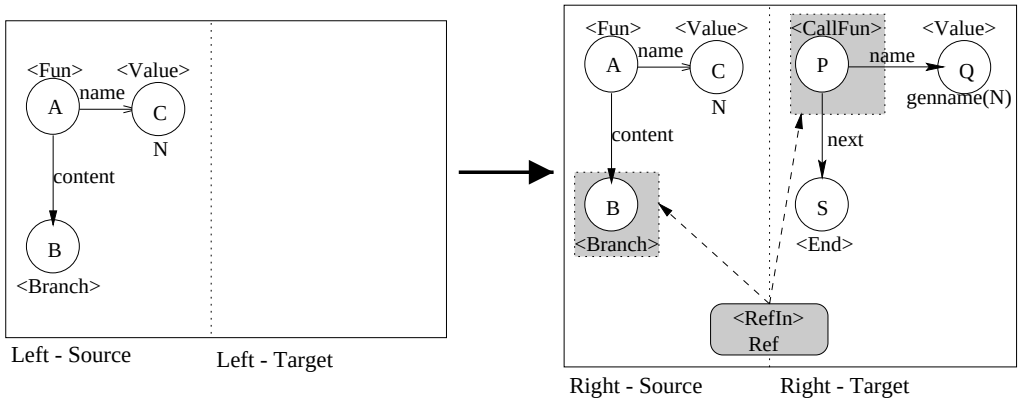


Figure 6.6: Rule of Branch node (PCRuleBranchIn)

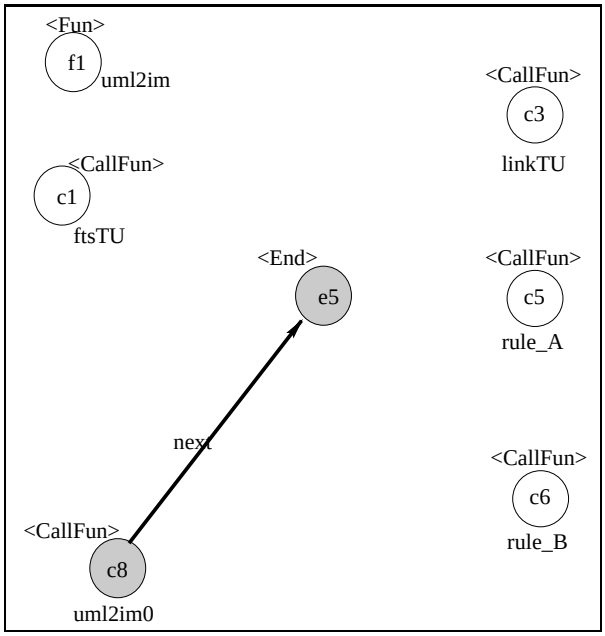


Figure 6.7: PCRuleBranchIn results

PCRuleForkIn The structure of forks is similar to branches': several parallel threads can lead from a fork but the order of their execution is arbitrary (and they have no condition attached). Thus the transformation of forks is carried out by the separation of these possible threads to different functions with the same name. The first step of this procedure is the creation an end (with an ***End*** node) of the thread after the call of the new function by ***CallFun*** node with a generated ***name*** from the containing `Fun` (see Fig. 6.8). To ensure the appropriate connections between the nodes, RefIn reference is used for the same purpose as in case of branches.

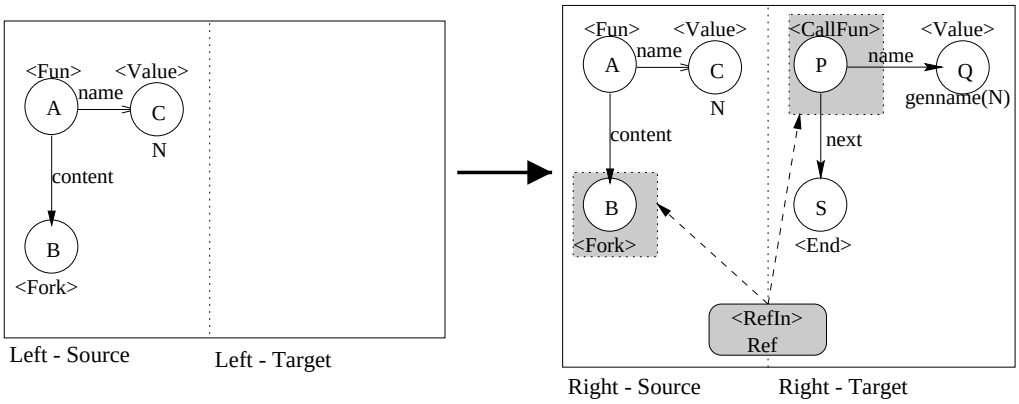


Figure 6.8: Rule of Fork node (PCRuleForkIn)

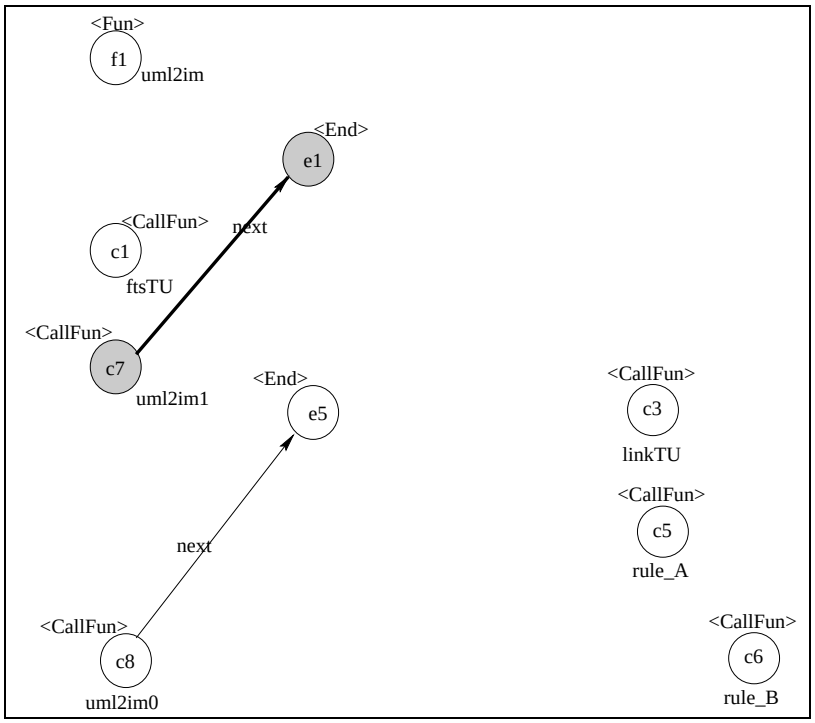


Figure 6.9: PCRuleForkIn results

PCRuleLoop Loops are implemented in CFG model as a `LoopNode` with a `loop` edge. In Prolog, loops are processed as function calls. In order to perform this requirement, generated names (from the `name` of the LoopNode) are assigned to loops and these functions (***CallFun*** nodes) are called in the appropriate time (depicted in Fig. 6.10). The RefIn and RefOut references are used to transform the connected incoming and outgoing `next` edges to the appropriate incoming and outgoing ***next*** edges between the corresponding nodes, while RefLoop serves the correspondence between the `LoopNode` and ***CallFun***.

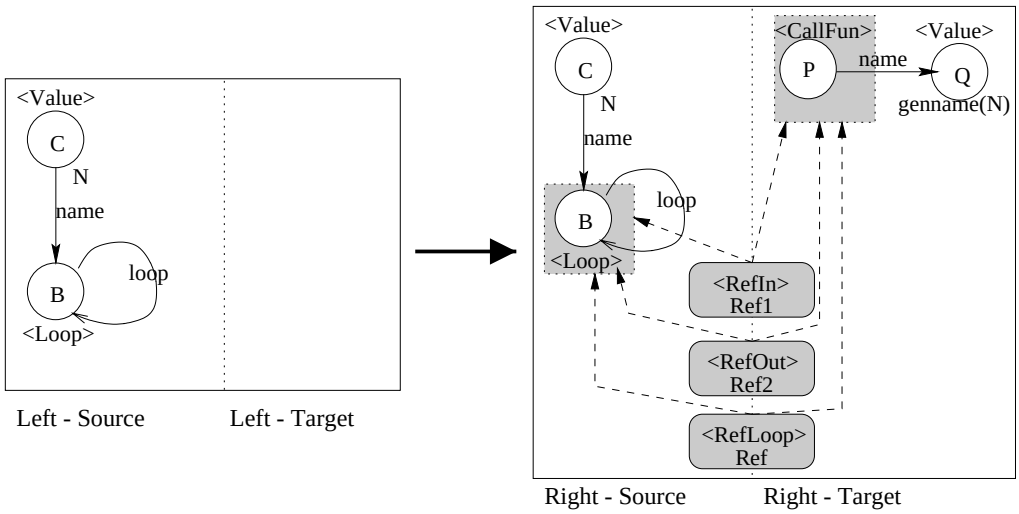


Figure 6.10: Rule of LoopNode (PCRuleLoop)

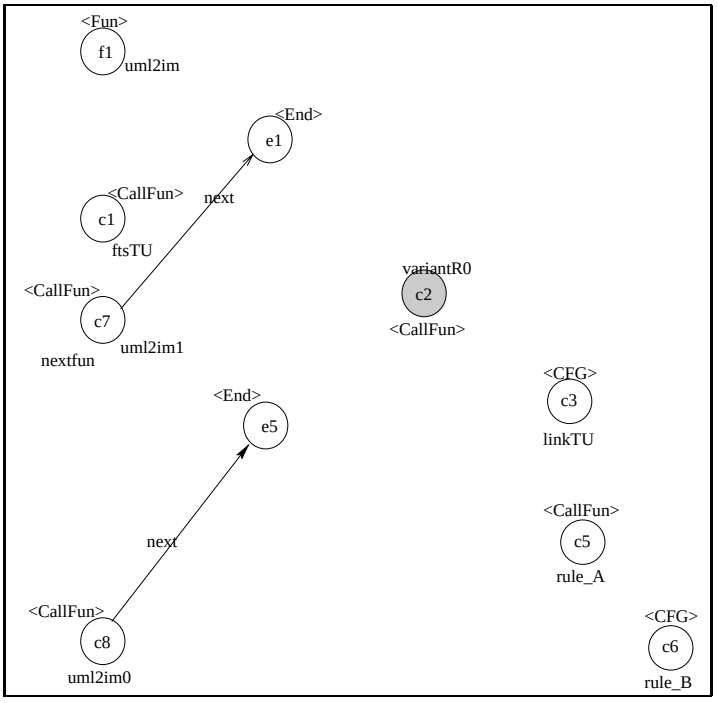


Figure 6.11: PCRuleLoop results

PCRuleLoopFirst (depicted in Fig. 6.12) According to the structure of loops in Prolog, the referring rules represent two functions with the same name (the ***name*** of the appropriate ***CallFun***, determined by RefLoop reference). The first function ***SubFun***, connected by ***nextfun*** edge to the referred ***CallFun*** node, contains the seed of the loop: the ***CallFun*** node with the name of LoopNode name and the built–in syntactical element ***"fail"*** (connected by a ***name*** edge and a ***Value*** node, which holds the name of the rule or transformation unit in the loop and the textual elements of Prolog) ensures the successful execution of the loop.

