ulm university universität
uulm

An Extensible Platform for the Analysis of Graph Transformation Systems using Constraint Handling Rules

Mathias Wasserthal
Universität Ulm
Fakultät für Ingenieurwissenschaften
und Informatik
Institut für Programmiermethodik und Compilerbau
Februar 2009
Diplomarbeit im
Studiengang Informatik

Gutachter:
Prof. Dr. Frühwirth
Prof. Dr. von Henke

Betreuer:
Frank Raiser

Fassung 10. Februar 2009

Satz: PDF-LaTeX 2_ε

Contents

List of Figures

List of Tables

Listings

List of abbreviations

AGG Attributed Graph Grammar, system for creation and analyzing GTS

CHR Constraint Handling Rules, a rule-based declarative language

CCHR CHR implementation in C

CD Compact Disc

DPO Double Pushout, construction in category theory

EMF Eclipse Modeling Framework, create source code from model definitions

GEF Graphical Editing Framework, framework for the creation of graphical editors for eclipse

GMF Graphical Modeling Framework, framework for the model-driven development of graphical editors for eclipse.

Groove Graphs for Object Oriented Verification, system for the creation and analyzing GTS

GTS Graph Transformation Systems, rule-based formalism for the modification of graphs

GUI Graphical User Interface

IDE Integrated Development Environment

JaCK Java Constraint Kit, CHR environment for java

JAR Java Archive file, file containing executable java programs

JASE Java Abstract Search Engine, search mechanisms for java

JavaDoc Documentation generated from annotated java source code

JCHR The K.U. Leuven JCHR system, a CHR environment for java

JCHRIDE Eclipse text editor for JCHR

JET Java Emitter Templates, a code generator for java

JGraph Framework for creating graphical editors for graphs in java

JGraphT Library for representing graphs in java

MVC Model View Controller, a paradigm for the realization of GUIs

oAW openArchitectureWare, a framework for model-driven development

PO Pushout, construction in category theory

PDE Plug-in Development Environment for eclipse

PDF Portable Document Format

RCP Rich Client Platform, client software based on eclipse

Swing Framework for GUI used in java applications

SWT Standard Widget Toolkit, framework for GUIs used by eclipse

TEF Textual Editing Framework, framework for text to model transformations in eclipse

TMF Textual Modeling Framework, framework for model-driven development of text to model transformations in eclipse

UI User interface

UML Unified Modeling Language

XPath Query language for XML files

XSLT Extensible style-sheet language transformations, language for text to text transformations

XText Textual editor and parser for the TMF

yFile Framework for creating editors for graphs in java

1 Introduction

Graphs are one of the fundamental structures in computer science and can be used to model a large variety of aspects. They represent an easy way of displaying certain structures (e.g. road maps), concepts (e.g. flowcharts), and relations (e.g. UML class diagrams).

Graph Transformation Systems (GTS) ([18]) provide the means to manipulate graphs in a rule based manner. GTS use rules which describe a graph before and after the application of the rule. These rules can be applied to host graphs in order to modify them. GTS have applications not only in theoretical computer science, but also in the fields of visual modeling techniques, model transformations, and concurrent distributed systems. Therefore, GTS are often embedded in other projects, but there are only few general purpose development environments that enable you to create and analyze GTS. *Graphs for Object Oriented Verification (Groove)* ([33]) and *Attributed Graph Grammar (AGG)* ([18]), to name two of these environments, both have graphical editors for GTS. Furthermore, the tools provide analysis methods for GTS, e.g. analysis of critical pairs ([18]).

Another rule based formalism are *Constraint Handling Rules (CHR)* ([20]). A CHR program is described by rules which modify a *constraint store* by adding and removing *constraints* from it. CHR has a large variety of applications including scheduling, agents, spatial or temporal reasoning, and verification. Additionally, CHR programs can be analyzed in various ways, e.g. termination analysis or operational equivalence analysis.

Implementations of CHR exist for various programming languages. Besides that, CHR can be used to describe, execute, and analyze GTS, as well ([32]). To do that, rules of a GTS are expressed as rules in CHR and graphs can be expressed as lists of constraints.

In the remainder of this chapter, I present the state of the art in GTS tools, CHR implementations, and integrated development environments in section 1.1. Based on this, I motivate my thesis in section 1.2. In section 1.3, I present two examples that are used throughout the thesis.

1.1 State of the art

The tools mentioned above for the creation and analysis of GTS (Groove, AGG) offer an impressive list of features. AGG provides a graphical editor to define graphs and rules. Further editing features are the introduction of graph constraints and negative application conditions which restrict the applicability of rules. It provides methods to test the rules for confluence, i.e. it is checked whether the resulting graphs are isomorphic when rules are applied in an arbitrary order. Groove also has a graphical editor to create and edit graphs. This editor is, however, based on entering keywords to describe properties of nodes and edges of a graph, but no graphical user interface elements like context menus are provided. Groove also has interesting analysis features. It can, for example, display the explored state space when running the GTS. Each intermediate graph, called the state of a graph, to which the original graph can be transformed is shown as a node. The rules that are needed to reach one state from another are displayed as edges in this graph. However, not the full state space is displayed but nodes representing isomorphic graphs are merged into one node. Both tools offer the possibility to apply rules to a given host graph, so that the GTS

can be tested on chosen inputs. An *Application Programming Interface (API)* to extend the functionality of the systems is not available in both tools.

Implementations of CHR exist in many languages, for example Prolog, java, and C ([39]). The earliest implementations of CHR are based on logic programming languages ([24]). The most widely used system is K.U. Leuven CHR which is implemented on top of certain Prolog based languages, e.g. SWI-Prolog ([38]) and Sicstus-Prolog ([36]). There are also implementations available for the functional language Haskell, e.g. HaskellCHR[1] and TaiChi ([5]). *Concurrent CHR* ([28]) is a CHR implementation that can be run concurrently and is implemented in Haskell. Implementations for java are the *Java Constraint Kit (JaCK)* ([2]) and the *K.U. Leuven JCHR System (JCHR)* ([40]). *CCHR* ([45]) is an implementation of CHR for the programming language C.

Eclipse[2] is an open-source platform for creating *Integrated Development Environments (IDE)*. Eclipse is written in java and has many technologies available that make the development of IDEs comfortable. For example, it already provides frameworks that allow the creation of custom graphical and textual editors. Eclipse is based around plug-ins, so that developers can extend the functionality of eclipse based on already available plug-in definitions or create new plug-in definitions that can be used by other developers. IDEs can also be distributed as *Rich Client Platforms (RCP)*, so that only the functionality of the IDE is provided without having to install a whole eclipse system.

1.2 Motivation

As mentioned above, AGG and Groove are not extensible with further analysis or editing methods. They do not provide all the features that may be needed for the design of a GTS. AGG, for example, offers a confluence analysis for the rules while Groove can explore the whole space of graphs that can be created with the rules. Both features are great for designing GTS, but they are not available in one single tool and none of the tools has a documented API for extending it with the desired functionality. Furthermore, Frank Raiser discovered that GTS can easily be encoded in CHR ([32]), so that GTS can be simulated with CHR. Because of that, analysis methods for CHR can also be applied to a GTS. The CHR notation of rules and graphs yields a simple textual representation of a GTS that can be used for editing, as well. Additionally, eclipse already provides frameworks to create graphical and textual editors and allows the creation of custom plug-in definitions that can be used by other developers to extend the functionality.

For these reasons, I have decided to develop a new platform that is based on eclipse and provides the possibility to create, edit, and analyze GTS in combination with CHR. This platform must have a graphical editor similar to those in AGG and Groove, but the Graphical *User Interface (GUI)* should offer a better usability. As GTS can be encoded in CHR, the graphs and rules of the GTS should also be displayed in this encoding. As the encoding provides a textual representation of a GTS, support for editing the GTS via the CHR encoding must also be provided. Additionally, this new platform should be easily extensible to support further analysis methods. Therefore, a common API must be available for the platform to add tools for analysis and editing purposes. To demonstrate the capabilities of the platform, selected analysis tools should be provided. The most important ones are a tool to simulate a GTS based on CHR and a tool that allows the easy embedding of CHR analysis methods.

[1] by Gregory J. Duck, 2004. Download: `http://www.cs.mu.oz.au/~gjd/haskellchr/`
[2] Available at `http://www.eclipse.org/`

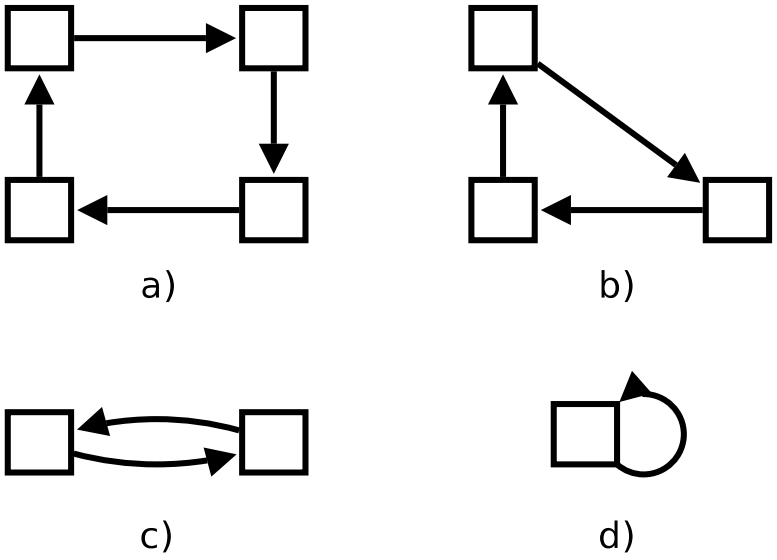


Figure 1.1: A circular list consisting of four nodes (a)), after removing one node (b)), with only two nodes (c)), and only one node (d)).

1.3 Examples

I have chosen two examples to show the features of the developed platform. First, an example for detecting circular lists from a graph is presented. The second example is the dining philosophers problem.

1.3.1 Finding circular lists

This is an example which finds circular lists in directed graphs. Figure 1.1 a) shows a graph consisting of four nodes that are connected in a circle. The algorithm to find circular lists tries to shorten linked lists of three nodes in a row. Therefore, from each linked list with three nodes in the graph, the middle node is removed and replaced by an edge connecting the two remaining nodes (figure 1.1 b)). This is repeated until only two nodes are left (figure 1.1 c)) that are connected to each other. These two nodes are then replaced by one node with a loop (figure 1.1 d)).

The result of this algorithm is a new graph. If the graph only contains one node with a single loop, then the original graph was a cyclic list. Else, the graph was no cyclic list. This is a useful example for demonstrating the encoding of GTS to CHR (section 2.3.2) and is used as an example throughout the thesis.

1.3.2 Dining philosophers

The dining philosophers problem is an illustrative example for multi-process synchronization problems which is a common computing problem in concurrency ([8]). The problem is that n philosophers are sitting next to each other on a round table. Every philosopher has a plate with spaghetti in front of him. There is also one fork between every two of them resulting in n forks on the table. Each philosopher can either think, wait until he can take

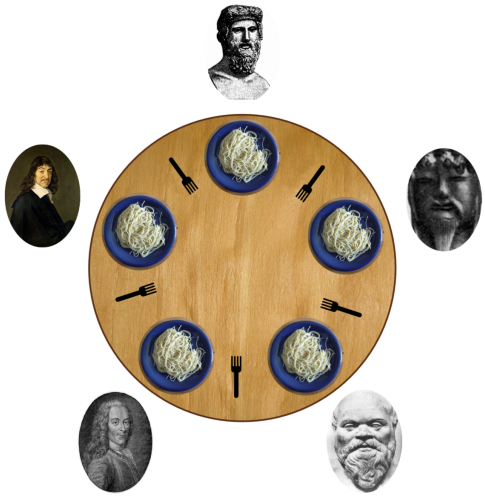


Figure 1.2: Illustration of the dining philosophers problem with five philosophers (from `http://en.wikipedia.org/wiki/File:Dining_philosophers.png`).

the two forks lying next to him, or eat his spaghetti (if he has the two forks). Figure 1.2 illustrates this. This problem is an example of multiple processes (the philosophers) which need multiple shared resources (the forks).

A problem that can arise is, that each philosopher takes the fork to his left and waits for the fork of the philosopher to his right, so that he can start to eat. But none of them will ever get two forks because all are waiting. This is called a deadlock state. Another problem is, that some of the philosophers may never get the chance to eat. This is the so called starvation problem. These are problems that can occur between multiple processes (the philosophers) which are concurrently accessing shared resources (the forks). The porting of this problem to GTS is shown in section 2.2 and its encoding to CHR in section 2.3.

1.4 Organization of the thesis

The main part of this thesis is grouped into four parts:

- Chapter 2 describes the theoretical background of GTS, CHR, and eclipse
- Chapter 3 describes the goals and requirements which are a result of the conception phase. Based on this, tools are analyzed which can realize the requirements.
- Chapter 4 describes the design and implementation of the project in order to fulfill the goals and requirements. Theoretical aspects needed for the realization are also explained here.
- Chapter 5 summarizes the results of the thesis and provides ideas on the further development of the project. Furthermore, possible application areas are shown.

2 Preliminaries

In this chapter, I give a short introduction to *Constraint Handling Rules (CHR)*, *Graph Transformation Systems (GTS)*, and *eclipse*. As mentioned in section 1.2, CHR can be used to simulate GTS. Therefore, I first give a small introduction to CHR. Afterward, I give an introduction to GTS. GTS are described here with an algebraic approach which is based on category theory. The mathematical background for category theory is covered in appendix B. Other approaches to GTS can be found in [41] and [19]. After the introductions to GTS and CHR, I show how to encode GTS to CHR. As the last topic, I present the architecture of the eclipse platform, because it will be used as framework for this project.

2.1 Constraint Handling Rules (CHR)

Constraint Handling Rules (CHR) are a constraint-based language developed by T. Frühwirth in 1998. The syntax and operational semantics of CHR are described in this section. This introduction is based on [20] which gives a more comprehensive overview of the topic of CHR.

CHR are a rule based language. A CHR program consists of the declaration of constraints (first-order logic predicates) and a set of rules. Constraints are separated into *built-in* and *user-defined constraints*. Built-in constraints are handled by a *constraint solver* and user-defined constraints are solved by the CHR program itself.

2.1.1 Syntax

Constraints are defined by giving them a unique combination of a name and an arity. A constraint c of arity n is defined as c/n (in Prolog-based systems). An n-ary constraint c is denoted by

$$c(t_1, \ldots, t_n)$$

where $t_1, \ldots, t_n$ are terms. These terms can either be ground, i.e. they represent a constant value, be a variable, or be an expression of further terms. Identifiers used in ground terms are lower case identifiers and variable names are given by a name which starts with a capital letter.

The input is a sequence of user-defined and built-in constraints which is called the *goal*

$$G = [G_1, \ldots, G_n]$$

where $G_1, \ldots, G_n$ are user-defined or built-in constraints. Rules are of the form:

$$Rulename \ @ \ H'_1, \ldots, H'_{i'} \backslash H_1, \ldots, H_i \Leftrightarrow D_1, \ldots, D_j | B_1, \ldots, B_k$$

where $Rulename$ is an optional unique name for the rule. $H_{kep} := [H'_1, \ldots, H'_{i'}]$, the *kept* constraints, and $H_{rem} := [H_1, \ldots, H_i]$, the *removed* constraints form the *head* (denoted by $H := H_{kep} + H_{rem}$) of the rule. $D := \{D_1, \ldots, D_j\}$ is called the *guard* and $B := [B_1, \ldots, B_k]$ is the body of the rule. The body consists of user-defined and built-in

constraints, whereas the head consists only of user-defined and the guard only of built-in constraints. Rules of these form are called *simpagation* rules.

If $H_{kep} = []$ then the rule is written as

$$Rulename \ @ \ H_1, \ldots, H_i \Leftrightarrow D_1, \ldots, D_j | B_1, \ldots, B_k$$

and called a *simplification rule*. If $H_{rem} = []$ then the rule is written as

$$Rulename \ @ \ H'_1, \ldots, H'_{i'} \Rightarrow D_1, \ldots, D_j | B_1, \ldots, B_k$$

and called a *propagation rule*. H_{kep} and H_{rem} must not both be empty.

2.1.2 Operational semantics

There exist several definitions of the operational semantics of CHR depending on how detailed the operations of a CHR program should be modeled. All operational semantics are based on a state transition system. The states are a combination of user-defined and built-in constraints. The transitions define how rules are applied to modify these states. Furthermore, a *Constraint Theory* CT is provided for the state transition system which can solve the built-in constraints. The starting state is defined by the *goal* which is a list of constraints. From a given starting state, the rules of the CHR program are applied continuously to the current state (using the defined transitions) until a contradiction of the built-in constraints occurs or no rules can be applied anymore. Rule application means to find constraints in the current state that *match* the constraints in a head H of a rule, so that the guard D of the rule is implied by the built-in constraints of the state under the meaning of the constraint theory CT. Informally, the constraint of a head is a pattern for which a constraint of the input is searched. A constraint from a rule's head can be matched to a constraint in the current state if their name and arity are equal and if there exists a substitution of the form $[T/c]$ or $[T/T']$ for variables T, T' and constant c, for the terms of the two constraints. For a more detailed explanation of a match, see [20].

A very detailed operational semantics, called the *refined semantics*, is used here. It is adopted from [20]. The CHR program itself is modeled as a sequence of rules and the head and body of rules are sequences of constraints. The guard of a rule is a conjunction of built-in constraints. Constraints in the heads of the rules are associated with numbers (their *occurrence*). The head constraints are numbered starting from 1 in a top-down order (from the first to the last rule) and from left to right. However, constraints in H_{rem} are numbered before constraints in H_{kep}.

A state is a tuple $\langle A, S, B, T \rangle_n^\nu$ where A, S, B, T, n, and ν have the following meaning: A is the *goal stack*, a sequence of built-in and user-defined (*unnumbered-*, *numbered-*, and *active*) constraints. S is the *CHR store*, a set of numbered constraints. B is the *built-in store*, a conjunction of built-in constraints. T is the *propagation history*, a set of tuples (r, I), where r is the name of a rule and I is the sequence of the identifiers of the numbered constraints that matched the head constraints of r. n is is the next number that can be used as an identifier for an unprocessed goal constraint. ν is the sequence of the variables of the initial goal constraints. Usually ν is omitted, because it does not change between states. An initial state is $\langle G, \emptyset, true, \emptyset \rangle_1^\nu$, where G are the unnumbered constraints of the goal and ν is the sequence of the variables of G.

The constraints in a state can be labeled in three different ways:

unnumbered constraint Constraints that are still unprocessed by the CHR program. They are denoted by their name and attributes. These constraints appear only in the goal stack.

numbered constraint User-defined constraints that have been processed and given an *identifier* to distinguish between several occurrences of the syntactically same constraint. These are denoted as $c\#i$, where i is the unique identifier of the constraint.

active constraint User-defined constraints that should only match with occurrence j of of constraint c in a given CHR program P. They are denoted as $c\#i : j$, where i is again the unique identifier of the constraint.

The function $chr(c\#i : j) = chr(c\#i) = c$ is defined to return the corresponding constraint to an active or numbered constraint. Accordingly, the function $id(c\#i) = i$ returns the id of a numbered constraint.

A state is *failed* if B contains contradictions (i.e. $CT \models \bar{\exists} B$). A state with consistent built-in constraints and empty goal stack ($A = []$) is called successful. A state is *final* if it is failed or if it is successful and no transitions can be applied anymore. Given a final state, its *answer* is $chr(S)$ and B (a failed state has always $false$ as answer, because B is inconsistent).

Figure 2.1 shows the transitions that are used by the refined semantics (for sequences, the Prolog list notation is used). In the refined semantics the constraints from the goal stack are executed from left to right. If the top-most constraint of the goal stack is a user-defined constraint (that may already be numbered), it is made an active constraint (transition **(re)activate**). Furthermore, if the constraint was not already numbered before it was activated, it is added to the CHR store as a numbered constraint with the new number n from the current state as its identifier. If the topmost constraint of the goal stack is a numbered constraint $c\#i : j$ (the occurrence j of constraint c is in rule r), if matching constraints H' for the rule r can be found in the CHR store, if the guard is implied by the built-in store under the resulting match H', and if the propagation history permits it, the rule is applied and the constraints of its body and the constraints matched to H_{kep} are added from left to right to the goal stack. The applied rule together with the identifiers of the matched constraints is then added to the propagation history (transition **apply**). The propagation history is used by the transition rule to reject the re-execution of propagation rules by comparing the constraints' identifiers of the match H' to the sequences of identifiers in the tuples of the propagation history.

If no match for the rule's head can be found with the current occurrence, the next occurrence is tested (transition **default**). However, if the active constraint could not be matched to any rule, then it is removed from the goal stack, but still remains in the CHR store (transition **drop**). If the top-most constraint in the goal stack is a built-in constraint, then it is added to the built-in store and constraints in the CHR store might be reconsidered (*woken*) if the newly added built-in from the goal stack further constrains variables of the user-defined constraint. It is then put back on top of the goal stack (transition **solve+wake**). The function *wakeUp(S,c,B)* defines the constraints that need to be reconsidered if a built-in c is added to the built-in store B.

Described in an informal way, a CHR program tries to apply rules in a top-down manner to a growing starting sequences of the goal stack. New constraints from the rule body are added during rule application to the stack. Therefore, the goal stack may grow during rule application just like function call stacks in imperative programming languages.