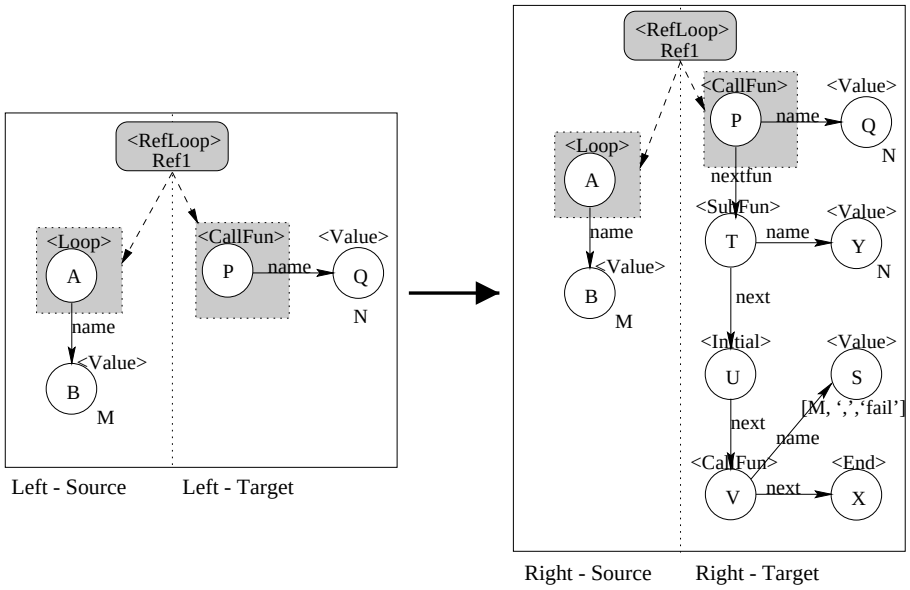


Figure 6.12: Rule of loop to create the first function (PCRuleLoopFirst)

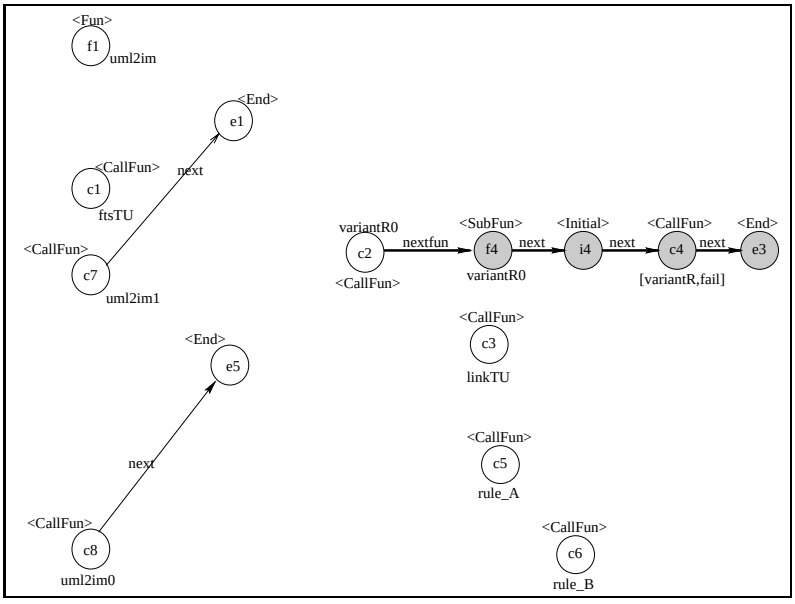


Figure 6.13: PCRuleLoopFirst results

PCRuleLoopLast The second function (see the rule in Fig. 6.14), ***SubFun***, connected by ***lastfun*** edge to the referred ***CallFun*** node, performs the end of the loop by representing an empty predicate with the ***name*** of CallFun.

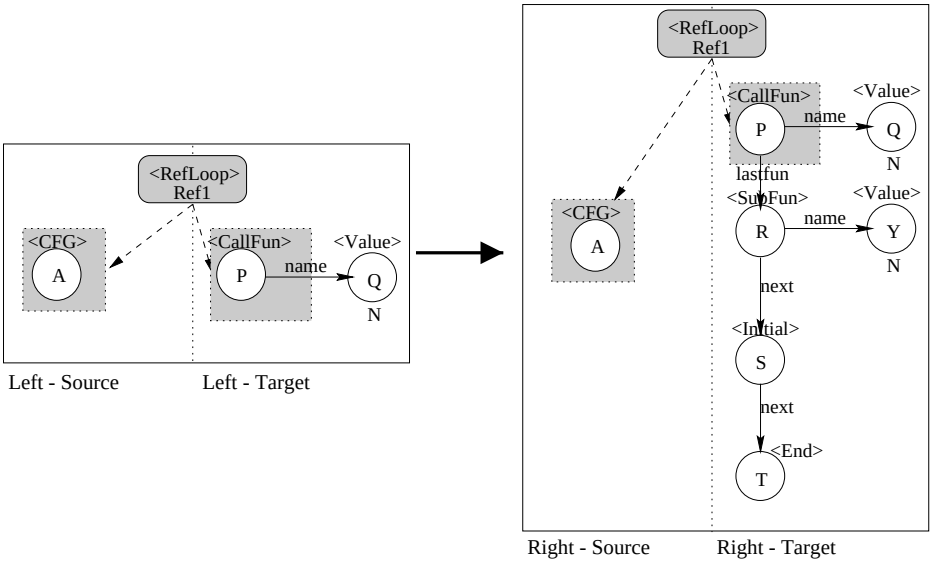


Figure 6.14: Rule of loop to create the last (second) function (PCRuleLoopLast)

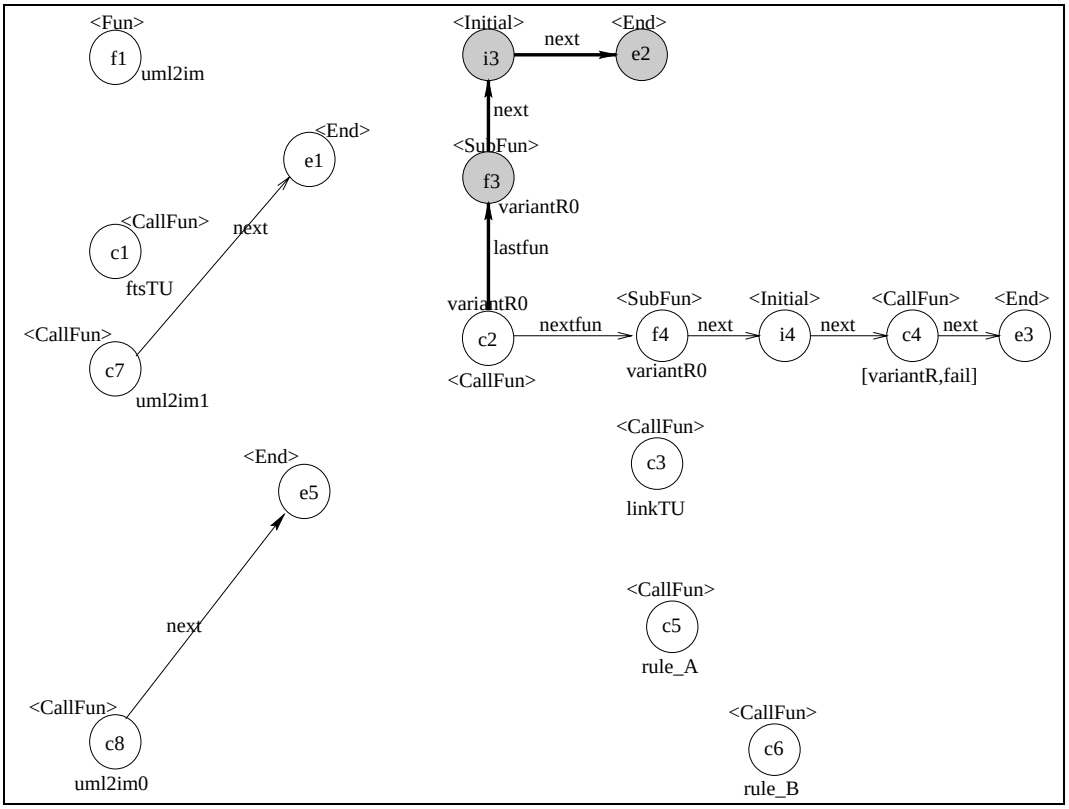


Figure 6.15: PCRuleLoopLast results

PCRuleFinal `Final` nodes denote the end point of a function in the source model, which is represented in our Prolog code model by an ***End*** node. The rule of Final nodes in Fig. 6.16 performs the conversion between `Final` and ***End*** nodes, and ensures the correspondence of the connected nodes (by next edges) by means of RefIn reference.

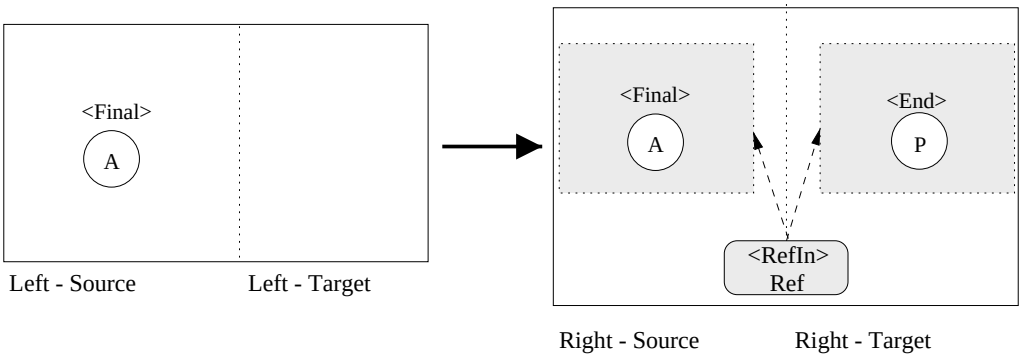


Figure 6.16: Rule of Final node (PCRuleFinal)

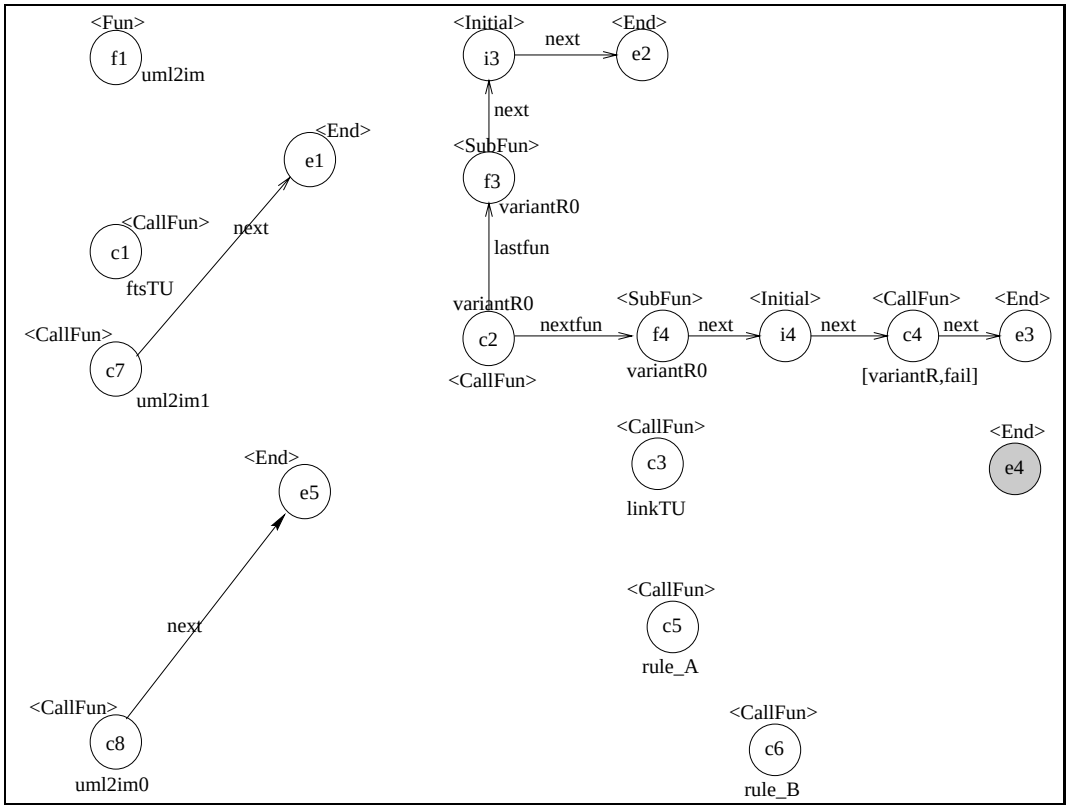


Figure 6.17: PCRuleFinal results

Rule of Initial nodes In the CFG model, `Initial` nodes are connected by `content` edges only to `Fun` nodes. The transformed `Initial` node represents the same functionality in Prolog Code model, i.e. it shows the starting point of the related function (connected by next edge to the corresponding Fun node). RefIn and RefOut references ensure further correspondence between the connected source and target nodes.