2.1.3 Example

In this section, I give an example for the syntax and operational semantics described above. The example is a CHR program for the computation of the greatest common divisor that

solve+wake

$\langle [c|A], S, B, T\rangle_n \mapsto_{solve+wake} \langle wakeUp(S, c, B) + A, S, B', T\rangle_n$
where c is a built-in constraint and $CT \models \forall((c \wedge B) \leftrightarrow B')$.

activate

$\langle [c|A], S, B, T\rangle_n \mapsto_{activate} \langle [c\#n : 1|A], c\#n \cup S, B, T\rangle_{n+1}$
where c is a user-defined constraint.

reactivate

$\langle [c\#i|A], S, B, T\rangle_n \mapsto_{reactivate} \langle [c\#i : 1|A], S, B, T\rangle_n$
where c is a user-defined constraint.

apply

$\langle [c\#i : j|A], H_{kep} \cup H_{rem} \cup S, B, T\rangle_n \mapsto_{apply} \langle C + H + A, H_{kep} \cup S, chr(H_{rem}) = H'_{rem} \wedge B, T \cup \{(r, id(H_{kep}) + id(H_{rem}))\}\rangle_n$
where the j^{th} occurrence of a user-defined constraint with the same symbol and arity as c is in the head $H'_{kep} \backslash H'_{rem}$ of a rule with newly created variables of the form
$H'_{kep} \backslash H'_{rem} \leftrightarrow D|B$
where $CT \models \exists(B) \wedge \forall(B \rightarrow \exists\bar{x}(chr(H_{kep}) = H'_{kep} \wedge chr(H_{rem}) = H'_{rem} \wedge D))$ and $(r, id(H_{kep}) + id(H_{rem})) \notin T$. $H = [c\#i : j]$ if the occurrence for c is in H'_{kep} and $H = []$ if it is in H'_{rem}.

drop

$\langle [c\#i : j|A], S, B, T\rangle_n \mapsto_{drop} \langle A, S, B, T\rangle_n$
if there is no occurrence j for c.

default

$\langle [c\#i : j|A], S, B, T\rangle_n \mapsto_{default} \langle [c\#i : j + 1|A], S, B, T\rangle_n$
if no other transition is possible in the current state.

Figure 2.1: Transitions for the refined semantics.

$$
\begin{array}{ll}
 & \langle [gcd(6), gcd(9)], \emptyset \rangle_1 \\
\mapsto_{activate} & \langle [gcd(6)\#1 : 1, gcd(9)], \{gcd(6)\#1\} \rangle_2 \\
\mapsto_{default} & \langle [gcd(6)\#1 : 2, gcd(9)], \{gcd(6)\#1\} \rangle_2 \\
\mapsto_{default} & \langle [gcd(6)\#1 : 3, gcd(9)], \{gcd(6)\#1\} \rangle_2 \\
\mapsto_{default} & \langle [gcd(6)\#1 : 4, gcd(9)], \{gcd(6)\#1\} \rangle_2 \\
\mapsto_{drop} & \langle [gcd(9)], \{gcd(6)\#1\} \rangle_2 \\
\mapsto_{activate} & \langle [gcd(9)\#2 : 1], \{gcd(6)\#1, gcd(9)\#2\} \rangle_3 \\
\mapsto_{default} & \langle [gcd(9)\#2 : 2], \{gcd(6)\#1, gcd(9)\#2\} \rangle_3 \\
\mapsto_{apply\ \mathbf{gcd2}} & \langle [K \text{ is } 9-6, gcd(K)], \{gcd(6)\#1\} \rangle_3 \\
\mapsto_{solve+wake} & \langle [gcd(3)], \{gcd(6)\#1\} \rangle_3 \\
\mapsto_{activate} & \langle [gcd(3)\#3 : 1], \{gcd(6)\#1, gcd(3)\#3\} \rangle_4 \\
\mapsto_{default} & \langle [gcd(3)\#3 : 2], \{gcd(6)\#1, gcd(3)\#3\} \rangle_4 \\
\mapsto_{default} & \langle [gcd(3)\#3 : 3], \{gcd(6)\#1, gcd(3)\#3\} \rangle_4 \\
\mapsto_{apply\ \mathbf{gcd2}} & \langle [K \text{ is } 6-3, gcd(K), gcd(3\#3 : 3)], \{gcd(3)\#3\} \rangle_4 \\
\mapsto_{solve+wake} & \langle [gcd(3), gcd(3)\#3 : 3], \{gcd(3)\#3\} \rangle_4 \\
\mapsto_{activate} & \langle [gcd(3)\#4 : 1, gcd(3)\#3 : 3], \{gcd(3)\#3, gcd(3)\#4\} \rangle_5 \\
\mapsto_{default} & \langle [gcd(3)\#4 : 2, gcd(3)\#3 : 3], \{gcd(3)\#3, gcd(3)\#4\} \rangle_5 \\
\mapsto_{apply\ \mathbf{gcd2}} & \langle [K \text{ is } 3-3, gcd(K), gcd(3)\#3 : 3], \{gcd(3)\#3\} \rangle_5 \\
\mapsto_{solve+wake} & \langle [gcd(0), gcd(3)\#3 : 3], \{gcd(3)\#3\} \rangle_5 \\
\mapsto_{activate} & \langle [gcd(0)\#5 : 1, gcd(3)\#3 : 3], \{gcd(3)\#3, gcd(0)\#5\} \rangle_6 \\
\mapsto_{apply\ \mathbf{gcd1}} & \langle [gcd(3)\#3 : 3], \{gcd(3)\#3\} \rangle_6 \\
\mapsto_{default} & \langle [gcd(3)\#3 : 4], \{gcd(3)\#3\} \rangle_6 \\
\mapsto_{drop} & \langle [], \{gcd(3)\#3\} \rangle_6
\end{array}
$$

Figure 2.2: Example computation of the gcd solver according to the refined semantics.

follows the algorithm of Euclid. This is called the *gcd example* in the remainder of this work. There's only one user-defined constraint

$$constraint\ gcd/1.$$

that has the arity one. There are two rules

$$gcd1 \ @\ gcd(0) \Leftrightarrow true.$$

$$gcd2 \ @\ gcd(I) \backslash gcd(J) \Leftrightarrow J \geq I, I > 0 | K \text{ is } J - I, gcd(K).$$

For these two rules in the given order, the constraint $gcd(0)$ has occurrence 1, $gcd(J)$ has 2, and $gcd(I)$ has 3. Figure 2.2 gives a sample computation with the goal $[gcd(6), gcd(9)]$. In this example, the built-in constraints and propagation history are not displayed for simplification reasons. The propagation history is not needed, because there are no propagation rules. Furthermore, the variables constrained in the built-in store are directly replaced in the goal and CHR store. As one can see, the original constraints from the store are added one after the other to the CHR store. If there are enough numbered constraints in the CHR store, so that a rule's head can match, the matching rule that comes first in the list of rules is applied and new constraints are added to the top of the goal stack.

2.2 Graph Transformation Systems (GTS)

The concept of *Graph Transformation Systems (GTS)* dates back to the late 1960s when it became a generalization of rewriting techniques for trees. The basic idea of GTS is the rule-based modification of graphs. This section describes the basic definitions to un-

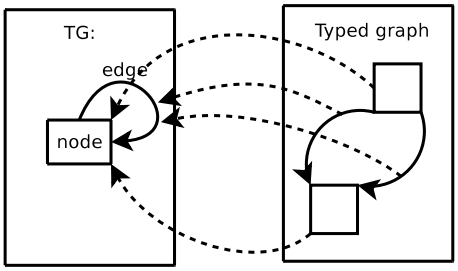


Figure 2.3: Type graph with a typed graph. The dotted arrows represent the graph morphism from the typed to the type graph.

derstand GTS. The theoretical basis on which GTS are described here is *category theory* which is described in appendix B. The definitions and notations are adopted from [18]. The example illustrations represent a GTS which finds circular lists in graphs as described in section 1.3.1.

2.2.1 Definitions: Graphs

A *graph* is a tuple $G = (V_G, E_G, src_G, tgt_G)$ with the sets of nodes (V_G) and edges (E_G) and two functions $src_G, tgt_G : E_G \rightarrow V_G$ which specify the source and target nodes of the edges in G. The degree of a node is further defined as $deg_G : V_G \rightarrow \mathbb{N}$, $v \mapsto |\{e \in E_G | src_G(e) = v\}| + |\{e \in E_G | tgt_G(e) = v\}|$. If the graph can be deducted from the context, the subscripts are omitted. A *graph morphism* $f = (f_V, f_E) : G_1 \rightarrow G_2$ between two graphs G_1 and G_2 is a tuple of two functions $f_V : V_{G_1} \rightarrow V_{G_2}$ and $f_E : E_{G_1} \rightarrow E_{G_2}$ that preserve the source and target functions. That means $f_V \circ src_{G_1} = src_{G_2} \circ f_E$ and $f_V \circ tgt_{G_1} = tgt_{G_2} \circ f_E$. A *type graph* TG is a graph with uniquely labeled nodes and edges. A *typed graph* is a tuple $G^{TG} = (G, TG, type_G^{TG})$ consisting of a graph G, a type graph TG, and a graph morphism $type_G^{TG} : G \mapsto TG$. A *typed graph morphism* $f^{TG} : G_1^{TG} \mapsto G_2^{TG}$ is defined as a graph morphism that fulfills $type_{G_1}^{TG} \circ f^{TG} = type_{G_2}^{TG}$. If the type graph can be deducted from the context, the superscript is omitted and a typed graph is denoted as $G = (V_G, E_G, src_G, tgt_G, type_G)$.

In figure 2.3 an example of a type graph and a typed graph can be seen. The type graph has uniquely labeled nodes and edges while the typed graph has no labels. The dotted arrows represent the graph morphism from the typed graph to the type graph. The type graph consists only of a node with a loop. That means, that the nodes in an according typed graph can be connected arbitrarily by edges.

2.2.2 Definitions: Graph transformation system

A *graph transformation system* $gts = (P, TG)$ is a tuple consisting of a set of typed graph production rules P and a type graph TG. A *typed graph production rule* (or just *rule* if the context is clear) is a tuple of the form $p = (L \overset{l}{\leftarrow} K \overset{r}{\rightarrow} R)$ where L, K, and R are typed graphs (called the *rule graphs*) and $l = K \rightarrow L$ and $r = K \rightarrow R$ are injective typed graph morphisms (the superscripts for the type graph are omitted, because it results from the context). Given a typed graph production $p = (L \overset{l}{\leftarrow} K \overset{r}{\rightarrow} R)$, a typed graph G (called the *host graph*), and a typed graph morphism $m : L \rightarrow G$ (the *match*), the application of p to G via m ($G \overset{p,m}{\Rightarrow} H$) is called a *typed graph transformation*. Intuitively,

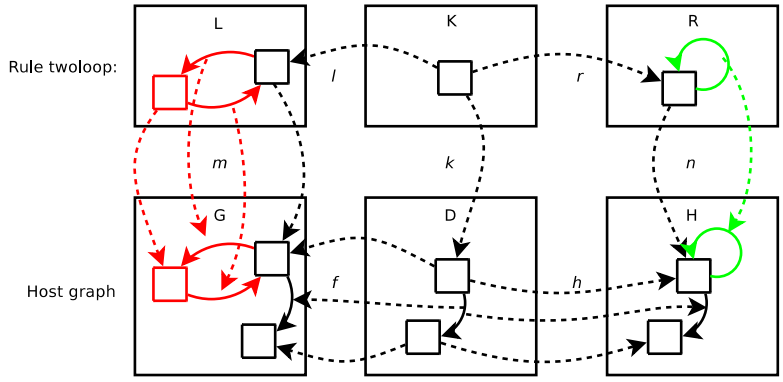


Figure 2.4: Typed graph transformation. The upper row shows the graphs L, K, and R of the production rule $twoloop$. The lower row shows a host graph that is modified by the rule.

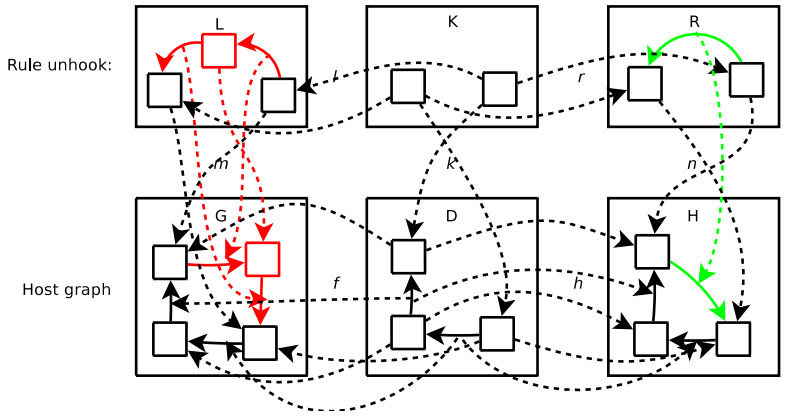


Figure 2.5: Typed graph transformation of the rule *unhook* from the circular list example.

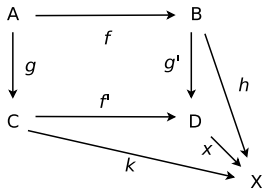
when L is matched (via the match morphism m) to a sub-graph of G, then all vertices and nodes in $G \backslash m(L \backslash l(K))$ are removed from G resulting in an intermediate graph D (called the *gluing graph*). After that, new nodes and edges are added from $R \backslash r(K)$, so that the resulting graph $H = D \uplus (R \backslash r(K))$ is the disjoint union of the graph D and the added nodes and edges in R. Figure 2.4 gives an example of a typed graph transformation. The top row shows the rule, the bottom row its application to a host graph. The L graph of the rule matches to a sub graph, consisting of two nodes connected to each other. One of the two nodes is removed together with the edges and a loop is added to the remaining node. Elements that are removed are displayed in red in the L graph and elements that are added are green in the R graph. The dotted lines between the upper and lower left graph is the match m. The rule and transformation example shown in figure 2.5 removes the middle node of a list of three nodes connected in a row. A new edge is inserted between the remaining two nodes. These two rules represent a GTS for finding circular lists in a graph which is explained in section 1.3.1.

2.2.3 Double pushout as a gluing construction

The informal description of a typed graph transformation given above corresponds to the category theoretical principle of two pushouts, a so called *double pushout (DPO)*. A short introduction to category theory can be found in appendix B.

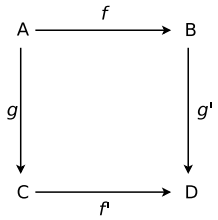
A category $C = (Obj_C, Mor_C, \circ, id)$ is a tuple and its elements have the following meaning: Obj_C is a class of objects. For each tuple $A, B \in Obj_C$ there is a set $Mor_C(A, B)$ of morphisms. $\circ$ is the concatenation of morphisms in Mor_C with $Mor_C(B, D) \circ Mor_C(A, B) \rightarrow Mor_C(A, D)$ and $id_A \in Mor_C(A, A)$ is the identity morphism for every $A \in Obj_C$. A morphism $f \in Mor(A, B)$ can also be written as $f : A \rightarrow B$.

Informally written, a pushout is the gluing of two objects in a category along a common object. Given $f : A \rightarrow B$ and $g : A \rightarrow C \in Mor_C$, then a *pushout (PO)* (D, f', g') is defined by a *pushout object* D and morphisms $f' : C \rightarrow D, g' : B \rightarrow D$ with $f' \circ g = g' \circ f$ such that the following universal property is fulfilled: For all objects $X \in Obj_C$ with morphisms $h : B \rightarrow X$ and $k : C \rightarrow X$ with $k \circ g = h \circ f$ there is a unique morphism $x : D \rightarrow X$ such that $x \circ g' = h$ and $x \circ f' = k$.



In category theory the category **GraphsTG** consists of typed graphs (over the type graph TG) as its objects, typed graph morphisms as its morphism and the concatenation of morphisms as concatenation operator.

The detailed definition of a pushout for the category **GraphsTG** can be found in appendix B. The pushout



in **GraphsTG** can be constructed by creating equivalence classes for the nodes and edges in B and C where two nodes or edges $b \in B$ and $c \in C$ are equivalent if $b = f(a)$ and $c = g(a)$. The morphisms f' and g' can then be defined as $f'(c) = [c]\ \forall c \in C$ and $g'(b) = [b]\ \forall b \in B$ (where $[c]$ and $[d]$ denote the according equivalence classes). This pushout construction fulfills the following properties:

1. If f is injective (or surjective), then f' is injective (surjective)
2. The pair (f', g') is jointly surjective (for every element in D there exists a preimage in C or B)
3. If f is injective and $x \in D$ has preimages $b \in B$ and $c \in C$ with $g'(b) = f'(c) = x$, then there is a unique preimage $a \in A$ with $f(a) = b$ and $g(a) = c$.

4. If f (and hence also f') is injective, then D is isomorphic to $D' = C \uplus B \backslash f(A)$.

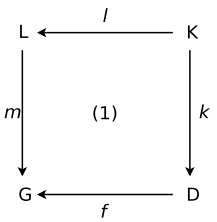
These properties can easily be proved ([18]): The first property can be shown with the properties of the equivalence relation. The second property is valid due to the construction of D, which also implies property three. For the last property, one can show that D' is isomorphic by giving a bijection $b : D' \to D$. $b(x) = [x]$ is such a bijection.

The possibility of constructing a pushout in the category **GraphsTG** can be used to check if a typed graph production rule $p = (L \overset{l}{\leftarrow} K \overset{r}{\rightarrow} R)$ can be applied to a host graph G via a match $m : L \to G$. The rule p is applicable to the typed graph G if there exists a valid context graph D, so that K together with $l : K \to L$ and $n : K \to D$ forms a pushout as described above. However, this definition provides no procedure to check whether a rule can be applied to a host graph. To apply a rule to a host graph, the context graph D must be a proper graph. An improper graph is a graph which contains edges, that are not connected to a node. These edges are called *dangling edges*. The *gluing condition* defines a criterion that must be fulfilled, so that a rule can be applied. Given the production rule p, the host graph G, and a match morphism $m : L \to G$, the following three sets are defined

1. The *gluing points* GP are those nodes and edges in L that are not deleted by p, i.e. $GP = l(K)$
2. The *identification points* IP are those nodes and edges in L that are identified by m, i.e. $IP = \{v \in V_L | \exists w \in V_L, w \neq v : m_V(v) = m_V(w)\} \cup$ $\{e \in E_L | \exists f \in E_L, f \neq e : m_E(e) = m_E(f)\}$
3. The *dangling points* DP are those nodes in L whose images under m are the source or target of an edge in G that does not belong to $m(L)$, i.e. $DP = \{v \in V_L | \exists\, e \in E_G \backslash m_E(E_L) : src_G(e) = m_V(v)$ or $tgt_G(e) = m_V(v)\}$

p and m satisfy the gluing condition if $DP \cup IP \subseteq GP$, i.e. a rule can only be applied if all dangling points and all identification points are also gluing points. However, Raiser ([32]) provides a simpler definition that covers only injective matches for nodes and edges, i.e. $m_V(v) \neq m_V(w) \forall v, w \in V_L$ and $m_E(e) \neq m_E(f) \forall e, f \in E_L$. The gluing condition can then be posted in the simpler form $DP \subseteq GP$, because there are no identification points. Furthermore, Raiser describes that non-injective matches can be simulated in the following way: Given the rule $p = (L \overset{l}{\leftarrow} K \overset{r}{\rightarrow} R)$ and a non-injective match $m : L \to G$ for which $m(v) = m(w), v \neq w, v, w \in V_L$, a new rule p' can be introduced, in which the nodes v and w are merged to a new node v_w in all three graphs of the rule. Therefore, the non-injective match $m(v) = m(w)$ is simulated by matching v_w to $m(v_w)$ where $m(v_w)$ is the same node as $m(v)$ (and $m(w)$) in G. Consequently, new rules can be introduced for all combinations of possible non-injective matchings simplifying the gluing condition to the form above.

Lemma 1. (*existence of the context graph*) For a typed graph production rule $p = (L \overset{l}{\leftarrow} K \overset{r}{\rightarrow} R)$, a typed graph G, and a match $m : L \to G$, the context graph D with the PO (1) exists if and only if the gluing condition is satisfied.



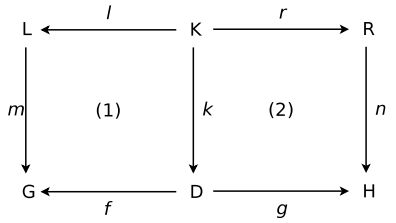


Figure 2.6: Double pushout for a rule $p = (L \overset{l}{\leftarrow} K \overset{r}{\rightarrow} R)$ and a host graph G.

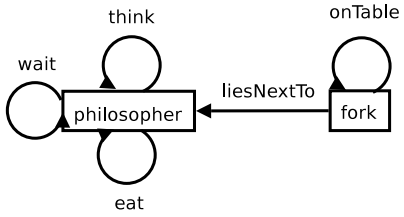


Figure 2.7: Type graph for the GTS representation of the dining philosophers problem.