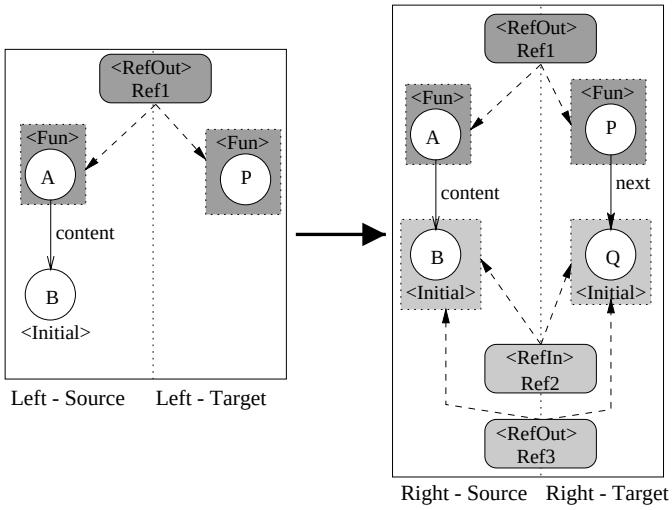


Figure 6.18: Rule of Initial node (PCRuleInitial)

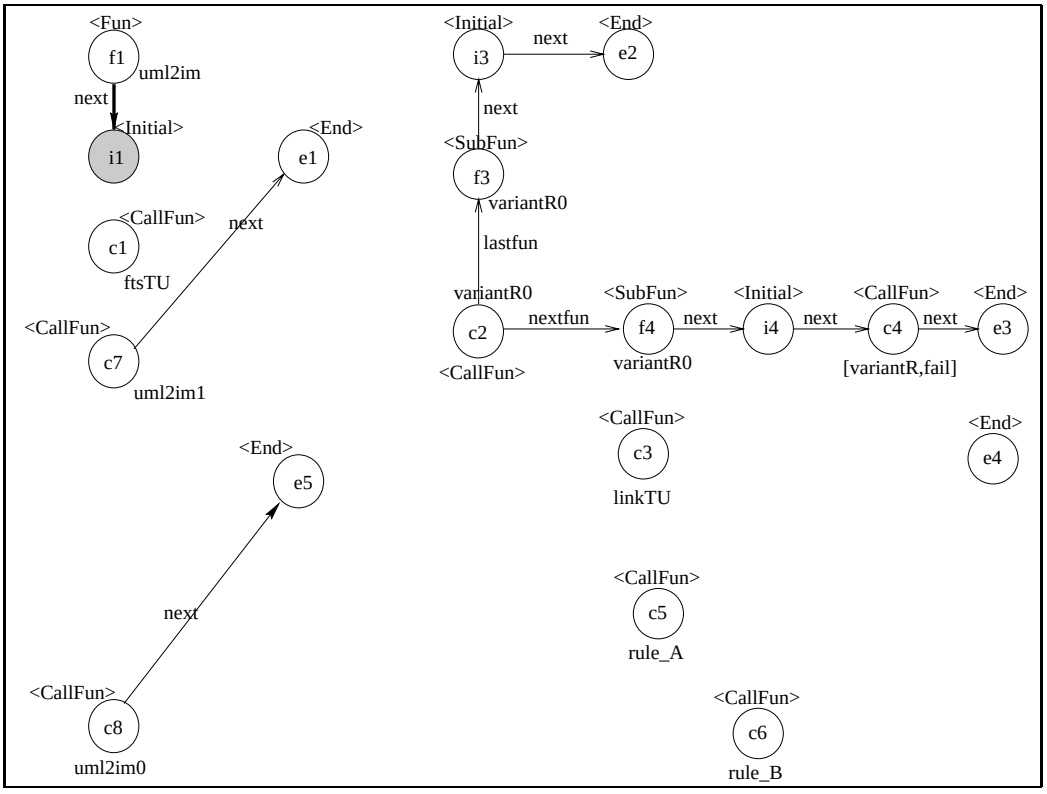


Figure 6.19: PCRuleInitial results

PCRuleCond The rule of `Cond` nodes with its condition `Cond` (held by the connected `Value` node) transforms them to a ***CallFun*** node, with the name ***Value***, which holds the condition and the corresponding built–in textual element of Prolog cut symbol (see if–then–else structure in Prolog, Section 6.1.3). The RefIn and RefOut serve for the correct process of the control flow, determining the place of the previous and subsequent edges.

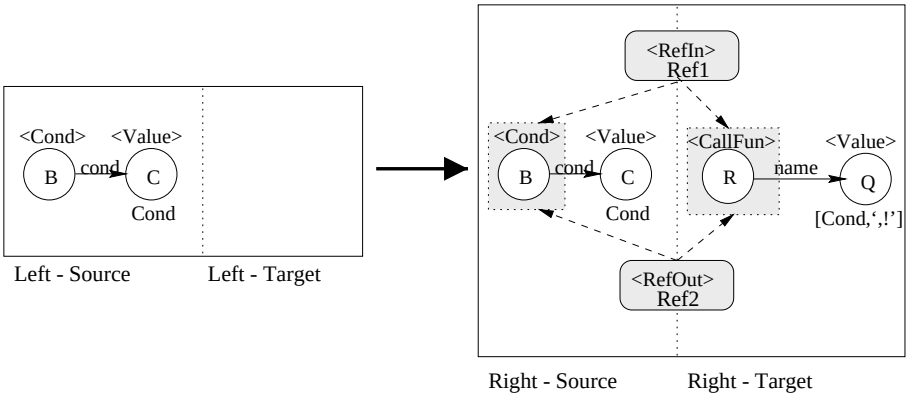


Figure 6.20: Rule of Cond node (PCRuleCond)

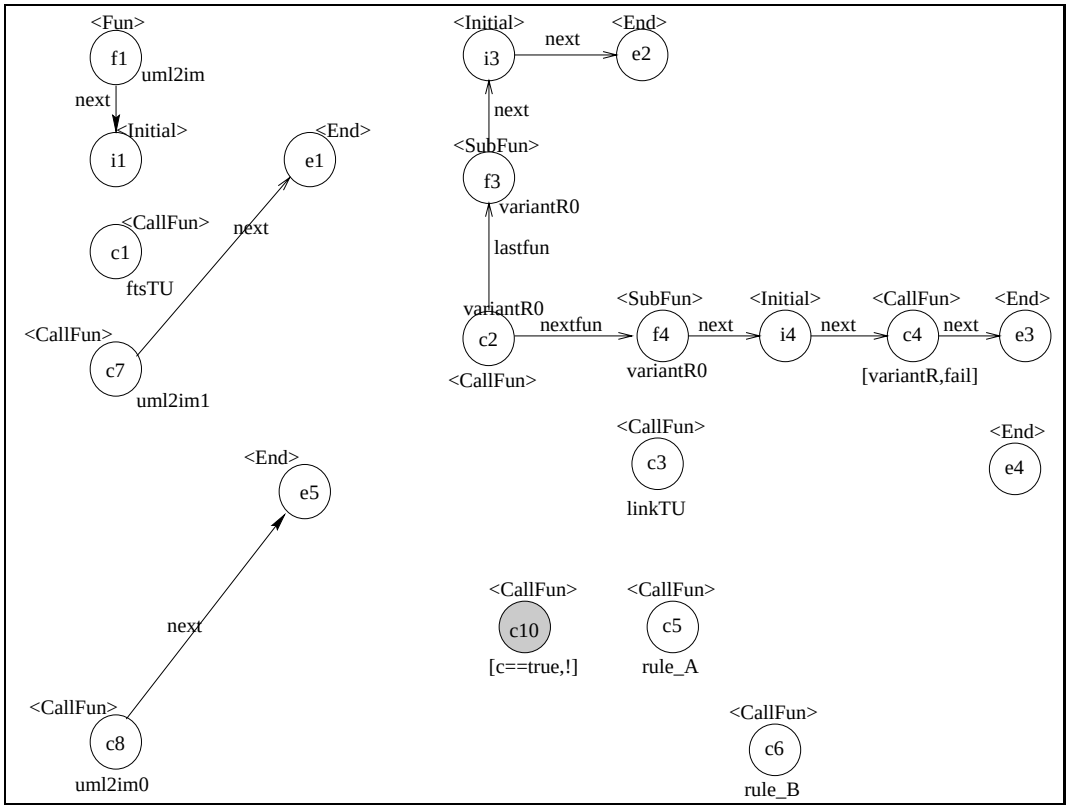


Figure 6.21: PCRuleCond results

PCRuleElse The rule of `Else` nodes transforms them to a ***CallFun*** node, with the name ***Value***, which holds the corresponding built–in textual element of Prolog cut symbol (see if–then–else structure in Prolog, Section 6.1.3). The RefIn and RefOut determine the correct process of the control flow.

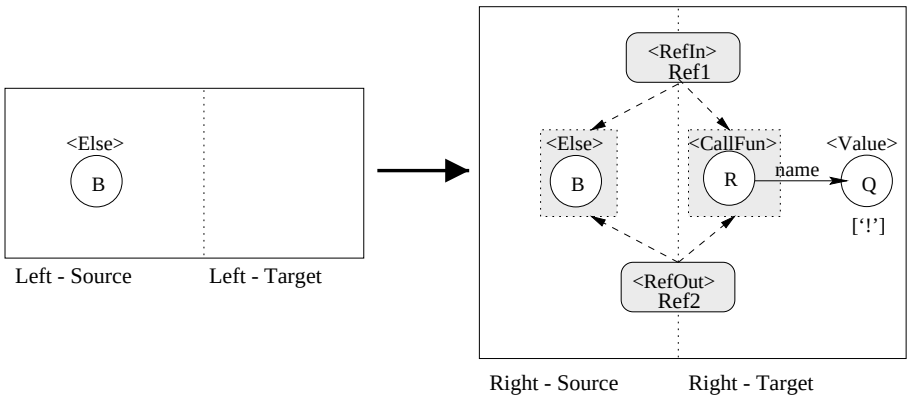


Figure 6.22: Rule of Else node (PCRuleElse)

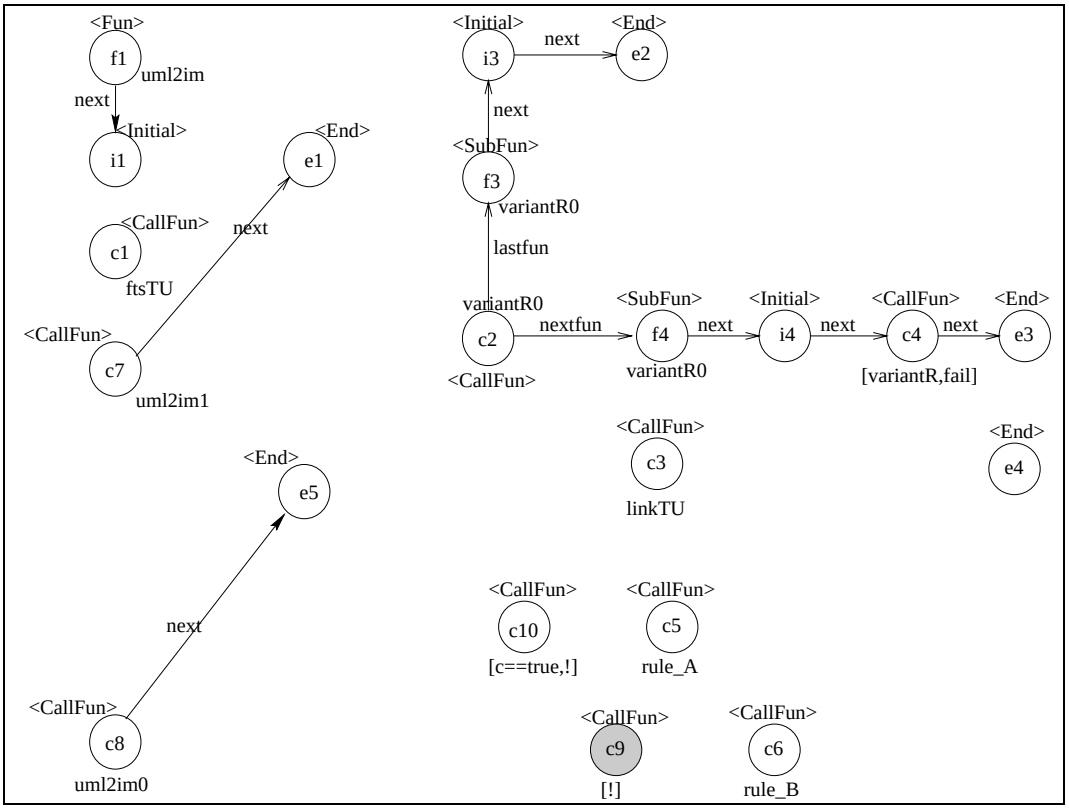


Figure 6.23: PCRuleElse results

PCRuleForkOut In order to process the outgoing threads of forks, the functions with the ***name*** of ***CallFuns*** are generated in the following way (in Rule 6.24): ***SubFun*** nodes are added to the final graph with ***nextfun*** edges. By means of `Fork` nodes and RefIn references of its subsequent nodes (each already transformed node is in RefIn reference), the corresponding nodes can be connected to each other through an ***Initial*** node which represents the beginning of ***SubFun***.

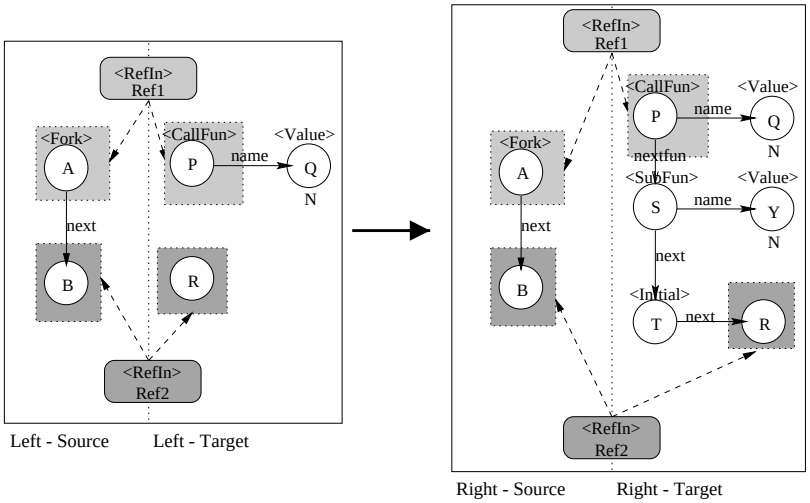


Figure 6.24: Rule of next edges to fork nodes (PCRuleForkOut)

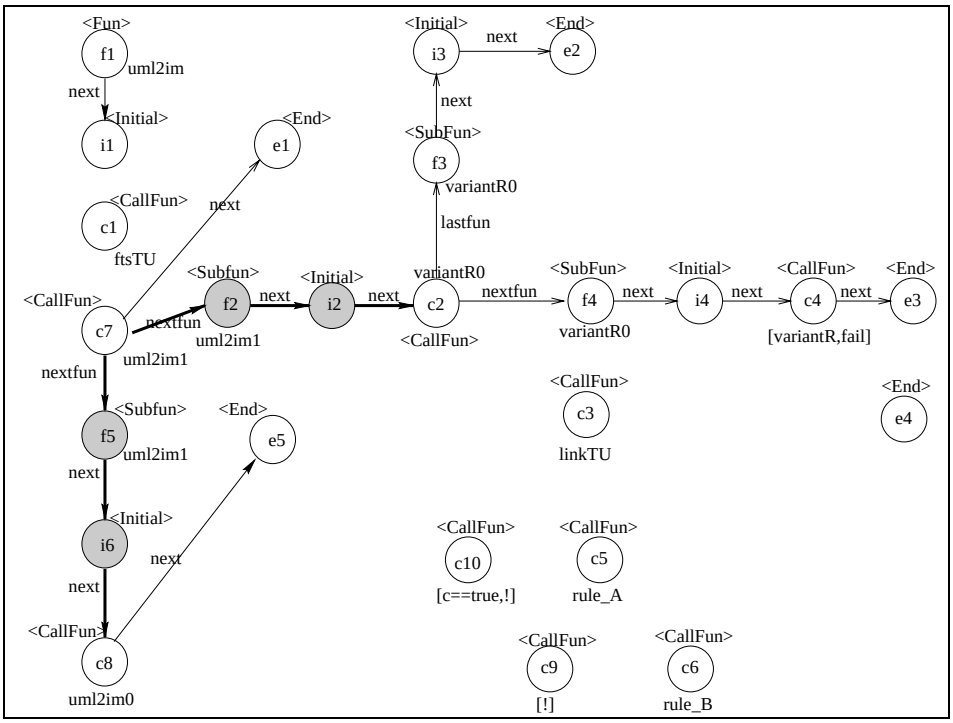


Figure 6.25: PCRuleForkOut results

PCRuleBranchOutCond As can be seen in Section 6.1.3, an if–then–else structure is implemented by separated predicates as functions. Conditional threads are transformed into ***SubFun*** nodes and ***Initial*** nodes (denoting the beginning of the function), connected by ***nextfun*** edges to the corresponding ***CallFun*** with the same ***name***. These rules ensure the correct place in the process of the outgoing next edges from the corresponding ***CallFun*** node.

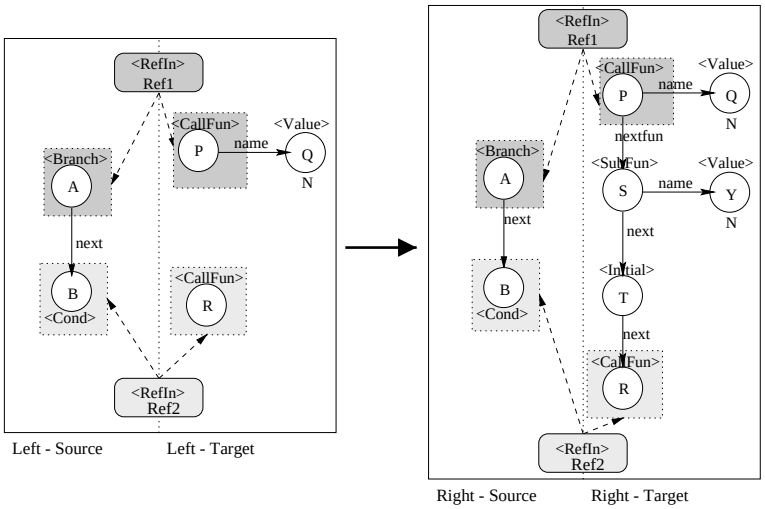


Figure 6.26: Rule assigning branches to conditional functions (PCRuleBranchOutCond)

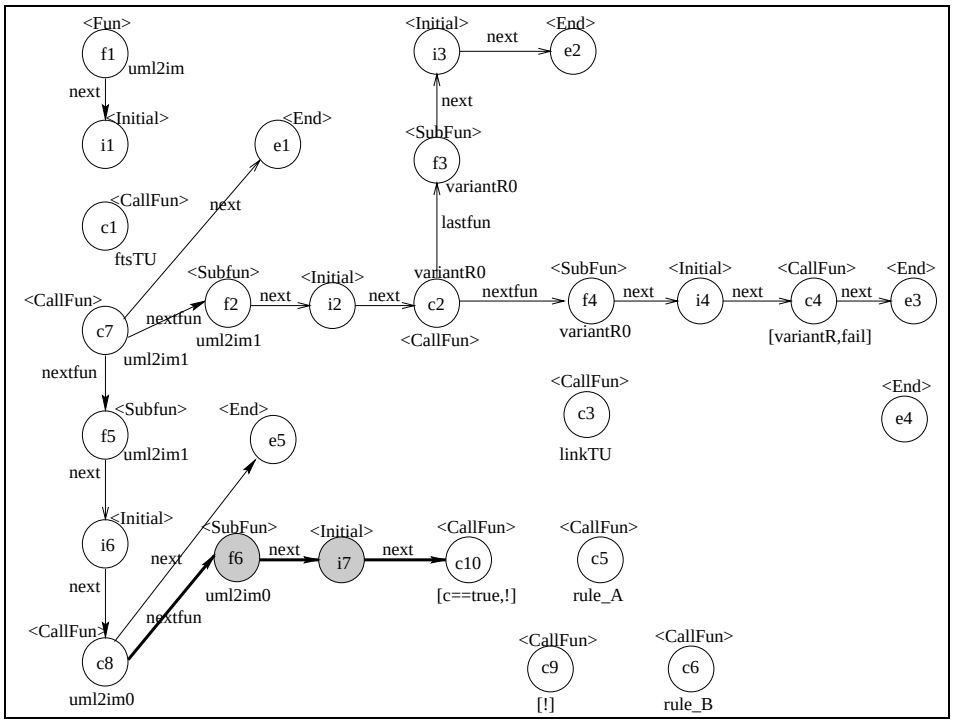


Figure 6.27: PCRuleBranchOutCond results

PCRuleBranchElseCond The else thread of branches is transformed in a similar way. According to the fact, that else thread does not have a condition, the corresponding function must be the last among the other function which are connected to the same branch (by ***CallFun*** and its ***name***). By the means of the ***lastfun*** edge, the algorithm generation can calculate it last. The rest notation of the rule mean the same as in the previous rule.

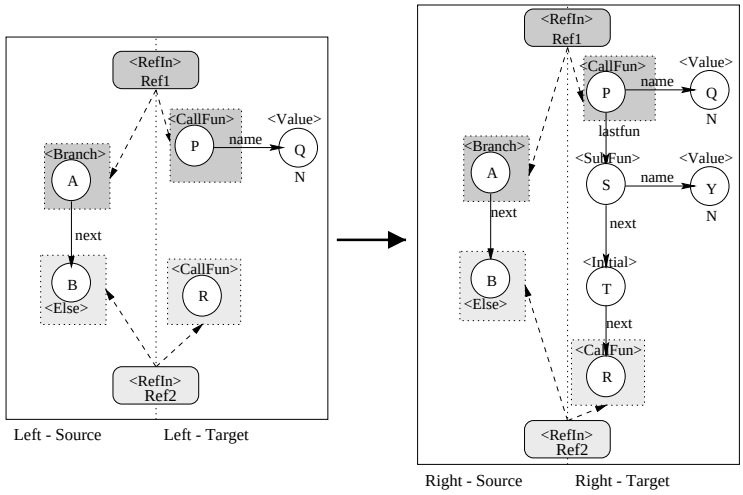


Figure 6.28: Rule assigning branches to its else function (pcRuleBranchOutElse)

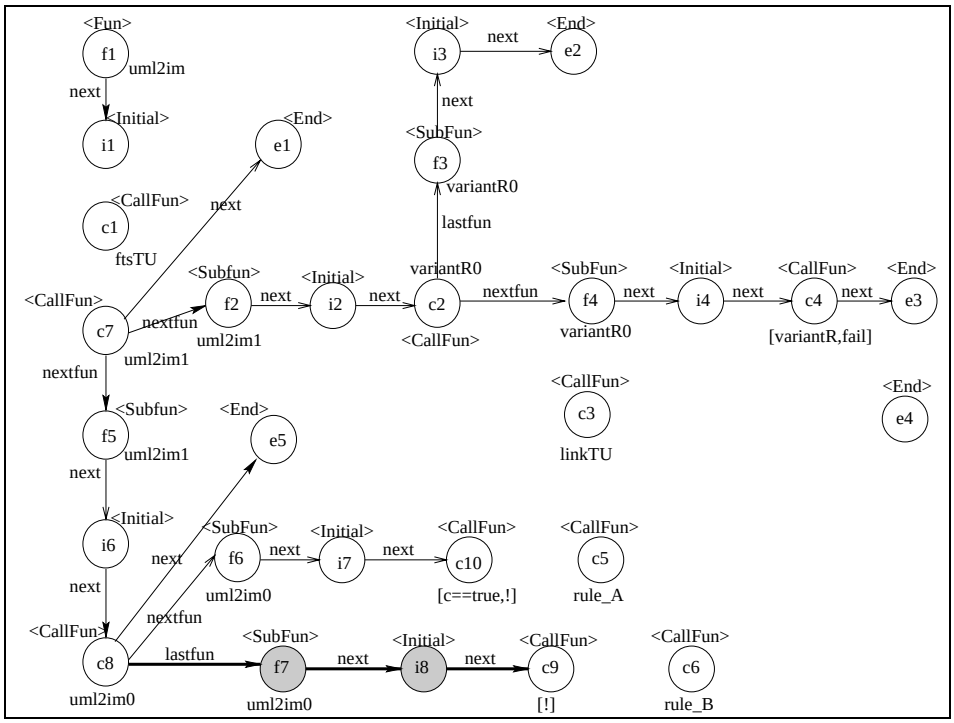


Figure 6.29: PCRuleBranchOutElse results

PCRuleNext (depicted in Fig. 6.30) To transform `next` edges between the corresponding (already transformed) nodes, RefIn and RefOut references are used, where *RefIn* references are between the nodes which previous nodes are in a correspondence, while RefOut reference assigns the nodes to the corresponding subsequent nodes. A `next` edge in model CFG between two nodes is transformed to a ***next*** edge in Prolog Code model connecting the referred nodes by the mentioned references.