The proof can be found in [18] on pages 45 to 46

The lemma stated above provides a way of constructing a direct graph transformation ([18] pp. 46–47). Given a production rule $p = (L \overset{l}{\leftarrow} K \overset{r}{\rightarrow} R)$, a typed graph G, and a match $m : L \rightarrow G$, the context graph D can be constructed as follows: $D = (G \backslash m(L)) \cup m(l(K))$ with the morphism $k(K) = m(l(K))$ and the morphism f is an inclusion. In a second step, the final pushout graph H can be constructed from K, $n : K \rightarrow D$, and $r : K \rightarrow R$ in the category **GraphsTG**. Figure 2.6 shows the two pushouts (1) and (2) that are used for a graph transformation.

Figure 2.4 shows an example for the construction of D and H. To construct D, the graph G is taken and $m(R \backslash l(K))$ is removed from it (the elements marked red). To the intermediate graph D, the elements of $R \backslash r(K)$ are added (marked green in the figure) to get the final graph H.

2.2.4 Example

Another example for a GTS can be created from the dining philosophers problem described in section 1.3.2. The type graph shown in figure 2.7 describes the nodes and edges that are needed. The meaning of the nodes is obvious, the edge between the philosopher and the fork means that this fork lies next to the philosopher. If the loop on the fork is present, then the fork is lying on the table, i.e. the loop is removed when a philosopher is eating with this fork. The loops on a philosopher describe the states he can be in: thinking, eating, or waiting. The loops on the type graph's nodes are also intended to be loops in a typed graph in order to describe the state of a node. Therefore, the type graph does not ensure that the typed graphs are valid instances of the dining philosophers problem. For example, the loops on the type graph's nodes can also result in edges connecting two different philosopher nodes in the typed graph. Figure 2.8 shows a correct instance of the dining philosophers problem with five philosophers where one philosopher is eating and one is waiting. One fork is between every two philosophers which is connected to each of them. No two eating

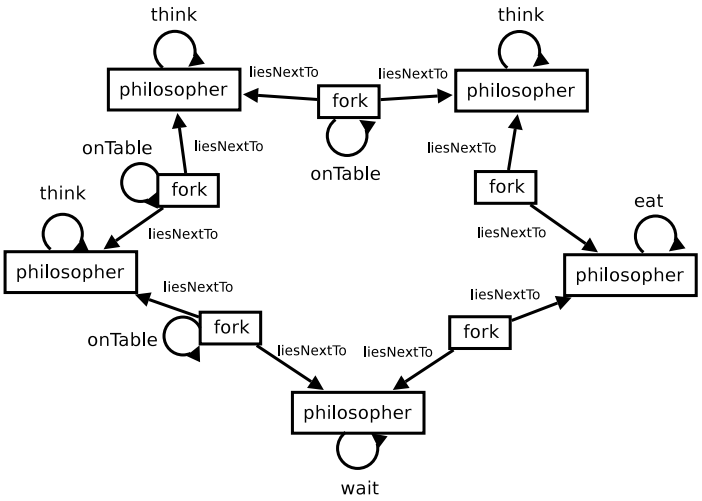


Figure 2.8: Typed graph for the dining philosophers problem with five philosophers and one philosopher eating.

philosophers are next to each other. The two *onTable* loops on the forks are missing where the philosopher is eating and each philosopher has exactly one loop.

To describe the change between the different states of the philosophers (eating, thinking, and waiting), three rules have to be introduced. The rules are designed in a way, so that no deadlocks can occur, i.e. as soon as two forks are available, they are both taken simultaneously. There are three rules in this GTS that describe the state transition of the philosophers from thinking to waiting, from waiting to eating, and from eating back to thinking. The rules are shown in figure 2.9. The first rule *thinkToWait* removes the think loop from a philosopher node and adds a wait loop. The second rule *waitToEat* removes the *onTable* loops from the forks that are connected to the philosopher, as well as the wait loop from the philosopher and adds a new eat loop to him which symbolizes the philosopher taking up the forks and starting to eat. The last rule *eatToThink* adds the *onTable* loops to the two forks connected to the philosopher, removes the eat loop, and adds the think loop to the philosopher. This means he stops eating, puts back his forks, and starts thinking again.

2.2.5 Notation

There exist several notations for the rules of GTS. The notation with three graphs K, L, and R, used in this section, is good for describing the DPO approach for rule applications. Another approach is to use only two graphs L' and R' for a rule and define a partial morphism $k' : L' \to R'$ that connects the nodes and edges that are connected via the K graph in the three graph notation. Figure 2.10 shows the *unhook* rule of the circular list GTS. The middle graph K is replaced by a single morphism $k' : L' \to R'$. For both, the three and two graph notation, the coloring of the nodes and edges in the graphs is only a visual help to see the nodes and edges that are not an image or preimage of the morphisms l, r, and k'. Another even more compact notation only uses one graph, but in this notation the coloring is important. The graph K of the three graph notation is displayed in black while the nodes and edges in the graphs L and R that are not part of the image of the morphisms l and r are displayed in red and green, respectively. As the rule is presented in one graph, no explicit morphisms are needed anymore. Figure 2.11 also shows the *unhook* rule, but this time in the one graph notation. The one graph notation is used for the remainder of

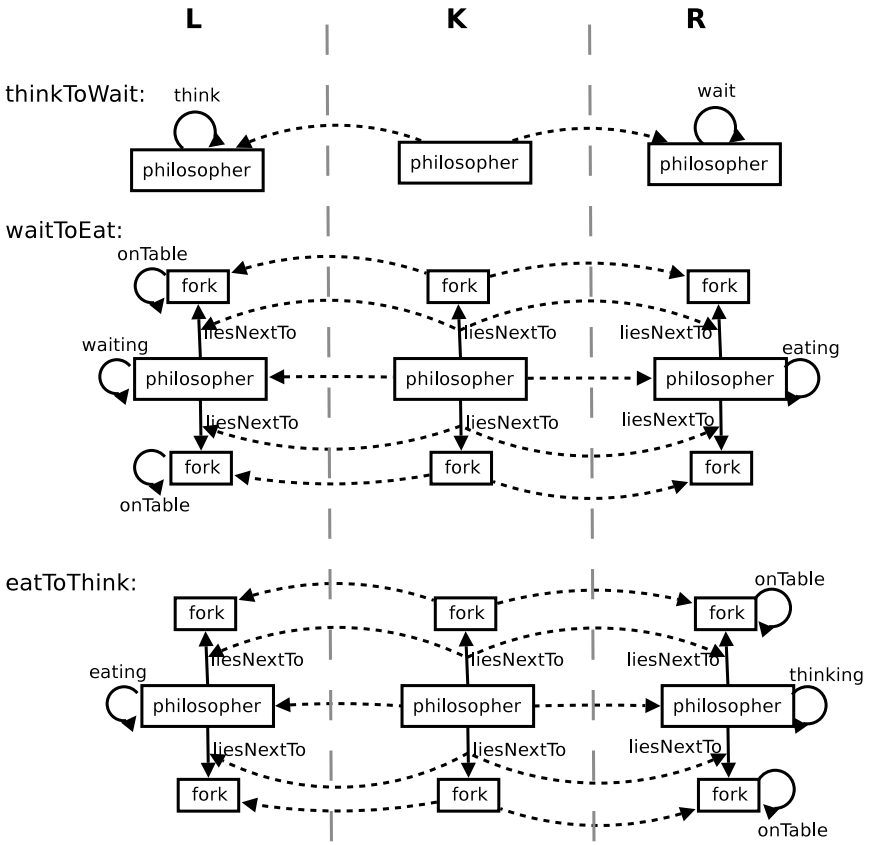


Figure 2.9: The rules *thinkToWait*, *waitToEat* and *eatToThink* of the GTS for the dining philosophers problem.

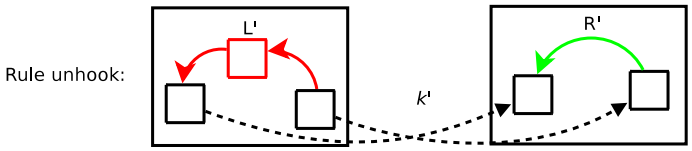


Figure 2.10: *Two graph* notation of the rule *unhook* of the circular list GTS.

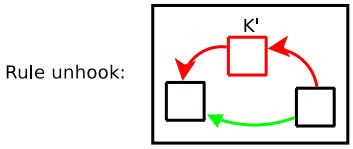


Figure 2.11: *One graph* notation of the rule *unhook* of the circular list GTS.

this thesis, because it provides a simple and compact way of representing GTS rules. To distinguish the nodes and edges in this graph, the symbols L', R', and K are used for red, green, and black nodes and edges of the graph. In the three graph notation L' is $L \backslash l(K)$, R' is $R \backslash r(K)$ and K is the K graph.

2.3 CHR encoding of GTS

Section 2.3.1 describes how to encode a GTS to CHR in general. Section 2.3.2 shows the encoding of the GTS for the examples introduced in section 1.3.

2.3.1 Encoding

The encoding of a GTS in CHR is adopted from [32]. To embed a GTS in CHR, an encoding must be provided for the typed graph production rules and for typed graphs in general. Production rules can be encoded as CHR rules and graphs in general as a sequence of constraints that represent the nodes and edges of the graph. As described in section 2.2, a GTS consists of three types of graphs: A type graph, multiple rules, and multiple host graphs. Let $gts = (P, TG)$ be a GTS with the type graph $TG = (V_{TG}, E_{TG}, src_{TG}, tgt_{TG})$ and a set of rules P which contains rules of the form $p = (L^{TG} \overset{l}{\leftarrow} K^{TG} \overset{r}{\rightarrow} R^{TG})$ that are typed over TG and let host graphs be of the form $G = (V_G, E_G, src_G, tgt_G, type_G)$ which are also typed over TG. The elements of the type graph provide the constraint definitions of the CHR program: for every node $v \in V_{TG}$ a constraint definition $v/2$ is added, and for every edge $e \in E_{TG}$ a constraint definition $e/3$ is added to the CHR program.

For the encoding of typed graphs, $[type_G(x)]$ is defined as the name of the type of the node or edge x in a typed graph. The functions $dvar, var : G \rightarrow \mathit{VARS}, x \mapsto X_x$ map nodes and edges to variables (*VARS* is an infinitely large set of variable names), so that X_x is a unique variable name associated with the node or edge x of a typed graph. With these mappings, the following encoding from nodes and edges of a typed graph to CHR constraints is defined:

$$chr_G(host, x) = \begin{cases} [type_G(x)](var(x), deg_G(x)) & , x \in V_G \\ [type_G(x)](del, var(src_G(x)), var(tgt_G(x))) & , x \in E_G \end{cases}$$

$$chr_G(keep, x) = \begin{cases} [type_G(x)](var(x), dvar(x)) & , x \in V_G \\ [type_G(x)](dvar(x), var(src_G(x)), var(tgt_G(x))) & , x \in E_G \end{cases}$$

A constraint that encodes a node is called a node constraint, and a constraint that encodes an edge is called an edge constraint. The encoding of a complete graph is the encoding of all the nodes and edges which are added to a sequence:

$$chr(host, G) = [chr_G(host, x_1), \ldots, chr_G(host, x_n)]$$

and

$$chr(keep, G) = [chr_G(keep, x_1), \ldots, chr_G(keep, x_n)]$$

where $x_1, \ldots x_n$ are the nodes and edges of G. The first attribute of a node constraint is the *identifier attribute*, the second argument is the *degree* (for the *host* encoding) or the *degree variable* (for the *keep* encoding). The edge constraints have three attributes. The first one is the *deletion* attribute that either contains a variable or a ground term. For *host* encodings, it is always a ground term (the constant del is used here). For *keep* encodings it is a variable. The second and third argument contains the identifiers of the node constraint they are connected to. These attributes are called *source* and *target attribute*. Therefore,

their values are called source and target identifiers. If a host graph G is encoded with $chr(host, G)$, the resulting sequence can be used as a list of goal constraints as input for a CHR program. For each rule $p = (L \xleftarrow{l} K \xrightarrow{r} R)$ of a GTS, a new CHR rule of the form

$$p \mathbin{@} C_L \Leftrightarrow C_R$$

with

$$\begin{aligned} C_L = \quad & \{chr(host, x) | x \in L \backslash K\} \cup \{chr(keep, x) | x \in K\} \\ C_R = \quad & \{chr(host, x) | x \in R \backslash K\} \cup \{chr(keep, e) | e \in E_K\} \cup \\ & \{chr(keep, v'), var(v') = var(v), \\ & dvar(v') = dvar(v) - deg_L(v) + deg_R(v) | v \in V_K\} \end{aligned}$$

is defined. This means that elements that are removed and added during a rule application are encoded as *host* elements and put into the head and body of the rule, respectively. Edges that are kept ($e \in E_K$) in the graph during the transformation are encoded as *keep* elements and put in the head and body. Each node that is kept ($v \in V_K$), is encoded as *keep* constraint in the head and in the body. In the body, however, a new node v' is introduced, together with built-in constraints that set the identifier of the nodes v and v' equal. A built-in constraint, to set the degree variable v' to the difference of the degrees of the graphs R and L is also added.

The rule *twoloop* (figure 2.4) from the GTS for finding circular lists in a graph is encoded as

$$\begin{aligned} twoloop \mathbin{@} \quad & node(N_0, Deg_0), node(N_1, 2), edge(del, N_0, N_1), edge(del, N_1, N_0) \\ & \Leftrightarrow node(N_0', Deg_0'), N_0' = N_0, Deg_0' = Deg_0 + 0 - 0, edge(del, N_0, N_0). \end{aligned}$$

This notation can be simplified to

$$\begin{aligned} twoloop \mathbin{@} \quad & node(N_0, Deg_0), node(N_1, 2), edge(del, N_0, N_1), edge(del, N_1, N_0) \\ & \Leftrightarrow node(N_0, Deg_0 + 0 - 0), edge(del, N_0, N_0). \end{aligned}$$

by replacing the variables introduced for v' with the variable and expression from the built-in constraints. This notation is used in the remainder of this thesis.

This encoding ensures that rules can be applied to the goal constraints if and only if the according graph production rule can be applied to the according host graph. More detailed information about this encoding and proofs for its soundness and completeness can be found in [32]. With these results, a GTS can be simulated by encoding it to a CHR program. Rule applications in this CHR program correspond to rule applications in the GTS. Therefore, if the input is a valid encoding of a graph, the constraint store also encodes a valid graph after rule applications.

2.3.2 Examples

The GTS for modeling the dining philosophers problem is described in section 2.2. The GTS consists of the three rules *thinkToWait*, *waitToEat* and *eatToThink*. The type graph can be encoded to the constraint definitions

$$philosopher/2, fork/2, think/3, wait/3, eat/3, onTable/3, liesNextTo/3$$

The rules are encoded (in the simpler notation) as

$$
\begin{aligned}
P_{phil} = [thinkToWait\ @\quad & philosopher(P, Deg_P), think(del, P, P) \\
& \Leftrightarrow philosopher(P, Deg_P), wait(del, P, P). \\
waitToEat\ @\quad & philosopher(P, Deg_P), fork(F_1, Deg_{F_1}), \\
& fork(F_2, Deg_{F_2}), liesNextTo(Del_0, F_1, P), \\
& liesNextTo(Del_1, F_2, P), onTable(del, F_1, F_1), \\
& onTable(del, F_2, F_2), wait(del, P, P) \\
& \Leftrightarrow philosopher(P, Deg_P), \\
& fork(F_1, Deg_{F_1} - 2), fork(F_2, Deg_{F_2} - 2), \\
& liesNextTo(Del_0, F_2, P), liesNextTo(Del_1, F_1, P), \\
& eat(del, P, P). \\
eatToThink\ @\quad & philosopher(P, Deg_P), fork(F_1, Deg_{F_1}), \\
& fork(F_2, Deg_{F_2}), liesNextTo(Del_0, F2, P), \\
& liesNextTo(Del_1, F1, P), eat(del, P, P) \\
& \Leftrightarrow philosopher(P, Deg_P), \\
& fork(F_1, Deg_{F_1} + 2), fork(F_2, Deg_{F_2} + 2), \\
& liesNextTo(Del_1, F_2, P), onTable(del, F_2, F_2), \\
& liesNextTo(Del_0, F_1, P), onTable(del, F_1, F_1), \\
& think(del, P, P).]
\end{aligned}
$$

In this GTS no nodes are added or removed. Because of that, all degree attributes in the heads are variables. The degree attributes of the bodies are expressions. The number that is added or subtracted from the variable represents the change in the number of edges that are connected to this node. In the $waitToEat$ rule, for example, which encodes the GTS rule $p = (L \overset{l}{\leftarrow} K \overset{r}{\rightarrow} R)$, the node encoded by $fork(F_1, Deg_{F_1})$ has a degree of three in the L graph and a degree of one in the R graph. Therefore, the constraint in the body of the CHR rule is $fork(F_1, Deg_{F_1} - 2)$, because the degree is decreased by two during rule application. The deletion variables of the edges that are removed or added are set to del while the edges that are kept receive a unique variable (Del_0 and Del_1). A host graph consisting of two philosophers with two forks where one philosopher is eating while the second one is waiting can be expressed with the list of constraints:

$$
\begin{aligned}
&TwoPhilos := [philosopher(p_1, 4), fork(f_1, 2), fork(f_2, 2), philosopher(p_2, 4), \\
&liesNextTo(del, f_1, p_2), liesNextTo(del, f_2, p_2), \\
&liesNextTo(del, f_1, p_1), liesNextTo(del, f_2, p_1), \\
&eat(del, p_2, p_2), wait(del, p_1, p_1)]
\end{aligned}
$$

The according starting state (compare section 2.1) of a the CHR program $Phil = (P_{phil})$ would be

$$\langle TwoPhilos, \emptyset, true, \emptyset \rangle_1$$

The rule *eatToThink* can be applied when all but the last constraint are added to the CHR store. After that, the graph encoded in the goal stack and CHR store contains again the two philosopher and fork nodes together with the edges from forks to the philosophers. The $eat(del, p_2, p_2)$ constraint is replaced by a $think(del, p_2, p_2)$ attribute, and two $onTable$ constraints are added for the $fork$ constraints.

The GTS for finding circular lists in graphs introduces the two constraint definitions $node/2$ and $edge/3$. The two rules are encoded as

$$\begin{array}{ll} unhook\ @ & node(N_0, Deg_{N_0}), node(N_1, 2), node(N_3, Deg_{N_3}), \\ & edge(del, N_0, N_1), edge(del, N_1, N_3) \\ & \Leftrightarrow node(N_0, Deg_{N_0}), node(N_3, Deg_{N_3}), edge(del, N_0, N_3). \\ twoloop\ @ & node(N_0, Deg_{N_0}), node(N_1, 2), edge(del, N_0, N_1), edge(del, N_1, N_0) \\ & \Leftrightarrow node(N_0, Deg_{N_0}), edge(del, N_0, N_0). \end{array}$$

In these rules, nodes are removed. For that reason, the degree attribute of the removed nodes is the number of edges that are connected to them (two in both rules). All edges that appear in the rules are removed or added. Therefore, they are all host encoded and contain the ground term del.

2.4 Eclipse

This section covers a short introduction to the eclipse platform and its architecture. Besides that, I describe shortly how to contribute plug-ins and define new extension points. Further information can be found in [22].

2.4.1 The platform

The Eclipse Platform is designed for hosting integrated development environments (IDEs) and arbitrary tools. It is written in Java. As can be seen in figure 2.13 the eclipse platform is split up into multiple layers.

An important concept of eclipse is its plug-in structure. Except a small core (the *Runtime Platform*), every functionality of eclipse is realized via plug-ins. Plug-ins define *extensions* and *extension points*. Basically, extensions are pieces of functionality that are plugged into extension points. When defining an extension point, at least an identifier must be provided, so that extensions can refer to this extension point. Furthermore, multiple attributes can be defined. Often a java class name is given that represents the code of the extension to be loaded. In most cases this class is requested to implement a certain interface, so that other extensions can communicate with this new one. Extensions implement the definition of the extension point, as well as the attributes requested by it.

The *Runtime Platform* is mainly responsible for finding all plug-ins at start-up and managing them. Plug-ins are usually not loaded when starting the platform, but they are dynamically loaded when they are needed. All other elements in figure 2.13 are plug-ins that are loaded on start-up by the Runtime Platform.

The *Workspace* is the root for the files and folders that can be used with the plug-ins loaded in the platform. It consists of multiple projects which are mapped to folders on the file system. Every project in a workspace can have multiple *natures* defining its properties, e.g. a java project has the *JavaCore* nature which provides java source folders and automated compilation of the sources. All files and folders in the workspace are so called *resources* which are a more fundamental concept than files and folders, because they let you extend the functionality of the resource elements. One example for this is the so called *marker mechanism* for annotations: Resource elements can be marked with annotations, e.g. compilation errors or warnings which are displayed in a file browser and in editors. Another example is the possibility to add resource listeners that get notified when the resource changes.