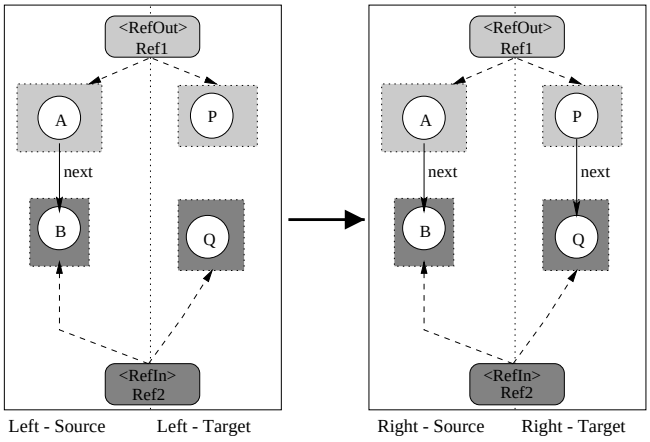


Figure 6.30: Rule of next edges (PCRuleNext)

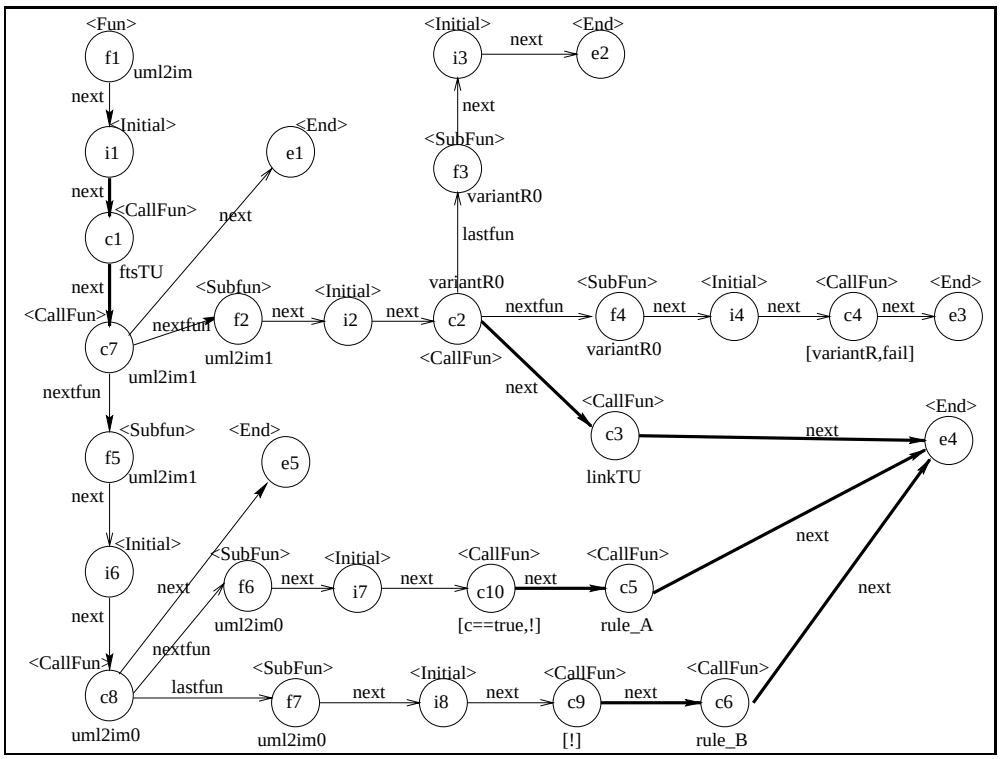


Figure 6.31: PCRuleNext results

6.3.2 Attaching Syntactic Elements

To support the code–writing, built–in textual elements of Prolog are connected by a name edge to a Value node, which holds these elements, as in case of some rules in the previous section (Section 6.3.1). In this section, textual elements are added to the instances of the target model (such as sequential order, the beginning and the end of functions) .

PCRuleDots The end of a function in Prolog is denoted by a dot. To represent the end of the functions, ***'.'*** is added to ***End*** nodes (and an additional ***'nl'*** newline to improve legibility).

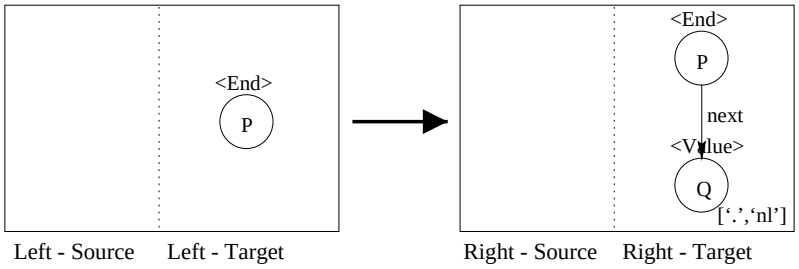


Figure 6.32: Punctuation for End of functions (PCRuleDots)

PCRuleComma The direction of the execution is represented in the final model by ***next*** edges. In Prolog, predicates are separated by commas. This rule adds a ***','*** and a ***'nl'*** (for the readability) value to each ***next*** edge in the target model.

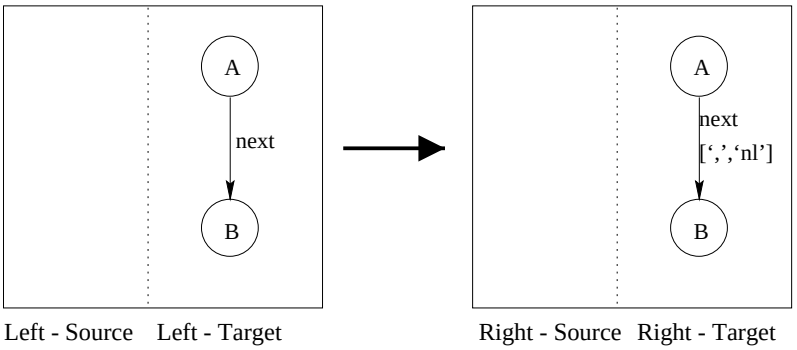


Figure 6.33: Punctuation for End of functions (PCRuleComma)

PCRuleInitialText In Prolog Code model, the starting point of functions are represented by ***Initial*** nodes. Prolog denotes the head of predicates (serve as functions) by the ***':-'*** symbol (and an additional 'nl' element).

PCRuleRemoveInitComma The previous rule, **PCRuleComma** added a ***','*** to each ***next*** edge, as well as the ***next*** edge from function nodes (Fun and SubFun nodes) to their ***Initial*** nodes (*':-,'* is not acceptable). In order to get an executable Prolog Code, these commas must be removed.

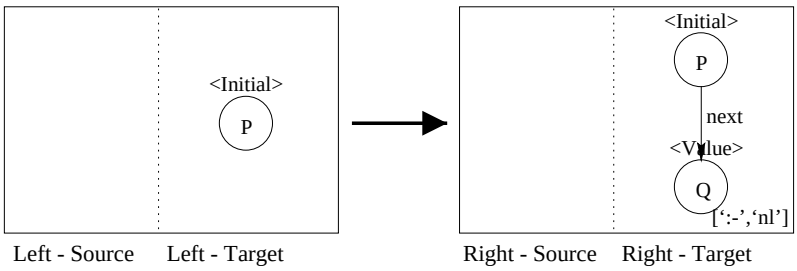


Figure 6.34: Punctuation for the beginning of functions (PCRuleInitialText)

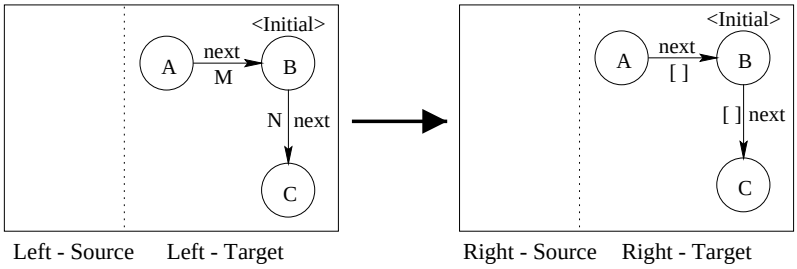


Figure 6.35: Rule for the correct punctuation for the beginning of functions (PCRuleRemoveInitComma)

PCRuleRemoveInitEnd Another failure can be caused by textual element ***','*** in Prolog programs, when an ***End*** node is the subsequent node of an ***Initial*** node (*',.'* is not acceptable). The following rule (in Fig. 6.36) removes these commas.

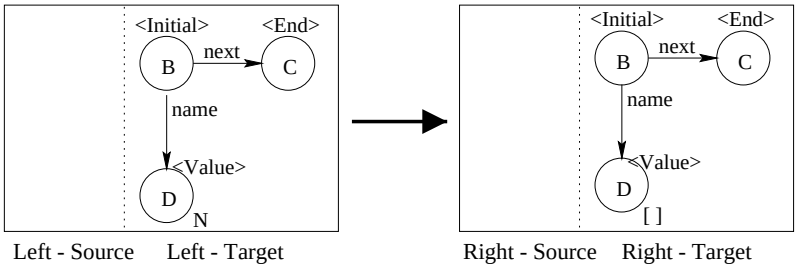


Figure 6.36: Removal of commas from next edges between Initial and End nodes (PCRuleRemoveInitEnd)

PCRuleRemoveFinalComma According to the previous rule, **PCRuleRemoveEnd**, ***next*** edges to ***End*** nodes has commas, which must be removed (see Fig. 6.37).

The result of the transformation (PCRuleNext result extended by textual elements) is illustrated in Fig. 6.38.

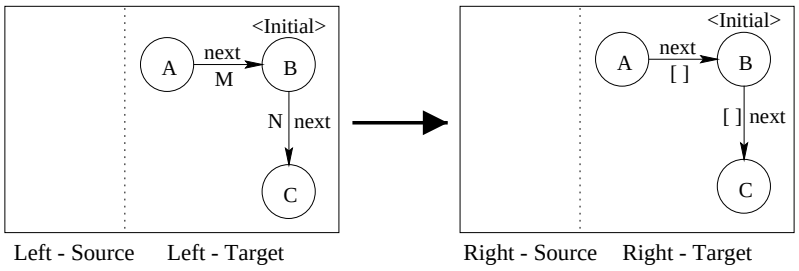


Figure 6.37: Rule for the correct notation at the end of functions (PCRuleRemoveFinalComma)

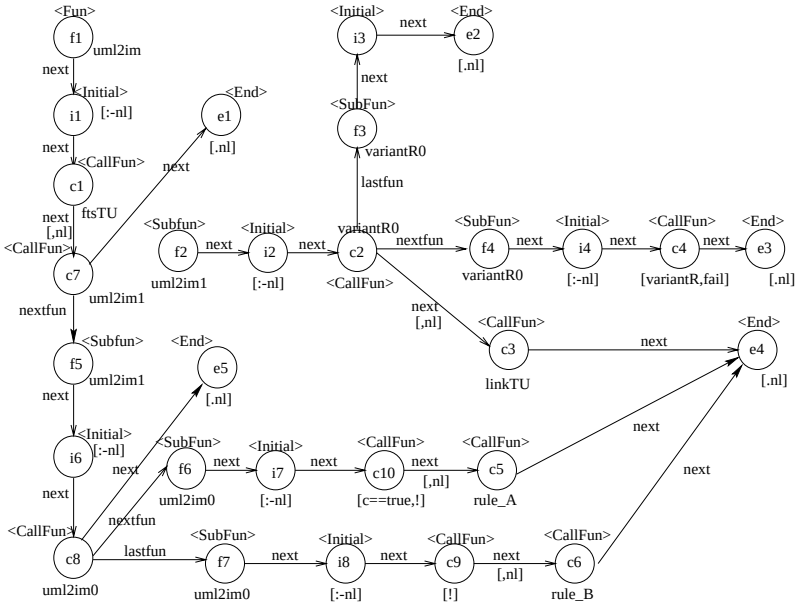


Figure 6.38: Result of the transformation

Chapter 7

Algorithm Generation

The outline of the UML–Prolog transformation is characterized by two model transformations as discussed previously. The first transformation (in Chapter 5) operates on UML statechart graphs and constructs CFGs, while the second one transforms the CFG model to a Prolog code specific graph (see Chapter 6). In order to generate the textual Prolog code for the control, a special graph traversal algorithm is used as the end.

Before discussing the code generation algorithm itself, the Prolog representations of models and are defined.

7.1 General Description of Models

As a starting point, please remember that both initial and internal models have the same underlying graph model, which structure is as follows.

- Each graph **node** has an **ID** and a **type**.
- **Edges** have a **type**, an own identifier, and an identifier to a **source** and a **target** node.
- Attributes **values** are stored by <**Value**> nodes.
- **Reference nodes** determine the correspondence between **source** and **target** objects by several **types** of references and also have an **ID** as their attributes.

Prolog implementation use the following predicates to form the graph structure used in the algorithm:

- Common model graph elements (the Prolog variable `Model` indicates the module containing the current model).
  - nodes are referred by the predicate:

    Model:type(ID)

  - special attribute nodes are denoted as follows

    Model:value(ID, data)

  - edges, directed from a `source` node (identified by `FromID`) to a `target` node (referred by `ToID`), are represented by:

Model:type(ID,FromID,ToID)

- `Reference nodes` of `reftypes`) in the reference model have `identifiers` in the Prolog implementation and two `node identifiers`, `LeftID` and `RightID`, referring to the identifiers of the appropriate participants on source and target sides respectively:

Model:reftype(ID,LeftID,RightID)

- The `Add` function is responsible for adding a new object to a specific model, assigning a new variable for object identifier at the same time.
  - in case of nodes

add(Model:type(NewID))

  - in case of attributes

add(Model:value(NewID, name))

  - in case of edges

add(Model:type(NewID, source, target))

  - in case of reference nodes

add(Model:reftype(NewID, LeftID, RightID))

- The `Remove` function is responsible for deleting graph objects from a specific model, defined by its identifier or attributes.
  - in case of nodes

remove(Model:type(ID))

  - in case of attributes

remove(Model:value(ID, name))

  - in case of edges

remove(Model:type(ID, source, target))

  - in case of reference nodes

remove(Model:reftype(ID, LeftID, RightID))

Please note, that variables in Prolog always begin with capital letters while the names of Prolog predicates are constrained to be an atom, thus they normally begin with non–capitals. More details on Prolog can be find in Section 6.1.