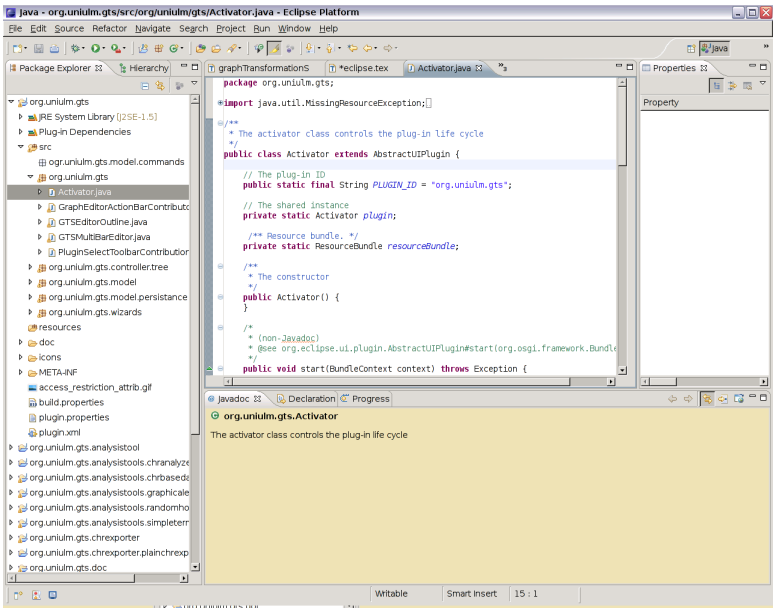


Figure 2.12: Screen-shot of the eclipse workbench.

All the elements described above are completely independent from a *graphical user interface (GUI)*. The *Workbench* is the framework for GUIs in eclipse (which is a plug-in itself). The workbench is based on the *Standard Widget Toolkit (SWT)* for drawing basic components and *JFace*, a GUI toolkit which is based on SWT and offers a lot of classes for handling many common GUI programming tasks. SWT comes in many versions that use different underlying window systems and defines itself an API which is independent from the underlying system. JFace provides components for imaging, font registry, dialogs, preferences, wizards, and progress reporting for long-running operations. The workbench itself supplies the structures in which tools interact with the user. The eclipse platform GUI paradigm is based around *views*, *editors*, and *perspectives*. While views and editors are elements visible to the user, the perspective gives information about the constellation of the elements. Editors are tools for editing and displaying files while views are tools for viewing objects currently worked with in the workbench. The eclipse platform takes care of integrating, initializing, and destroying editors and views. Further basic features of the eclipse platform is the *team support* and the *help* which are not further considered here. The Screen-shot in figure 2.12 shows the java perspective with an editor and multiple views.

2.4.2 Plug-ins and extension points

Creating a plug-in in eclipse is done as follows: The user defines two files, `plugin.xml` and `manifest.mf`. The plug-in file is an *Extensible Markup Language (XML)* file, in which the user describes all the extensions in his plug-in. For every extension, a unique identifier must be given to identify the given extension, as well as the identifier of the extension point being extended. Furthermore, the attributes described by the extension point definition must be added. The most common attribute is the *class* attribute which provides the extension with the information about what class should be loaded when the extension is activated. This class must also implement the interface given in the description of the extension point. Defining an *extension point* works in a similar manner. In the `plugin.xml`

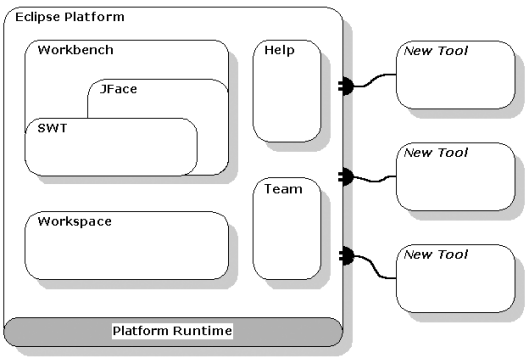


Figure 2.13: Overview of the components of the eclipse platform (from `http://www.eclipse.org/articles/Whitepaper-Platform-3.1/eclipse-platform-whitepaper.html`).

```
<plugin>
<extension
         id="example.exampleview"
         point="org.eclipse.ui.views">
      <view
            class="example.Example"
            icon="icons/sample.gif"
            id="example.ExampleView"
            name="Example View">
      </view>
   </extension>
   <extension-point
               id="example.exampleExtensionPoint"
               name="An exampe extension"
               schema="example.example.exsd"/>
</plugin>
```

Listing 2.1: A `plugin.xml` example.

file, a tag must be inserted that describes the extension point. The *schema* file is a XML file that describes the attributes an extension must provide, as well as a textual description of the extension point for documentation. Listing 2.1 shows an example plug-in file for an extension of the `org.eclipse.ui.views`. The *extension* tag is used to create a new extension by providing the *id* attribute and the *point* attribute which specifies the identifier of the used extension point. The definitions for the extension point `org.eclipse.ui.views` states that there must be a *view* tag with several attributes. In the class attribute, a class must be given that implements the `org.eclipse.ui.IViewPart` interface (this information is also contained in the extension point definition). The listing also shows an extension point which is defined by an *extension-point* tag. This tag requires a unique *id*, so that all extensions to this point can be found, a *name* that describes the extension point, and a *schema* file, which describes the extension point. The schema file contains the structure of the XML tags for the extensions (like the *view* tag described above). Usually, a java interface is provided together with an extension point, that must be implemented by the extensions. This interface is then used to communicate with the extensions. The schema file lists which interface must be implemented by the java class that is given in the *class* attribute.

The `manifest.mf` file describes the packages that are exported by the plug-in and the dependencies of this plug-in. Besides that, it defines the execution environment and other information needed to properly manage this plug-in.

3 Conception

In this chapter, I present already available programs for editing and analyzing GTS in section 3.1. Furthermore, I analyze the goals mentioned in section 1.2 and refine them further in section 3.2. For these goals, I make a requirements analysis in section 3.3. In section 3.4, I present possible tools that can be used for the realization of the project and describe which tools have been chosen for realization.

3.1 Available GTS tools

In this section, I analyze the tools *Attributed Graph Grammar (AGG)* ([18]) and *Graphs for Object-Oriented Verification (Groove)* ([34]) that have already been mentioned in the introduction to give a conclusion about the pros and cons of each tool. I analyze the usability, available analysis features, the possibility to extend the tools, and whether they can be used in other applications. These topics are chosen for analysis, because they are also the main features of the platform developed during this thesis.

3.1.1 AGG

AGG features a user interface divided into four parts. On the left is a list of rules, host graphs, and the type graph (the graph pane). On the right is a list of the type nodes and type edges (type pane). In the middle are three fields. The upper two fields are for drawing rule graphs (rule pane) in a two graph notation (compare section 2.2.5) and the lower one is for drawing the type or host graph, respectively (host or type pane). Before creating a type graph, one must define the nodes and edges in the list on the right side. See also the screen-shot in figure 3.1.

When creating edges in host and rule graphs, there is no visual feedback if the connection is valid, according to the type graph, until the user finishes the connection. If the connection is invalid, an error message dialog pops up informing about the problem. Furthermore, the nodes must be mapped manually between the rules's left and right side. The left side can also be copied over to the right, then the mapping is done automatically. When deleting nodes or edges, a dialog pops up to confirm the deletion.

AGG can apply the given rules automatically to a host graph or by defining the matches manually. When selecting matches manually, they can also be auto-completed. There is also the possibility of undoing transformation steps. Another feature is the non-deterministic transformation mode where rules are applied in a non-deterministic mode. One analysis features of AGG is the critical pair analysis for checking confluence. Confluence is the property that rules can be applied in arbitrary order and the final result, when no rule can be applied anymore, is always the same. The other feature is to test the applicability of a rule sequence to a host graph. The advanced features are the creation of attributed GTS that let you assign attributes to edges and nodes. Furthermore, it supports graph constraints, to further restrict the applicability of rules, as well as positive and negative application conditions. There are additional analysis methods for these advanced features, but they are not discussed here (see [18] for more information about these topics), because this is beyond the scope of this work. AGG provides no functionality to add further plug-ins for analyzing

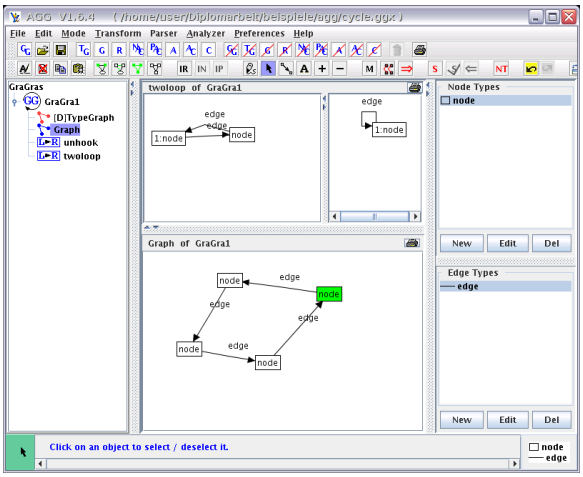


Figure 3.1: Screen-shot of AGG.

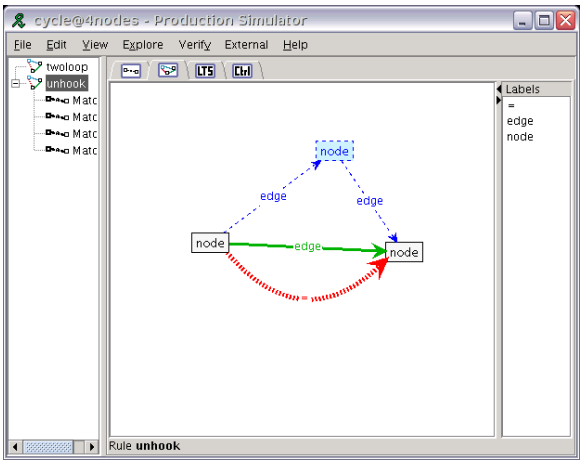


Figure 3.2: Screen-shot of Groove.

or editing a GTS. But the kernel of AGG (without GUI) can be used in other applications. Therefore, an API for creating GTS is provided, so that rules can be applied to host graphs in arbitrary applications.

3.1.2 Groove

Groove provides a similar appearance to AGG. On the left side is a list of rule graphs with their matches on the host graph as sub-elements. There are no type graphs in Groove, nodes and edges can only be labeled with strings. Therefore, there is no consistency check for the validity of the graph. On the right is a list of labels which can be used for nodes and edges (label list). In the middle is a tabbed pane with four tabs, one for the host graph (this is called the start graph), one for the rules, one for the exploration of all the graphs that can be created with the given rules and one for analysis purposes. See also the screen-shot in figure 3.2.

The graphs cannot be directly edited in this middle pane, but an editor can be started from there. This editor is rather simple, it lets you create nodes and edges and add labels. The labels are not imported from the other graphs in the main program and they must be entered by hand for each node. Rules are displayed in the one graph notation (compare section 2.2.5) where light blue dashed lines represent the elements that are removed and broader green lines symbolize elements that are added during rule application. Elements that stay, are kept in black. Groove supports non-injective match morphisms, i.e. it can match one element in a rule application multiple times. If this is not intended, the two elements in the rule have to be marked as unequal, so that they are not matched to the same element. This is done by adding a connection between them with the label "`not:=`" which is then displayed as a red edge between the nodes. However, this is not a transitive operation. Because of that, if n nodes can be matched non-injectively, $O(n^2)$ edges have to be inserted. For the editing of the rules, the same editor is used as for graphs, so there are no additional GUI elements for creating it. The properties of an element are, therefore, described by adding special keywords to the label, e.g. elements that are added during rule applications have a preceding "`new:`" in their label. This makes editing rules and graphs very uncomfortable.

As described above, Groove itself is not an editor but a simulator and the editor for the graphs is another tool, so the main feature of Groove is the application of the rules to a start graph. On the left pane, all rules with their corresponding matches on the start graph are displayed. By selecting one of the matches, it is displayed in the start graph and the corresponding rule can be applied. The new matches are directly updated on the left pane. The other feature of Groove is to display the space of all graphs that can be reached by applying rules to the start graph. This space is itself displayed as a graph where each node represents a graph in the transformation and each edge represents the rule that is applied to transform the graph of the source node to the graph of the target node. By clicking on the nodes, the corresponding graph is displayed. Groove automatically merges nodes that contain isomorphic graphs. The space can either be fully explored or only in a linear way (isomorphic graphs are evaluated only once), which supports a non-deterministic order of rule application, as well. Groove can try to find final states of a GTS. Therefore, it tries to apply rules in depth- or breadth-first way until no rule is applicable anymore. The final and the intermediate graphs are also displayed in the explored space tab. The extended features of Groove are the possibility to add wild-cards and regular expressions to labels which are evaluated when applying rules. Furthermore, simple data types can be associated with nodes and edges which can be changed during rule application. For more information about these advanced features see the Groove documentation ([33]). They are not covered here, because that would be beyond the scope of this thesis. There is no documented way of extending Groove with a better editor or further analysis possibilities. An API for using Groove without its GUI in external programs is not documented.

3.1.3 Conclusion

Both tools provide a different approach in graphical editing, but both have disadvantages. AGG informs the user very late about editing errors by popping up messages instead of giving earlier visual feedback. Groove's editing mechanism is based on entering keywords in the nodes' and edges' labels without the use of further graphical interfaces. That requires a constant switching from the keyboard to the mouse. Groove does not even provide support for type graphs, it is based only on the matching of labels. That is why no type system is provided to check the consistency of the start and rule graphs. Both tools provide interesting features, e.g. Groove provides the exploration of all intermediate graphs that can be created by applying rules, while AGG provides a confluence analysis. However, none of the tools

is extensible in order to add other missing features. Only AGG can be integrated in other projects to use GTS.

3.2 Goals

Raiser describes in [32] how GTS can be ported to CHR. Given this fact, methods for analyzing CHR programs can directly be used to analyze GTS, e.g. confluence, termination, and operational equivalence analysis. But these methods cannot be integrated into the existing tools described above. Furthermore, both tools are stand-alone tools that are not embedded in a common framework. Therefore, there is no common API to extend their functionality.

For that reasons, the subject of this thesis is to create a platform that enables you to create and analyze GTS. Editing should be possible in graphical notation and by using the encoding in CHR. The textual editor for the CHR encoding should be provided, because it represents a simple textual notation for editing a GTS and it helps to understand the results of analysis tools that are based on CHR. The platform must not be a stand-alone tool, but it should be embedded in the eclipse framework to allow easy extensibility. For that reason, no analysis methods are provided by the platform itself, but by further plug-ins. Because of that, an API must be made available, so that analysis tools that work directly on the GTS and analysis tools that are based on CHR can be integrated with little work.

To demonstrate the power of the platform, another goal is to create several prototypical analysis tools. One analysis method, available in both AGG and Groove, is the application of the created rules to a host graph. Therefore, an analysis tool must be offered, that adds this feature to the platform. This tool is based on CHR, because CHR can be used to simulate a GTS. Another goal is to create a tool that provides another API, so that CHR analysis methods (called *CHR tool* in the remainder) can be added to the platform. A prototypical CHR tool should be added that creates an executable CHR program from the GTS, because this can be needed by several other CHR tools, e.g. confluence analysis. Furthermore, the executable CHR program can also be embedded into other applications, so that the created GTS can be used there. Exemplary tools, that are independent from CHR should also be provided. That is why a possibility to analyze the termination of the GTS should be realized. The creation of random host graphs is an example for a tool that modifies the GTS. Realizing more of the analysis methods found in AGG or Groove, e.g. graph space exploration of the GTS or confluence analysis, would be out of the scope of this thesis and are left for future work.

To summarize, the functional goals are

- Provide editors GTS
  - in a graphical way
  - and in textual way via CHR
- Design of extension possibilities to add arbitrary analysis tools for GTS
  - Provide several prototypical analysis tools:
    - Simulation of GTS via CHR
    - Generation of random host graphs
    - Termination analysis for rules
  - Provide an analysis tool for embedding CHR based analysis methods

A non-functional goal is to provide a good usability for the editors and tools to allow rapid creation and analysis of new GTS.

3.3 Requirements analysis

In the following, I work out the requirements to achieve the goals described in the previous section. As described in section 3.2, there are two main goals for this thesis: the development of editing capabilities for GTS and the design of an extensible analysis platform.

As described in the previous section, there will be graphical, as well as textual editors which can be used in parallel. Therefore, an important requirement is the development of a front-end that displays two editors in parallel. As there are multiple editors, loading and saving of a GTS from and to a file must be handled by the platform itself. For the graphical editors it is very important to prevent the creation of inconsistent graphs and rules. Additionally, they should give early visual feedback of allowed editing steps. As a GTS $gts = (P, TG)$ consists of a set of rules P and a type graph TG, at least two editors are needed. Furthermore, a set of host graphs to apply the rules to must be available, therefore, an additional editor is needed. For the textual editors, a parsing mechanism is needed, so that it can be checked whether the CHR source encodes a GTS. Besides that, the textual editor must provide specific feedback of errors in the encoding. The encoding described in section 2.3 is too complex for editing a GTS, because it also provides some redundant information (the degree, and deletion attribute of constraints) that is only required when running the CHR program and for confluence analysis. As a consequence, a simpler encoding must be provided to make the editing more comfortable.

To synchronize the editors, a common model of the GTS, the *GTS model*, must be provided that is used by both editors and the platform. The editors must work synchronously and always display the same information to allow comfortable editing. Furthermore, an undo/redo stack is needed, so that changes made to the model can be undone. This stack must be shared between the editors to prevent inconsistencies of the GTS.

The second main goal is to provide a platform for easy extensibility with analysis tools for GTS. As a consequence, a plug-in structure must be developed to share the currently edited GTS model with the desired plug-in. Additionally, a simple GUI must be designed for starting these plug-ins. These plug-ins are called *(GTS) analysis tools*. For the CHR based analysis of GTS, an analysis tool called the *CHR based analysis tool* that creates CHR source code and opens a *CHR analysis tool* with this generated file must be provided. For the CHR code generation, the possibility to activate and deactivate rules, as well as to change their order is required, because this is important for the execution of CHR programs (section 2.1). Current implementations of CHR for java are all deterministic. That is why a non-deterministic mode cannot be provided here. Furthermore, the tool requires a possibility to encode host graphs as input for the CHR analysis tools. When rules or host graphs are selected, they should be displayed in the editor, so that the user has direct visual feedback of the activated rules. In order to add available analysis methods for CHR to the platform, another plug-in system must be developed to integrate them into the platform. The most basic functionality that is needed is the creation of an executable environment of the CHR source, so a CHR analysis tool must be developed that creates this environment. A good example for a CHR analysis tool is a confluence analysis. Therefore, a strategy has to be developed on how to embed an automatic confluence analysis into the platform. The CHR analysis tools should also have the possibility to display rules or parts of them in the editors that correspond to CHR rules. For that reason, the CHR based analysis tools must transform the request from a CHR tool to a request for the platform in order to display a certain graphs or highlight parts of it.

The analysis tool for rule application to a host graph via CHR must have a graphical front end to display the graph represented by the CHR store, so a synchronization between the CHR store and the graphical front end must be developed. Layout algorithms are needed for proper display of the resulting graph. AGG and Groove both provide the possibility to apply rules step-by-step and manually select the next match for a rule to test the GTS

thoroughly. These features should be included in this tool, as well. Therefore, there must be a possibility to enable and disable CHR rules and change their order. Additionally, a method has to be developed, so that the CHR rules can be applied step by step and the match of the rules can be specified manually. When rules are disabled or enabled, they should also be displayed in the editors.

The tool for creating random host graphs must provide a simple interface to enter the number of nodes and edges that should be created. Because of that, an algorithm must be provided to create a consistent typed graph from this information.

Termination can be analyzed by using a ranking function. Therefore, a method must be provided to enter the ranking function and analyze the rules with it.

Some of the tools require a communication back to the platform in order to display certain graphs or at least parts of them. Therefore, a communication infrastructure must be developed, so that the platform can also receive information from the tools to display graphs or highlight parts of it. More of the features that can be found in AGG and Groove are not provided as additional tools, because the goal of this thesis is the development of an extensible platform and not to implement all the features of AGG and Groove into a single program.

Here is a list of the resulting requirements:

- Platform in general:
  - Shared model for GTS, used by editors and tools
  - Shared command stack
  - Saving/Loading mechanisms for GTS models
- Editors in general:
  - Synchronization of view
- Graphical editor:
  - Forbid inconsistencies in the GTS
  - Early visual feedback about invalid model states
  - Multiple embedded editors for host, rule and type graphs
- Textual editor:
  - Create model from CHR source
  - Algorithm to check for valid encoding of a graph
  - Update the GTS model from the CHR source
  - Error feedback in the editor
  - Simpler encoding for easier editing
- Plug-in system:
  - Provide model, command stack, and file for the tool
  - Receive information from tools to update the view in the editors
- CHR based analysis tool:
  - Plug-in structure for CHR analysis tools
  - Select rule order

  - CHR code generation
  - Create executable environment of CHR source
  - Confluence analysis tool
  - Display graphs selected in CHR tools in the editors.

- Tool for running a GTS based on CHR with graphical front-end:
  - Develop a method to execute CHR programs step by step and define matches manually
  - Create executable environment of CHR program
  - Graphical front-end with automatic graph layout
  - Synchronization of CHR store and graphical front-end
  - Select order of the rules
  - Display rules in editors

- Random host graph tool:
  - Simple GUI
  - Creation of valid models

- Termination analysis tool:
  - Manual definition of ranking function
  - Display non-terminating rules in editors

3.4 Finding the right tools

In this section, I give an overview of the tools that are available to possibly provide the functionality requested above. The selection of the tools is covered at the end in section 3.4.7.

3.4.1 Model creation

The most basic concept of the platform is a common model on which the editors and analysis tools can operate. The model can be designed manually or a library providing graph models can be used.

3.4.1.1 JGraphT

JGraphT[1] is a graph library which is based on java and supports many different types of graphs, e.g. directed and undirected graphs. Furthermore, it implements the listener pattern ([23]) to get notified about changes made to the graph. JGraphT is designed to be simple and highly efficient with large graphs. Additionally, many algorithms from graph theory are available. JGraphT graphs can be used as models for JGraph which is described when discussing about graphical editors in section 3.4.2.3.

[1] Available at `http://www.jgrapht.org/`

3.4.1.2 JGraph

JGraph[2] is a graphical framework for displaying graphs, but it also provides an own model for the representation of graphs that can also be used without the JGraph graphical framework. The graph model is meant for extending it for custom application, but a standard graph model is already provided that works for most applications.

3.4.1.3 EMF

Eclipse features a framework for creating arbitrary models and generate code from it. It is called the *Eclipse Modeling Framework (EMF)* ([15]). EMF creates java interfaces from the model definition, as well as implementations of it. It provides structures to keep the model consistent, i.e. if two classes are connected to each other and a reference is deleted or added in either of the two objects, it also gets deleted or added in the other one. Another benefit of EMF is that it implements the listener pattern ([23]), so that editors and viewers can be updated automatically upon changes to the model. Aside from references between objects, the concept of *containers* and *containments* is provided to create a tree like structure for the objects, i.e. elements can only be contained in one containment at the same time. EMF also offers the possibility to add custom code to the models.

3.4.2 Graphical editors

For the graphical editors, eclipse itself provides two solutions: The *Graphical Editing Framework (GEF)* and the *Graphical Modeling Framework (GMF)*. There are third party graphical editor frameworks that can be used without eclipse: *JGraph* and *yFiles*.

3.4.2.1 Graphical Editing Framework (GEF)

GEF ([16]) is a framework for graphical editors in the eclipse platform. It is based on the model view controller (MVC) paradigm which is further described in [27]. Figure 3.3 shows the relation between model, view, and controller objects in the MVC pattern. The model represents the data that should be displayed and edited. The view renders the contents of the model. The view can query the model itself for changes or add itself as listener to it, so that it gets notified when the model changes. The controller translates interactions from the user with the view into modifications of the model. In this design, the controller only receives information from the view and propagates it to the model. However, in GEF the controller has a more central part and is named *edit-part*. It registers itself as listener to the model and updates the view object when the model is changed. Therefore, the view and the model are decoupled. Edit-parts can have multiple *edit-policies*. *Requests* which are sent by *tools* to edit the model, are send from the edit-part to the corresponding edit-policies which itself create *commands* that modify the model. The tools are provided by GEF and give the user the possibilities of editing the model. Tools for creating, deleting, and moving objects are already provided by GEF, but they can also be extended to provide more specialized functionality. The creation of the views is based on *draw2D*, a simple toolkit for creating graphical figures. The technology used for the model is not defined by GEF. The model has to implement the listener pattern. EMF can be used for creating the model, for example. GEF is widely used in the eclipse community for creating graphical editors and good documentation, as well as sample code is available.

[2] Available at `http://www.jgraph.com`

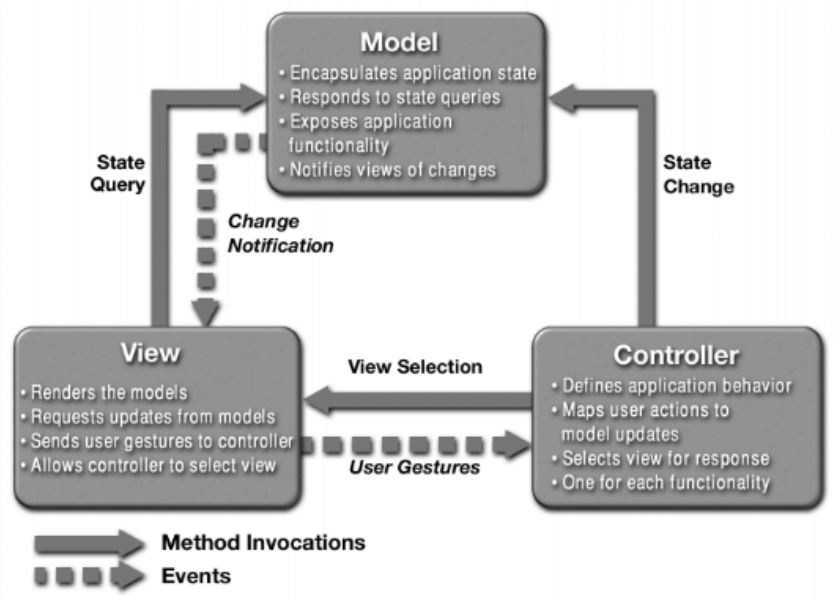


Figure 3.3: Schematics of the MVC pattern (from `http://java.sun.com/blueprints/patterns/MVC-detailed.html`).

3.4.2.2 Graphical Modeling Framework (GMF)

GMF ([17]) is based on GEF and EMF. GMF lets you define an EMF model, as well as views, tools, and a mapping between them in a graphical editor. From this information, code for a graphical editor is generated. Many standard functions often used in GEF editors are automatically implemented such as zooming and adding comments to elements. A separation of the model itself and a model for the graphical representation is available, as well as the possibility to save them to and load them from a file. To conclude, GMF is a tool to rapidly create standard graphical editors.

3.4.2.3 JGraph

JGraph[3] is a complete framework for the graphical editing of graphs. The model for editing is already available in JGraph, but it can also be customized. Other models can be used, as well, e.g. the JGraphT model. The JGraph editor components are based on the Swing API. There is a free open-source core of JGraph that provides classes for creating own editors. However, there are also some commercial add-ons for laying out graphs (*Layout Pro*) and for creating feature rich editors (*GraphPad Pro*). A complete programmers guide is available together with a few example projects on the project website.

3.4.2.4 yFile

yFile[4] is a commercial project. It consists of an extensive library that features algorithms for the analysis, visualization, and automatic layout of graphs. The library is written in java

[3] Available at `http://www.jgraph.com`
[4] Available at `http://www.yworks.com/`

and uses the Swing API for visualization. It can only be extended in limited ways, because most licenses are only byte code licenses, which are not offering access to the source code.

3.4.3 Textual editors

Eclipse itself already provides base classes for textual editors, however, these editors do not offer parsing mechanisms to retrieve the information that is encoded in the text. An editor made especially for JCHR (a java-based CHR implementation) is currently developed at the university of Leuven. Two other frameworks are for creating an EMF model from the contents of a text editor based on a grammar: The *Textual Editing Framework (TEF)* and the *Textual Modeling Framework (TMF)*, which are both based on eclipse.

3.4.3.1 JCHRIDE

The JCHRIDE ([1]) is an eclipse text editor with syntax highlighting and rudimentary code completion. Furthermore, support for compiling JCHR source code files with the K.U. Leuven JCHR compiler is provided. The compilation, however, works only on Microsoft Windows systems. The editor does not generate a model of the JCHR source, but only checks for syntactic correctness.

3.4.3.2 XText

XText ([13]) is part of the eclipse Textual Modeling Framework (TMF). For XText, a grammar and an according EMF model has to be created. Then XText creates a parser, a serializer, and a text editor from the model and the grammar. Therefore, models can be created from text or serialized into a string according to the grammar. The editor has many features (syntax highlighting, auto-completion...) and is integrated into eclipse. At the time of implementation XText has still been part of the *openArchitectureWare (oAW)* project ([30]), a framework for model driven development, which is based on eclipse. However, XText has been on the move from the oAW project to the eclipse TMF project, but no version has been available for TMF at the time of implementation.

3.4.3.3 Textual Editing Framework (TEF)

A similar tool is the *Textual Editing Framework* (TEF), which is currently in development at the technical university of Berlin ([35]). TEF provides a text editor base class, which is supplied with a context-free grammar, annotated with classes and properties of classes from an EMF model. This grammar is then interpreted by the TEF framework to create an instance of the EMF model described by the content of the text editor. Furthermore, TEF features error annotation in the editor for syntax errors and reference errors in the generated model instance. Models can also be serialized to strings and displayed in the text editor. The grammar also supports special white space terminals, that are only considered when creating a string from a model. TEF uses the method of background parsing to continuously create a model in the background. Depending on the type of the TEF editor, a new model is created on every background parsing step or an existing model is being edited. There is a rudimentary auto-completion function available that can be further extended.

3.4.4 CHR environments

CHR is an extension of a host language, therefore, there exist a lot of implementations for several languages. [39] contains an extensive list of available CHR implementations. Most

implementations are based on logical programming languages like Prolog, but only few are available for imperative languages like java or C. In the following, I present the two most used implementations for java.

3.4.4.1 Java Constraint Kit (JaCK)

JaCK ([2]) is the earliest CHR implementation for java. It has been started in 2001. JaCK is a combination of several components. JCHR is a CHR dialect that combines Prolog based CHR syntax with java syntax. VisualCHR is a tool for visualizing the execution of JCHR programs. The last component is the *Java Abstract Search engine (JASE)* in which tree-based search strategies can be specified. It provides the possibility to implement search algorithms to use backtracking when inconsistent states occur during the execution of JCHR. However, this project is no longer maintained.

3.4.4.2 K.U. Leuven JCHR

The K.U. Leuven JCHR system (not to be confused with the JCHR component of JaCK) is currently in development at the Catholic University of Leuven ([40]). The term JCHR is used in the remainder of this thesis for the K.U. Leuven JCHR system. JCHR is not only an implementation of CHR in java, but an integration of the CHR features in java. For an introduction to the features of JCHR, please refer to the official K.U. Leuven JCHR documentation [43]. Basically, the K.U. Leuven JCHR compiler creates a java 1.5 compatible source file from a `.jchr` file. The resulting class is called *handler*. The syntax of `jchr` files is a combination of Prolog based CHR and java syntax. JCHR offers the possibility to embed arbitrary java classes and methods into a handler. Therefore, elements of the java language can also be used in JCHR handlers. Furthermore, the generated handlers can be embedded in other java programs. To do that, the handler provides methods to retrieve a list of the constraints in the CHR store and methods to add new constraints to the store.

3.4.5 Code generation

As described in 3.3, there are analysis tools that might rely on CHR source code files. As a consequence, code generators are needed. I describe Extensible Style-sheet Language Transformations (XSLT) and Java Emitter Templates (JET).

3.4.5.1 XSLT

XSLT ([26]) is an XML based transformation language for transforming XML documents into other document types like HTML or java source code. The XML source file is not modified, but a new file is created from the content of the old document. XSLT uses XPath ([4]) to query the source XML document for the information needed in the new document. An XSLT file represents the new file with embedded tags that represent the information from the input XML document. Tags are available for iteration and conditional execution based on the input document. Furthermore, a processor is needed that interprets the XSLT program and generates the output document. The two best known processors for java are Xalan-J[5] which is developed by the Apache foundation and Saxon[6] which is based upon SAX the java XML parsing tool.

[5]Available at `http://xml.apache.org/xalan-j`
[6]Available at `http://www.saxonica.com`

3.4.5.2 Java Emitter Templates (JET)

JET ([12]) is used to create source files in the EMF project, but this tool can also be used standalone. Originally, JET was only used to generate code from structures defined by XML files but has been extended to accept EMF model instances as input. The definitions for the generated source is provided via a text file in which the static parts of the source file are written as they should appear in the generated source and XML tags can be embedded to insert dynamic information gained from the input XML file or the EMF model. The syntax is similar to that of XSLT. There are tags for iteration over multiple elements and conditional insertion of parts of the source file. Further tags can be created as *tag libraries* by creating an eclipse extension to JET. The input is queried with XPath ([4]). XPath is basically a method for querying XML files in an easy and intuitive way. XPath expressions can also be used as input for so called *XPath functions*. These functions include, for example, counting the number of elements returned by the query or manipulate lists. Further XPath functions can be added by creating an according JET extension which contains the implementation of the function in java. JET offers the possibility to store calculated values in the model during transformation.

3.4.6 Graph layout and visualization

The tool for the execution of GTS based on CHR requires a visual representation of the CHR store as a graph. Therefore, a tool for displaying and laying out a graph is needed.

3.4.6.1 JGraph, yFiles

The tools, mentioned above for the graphical editors that offer layout algorithms can be used as visualization tool-kits. The only interaction that is allowed, is selecting nodes and edges and moving them around. All the other editing features are left away. JGraph and yFiles are the two tools that also provide layout functionality.

3.4.6.2 Zest

For displaying and laying out arbitrary graphs *Zest* ([14]) can be used. It uses draw2d from the GEF project for vizualization. Zest provides a graph object to which nodes and edges can be added. Zest is based on eclipse and uses the *Standard Widget Toolkit (SWT)* API. Furthermore, Zest already features several layout algorithms. They can be also be extended to create custom layout algorithms. The algorithms can also be used independently from Zest. In addition to that, possibilities of selecting nodes and edges and moving nodes around are provided. Listeners can be added to get notified about selection events occurring at the node and edge widgets.

3.4.7 Suitability of the tools

As mentioned above, it can hardly be decided for each category on its own which tools should be used, because some of them rely on each other. First, I exclude some of the tools. Tools with graphical components (i.e. the graphical editors and the visualization tools) that use the Swing API are difficult to use, because eclipse uses the SWT API for its graphical user interface. Although there are libraries that let you integrate Swing components into eclipse, this requires much additional work and is error-prone. That is why JGraph and yFile are not used as editors or visualization tools. This leaves GEF or GMF

for the graphical editor and Zest for the visualization and layout toolkit. GMF is based on GEF and EMF and provides the automatic generation of standard editors. However, the platform contains a textual editor besides the graphical one and these editors interact with each other. So, if GMF was chosen, the generated code would have to be modified a lot, so that it could be integrated into the platform and the inter-editor communication structure. As GMF editors are generated from model specifications, every change in the specification would result in a newly created editor and the changes that must be made for the integration into the platform would have to be redone. GEF, on the other hand, can be better customized and is not generated from specifications, but implemented manually. Therefore, GEF editors can be used better in combination with other editors. For these reasons, GEF has been chosen as the framework for graphical editors.

One requirement of the textual editor is that the content must be parsed to retrieve the information about the encoded GTS model. JCHRIDE is not used, because it only checks for syntactical correctness and does not provide further parsing mechanisms. That leaves XText or TEF as possible solutions. Both frameworks have all required features: they can both create an EMF model instance from the content of a text editor and both can print a model instance to the text editor. However, XText has been on the move from the oAW to the TMF project at the time of implementation and no version has been available, which is specifically designed for the eclipse TMF framework. Because of that, TEF is the framework that could be embedded better into eclipse and has, therefore, been chosen as textual editor.

The decision for the code generation tool was mainly driven by the eclipse platform. JET is specifically designed for eclipse and seamlessly integrates in the workspace concept by providing tags for creating files, folders, and new projects. Furthermore, XSLT with Saxon or Xalan-J only takes XML documents as input, while JET also provides the possibility to use instances of EMF models as input. For these reasons, JET is chosen as code generator.

The choice how to provide the model for the GTS is heavily influenced by choosing TEF and GEF as editors and JET as code generator. Although JGraphT provides an elaborate model for creating own graph models, many tools based on the eclipse framework use EMF models as input. Therefore, I have decided to design a GTS model based on EMF. Another advantage of EMF over JGraphT is, that it directly supports saving and loading of models to and from XML files. In combination with JET, EMF model instances can be used directly as input for code generation without having to save it to an XML file first.

The K.U. Leuven JCHR system is currently the best supported and fastest ([39]) CHR implementation for java. JaCK is no longer maintained and does not perform as well as the K.U. Leuven JCHR system. Therefore, the K.U. Leuven JCHR system is used for creating executable environments of CHR programs. Because of that, all generated CHR source is in JCHR syntax.

Further details about the tools and how they are used is provided in chapter 4 as needed.

4 Realization

In this chapter, I describe the design and implementation of the project. First, I present the model for GTS in section 4.1. In section 4.2, I show how I have designed the platform itself including the plug-in system for the analysis tools and the elements to display two editors synchronously. The graphical and textual editors together with synchronization algorithms are described in sections 4.3 and 4.4. In sections 4.5 to 4.8, I present the realization of the plug-ins. I pay special attention to the tool for CHR based analysis (section 4.6) and the simulation of a GTS based on CHR (section 4.7). Section 4.5 is a detailed guide that shows how to create custom analysis tools with the example of a tool for termination analysis. In section 4.8, I present a tool for generating random host graphs and provide a guide how a confluence analysis for CHR could be added as a CHR analysis tool. Each topic will be divided in showing the design first and then giving details about the implementation. Chosen parts also have an additional section which show a sample computation.

The implementation sections give reference to the eclipse project names, where the source can be found. These project can be found on the CD-ROM to this thesis (see appendix A).

4.1 GTS model

In this section, I explain how the model is designed and present the possibilities for editing it. The model is not directly edited by the tools or by the editors, but commands are provided to edit the model. These commands are managed by a command stack in order to provide undo and redo functionality.

4.1.1 Design

In the following, I describe the design of the model and the design of the commands.

4.1.1.1 Model

The model for the GTS (*model* or *GTS model* for the remainder of this work) consists of multiple graph models. As described in section 2.2, a GTS consists of a set of rules P typed over a type graph TG. Furthermore, the possibility to specify a set of host graphs HG typed over TG should be provided. One thing that all the graphs mentioned have in common is that they consist of nodes and edges that connect the nodes. Each node has a list of edges that have it as source and a separate list for the target edges, whereas each edge has exactly one source and target node. Furthermore, the edges and nodes belong to exactly one graph. This description of a graph can be seen in the *Unified Modeling Language (UML)* class diagram in figure 4.1. The nodes and edges are modeled as a composition, because nodes and edges can only belong to one graph. Graphs have an identifier, as well.

Figure 4.2 shows the UML class diagram of the complete GTS model. A type graph (*TypeGraph*) is essentially a graph as described above, only that it has an additional list of its typed graphs. The nodes and edges of the type graph (*TypeNode* and *TypeEdge*) also have a list of their typed nodes and edges (*TypedNode* and *TypedEdge*), respectively. The type

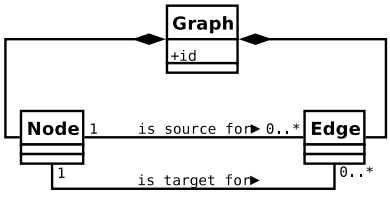


Figure 4.1: General UML model of a graph in this GTS model.

graph, type nodes, and type edges are inherited from the standard graph model (*Graph*, *Node*, and *Edge*). The type nodes and edges have an additional unique identifier as attribute.

Typed graphs (*TypedGraph*) are the counterparts of the type graphs, so the typed graphs and its nodes and edges contain exactly one reference back to their type graph, type node, or type edge, respectively. In regard to the encoding of GTS to CHR, every node in a typed graph receives an unique id. The graphs in HG (*HostGraph*) are inherited from the typed graph class and have a reference back to the GTS they belong to. A GTS can contain multiple host graphs, as described in section 2.2. Rule graphs consist of three graphs that are in relation by two graph morphisms of the form $p = (L \xleftarrow{l} K \xrightarrow{r} R)$ with the graphs L, K, and Ro and the graph morphisms l and r. I decided not to use this approach directly in the model, because nodes and edges in K are additionally stored in L and R and the morphisms have to be provided explicitly which would require additional logic in the model. As a result, I have designed the rules as a triple of the form $p' = (L', K, R')$ where $L' = L \backslash K$ and $R' = R \backslash K$. A transformation from p' back to p can be achieved by defining $R = R' \uplus K$ and $L = L' \uplus K$ and setting the morphisms l and r to map the nodes from the graph K to the nodes that are added from K to L' and R', respectively. With this new way of presenting a rule, the three graphs can be encoded in one typed graph $T = L' \uplus K \uplus R'$ (*RuleGraph*) by giving each node and edge (*TransformNode* and *TransformEdge*) an attribute that describes whether they are in L', R', or K. The nodes and edges in L' correspond to the red elements in the *one graph notation* described in 2.2, the nodes and edges in R' to the green elements, and the nodes and edges in K to the black elements. All these types of graphs are bundled together in one graph transformation system class (*GTS*) by two lists of host and rule graphs and a single type graph over which the other graphs are typed. Another approach to model a GTS that is closer to the category theoretic basics described in 2.2 would be possible, but in the implementation section I explain why I have chosen the approach described above.

4.1.1.2 Editing commands

Commands can be used by the editors to edit a GTS model. There are multiple tasks, when editing a GTS. First, rule and host graphs can be added and removed and their name can be changed. For each type of graph, i.e. host, rule, and type graph, nodes and edges can be removed. The type graph's edges and nodes can also be renamed. The identifier of the nodes in rule and host graphs can also be modified. Finally, the nodes and edges of a rule graph $p' = (L', K, R')$ can be moved between the graphs L', K, and R'. Table 4.2 lists the modifications that can be applied to GTS model. However, the GTS model does not imply, for example, that type nodes can only be added to a typed graph, and not to a host graph. Therefore, commands must be provided that ensure that no such *invalid model states* can occur. In the following, I describe the possibilities of returning an invalid model. The according figures illustrate invalid model states.