In the following the code of the algorithm will be discussed.

```
Predicates = set of Prolog Code predicates — elements (nodes and edges, generated
        from an initial UML statechart graph)
CONTROL = resulted algorithm of control flow
name(id) = the name of an element
type(id) = the type of an element
next(id) = the identifier of an element connected by next edge to the element referred
        by id
nextfun(id) = the identifier of an element connected by nextfun edge to the element
        referred by id
lastfun(id) = the identifier of an element connected by lastfun edge to the element
referred by id

controlgeneration =
   open2write(CONTROL);
   gen_fun;
   geb_subfun;
end.

gen_fun =
   for each (id: type(id)="Fun") do
      write(CONTROL, name(id));
      gen_body(next(id));
end.

gen_body(id) =
   CurrId = id;
   write(CONTROL, name(id));
   repeat
      CurrId = next(CurrId);
      write(CONTROL, name(id));
   until type(CurrId) = "End";
end.
gen_subfun =
   for each (id: type(id)="CallFun") do
      if ∃ nextfun(id) then
         gen_body(nextfun(id));
      else if ∃ lastfun(id) then
         gen_body(lastfun(id));
end.
```

Figure 7.1: The algorithm of the control

7.2 Prolog Code of the Transformation

The skeleton of the graph traversal algorithm is introduced as follows (in Fig. 7.1) in a Prolog style. The original Prolog source code can be found in Appendix A.3.

1. The algorithm processes the `Fun` nodes of the PC graph one by one.
   (a) the head of the clause is printed with respect to the `name` attribute of `Fun` nodes and `Initial` nodes;
   (b) its body is processed by listing the subsequent predicates represented by `CallFun` nodes according to `next` edges one by one (separated by commas);
   (c) when the `End` node is reached the code generation for the current `Fun` node is finished (by printing the final dot symbol).
2. At a next phase, the algorithm processes the `SubFun` nodes of the PC graph in order to create further Prolog predicates related to `CallFun` nodes by `nextfun` and `lastfun` edges in order to keep same predicates in an appropriate order. The code generation for `SubFun` nodes is carried out similarly to the method in case of `Fun` nodes (1.a, 1.b, 1.c).

7.3 Prolog Code of the Sample Control

The following few lines of Prolog code can be regarded as the overall aim of the previous chapters, as being the Prolog equivalent of the UML statechart of Fig. 3.4, the control flow graph of Fig. 5.23, and finally, the Prolog Code model of Fig. 6.38.

```
uml2im:-
  ftsTU,
  uml2im1.
uml2im0:-
  c==true,!,
  rule_A.
uml2im0:-
  !,
  rule_B.
uml2im1:-
  variantR0,
  linkTU.
uml2im1:-
  uml2im0.
variantR0:-
  variantR,fail.
variantR0.
```

Table 7.1: Prolog source code of the control algorithm

One can easily verify that the Prolog code conforms with the execution order prescribed by the visual control flow specification represented by UML statecharts (see Fig. 3.4) and is equivalent all the intermediate models used during the automatic code generation process.

Chapter 8

Conclusion and Result Evaluation

8.1 Benchmark Examples

Finally, as a benchmark application, our automated algorithm generation method was tested on a UML statechart model of medium size in order to assess its run–time performance.

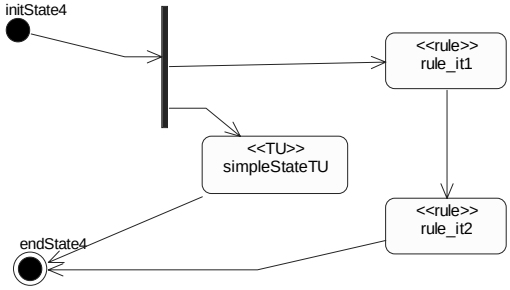
The run–time performance of the automated algorithm generation method based on model transformation was tested on the following benchmark examples.

Benchmark Example 1 *Our well–known UML statechart specifying the control flow of model transformation unit* `uml2imTU` *is assessed as a transformation of small size.*

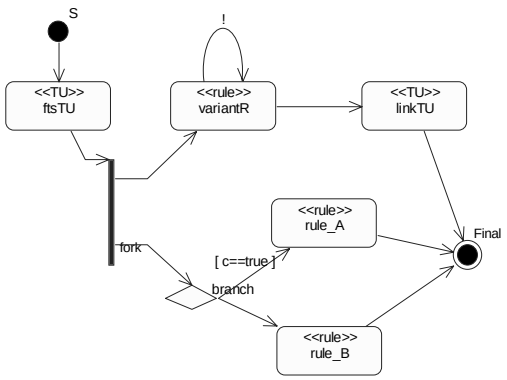
Benchmark Example 2 *The following figures (depicted in Fig. 8.1) depict the components of a larger UML statechart (called* ***TUlarge****) consisting of the following six transformation units:* `compositeStateTU`*,* `simpleStateTU`*,* `transitionTU`*,* `nonInterlevelTU`*,* `interlevelTU`*,* `initialStateTU`*.*

The process of tests was the following:

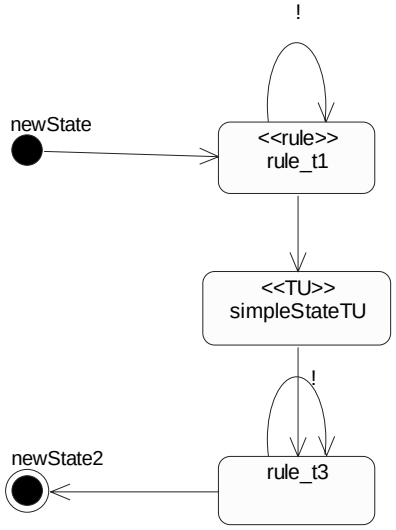
- The corresponding UML statecharts were created in Rational Rose (leading UML CASE tool), and the models were exported into an XMI format. The size of the XMI file was recorded.
- A filtering Prolog program converted this XMI description into the corresponding Statechart graph model. The overall number of graph nodes edges, and attributes was encountered this time.
- The first transformation (carried out by our Prolog program found in Appendix A.1) from statechart graphs to control flow graphs were performed and its execution was timed.
- As a result, a control flow graph was obtained and its size was measured with respect to the number of contained graph objects.
- The intermediate CFG model was transformed to a Prolog Code graph (please find the corresponding Prolog source code of transformation rules in Appendix A.2), and the run–time of the transformation algorithm was measured again.



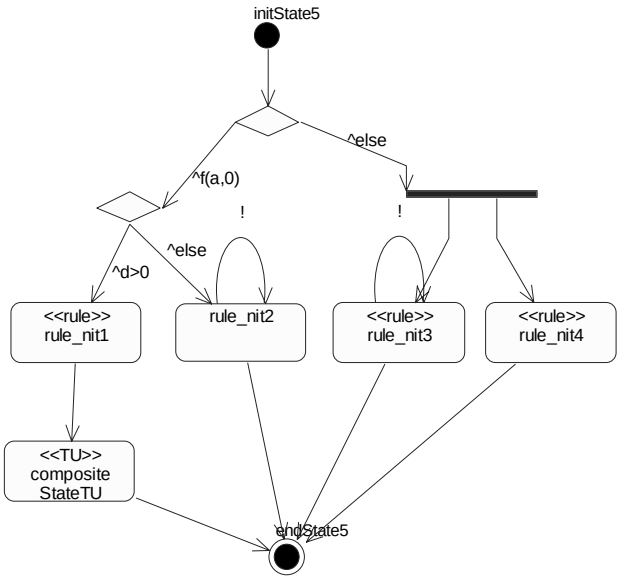
(a) Control of compositeStateTU



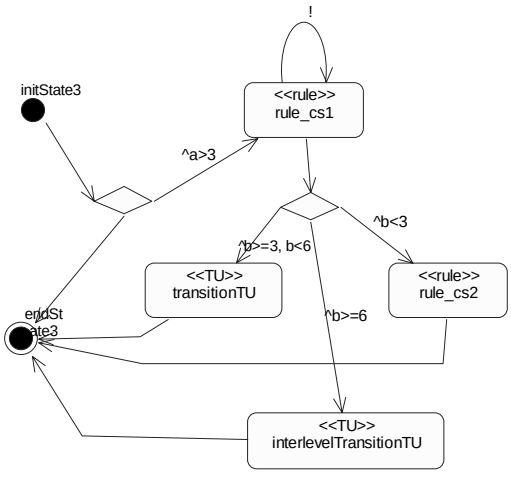
(b) Control of simpleStateTU



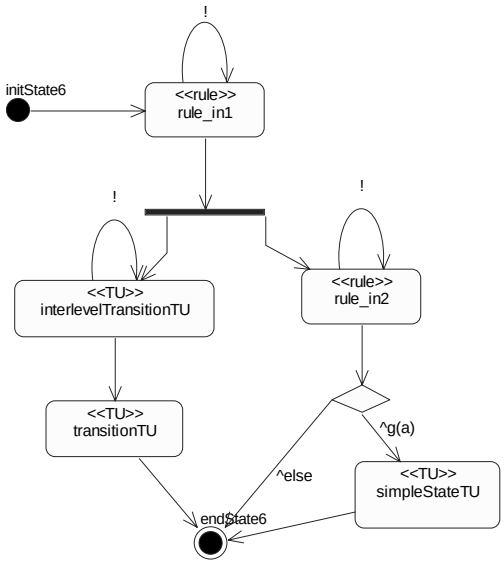
(c) Control of transitionTU



(d) Control of nonInterlevel



(e) Control of interlevelTU



(f) Control of initialStateTU

Figure 8.1: An overview of model transformation

- The overall number of Prolog Code graph objects (nodes, edges, and attributes) was counted as well.
- Finally, after having executed the graph traversal algorithm (found in Appendix A.3), the Prolog code implementing the visually specified control structure was recorded with respect to the number of lines of automatically generated code.

8.2 Benchmark Results

The results of the tests are shown in Table 8.1, according to the previous phases.

The following abbreviations are used in the tables for the two benchmark examples:

- `XMI size` is the XMI file size of the UML statecharts (displayed in bytes)
- `UML size` is the size of the generated UML statechart graph (displayed in the overall number of graph objects)
- `UMLtoCFG time` indicates the CPU time required for executing the UML2CFG program on the benchmark example UML files (displayed in seconds)
- `CFG size` refers to the size of the generated CFG model (displayed in graph objects)
- `CFGtoPC time` is the CPU time required for executing the CFG2PC program on the benchmark example UML files,
- `PC size` indicates the size of the final Prolog Code model (displayed in number of graph objects).
- `Code size` measures the number of source code lines in the automatically generated Prolog program implementing the control structures (displayed in number of source code lines).

As the final step of code generation is a simple graph traversal algorithm (executed usually in linear time), execution time was not measured in this case.

Results/Examples	uml2imTU	TUlarge
XMI size	45351	300484
UML size	194	1008
UMLtoCFG time	0.02	0.14
CFG size	52	287
CFGtoPC time	0.02	0.19
PC size	141	860
Code size	17	100

Table 8.1: Test results of a dense graph

The following facts can be observed:

- Prolog programs (`UMLtoCFG` and `CFGtoPC`) are extremely powerful (indicated by the time of the running) for statecharts of small and medium size.

- The Prolog code representation drastically reduces the size of `UML` models represented in a verbose `XMI` model.
- The size of `PC` target files are larger than the size of their corresponding `CFG` description since the former model contains special textual elements of Prolog.

We can draw the following conclusions:

- Control Flow Graphs may provide a general and compact means to represent control structures for code generation of arbitrary target language.
- Choosing Prolog for implementing complex model transformations concerning automatic code generation seems to be a correct design decision indicated by the run–time performance of transformation rules for models of a large scale.
- The code generation algorithm (described by the two model transformations and the graph traversal method) is correct, which was informally verified through Chapters 5 – 7.

8.3 Future Work

Further research aiming to extend the results presented in the current paper may be required at least in the following fields.

- Although Prolog, as providing powerful graph pattern matching, was chosen for our automated code generation (described in Chapter 4), the application method of transformation process is general, i.e. commonly used **Control Flow Graphs may serve as a basic description mechanism** for control flow representation. Thus, the transformation method should be extended from CFGs to **further programming languages** including object–oriented and functional ones.
- The further line of research could be the **formal verification of the automatic generation algorithm**. For this reason, formal methods have to be applied to the transformation to verify its correctness.

8.4 Conclusion

In the current paper, an automated algorithm generation of the control structures in transformation units is elaborated by means of model transformation rules.

The algorithm generation consists of three distinct phase:

- At first, the UML based visual specification which represents the control flow by using UML statecharts is transformed into an abstract model called **Control Flow Graph (CFG)**.
- Secondly, a **Prolog code specific (PC) graph model** is generated from the CFG model that is closely related to the final executable Prolog code.
- Finally, a simple algorithm traverses the Prolog code graph and prints the syntactic elements attached to nodes and edges into the target file.

The code generation approach was implemented in Prolog, tested on benchmark applications with valuable results, and informally verified.

As a conclusion, we hope that the automatic algorithm generation method presented in the report might serve as a basis for the design of future automatic code generation problems.

Appendix A

The Prolog Source Code of Transformations

A.1 From UML Statecharts to Control Flow Graphs

```
uml2cfgTU:-
        ruleCS,
        ruleSS,
        ruleBranch,
        ruleFork,
        ruleInitial,
        ruleFinal,
        ruleSubvertex,
        ruleTrans,
        ruleLoop,
        ruleElse,
        ruleCond.

%Rule CompositeState
ruleCS:-
        models(Src, REF, Targ),
        %LHS
        Src:compositeState(A),
        Src:name(E1, A, B),
        Src:value(B, N),
        %RHS
        add(Targ:fun(P)),
        add(Targ:value(Q, N)),
        add(Targ:name(E2, P, Q)),
        add(REF:subRef(Ref1, A, P)),
        fail.
ruleCS.

%Rule SimpleState
ruleSS:-
        models(Src, REF, Targ),
        %LHS
        Src:simpleState(A),
        Src:name(E1, A, B),
        Src:value(B, N),
        %RHS
        add(Targ:cfgNode(P)),
        add(Targ:value(Q, N)),
```

```
        add(Targ:name(E2, P, Q)),
        add(REF:subRef(Ref1, A, P)),
        fail.
ruleSS.

%Rule Pseudostate Branch
ruleBranch:-
    models(Src, REF, Targ),
    %LHS
    Src:pseudostate(A),
    Src:kind(E, A, B),
    Src:value(B, 'branch'),
    %RHS
    add(Targ:branch(P)),
    add(REF:subRef(Ref1, A, P)),
    fail.
ruleBranch.

%Rule of PseudoState Fork
ruleFork:-
    models(Src, REF, Targ),
    %LHS
    Src:pseudostate(A),
    Src:kind(E, A, B),
    Src:value(B, 'fork'),
    %RHS
    add(Targ:fork(P)),
    add(REF:subRef(Ref1, A, P)),
    fail.
ruleFork.

%Rule of PseudoState Initial
ruleInitial:-
    models(Src, REF, Targ),
    %LHS
    Src:pseudostate(A),
    Src:kind(E, A, B),
    Src:value(B, 'initial'),
    %RHS
    add(Targ:initial(P)),
    add(REF:subRef(Ref1, A, P)),
    fail.
ruleInitial.

%Rule of FinalState
ruleFinal:-
    models(Src, REF, Targ),
    %LHS
    Src:finalState(A),
    %RHS
    add(Targ:final(P)),
    add(REF:subRef(Ref1, A, P)),
    fail.
ruleFinal.

%Rule of subvertex edge
ruleSubvertex:-
        models(Src, REF, Targ),
        %LHS
```

```
        Src:subvertex(E1, A, B),
        REF:subRef(Ref1, A, P),
        REF:subRef(Ref2, B, Q),
        %RHS
        add(Targ:content(E2, P, Q)),
        fail.
ruleSubvertex.

%Rule of Transition
ruleTrans:-
        models(Src, REF, Targ),
        %LHS
        Src:transition(B),
        Src:source(E1, B, A),
        Src:target(E2, B, C),
        REF:subRef(Ref1, A, P),
        REF:subRef(Ref2, C, Q),
        %RHS
        add(Targ:next(E3, P, Q)),
        fail.
ruleTrans.

%Rule of self--transition
ruleLoop:-
        models(Src, REF, Targ),
        %LHS
        Src:signalEvent(C),
        Src:name(E4, C, D),
        Src:value(D, '!'),
        Src:trigger(E3, B, C),
        Src:transition(B),
        Src:target(E2, B, A),
        Src:source(E1, B, A),
        Src:simpleState(A),
        REF:subRef(Ref1, A, P),
        Targ:next(E5, P, P),
        %RHS
        remove(Targ:next(E5, P, P)),
        remove(Targ:cfgNode(P)),
        add_old(Targ:loopNode(P)),
        add(Targ:loop(E6, P, P)),
        fail.
ruleLoop.

%Rule of condition
ruleCond:-
        models(Src, REF, Targ),
        %LHS
        Src:guard(C),
        Src:guard(E1, B, C),
        Src:transition(B),
        Src:source(E2, B, A),
        Src:target(E3, B, E),
        Src:name(E4, C, D),
        Src:value(D, Cond),
        REF:subRef(Ref1, A, P),
        REF:subRef(Ref2, E, Q),
        Targ:next(E5, P, Q),
        %RHS
```

```
        add(Targ:cond(R)),
        add(Targ:next(E6, P, R)),
        add(Targ:next(E7, R, Q)),
        add(Targ:value(S, Cond)),
        add(Targ:cond(E8, R, S)),
        remove(Targ:next(E5, P, Q)),
        fail.
ruleCond.

%Rule of else structure
ruleElse:-
        models(Src, REF, Targ),
        %LHS
       Src:guard(C),
        Src:guard(B, C),
        Src:name(E0, C, D),
        Src:value(D, 'else'),
        Src:transition(B),
        Src:source(E3,B, A),
        Src:target(E4, B, E),
        REF:subRef(Ref1, A, P),
        REF:subRef(Ref2, E, Q),
        Targ:next(F, P, Q),
        %RHS
        add(Targ:else(R)),
        add(Targ:next(G, P, R)),
        add(Targ:next(H, R, Q)),
        remove(Targ:next(F, P, Q)),
        fail.
ruleElse.
```

A.2 From Control Flow Graphs to Prolog Code Model Graphs

```
genname(Name, NewName):-
        names(Name,ID), !,
        % Creating a new name
        create_name(Name,ID,NewName),
        % Updating the internal database
        retract(names(Name,ID)),
        ID1 is ID + 1,
        assert(names(Name,ID1)).
genname(Name,NewName):-
        create_name(Name,0,NewName),
        assert(names(Name,1)).

create_name(Name,ID,NewName):-
        atom_chars(Name,NameLs),
        number_chars(ID,NumLs),
        append(NameLs,NumLs, NewNameLs),
        atom_chars(NewName,NewNameLs).


cfg2pcTU:-
        pcRuleFun,
        pcRuleCFG,
        pcRuleTop,
        pcRuleBranchIn,
        pcRuleForkIn,
        pcRuleLoop,
        pcRuleLoopFirst,
        pcRuleLoopLast,
        pcRuleFinal,
        pcRuleInitial,
        pcRuleCond,
        pcRuleElse,
        pcRuleForkOut,
        pcRuleBranchOutCond,
        pcRuleBranchOutElse,
        pcRuleNext,
        pcRuleDots,
        pcRuleComma,
        pcRuleInitialText,
        pcRuleTab,
        pcRuleRemoveInitComma,
        pcRuleRemoveInitEnd,
        pcRuleRemoveFinalComma.

%Rule of Fun node
pcRuleFun:-
        models(Src, REF, Targ),
        %LHS
        Src:fun(A),
        Src:name(E1, A, B),
        Src:value(B, N),
        %RHS
        add(Targ:fun(P)),
        add(Targ:value(Q, [N])),
        add(Targ:name(E2, P, Q)),
```

```
        add(REF:refIn(Ref, A, P)),
        add(REF:refOut(Ref1, A, P)),
        fail.
pcRuleFun.

%Rule of CFG node
pcRuleCFG:-
        models(Src, REF, Targ),
        %LHS
        Src:cfgNode(A),
        Src:name(E1, A, B),
        Src:value(B, N),
        %RHS
        add(Targ:callFun(P)),
        add(Targ:value(Q, [N])),
        add(Targ:name(E2, P, Q)),
        add(REF:refIn(Ref, A, P)),
        add(REF:refOut(Ref1, A, P)),
        fail.
pcRuleCFG.

%Rule of Branch node
pcRuleBranchIn:-
        models(Src, REF, Targ),
        %LHS
        Src:fun(A),
        Src:name(E1, A, C),
        Src:value(C, N),
        Src:content(E2, A, B),
        Src:branch(B),
        %RHS
        add(Targ:callFun(P)),
        genname(N,N0),
        add(Targ:value(Q, [N0])),
        add(Targ:name(E3, P, Q)),
        add(Targ:end(S)),
        add(Targ:next(E5, P, S)),
        add(REF:refIn(Ref, B, P)),
        fail.
pcRuleBranchIn.

%Rule of Fork node
pcRuleForkIn:-
        models(Src, REF, Targ),
        %LHS
        Src:fun(A),
        Src:name(E1, A, C),
        Src:value(C, N),
        Src:content(E2, A, B),
        Src:fork(B),
        %RHS
        add(Targ:callFun(P)),
        genname(N,N0),
        add(Targ:value(Q, [N0])),
        add(Targ:name(E3, P, Q)),
        add(Targ:end(S)),
        add(Targ:next(E5, P, S)),
        add(REF:refIn(Ref, B, P)),
        fail.
```

```
pcRuleForkIn.

%Rule of Loopnode
pcRuleLoop:-
        models(Src, REF, Targ),
        %LHS
        Src:loopNode(B),
        Src:loop(E0, B, B),
        Src:name(E1, B, C),
        Src:value(C, N),
        %RHS
        add(Targ:callFun(P)),
        genname(N,N0),
        add(Targ:value(Q, [N0])),
        add(Targ:name(E2, P, Q)),
        add(REF:refLoop(Ref, B, P)),
        add(REF:refIn(Ref1, B, P)),
        add(REF:refOut(Ref2, B, P)),
        fail.
pcRuleLoop.

%Rule of loop to create the first function
pcRuleLoopFirst:-
        models(Src, REF, Targ),
        %LHS
        REF:refLoop(Ref1,A,P),
        Src:loopNode(A),
        Src:name(E0,A,B),
        Src:value(B,M),
        Targ:callFun(P),
        Targ:name(E1, P, Q),
        Targ:value(Q, N),
        %RHS
        add(Targ:subFun(T)),
        add(Targ:nextfun(E3, P, T)),
        add(Targ:value(Y,N)),
        add(Targ:name(E4, T, Y)),
        add(Targ:initial(U)),
        add(Targ:next(E5, T, U)),
        add(Targ:callFun(V)),
        add(Targ:next(E6, U, V)),
        add(Targ:value(S,[M, ',', 'fail'])),
        add(Targ:name(E7, V, S)),
        add(Targ:end(X)),
        add(Targ:next(E8, V, X)),
        fail.
pcRuleLoopFirst.


%Rule of loop to create the last function
pcRuleLoopLast:-
        models(Src, REF, Targ),
        %LHS
        REF:refLoop(Ref1,A,P),
        Src:loopNode(A),
        Targ:callFun(P),
        Targ:name(E1, P, Q),
        Targ:value(Q, N),
        %RHS
```

```
        add(Targ:subFun(R)),
        add(Targ:lastfun(E2, P, R)),
        add(Targ:value(Y,N)),
        add(Targ:name(E3, R, Y)),
        add(Targ:initial(S)),
        add(Targ:next(E4, R, S)),
        add(Targ:end(T)),
        add(Targ:next(E5, S, T)),
        fail.
pcRuleLoopLast.

%Rule of Final node
pcRuleFinal:-
        models(Src, REF, Targ),
        %LHS
        Src:final(A),
        %RHS
        add(Targ:end(P)),
        add(REF:refIn(Ref, A, P)),
        fail.
pcRuleFinal.

%Rule of Initial node
pcRuleInitial:-
        models(Src, REF, Targ),
        %LHS
        Src:initial(B),
        Src:content(E0, A, B),
        Src:fun(A),
        REF:refOut(Ref1, A, P),
        % RHS
        add(Targ:initial(Q)),
        add(Targ:next(E1,P,Q)),
        add(REF:refIn(Ref2,B,Q)),
        add(REF:refOut(Ref3,B,Q)),
        fail.
pcRuleInitial.

%Rule of conditional thread of branches
pcRuleCond:-
        models(Src, REF, Targ),
        %LHS
        Src:cond(B),
        Src:cond(E2, B, C),
        Src:value(C, Cond),
        %RHS
        add(Targ:callFun(R)),
        add(Targ:value(Q, [Cond, ',!'])),
        add(Targ:name(E5, R, Q)),
        add(REF:refIn(Ref1, B, R)),
        add(REF:refOut(Ref2, B, R)),
        fail.
pcRuleCond.

%Rule of else thread of branches
pcRuleElse:-
        models(Src, REF, Targ),
        %LHS
        Src:else(B),
```

```
        %RHS
        add(Targ:callFun(R)),
        add(Targ:value(Q, ['!'])),
        add(Targ:name(E5, R, Q)),
        add(REF:refIn(Ref1, B, R)),
        add(REF:refOut(Ref2, B, R)),
        fail.
pcRuleElse.


%Rule of outgoing next edges from Fork nodes
pcRuleForkOut:-
        models(Src, REF, Targ),
        %LHS
        Src:fork(A),
        REF:refIn(Ref1, A, P),
        Targ:name(E1, P, Q),
        Targ:value(Q, N),
        Src:next(E2, A, B),
        REF:refIn(Ref2, B, R),
        %RHS
        add(Targ:subFun(S)),
        add(Targ:nextfun(E3, P, S)),
        add(Targ:value(Y,N)),
        add(Targ:name(E4, S, Y)),
        add(Targ:initial(T)),
        add(Targ:next(E5, S, T)),
        add(Targ:next(E6, T, R)),
        fail.
pcRuleForkOut.

%Rule of outgoing next edges from Branch nodes
pcRuleBranchOutCond:-
        models(Src, REF, Targ),
        %LHS
        Src:branch(A),
        REF:refIn(Ref1, A, P),
        Targ:name(E1, P, Q),
        Targ:value(Q, N),
        Src:next(E2, A, B),
        Src:cond(B),
        REF:refIn(Ref2, B, R),
        %RHS
        add(Targ:subFun(S)),
        add(Targ:nextfun(E3, P, S)),
        add(Targ:value(Y,N)),
        add(Targ:name(E4, S, Y)),
        add(Targ:initial(T)),
        add(Targ:next(E5, S, T)),
        add(Targ:next(E6, T, R)),
        fail.
pcRuleBranchOutCond.

%Rule of outgoing next edges from Else nodes
pcRuleBranchOutElse:-
        models(Src, REF, Targ),
        %LHS
        Src:branch(A),
        REF:refIn(Ref1, A, P),
```

```
        Targ:name(E1, P, Q),
        Targ:value(Q, N),
        Src:next(E2, A, B),
        Src:else(B),
        REF:refIn(Ref2, B, R),
        %RHS
        add(Targ:subFun(S)),
        add(Targ:lastfun(E3, P, S)),
        add(Targ:value(Y,N)),
        add(Targ:name(E4, S, Y)),
        add(Targ:initial(T)),
        add(Targ:next(E5, S, T)),
        add(Targ:next(E6, T, R)),
        fail.
pcRuleBranchOutElse.

%Rule of next edges
pcRuleNext:-
        models(Src, REF, Targ),
        %LHS
        Src:next(E1, A, B),
        REF:refOut(Ref1, A, P),
        REF:refIn(Ref2,B,Q),
        %RHS
        add(Targ:next(E2, P, Q)),
        fail.
pcRuleNext.

%Dots assigned to End nodes
pcRuleDots:-
        models(_Src,_REF,Targ),
        % LHS
        Targ:end(P),
        % RHS
        add(Targ:value(Q,['.','nl'])),
        add(Targ:name(E0,P,Q)),
        fail.
pcRuleDots.

%Commas assigned to next edges
pcRuleComma:-
        models(Src,REF,Targ),
        % LHS
        Targ:next(E, A, B),
        % RHS
        add_old(Targ:value(E,[',','nl'])),
%       add(Targ:name(E0,E,Q)),
        fail.
pcRuleComma.

%':-' assigned to Initial nodes
pcRuleInitialText:-
        models(_Src,_REF,Targ),
        % LHS
        Targ:initial(P),
        % RHS
        add(Targ:value(Q,[':-','nl'])),
        add(Targ:name(E0,P,Q)),
        fail.
```

```
pcRuleInitialText.

%Removal of commas after Initial nodes
pcRuleRemoveInitComma:-
        models(Src,REF,Targ),
        % LHS
        Targ:initial(B),
        Targ:next(E0,A,B),
        Targ:value(E0,M),
        Targ:next(E1,B,C),
        Targ:value(E1,N),
        % RHS
        remove(Targ:value(E0,M)),
        remove(Targ:value(E1,N)),
        add_old(Targ:value(E0,[])),
        add_old(Targ:value(E1,[])),
        fail.
pcRuleRemoveInitComma.

%Removal of unnecessary commas and ':-'
pcRuleRemoveInitEnd:-
        models(Src,REF,Targ),
        % LHS
        Targ:initial(B),
        Targ:next(E1,B,C),
        Targ:end(C),
        Targ:name(E2,B,D),
        Targ:value(D,N),
        % RHS
        remove(Targ:value(D,N)),
        add_old(Targ:value(D,[])),
        fail.
pcRuleRemoveInitEnd.

%Removal of commas before End nodes
pcRuleRemoveFinalComma:-
        models(Src,REF,Targ),
        % LHS
        Targ:end(C),
        Targ:next(E1,B,C),
        Targ:value(E1,N),
        % RHS
        remove(Targ:value(E1,N)),
        add_old(Targ:value(E1,[])),
        fail.
pcRuleRemoveFinalComma.
```