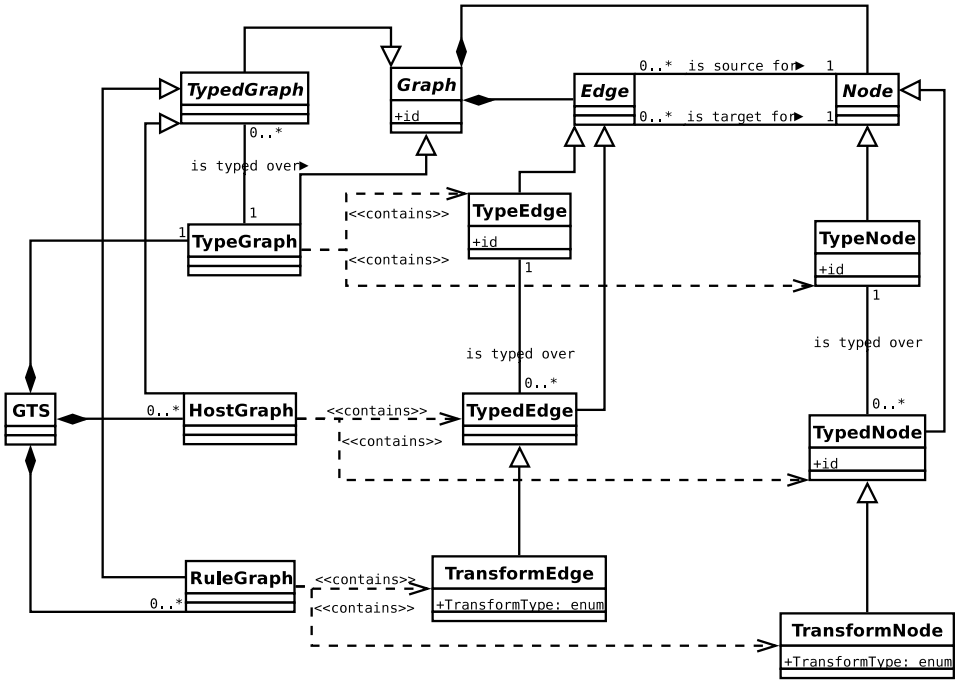


Figure 4.2: UML class diagram of the model of a GTS.

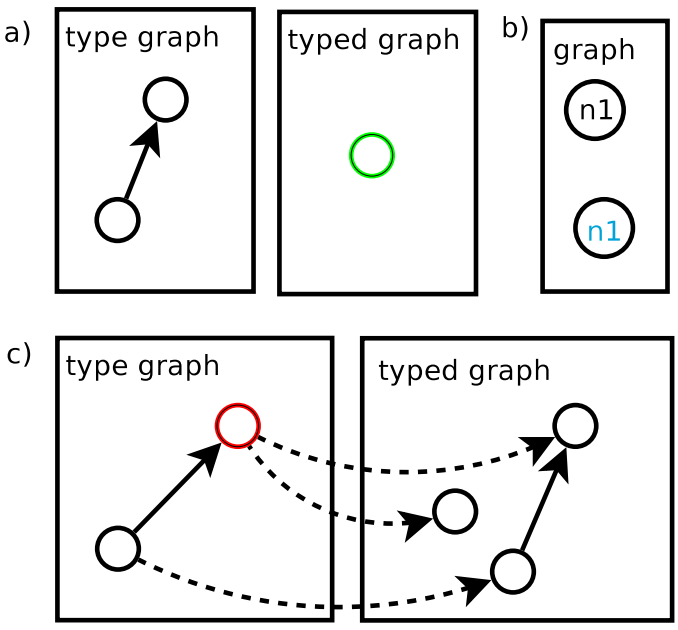


Figure 4.3: Problematic states when editing nodes in a graph (removed elements are red, added elements are green, and modifications are blue).

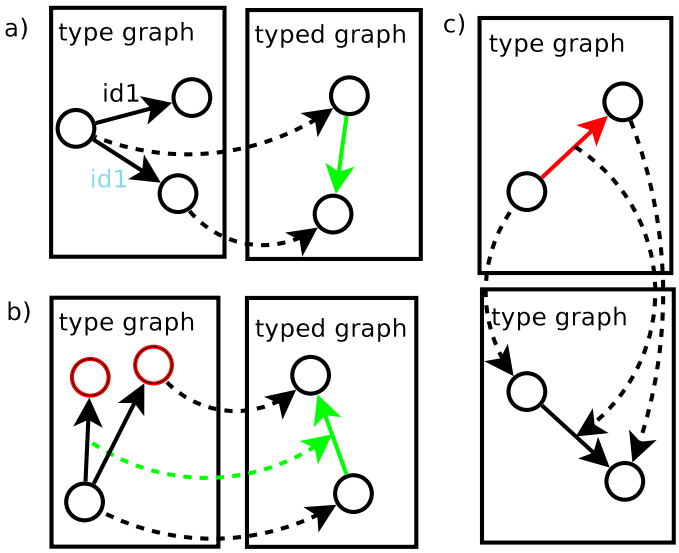


Figure 4.4: Problematic states when editing edges in a graph (removed elements are red, added elements are green, and modifications are blue).

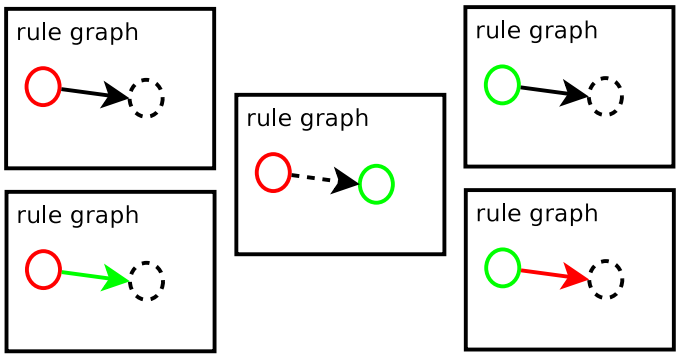


Figure 4.5: Impossible combinations of nodes and edges in the graphs R',L', and K of a rule. Elements in L' are red, R' is green, and K is black. A dashed line means, the element can be in any graph.

src/tgt	L'	K	R'
L'	L'	L'	$\emptyset$
K	L'	L',K,R'	R'
R'	$\emptyset$	R'	R'

Table 4.1: Allowed combinations of edges with source and target nodes for rule graphs.

Node commands When adding nodes in a typed graph with no type node being specified (figure 4.3 a) shows a typed node without its type morphism), the model is invalid. The identifier of a new node must be unique among the other nodes (figure 4.3 b) shows two nodes with the same identifier). When a node's identifier is changed, it must also be ensured that it is unique. The uniqueness of the identifiers is important for the encoding of a GTS in CHR (section 2.2). Removing a node can be problematic: Edges can be left dangling and if the removed node is a type node, then typed nodes that reference it have no type anymore (figure 4.3 c)).

Edge commands When changing or setting the id of a type edge in a type graph, it must be ensured that the identifier is unique (figure 4.4 a)). When adding typed edges in typed graphs, the types of the source and target nodes must match the source and target of the type edge (figure 4.4 b) shows typed nodes that are connected by the wrong edge), i.e. the typing morphism described in section 2.2 must be valid. When creating a typed edge, a type edge must be provided (figure 4.4 a)). Deleting an edge only raises problems when it is a type edge, because typed edges referencing it would not have a type anymore (figure 4.4 b)).

Graph commands Adding and removing graphs does not impose any problems, except when changing the name of typed graphs, attention must be paid that the new name is unique for host and rule graphs, respectively.

Commands for nodes and edges in rule graphs The problems stated above for edges and nodes are also valid for rule graphs, but editing rule graphs can lead to additional invalid model states. When changing the graph to which the nodes and edges belong (L',R', or K), it must be ensured that edges cannot arbitrarily connect nodes from the three graphs, e.g. an edge can never connect a node in R' to a node in L'. See figure 4.5 for the combinations that are invalid. The valid combinations of nodes and edges in the graphs are listed in table 4.1. The table shows the graphs in which an edge can be in depending on the fact in which graphs the source and target nodes are in. An edge in L' may only be connected to nodes in L' and K. Accordingly, an edge in R' may only be connected to nodes in R' and K. Edges in K may only be connected to nodes in K.

To prevent the invalid model states described above, commands must be designed in a way, so that they ensure that modifications to a model do not lead to any invalid model state. The problems of checking for unique identifiers when adding or removing nodes, edges, or graphs can be solved by rejecting the modification if the identifier is not unique. The same solution holds for adding typed nodes or typed edges when no type node or edge is present. Then the modification is rejected. However, one problem that must be discussed is how to handle the removal of node and edges. As described above, by removing nodes, edges can be left dangling. By removing type edges or nodes, there can exist typed edges or nodes that have no type any more. There are two solutions for this problem: Either forbidding the deletion of those elements until all other elements that reference it are removed or removing the referenced elements. I have decided to remove the referenced objects, as well, in order to make the editing of the model more comfortable.

A similar problem must be discussed for rule graphs. In fact, a rule consists of three graphs L', K, and R' which are encoded in one graph in the model. The change of the attribute to

which graph a node or edge belongs is called a *move* between the graphs. When moving a rule node in a rule graph, the resulting graph may become invalid according to figure 4.5. There are again two solutions: modifying the connected edges, so that the model is valid again or forbidding the the modification until the according edges are in the correct graph. I have decided to modify the connected edges in order to simplify the editing of the model. However, there are, in fact, three possibilities that can occur when moving a node:

1. After moving the node, the graph is still valid
2. The graph can be made valid, by moving edges
3. The graph cannot be made valid by moving the edges

In the first case, nothing has to be done, because the model is already valid. In the second case, the edges that are connected to the moved node can also be moved to other graphs, so that the model becomes invalid. In the last case, there is an edge that connects a node in R' to a node in L'. For this combination, no edge can connect these two nodes (table 4.1). In this case, there are two possible strategies: forbidding the move to a new graph for the node or deleting the edges that would render the model invalid. I have decided again to delete the edges, because this simplifies the editing of the model.

The last task of editing is the moving of an edge between the three graphs of a rule.However, solving invalid model states by moving connected nodes, too, is problematic, because there can exist multiple ways of moving the nodes, so that the model becomes valid again. Because of that, the move of an edge is simply forbidden if the resulting model would be invalid.

4.1.2 Implementation

In this section, I describe how the model and editing commands are implemented. The model can be found in the projects `org.uniulm.gts.model` and `org.uniulm.gts.model.edit`. The commands are bundled in the project `org.uniulm.gts.model.commands`.

4.1.2.1 Model

For the implementation, I use the two java technologies EMF and GEF. I decided to use a code generator for creating the source code for the model, because generated code tends to contain less errors and saves a lot of time. I chose to use EMF, the eclipse based code generation tool for modeling. Despite its advantages, code generators often provide only few possibilities to add user generated code which might be necessary to describe logic that would otherwise make the model itself very complex. Because of that, the code to keep the model valid is not put into the model itself, but it is realized by using GEF commands. GEF commands are used, because the graphical editor uses GEF. Therefore, the commands can be better integrated in the editor.

The UML class diagram in figure 4.2 can be ported directly to EMF by creating an *ecore* file from it which describes all the classes, their attributes, relations, and further the hierarchic compositional structure of the elements (this is called a containment/container relationship in EMF). The containment structure is especially important for serializing the model to XML files, which is described in section 4.2.1. The nodes and edges for rule graphs contain an attribute to express to which graph of the rule they belong to. This is realized by the java enumeration `RuleGraphType` with three states, namely `K`, `R`, and `L`, representing the belonging of the element to the graphs K, R', and L'.

The EMF generates two projects out of an *ecore* file: the interfaces and implementations of the classes (`org.uniulm.gts.model`) and classes that provide rudimentary editing capabilities for them (`org.uniulm.gts.edit`). Other projects like testing environments and editors can also be generated, but they are not considered here. Furthermore, the generated implementation provides the listener pattern ([23]) for every class. Automatic consistency for elements that have a binary relation is also available. When changing the relation in one object, it gets also changed in the other one automatically, e.g. when the source attribute of an edge is set to null, the edge is automatically removed from the list of edges of the former source node. The reason why the model design is not closer to the theory described in 2.2 is that EMF is based around classes and their connections to each other. Morphisms for typed graphs or between the graphs of a rule could be realized as methods of a class, but the implementation would not be provided automatically. As a result, the morphisms are expressed as relations and compositions between the classes. In addition to that, the model described above is much more convenient for the editors, as described in section 4.3. Some further adjustments have to be made to the *ecore* model in regard to the graphical editor: The classes for the nodes, edges, and graphs have a reference to a view object that saves visual information about the corresponding element, i.e. size and position. As described in [3], a model could also be divided into two models where the second model only saves the graphical representation. This would provide a separation of the display information from the information of the GTS itself which is in general a good feature of a model, but there is no advantage specifically for this project. If there were multiple graphical editors, all with a different way of displaying the graphs, it would justify the separation. Furthermore, the implementation of such a separation would have been too time-consuming. Because of that, the visual information is saved in the same model.

4.1.2.2 Editing commands

The commands are implemented as GEF commands, so they can easier be integrated into the graphical editors just as described in [29]. GEF provides a base class `Command` where methods for executing, undoing, and redoing are provided. This class is meant for extending it. Additionally, commands have a `canExecute()` method that is checked by the command stack prior to executing it. This method can be used to check for possible invalid model states. In the following, I show how I have solved the problems of editing the model (section 4.1.1) by using commands. A full list of commands for the modification of the model is listed in table 4.2 (commands for changing the visual presentation of the model are not listed). From this list, one can see that there is not one command for each element of the GTS. The commands are more generic and determine the correct object of the model from the given information, e.g. if the graph to which the node or edge is added is a rule graph, the node or edge is also a rule node or edge, respectively. Each command checks in the `canExecute()` command if it can be executed without rendering the model invalid. The `execute()` method saves values for undoing the command. If further modifications have to be applied to the model to keep it valid, the corresponding commands are also created in the `execute()` method. At the end of this method, the `redo()` method is called. In the `redo()` method, the changes are in fact executed. Whereas in the `undo()` method, the previously saved values are restored and commands executed in the `redo()` method are undone

In appendix C.1, two exemplary commands are shown: one for deleting nodes and one for moving a node or an edge between the graphs R', L', and K in a rule. I have chosen these two commands, because they present the two more complex situations that can render a model invalid. Listing 4.1 shows an excerpt of the `NodeDeleteCommand`. In this command, further commands are created in the `execute()` method (lines 1–19) in order to delete edges connected to the node. If it is a type node, delete commands for the typed nodes referencing it are created. These commands are executed in the `redo()`

`EdgeCreateCommand`	Create an edge
`EdgeDeleteCommand`	Delete an edge
`EdgeModifyCommand`	Change the identifier of an edge
`NodeCreateCommand`	Create a node
`NodeDeleteCommand`	Delete a node
`NodeModifyCommand`	Change the identifier of a node
`HostGraphCreateCommand`	Create a new host graph
`RuleGraphCreateCommand`	Create a new rule graph
`GraphDeleteCommand`	Delete a typed graph
`ChangeGraphNameCommand`	Change the name of a typed graph
`ChangeRuleGraphCommand`	Change the belonging of a node or an edge to another rule graph

Table 4.2: Table of available editing commands.

```
1  public void execute() {
2    removedEdges=new LinkedList<EdgeDeleteCommand>();
3    removedTypedNodes=new LinkedList<NodeDeleteCommand>();
4    if(child instanceof ITypeNode){
5     for(IAbstractNode n:((ITypeNode)child).getTypes())
6      removedTypedNodes.add(new NodeDeleteCommand(n));
7    }
8    if(child instanceof INode){
9     typeNode = ((INode)child).getType();
10   }
11   for(IAbstractEdge e:child.getSrcEdg()){
12    removedEdges.add(new EdgeDeleteCommand(e));
13   }
14   for(IAbstractEdge e:child.getTgtEdg()){
15    removedEdges.add(new EdgeDeleteCommand(e));
16   }
17   redo();
18  }
19
20  public void redo() {
21   for(EdgeDeleteCommand e:removedEdges){
22    if(e.canExecute()) e.execute();
23   }
24   if(child instanceof ITypeNode){
25    for(NodeDeleteCommand n:removedTypedNodes)
26     if(n.canExecute()) n.execute();
27   }
28   if(child instanceof INode) ((INode)child).setType(null);
29   child.setGraph(null);
30  }
31
32  public void undo() {
33   child.setGraph(parent);
34   if(child instanceof INode){
35    ((INode)child).setType(typeNode);
36   }
37   if(child instanceof ITypeNode){
38    for(NodeDeleteCommand n:removedTypedNodes)
39     n.undo();
40   }
41   for(EdgeDeleteCommand e:removedEdges) e.undo();
42  }
```

Listing 4.1: Excerpt from the `NodeDeleteCommand`.

```
1  public boolean canExecute(){
2   if(model instanceof ITransformEdge){
3    TransformElementType src=((ITransformNode)((ITransformEdge)model).getSrc()).getTrans
         ();
4    TransformElementType tgt=((ITransformNode)((ITransformEdge)model).getTgt()).getTrans
         ();
5    if(newType.equals(TransformElementType.K) &&
6      src.equals(TransformElementType.K) &&
7      tgt.equals(TransformElementType.K))
8     return true;
9    else if(!newType.equals(TransformElementType.K) &&
10     (src.equals(TransformElementType.K) || src.equals(newType)) &&
11     tgt.equals(TransformElementType.K) || tgt.equals(newType)))
12    return true;
13   else return false;
14  }
15  if(model instanceof ITransformNode) return true;
16  return false;
17 }
```

Listing 4.2: Excerpt from the `ChangeRuleGraphCommand`.

method (lines 20–30) by first removing the typed nodes (if the node itself is a type node), then the edges, and at last the node itself. Especially note the `undo()` method (lines 32–42) where all the commands are undone, but in opposite order to keep the model valid in intermediate states. Listing 4.2 shows the `canExecute()` method from the `ChangeRuleGraphCommand`. This command works on both, edges and nodes. For edges, the `canExecute()` method checks whether the combination in the new graph is valid according to table 4.1 (lines 2–14). Nodes may always be moved between the graphs (line 15), because edges connected to them are moved or deleted, as well.

4.2 The platform

The design of the platform itself is very important, because all the tools and the editors are embedded there. The platform consists, on the one hand, of GUI elements for displaying the editors and for giving a choice for the analysis tools and, on the other hand, of the eclipse extension point where analysis tools can be plugged in.

4.2.1 Design

In this section, I explain the design of the GUI and the extension point, as well as the flow of information between analysis tools and editors.

4.2.1.1 Graphical user interface

The GUI of the platform consists of an editor which is placed in the eclipse workbench and a corresponding *action bar*. The editor must be able to display two embedded editors at the same time (the graphical and the textual one). This split editor is called *multi-bar editor* (in relation to the already existing multi-page editor provided by eclipse). In the multi-bar editor, the editors are arranged vertically below each other, each one having a bar above containing its name. The screen-shot in figure 4.6 shows a multi-bar editor instance with a textual editor expanded and a graphical editor collapsed. Each editor can be collapsed and expanded by clicking on an arrow in the bar next to its name.

The *GTS editor* is extended from the multi-bar editor and is used as base editor for the platform. It handles loading and saving of the GTS model and provides a shared command stack for the editors and analysis tools.

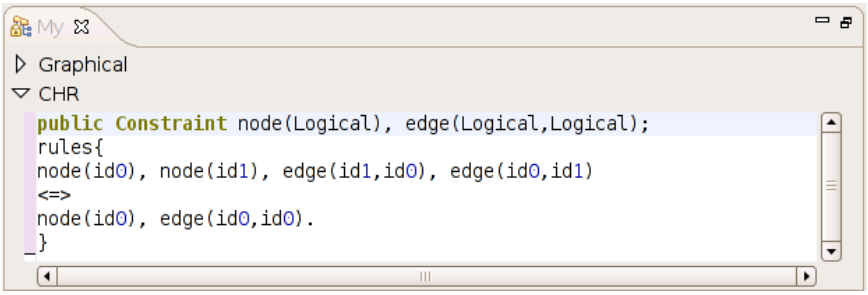


Figure 4.6: The multi-bar editor.

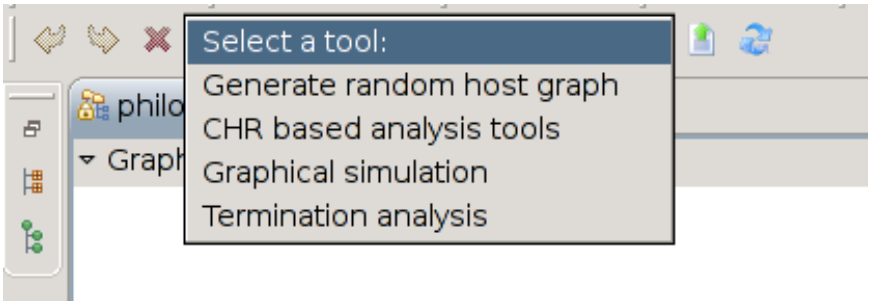


Figure 4.7: The action bar of the multi-bar editor.

The second GUI element is an action bar for the multi-bar editor which can be seen in figure 4.7. This action bar shows actions that are available for the editors which are embedded in the corresponding multi-bar editor. Furthermore, it contains a combo box that lists all the available analysis tools. Buttons for undo, redo, and delete operations are inserted at the left side of the combo-box. The analysis tools can be launched by clicking the entries of the combo box. The buttons on the right side of the combo-box are managed by the embedded editors. These buttons can be updated by the embedded editors. For example, the graphical editor can display buttons to add new host and rule graphs depending on which graph is displayed at the moment.