A.3 The Prolog Source Code of Control Algorithm Generation

```
%Process of function nodes from the beginning of them
generate_code:-
        gen_fun,
        gen_subFun.

gen_fun:-
        model(Src),
        Src:fun(ID),
        get_name(ID, Name),
        print_value(Name),
        gen_body(ID),
        %process the subsequent object
        fail.
gen_fun.

%continue the codewriting at the next node
gen_body(ID):-
        model(Src),
        Src:next(E, ID, ToID), !,
        get_edge_name(E,Name),
        print_value(Name),
        get_type(ToID, Type),
        gen_body2(ToID, Type).

%special process, according to the type of the current node
gen_body2(ID, 'callFun'):-
        !,
        get_name(ID, Name),
        print_value(Name),
        gen_body(ID).

gen_body2(ID, 'initial'):-
        !,
        get_name(ID, Name),
        print_value(Name),
        gen_body(ID).


gen_body2(ID, 'end'):-
        !,
        get_name(ID, Name),
        print_value(Name).

gen_subFun:-
        model(Src),
        Src:callFun(ID),
        gen_subFun2(ID),
        fail.
gen_subFun.

%branch, fork or loop process, the coherent threads must have been
%together in the code
gen_subFun2(ID):-
        model(Src),
        Src:nextfun(_E, ID, ToID),
```

```
        Src:subFun(ToID),
        get_name(ToID, Name),
        print_value(Name),
        gen_body(ToID),
        fail.

%the lastfun determines the last fun
gen_subFun2(ID):-
        model(Src),
        Src:lastfun(_E, ID, ToID),
        Src:subFun(ToID),
        get_name(ToID, Name),
        print_value(Name),
        gen_body(ToID).

%get the name of the current node by connected name edge
get_name(ID, NameLs):-
        model(Src),
        Src:name(_E, ID, ToID),
        Src:value(ToID, NameLs), !.

get_edge_name(ID, NameLs):-
        model(Src),
        Src:value(ID, NameLs), !.

get_type(ID,Type):-
        model(Model),
        current_predicate(Type, Model:Term),
        arg(1,Term,ID),
        (Model:Term).

print_value([]).
print_value([nl|Ls]):-
        nl, !,
        print_value(Ls).
print_value([Val|Ls]):-
        write(Val),
        print_value(Ls).
```

Bibliography

[1] *APPLIGRAPH Subgroup Meeting on Exchange Formats for Graph Transformation Systems*, Paderborn, September 2000.

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

[3] M. Andries, G. Engels, A. Habel, B. Hoffmann, H.-J. Kreowski, S. Kuske, D. Plump, A. Schürr, and G. Taentzer. Graph transformation for specification and programming. *Science of Computer Programmming*, 1999.

[4] A. Bondavalli, M. D. Cin, D. Latella, and A. Pataricza. High-level Integrated Design Environment for Dependability. In *WORDS'99, 1999 Workshop on Real-Time Dependable System*, 1999.

[5] A. Bondavalli, I. Majzik, and I. Mura. Automatic dependability analyses for supporting design decisions in UML. In *HASE'99: the 4th IEEE International Symposium on High Assurance Systems Engineering*, 1999.

[6] M. D. Cin, G. Huszerl, and K. Kosmidis. Evaluation of safety-critical system based on guarded statecharts. In *HASE'99 4th IEEE International Symposium on High Assurance Systems Engineering*, 1999.

[7] A. Corradini, U. Montanari, F. Rossi, H. Ehrig, R. Heckel, and M. Löwe. *Handbook of Graph Grammars and Computing by Graph Transformations*, volume 1: Foundations, chapter Algebraic Approaches to Graph Transformation — Part I: Basic Concepts and Double Pushout Approach, pages 163–245. World Scientific, 1997.

[8] H. Ehrig, R. Heckel, M. Korff, M. Löwe, L. Ribeiro, A. Wagner, and A. Corradini. *Handbook of Graph Grammars and Computing by Graph Transformations*, volume 1: Foundations, chapter Algebraic Approaches to graph transformation — Part II: Single pushout approach and comparison with double pushout approach, pages 247–312. World Scientific, 1997.

[9] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, 1987.

[10] G. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, (23):279–295, 1997.

[11] Intelligent Systems Laboratory, Swedish Institute of Computer Science. *Sicstus Prolog User's Manual*, November 1997.

[12] R. A. Kowalski. Logic for problem solving. Technical report, University of Edinburgh DCL Memo 75, Dept of Artificial Intelligence, March 1974.

[13] H.-J. Kreowski and S. Kuske. On the interleaving semantics of transformation units — a step into GRACE. In J. E. Cuny, H. Ehrig, G. Engels, and G. Rozenberg, editors, *Graph Grammars and Their Application to Computer Science*, volume 1073 of *LNCS*, pages 89–108. Springer, 1996.

[14] S. Kuske. More about control conditions for transformation units. In H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors, *Theory and Application of Graph transformations*, volume 1764 of *LNCS*, pages 323–337, 2000.

[15] D. Latella, I. Majzik, and M. Massink. Towards a formal operational semantics of UML Statechart Diagrams. In *IFIP TC6/WG6.1 3rd International Conference on Formal Methods for Open Object-Oriented Distributed Systems*, february 1999.

[16] M. Nagl. Set theoretic approaches to graph grammars. In H. Ehrig, M. Nagl, G. Rozenberg, and A. Rosenfeld, editors, *Graph Grammars and Their Application to Computer Science*, volume 291, pages 41–54, Berlin, 1987. Springer.

[17] Object Management Group. *UML Semantics Version 1.1*, September 1997. http://www.rational.com/uml.

[18] Object Management Group. *XML Metadata Interchange*, October 1998. http://www.omg.org.

[19] Object Management Group. *Meta Object Facility Version 1.3*, September 1999. http://www.omg.org.

[20] Object Management Group. *Object Constraint Language Specification Version 1.3*, June 1999. http://www.rational.com/uml.

[21] S. Owre and N. Shankar. The formal semantics of PVS. Technical report, Computer Science Laboratory, SRI International, August 1997.

[22] J. A. Robinson. A machine–oriented logic based on the resolution principle. *Journal of the ACM*, 12:23–44, 1965.

[23] P. Roussel. *Manuel de Reference et d'Utilisation.* Groupe d'Intelligence Artificielle, Marseille–Luminy, 1975.

[24] A. Schürr. Introduction to PROGRES, an attributed graph grammar based specificatoin language. In M. Nagl, editor, *Graph–Theoretic Concepts in Computer Science*, volume 411, pages 151–165, Berlin, 1990. Springer. Lecture Notes in Computer Science.

[25] D. Varró. Automatic transformation of UML models. Master's thesis, Budapest University of Technology and Economics, 2000.

[26] D. Varró, P. Domokos, and A. Pataricza. UML specification of model transformation systems. In *TACAS 2001: Tools and Algorithms for the Construction and Analysis of Systems*, 2001. Submitted paper.

[27] D. Varró and G. Varró. Designing the automatic transformation of visual languages. scientific report (TDK), Technical University of Budapest, November 1999.

[28] D. Varró, G. Varró, and A. Pataricza. Designing the automatic transformation of visual languages. *Science of Computer Programming*, December 2000. Submitted paper.