Other needed GUI elements are file and project creation wizards that guide the user through the creation of new GTS. There must be a wizard to create a new GTS project which can host multiple GTS files. For creating files that contain a GTS, another wizard must be created. Both wizards are kept very simple and only ask the user to enter the name of the GTS project or select the project where the new GTS file should be added to and provide the name of the file. Both wizards are shown in figure 4.8.

4.2.1.2 Extendability

Before discussing how the platform's extensibility can be designed, I describe the flow of information in the platform between the tools and the editors, first. Information about changes in the model do not need to be exchanged between tools and editors, because the editors and tools can be directly updated from the model itself. Therefore, the only information that must be send to the analysis tools is the model itself, the file it was loaded from, and the command stack, so that the tools can read the model, modify it, or create further files depending on the file name of the model.

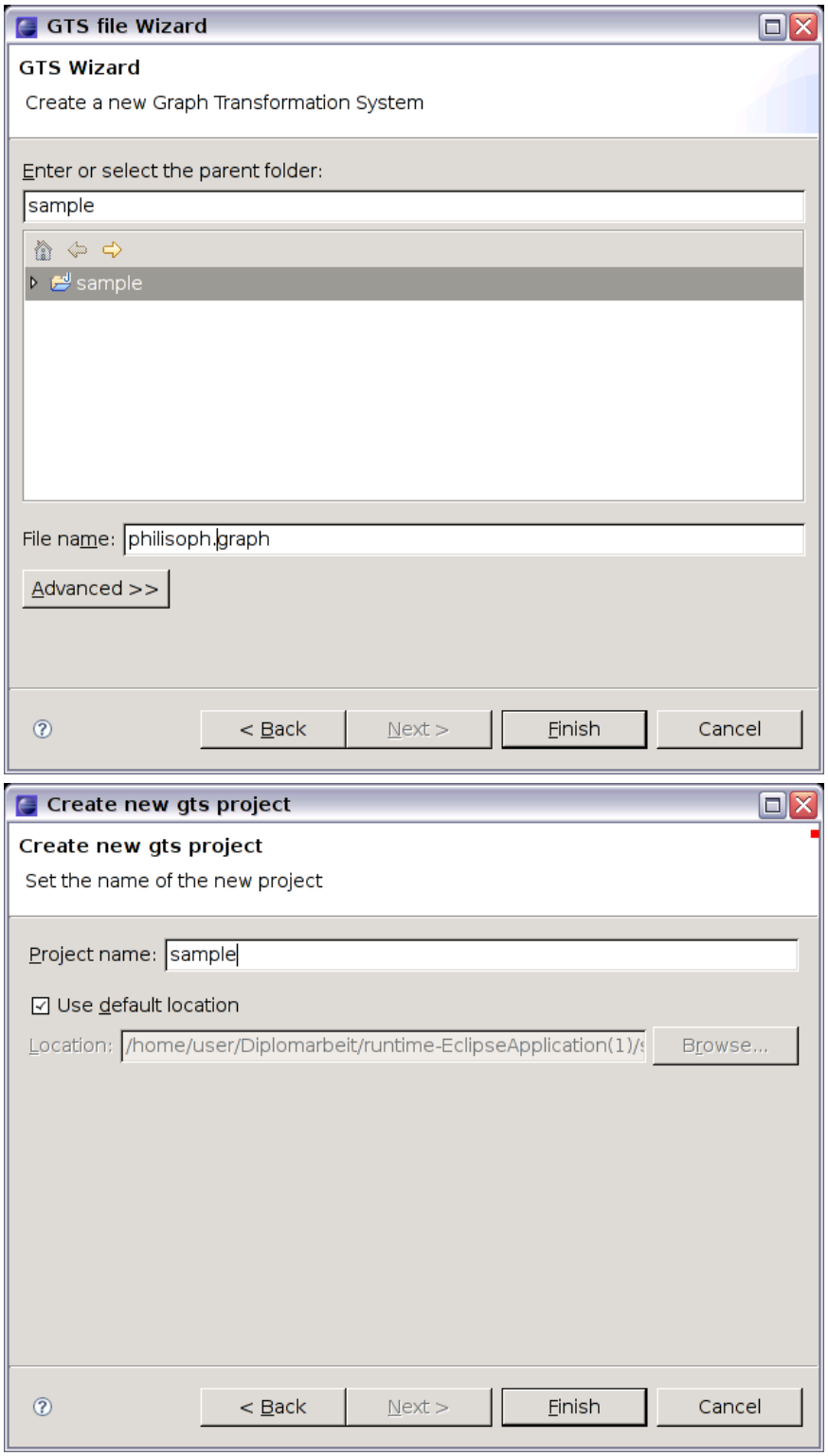


Figure 4.8: File (upper) and project (lower) creation wizards.

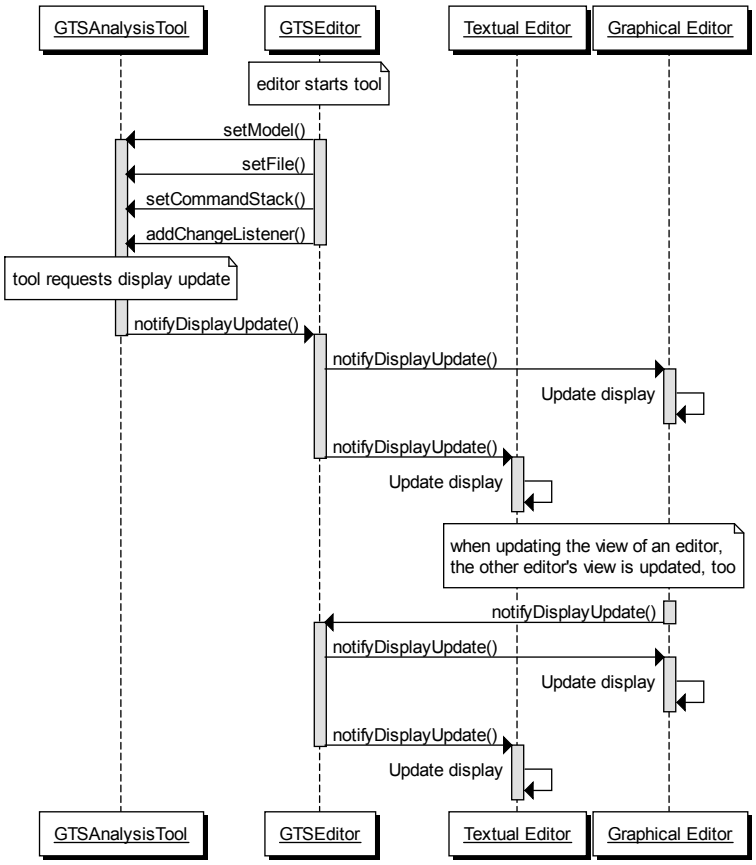


Figure 4.9: UML sequence diagram of the flow of information between the editors and the tools.

The information that is displayed in the editors and the tools should always be synchronized. For that reasons, the information that must be transferred is which graphs, nodes, and edges are currently displayed, so that tools can display graphs and selections of nodes and edges directly in the editors. The sequence diagram in figure 4.9 shows that the GTS editor is the central point of information transfer. Analysis tools can send information to the GTS editor about activated graphs and selections. This information is then sent to all the editors which update their view accordingly. This mechanism can also be used for synchronizing the view between the embedded editors by sending a notification with the graph to display from one of the editors to the platform. The platform then updates the view of both editors.

An eclipse extension point is used for plugging in analysis tools. As described in section 2.4, creating an extension point comes in two parts: designing the extension point definition and providing an interface which must be implemented by the extension. Basically, extension points only offer a flow of information *to* the extension, but a flow of information *from* the extension is not realized. However, the analysis tools need to send back notifications, so that graphs can be displayed in the editors. This is described in figure 4.9. Therefore, the interface of the extension point must provide methods to set the GTS model, the command stack, and the file in the analysis tools. The file is needed, so that analysis tools that generate further files can name them reasonably. The command stack is needed for analysis tools that modify the model. To receive information from the analysis

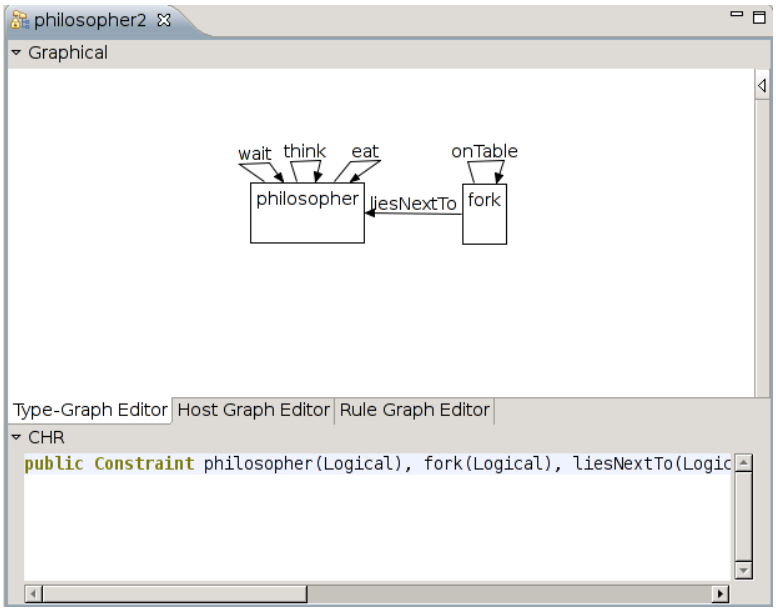


Figure 4.10: This screen-shot shows a multi-bar editor with two editors embedded. The upper editor is a multi-page editor, the lower one is a text editor.

tools, methods for the listener pattern must be added, so that the platform can add itself as listener to the analysis tool to get notified when a new graph or a part of a graph should be displayed. Because of that, a listener interface must be created that can be implemented by the platform to receive notifications from the analysis tools. The extension point definition contains only three attributes: The *identifier* of the extension, the *name* that is displayed to the user when the extensions are listed, and the *class* that is loaded when starting the extension.

4.2.2 Implementation

This section is split up into several parts: First, I provide some details about the multi-bar editor implementation, then I show how to save and load EMF models from files. After that, I explain the methods of the interface for the extension point. The complete interface is also listed in appendix C.2. The next topic is the registration and visibility of actions in the action bar. Then, there is a short section on how to load extensions and display them in the combo-box of the action bar. At the end, I show how the communication of the plug-ins with the editors is realized. The platform itself can be found in the main project `org.uniulm.gts`. The extension point definition is found in the project `org.uniulm.gts.analysistool`.

4.2.2.1 Multi-bar editor

The multi-bar editor is implemented in the class `MultiBarEditorPart` which is similar to a `MultiPageEditorPart`, but with a `MultiBar` widget instead of a `CTabFolder` widget as underlying structure. Figure 4.10 shows an example of a multi page editor and a multi-bar editor where the multi-page editor is embedded in a multi-bar editor. These editors embed further editors on separate pages. Extending the

`MultiPageEditorPart` and overwriting all the methods that access the `CTabFolder` widget is, however, problematic, because there are methods that are marked *final* and can not be overwritten. As a consequence, the editor has to be reimplemented. Furthermore, some concepts from the `MultiPageEditorPart` cannot be supported. The `MultiPageEditorPart` provides several means of modifying the *active* page which refers to the currently displayed page in the `CTabFolder` widget. However, a multi-bar does not have an active bar because everything is displayed in parallel. For that reason, these features are not implemented. Additionally, the method for adding new editors has been changed. A float value has to be given when adding a new bar for an editor. This float value gives the relative height of the newly added bar. When resizing the window, collapsing or expanding a bar, these additional float values are evaluated to recalculate the height of the expanded bars. The higher the float value is, the more space the editor gains. The upper editor in the multi-bar editor in figure 4.10 has a float value that is twice as high as the one of the lower editor. This `MultiBarEditorPart` is extended by the `GTSEditor` (which is referred to as *GTS editor* in the remainder).

4.2.2.2 Saving/Loading GTS models

The GTS editor is the main editor of the platform. It features loading and saving mechanisms for GTS models. Loading and saving EMF models is done by serializing them into XML files. This service is directly provided by EMF: An EMF `Resource` object is created from an eclipse `Path` object, then save and load operations can be applied to the `Resource` object. An eclipse `Path` object contains the path to a file or folder. After doing a load on a resource one can retrieve a list of its contents. There is the possibility of saving multiple contents, i.e. multiple GTS models, but for this application only one model is saved. As described in section 4.1.2, the modeling technique of containers and containments is used in EMF. Objects can be inside of at most one containment. This gives the model a tree-like structure which is needed for persistence, because EMF uses XML to serialize the model and XML has a tree-like structure. If there are elements that are currently not in a containment, saving fails. Therefore, special attention must be paid when modifying the model (see section 4.1.1), so that all referenced objects are contained in another object. The functionality of loading and saving EMF models is bundled in the class `ModelManager` which provides the functionality to load and save models to files and to create new models when starting a fresh GTS.

4.2.2.3 Wizards

Other features that have to be implemented are project and file creation wizards for GTS. In section 4.7, I explain why an own project is needed for GTS. However, the project creation wizard (bundled in the class `NewGTSProjectWizard`) creates a new eclipse plug-in project, so that plug-ins can be added as dependencies. The creation is done via JET which is described in section 4.6. JET is used, because it features an easy way to create projects and files in the eclipse workspace. The new file wizard (bundled in the class `ModelWizard`) creates a new GTS model with only the type graph, but without host and rule graphs. This model is then serialized to the file which the user has entered in the wizard. The creation of the initial model is done by the `ModelManager` class mentioned in the previous section.

4.2.2.4 Extension point

The implementation of the extension point is very close to the design described above. The interface's name is `IGTSAnalysisTool`. However, an additional method is added to the

```
1  public void computeTools(List<String> names, List<IGTSAnalysisTool> tools){
2   names.clear(); //save the names
3   tools.clear(); //save the tools instances
4   IExtensionRegistry registry=Platform.getExtensionRegistry();
5   IExtensionPoint ep = registry.getExtensionPoint("org.uniulm.gts.analysistool");
6   IExtension[] exts=ep.getExtensions();
7   for(IExtension ext:exts){
8    for(IConfigurationElement c:ext.getConfigurationElements()){
9     try{
10     Object tool=c.createExecutableExtension("class");
11         if(tool instanceof IGTSAnalysisTool){
12      names.add(c.getAttribute("name"));
13      tools.add((IGTSAnalysisTool)tool);
14         }
15    }catch(Exception e){...}
16   }
17  }
18 }
```

Listing 4.3: Load extensions from an extension point.

interface in order to tell the analysis tool that it should start its analysis (`runAnalysis()`). This method is called after the command-stack, the file, and the model are set for the tool. The complete code for the interfaces and the extension point definition can be found in appendix C.2. A guide for creating an analysis tool can be found in section 4.5.2.

Loading extensions in eclipse is done in multiple steps. Listing 4.3 shows the used code that is used to load all the analysis tools from the extension point and save them to two lists. First, the extension registry must be fetched (line 4). From there, one gets the desired extension point by giving its identifier (line 5). The extension point contains a list of its extensions (line 6). Each extension consists of a list of `ConfigurationElements` where each element contains a definition of an extension. The attributes of the extension can be queried for each `ConfigurationElement`. From the attribute `class` an executable class is loaded and instantiated (line 10). This object is then saved in a list of tools (line 13). The `name` attribute is saved in a list of strings. This list is used for displaying the analysis tools in a combo box (line 12).

4.2.2.5 Action bar

As described in the design section (4.2.1), the platform has a common action bar for all the editors. This action bar's main element is a combo-box that displays all available analysis tools. This combo box is implemented in the class `PluginSelectToolbarContribution`. Elements, except buttons, that are added to an action bar, are called *action bar contributions* in the eclipse framework or *contribution* if the context is clear. This combo-box contribution handles the loading of the extensions as described in the previous section and launches the tools by clicking on their entry. Listing 4.4 shows how an analysis tool extension is started. When launching an analysis tool extension, the model, the file, and the command stack of the currently active editor are set in the tool. Furthermore, the currently used GTS editor is added as a listener to the tool after removing all old listeners (lines 9–14). Finally, the `runAnalysis()` method is called (line 15) within a `SafeRunnable` instance which provides an extra thread that catches exceptions (lines 2–15). The active model, file, and command stack are always saved in the combo-box in corresponding variables. Further attention has to be paid when using multiple instances of the GTS editor in parallel, because they all share the same action bar. Therefore, each editor has a reference to the combo box in the action bar and every time an editor receives focus, it sets the model, file, and command stack in the combo box, so that the tools receive the content of the currently displayed editor.

```
1  final IGTSAnalysisTool t=tools.get(cb.getSelectionIndex()-1);
2  ISafeRunnable run= new ISafeRunnable(){
3   public void handleException(Throwable exception) {
4    String[] but={"Ok"};
5    MessageDialog md=new MessageDialog(null, "Error executing tool", null, "An
         unexpected error occurred during tool execution", 0, but, 0);
6    md.open();
7   }
8   public void run() throws Exception {
9   t.setModel(model);
10  t.setCommandStack(commandStack);
11  t.setFile(file);
12  t.removeAllListeners();
13  if(editor!=null)
14   t.addChangeListener(PluginSelectToolbarContribution.this.editor);
15  t.runAnalysis();
16  }
17 };
```

Listing 4.4: Start selected extension from `PluginSelectToolbarContribution`.

As already mentioned, the editors can register their own actions to the action bar. To realize that, the action bar has the method `registerRetargetActions(List<IAction>)` to add actions to it. Not all actions are shown permanently, but only when the displayed editors need them. Because of that, every time when the platform is notified about a display update for the editors, it queries the currently active actions of the editor and sends them to the action bar, which updates the displayed actions accordingly.

Up to now, only tools can send information about active graphs, nodes, and edges to the platform, but there is no communication between the tools. However, the interface already provides methods for setting active graphs, nodes, and edges. This might be especially useful for tools that combine multiple other tools, e.g. analysis environments combining several analysis tools or alternative display modes.

4.3 The graphical editor

This section covers the realization of the graphical editor. The design section explains the graphical user interface of the editor. The editor is based on GEF and implements the MVC paradigm. This is described in the implementation section together with the tools and actions that are provided for the editor. A sample computation for the creation of a GTS is shown at the end of this section.

4.3.1 Design

This section covers the design of the graphical editor's GUI. In a GTS, there are multiple types of graphs (type graph, host graphs, and rule graphs) which need to be edited by this editor. The editor should provide an intuitive method of editing together with visual feedback about invalid model states (section 4.1.1.2). Editing type graphs is basically editing a normal graph where identifiers for nodes and edges must be supplied. Because of that, a standard editor with tools for creating nodes and edges can be chosen. Figure 4.11 shows a standard GEF editor with a palette on its right containing a selection, a node-, and an edge-creation tool. Typed graph editors are needed for editing rule and host graphs. For typed graphs, the type graph or at least a list of the types for nodes and edges, must be available. However, displaying the type graph directly to choose the edges and nodes that should be created next is better, because it shows directly the context of the node or edge type that should be created. Using only a list of types (like in Groove or AGG) leads to switching back to the type graph editor to look up the adjacent edges and nodes of the selected node

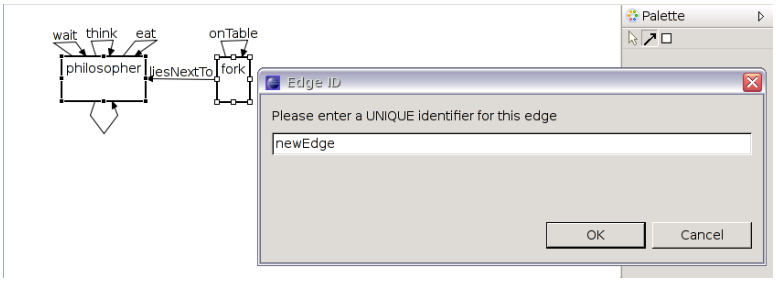


Figure 4.11: Standard graphical editor for creating type graphs where the identifier of a new edge is queried.

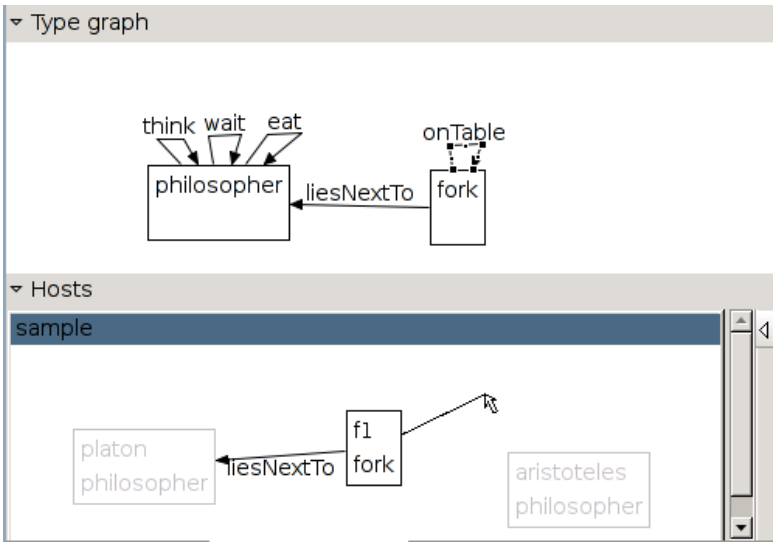


Figure 4.12: Host graph editor together with a type graph, to select the types of nodes and edges.

or edge. Furthermore, when creating new edges in a typed graph, the nodes that cannot be selected as source (or target) should be grayed out, so that the user cannot add edges to the graph that are incorrect according to the type graph. An editor for creating a host graph (which is in fact a typed graph) could be realized as shown in figure 4.12 with the type graph displayed above the typed graph editor. The type graph is displayed above the host graph editor and can be used to select the type of the node or edge that should be created next. A new edge of the type *onTable* is currently created in the screen-shot. This edge can only connect nodes of the *fork* type, therefore, the *philosopher* nodes are grayed out. Typed nodes and typed edges can be created in the type graph by selecting the corresponding type node or edge in the type graph and then clicking at the appropriate position in the typed graph editor.

In a GTS, there are in general several host graphs and rules. Because of that, the typed graph editor must offer a way to select the graph that should be displayed in the editor. Figure 4.13 shows a typed graph editor with an embedded list of the available graphs on the right side. A new graph can be displayed by clicking on the according entry. Furthermore, the name of the graph must be displayed in the graphical representation of the graph.

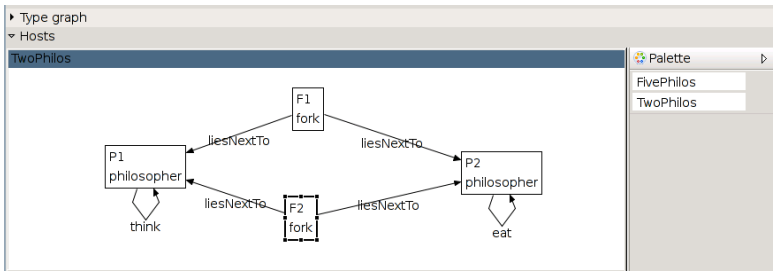


Figure 4.13: Host graph editor together with a list of available graphs on the right side. The type graph is collapsed.

Another variant of typed graphs are rule graphs. There are several ways of giving a graphical notation of rule graphs, as described in section 2.2. In the two and three graph notations, the morphisms between the graphs must be provided explicitly. The one graph notation implicitly contains these morphisms. I have decided to use the one graph notation, because it is a compact and simple notation without the need of manually specifying morphisms. Another benefit of this is that the typed graph editor described above can also be used for editing rule graphs. However, for rule graphs, a context menu has to be added, so that the nodes and edges can be made green, red, or black. Figure 4.14 shows the rule editor with the context menu for moving an edge between the red, black, and green graph of a rule.

As the editors for host and rule graphs both enable the user to edit several graphs, actions must be provided to add and remove host graphs and rules. These actions can be embedded in the action bar of the platform, as shown in figure 4.15.

As just stated, the graphical editor should consist of three editors (for type, host, and rule graphs). These editors are grouped together on multiple pages, so that the user can switch between them. The actions in the action bar contributor should be updated accordingly when the editors are switched. Figure 4.16 shows these pages for the type, rule, and host graph editor. The type graph editor is currently displayed.

4.3.2 Implementation

The implementation is divided into multiple parts. First, I describe how the model view controller (MVC) ([27]) paradigm is implemented in GEF, then I show the implementation of the *edit-policies* for creating the commands to edit the models. This can all be used by the editors for type, rule, and host graphs in common. The next section describes how the editors themselves are implemented and the last section shows how further actions for the editors are added. The graphical editors can be found in the project `org.uniulm.gts.graphicaleditors`.

4.3.2.1 Model View Controller (MVC)

In GEF, the controller objects are called `Editparts`. GEF already provides a base implementation in form of an `AbstractGraphicalEditpart`, which contains several methods that can be overwritten to add functionality. Table 4.3 shows the methods that are of interest. There are a few more methods, but I present only the ones used in this project.

For the graph model defined in section 4.1.1, there are basically three `EditPart` classes, one for edges, one for nodes, and one for graphs. The `EditPart` instances are created via

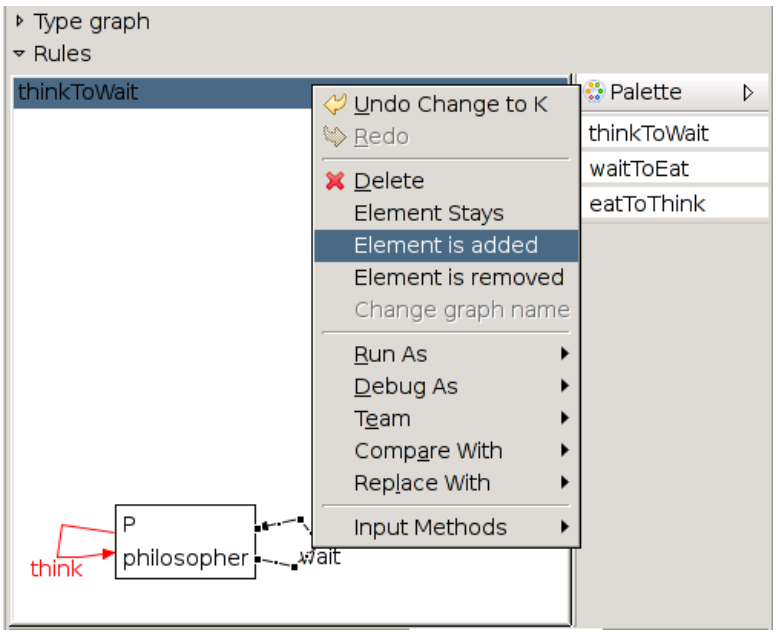


Figure 4.14: Rule graph editor with context menu for changing the graph of an edge or a node in a rule.

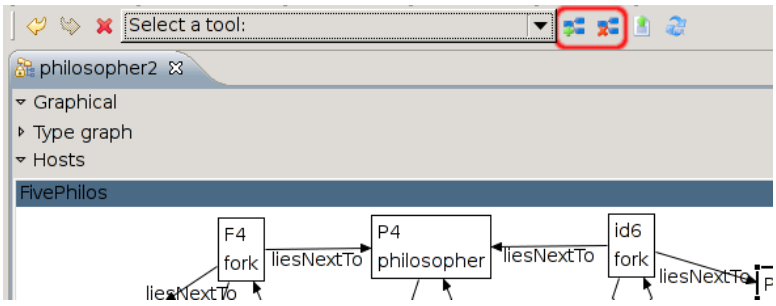


Figure 4.15: Actions for adding and removing host graphs in the action bar of the platform (marked with a red rectangle).

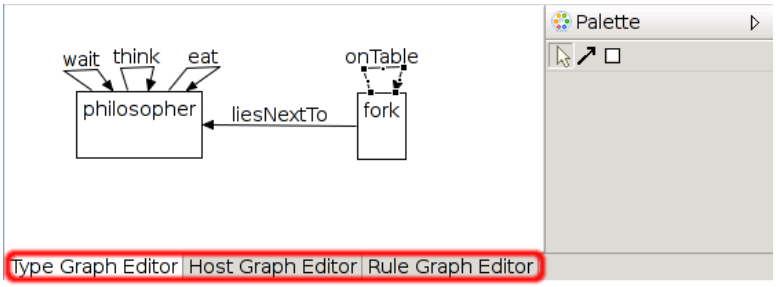


Figure 4.16: Multi page editor containing a rule, host and type graph editor, the type graph editor is currently the active page.

`createFigure()`	create the *view* object
`refreshVisuals()`	refresh the view from the model
`getModel()`	return the model object
`getModelChildren()`	return all children elements of the model
`getModelSourceConnections()`	return all edges that are connected as source
`getModelTargetConnections`	return all edges that are connected as target
`activate()`	activate the edit-part
`deactivate()`	deactivate the edit-part
`createEditPolicies()`	create edit policies for the edit-part

Table 4.3: Important methods, provided by the `AbstractGraphcialEditpart`, that should be implemented by the extending class.

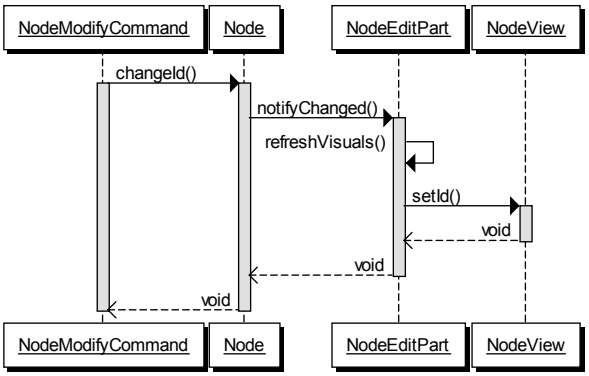


Figure 4.17: UML sequence diagram, showing how the view of a node is updated from the model.

a factory used by the editor. This factory is implemented in the class `GraphEditPartFactory` and returns the corresponding `EditPart` instance for a model element. `EdgeEditPart` instances are created for objects which are instances of the class `IAbstractEdge`. Accordingly, a `NodeEditPart` is created for instances of the `IAbstractNode` class and a `GraphEditPart` is created for objects of the class `IGraphModel`. The edit parts together with their model instances are listed in table 4.4.

All of the edit parts implement the `activate()` and `deactivate()` method in which they add or remove themselves as listener to their model element. To do that, they must implement the EMF `Adapter` interface, which provides the `notifiyChanged()` method. This interface is used by the EMF to notify its listeners. In this method, the edit-parts get notified about changes in the model. The UML sequence diagram in figure 4.17 shows how the edit parts are updated from the model. This example shows how a `Node` object is modified by a *NodeModifyCommand* and how the corresponding `NodeEditPart` is updated. When the `Node` is modified, e.g. by changing its `id` attribute, the `notifyChanged()` method of the corresponding `NodeEditPart` instance is called. In this method, the edit-part itself updates the corresponding view element with the method `refreshVisuals()`. The class `NodeView` implements the graphical representation of a node. Depending on the type of the edit-part, other refresh methods are called. The `GraphEditPart` calls the `refreshChildren()` method which fetches the model element's children, i.e. the nodes, by calling the `getModelChildren()` method. `getModelChildren()` returns a list of all the nodes in the `GraphEditPart`'s model. The `GraphEditPart`

`NodeEditPart`	Controller for (type-, host- and rule-) nodes
`EdgeEditPart`	Controller for (type-, host- and rule-) edges
`GraphModelEditPart`	Controller for (type-, host- and rule-) graphs

Table 4.4: Available `EditPart` classes.

then checks if any nodes or edges are removed or added and updates the edit-parts for the nodes accordingly.

In the `createFigure()` method which is also implemented by all edit parts, a *draw2d* figure is created. Edges are represented as `PolylinePonnection`, nodes as `NodeView`, and graphs as `FreeformLayer` instances. Only for the nodes, the new class `NodeViewPart` is extended from the draw2d `Rectangle` class. Graphs and edges are displayed using standard classes from draw2d. A new class is created for the nodes, because it contains multiple labels what makes updating this view more complex.

4.3.2.2 Edit policies

Before describing the `createEditPolicies()` method of the edit-parts (see table 4.3), I want to explain how editing works in GEF. The root of all editing actions is the `Tool` class. This class sends (upon user interaction) a `Request` instance to an `EditPart`. This `Request` signals that the model shall be edited. The `EditPart` sends this `Request` to its `EditPolicy` instances which create the corresponding `Command` instances to manipulate the model. The commands are already described in section 4.1.2. Therefore, the `GraphEditPart` needs an edit policy for moving around and resizing nodes, as well as adding them. This is the `GraphXYLayoutPolicy`. Changing the name of a graph is handled by the `ChangeGraphNameEditPolicy` that is only added for typed graphs, i.e. rule and host graphs. `NodeEditParts` need edit policies for deleting nodes (`NodeComponentEditPolicy`), adding edges (`EdgeCreateEditPolicy`) , moving the nodes between the graphs of a rule (`TransformTypeEditPolicy`, only for rule graphs), and changing the name of a node (`LabelDirectEditPolicy`). `EdgeEditParts` for edges of a type graph have an edit policy for changing the identifier (`LabelDirectEditPolicy`). For creating edges, another edit-policy must be installed (`ConnectionEndpointEditPolicy`). Edges in rule graphs have a `TransformTypeEditPolicy` installed that generates commands for moving the edge between the graphs of the rule. Handling bend-points of the edges is done in the `EdgeBendpointEditPolicy`. Another edit policy is installed for deleting edges (`ConnectionEditPolicy`). All these edit policies create the commands depending on the model element contained by the edit part. Creating the command is done in the `getCreateCommand()` method. Listing 4.5 shows the `getCreateCommand()` method from the `GraphXYLayoutEditPolicy` class for creating a new `NodeCreateCommand`. In this example, a new command for adding a node to a graph is created. First, it is checked whether the object contained in the request is of the correct class `INode` (line 3). If that is not the case, the edit-policy returns `null`. This means for edit-policies in general that they are not responsible for this request. If the request to create a node is `TypedNodeCreateRequest`, i.e. when a typed node should be created, then the according type node is queried from the request (line 4–7). At the end, a new `NodeCreateCommand` instance is returned (line 8–11) which is initialized with the model object for the graph, the size and location of the new node, the type node, and an identifier for the node which is left blank. The command (section 4.1.2.2) modifies the model according to the input. If the type is `null` and the graph model is a type graph then a type node is created. If the type node is not null, then depending on the type of the graph,

```
protected Command getCreateCommand(CreateRequest request) {
 Object childClass = request.getNewObject();
 if (childClass instanceof INode) {
  ITypeNode type=null;
  if(request instanceof TypedNodeCreateRequest){
   type=((TypedNodeCreateRequest)request).getTypeNode();
  }
  return new NodeCreateCommand(
    (IGraphModel)getHost().getModel(),
    (Rectangle)getConstraintFor(request),
    type,"");
 }
 return null;
}
```

Listing 4.5: `getCreateCommand()` from the `GraphXYLayoutCreateEditPolicy`.

a node for a rule graph or a node for a host graph is created. If the identifier for the node is left blank, then the command queries the name from the user.

4.3.2.3 Editor

GEF has several base classes for creating editors. I have chosen the `GraphicalEditorWithFlyoutPalette`, because it already contains a palette which can hold tools for creating nodes and edges. This base class has to be extended to create custom editors. Basically, the two methods `initializeGraphicalViewer` and `configureGraphicalViewer` have to be implemented for setting the model that should be displayed and for configuring the editor. Configurations include setting context menus and an edit-part factory. At first, details are given for the type graph editor and then for the typed graph editors, i.e. the host and rule graph editors.

For the type graph editor, an instance of the `GraphEditPartFactory` as the edit part factory is set and a palette with two tools, one for creating nodes and one for creating edges (bundled the class `TypeGraphPaletteFactory`), is added. The two tools are already provided by GEF.

Implementing the typed graph editors is more complex, as the type graph is also needed for selecting the type nodes and edges. This is realized by embedding two editors in a multi-bar editor, as shown in figure 4.12. The upper one displays the type graph (`GraphViewer`). The lower one is the editor for typed graphs itself (`TypedGaphEditor`). The `GraphViewer` is a GEF editor with only a selection tool and no palette, so that no editing can be done there. The typed graph editor has a reference to a `GraphViewer` and adds itself as selection listener to it. Therefore, every time the user selects a new node or edge in the `GraphViewer`, a notification is sent to the `TypedGraphEditor`. When it receives notification that a new node or edge was selected, the active tool is changed. If a node is selected in the graph viewer, a `TypedNodeCreationTool` becomes the active tool. When an edge is selected, a `TypedEdgeCreationTool` becomes active. Therefore, the graph viewer is the tool selector for this editor. When no node or edge is selected, a `SelectionTool` is activated in the `TypedGraphEditor`. The palette is used, as well, but it displays a list of available typed graphs and provides the possibility to select them (see figure 4.13). To use the palette as a list of typed graphs, the editor adds itself as `PaletteListener` to get notified when the active *tool* is changed. During a tool change event, the editor removes the old graph from the display and sets the selected graph as the new model. The elements in the palette are in fact selection tool entries with different labels, so that the identifier of the graph is displayed. The typed graph editor also adds itself as listener to the `GTS` model element, so that it gets notified when graphs are

```
1 public void run(){
2  if(tge.getCurrentGraphModel() instanceof IHostGraphmodel){
3   ITypedGraphModel g=(ITypedGraphModel)tge.getCurrentGraphModel();
4   GraphDeleteCommand host=new GraphDeleteCommand(g);
5   cs.execute(host);
6  }
7 }
```

Listing 4.6: `run()` method from the `DeleteGraphAction` super class.

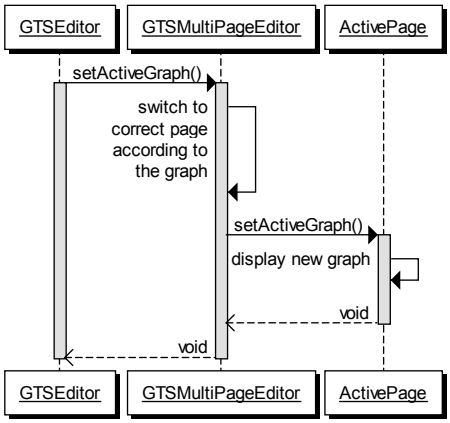


Figure 4.18: Flow of information from the platform to the graphical editor.

added or removed. When a new graph is added, the editor displays this new graph and adds an according entry to the palette. When a graph gets deleted, it displays the next graph in the list and removes the current one from the palette. Furthermore, the editor needs a reference to the list of typed graphs it edits, the command stack, and the GTS model itself.The editor described above can be used as host and rule graph editor.

For each typed graph editor two actions are registered in the action bar to add and delete host and rule graphs, respectively. Actions provide a method `run()`, that is called when clicking on a toolbar entry associated with this action. The actions for adding and removing typed graphs each have super classes that implement the `run()` method. Listing 4.6 shows the `run()` method of the `DeleteGraphAction`. The action has references to the typed graph editor in the variable `tge` and to the command stack (`cs`). The method retrieves the current graph model from the editor, creates a `GraphDeleteCommand` from it, and adds it to the command stack. The editor is then updated due to the changes that are made in the model. The `run()` method of the `NewGraphAction` first asks the user to input a unique identifier and then creates the according command for creating a new graph. The methods for retrieving the correct command and for retrieving a list of available graphs (to check if the entered identifier is unique) must be implemented by the actions for host and rule graphs, respectively.

The three editors described above (for type, host, and rule graphs) are each added on a separate page in a `MultiPageEditorPart` to become the final graphical editor for GTS (`GTSMultiPageEditor`), as can be seen in figure 4.16. All editors, as well as the `MultiBar`- and `MultiPageEditorParts` implement the analysis tool interface from the extension-point, described in section 4.2.2.4, to simplify the flow of information between the type-, rule-, and host graph editors and the platform. Figure 4.18 shows a sequence diagram for the flow of information when the platform notifies the editor about a new graph that should be displayed. The multi page editor may receive notifications

```
1  public boolean evaluate(EditPart editpart) {
2   if(editpart != null){
3    if(TypedEdgeCreationTool.this.getState() == TypedEdgeCreationTool.STATE_INITIAL
4        && editpart instanceof NodeEditPart
5        && ((NodeEditPart)editpart).getModel() instanceof INode
6            && typeEdge.getSrc().equals(((INode)((NodeEditPart)editpart).getModel()).
                 getType()))
7      return true;
8    else if(TypedEdgeCreationTool.this.getState() == TypedEdgeCreationTool.
         STATE_CONNECTION_STARTED
9        && editpart instanceof NodeEditPart
10       && ((NodeEditPart)editpart).getModel() instanceof INode
11       && typeEdge.getTgt().equals(((INode)((NodeEditPart)editpart).getModel()).getType
             ()))
12     return true;
13   }
14   return false;
15  }
```

Listing 4.7: `evaluate()` method of the `Conditional` class.

from the platform to display a new graph, so it checks what type of graph it is (type, rule, or host), displays the corresponding page, and signals the editor on this page to display the given graph. When certain nodes and edges shall be selected, the multi page editor forwards the list of nodes and edges to the active editor. The active editor then translates this list into a `Selection` for the GEF editor. Selections are a list of objects that are marked as *selected* in the editor. When the user selects another graph in one of the typed graph editors, the editor notifies the multi page editor about a graph change. When the multi page editor gets notified about a graph change, it forwards this information to the platform. When a page change occurs, it also notifies the platform, about the new displayed graph. This is shown in figure 4.9. The platform can then update the other editor, so that the same graphs are displayed in both editors.

4.3.2.4 Tools and actions in the editors

For the creation of typed nodes and edges, two special tools are needed that contain the type of the new element. These two tools can be found in the classes `TypedNodeCreationTool` and `TypedEdgeCreationTool`. The custom tools for creating typed edges and nodes are made the active tools when a selection change event in the graph viewer occurs. Consequently, the first element of the current selection is set as the type for the next created element. The tool sends a `TypedNodeRequest` (or `TypedEdgeRequest`) which contains the currently active type node (edge) to the according controller object. The typed edge creation tool has an additional feature. As the user should be informed very early about valid (and invalid) editing options, the editor adds itself as state change listener to the tool, so that the editor gets notified when the tool is selected when the source has been selected and when the operation is finished. The editor reacts on these events with graying out the nodes to which the edge cannot be connected, i.e. all nodes of the wrong type. This change of colors is done in the model itself. As described in 4.1.2, each model references an object that saves its visual information. This object is used to store whether a node may be used or not. This is then displayed accordingly by the editor. The nodes are not only grayed out but also inaccessible for the tool. To realize this, the `getTargetConditional()` method from the `CreationTool` base class is used. It returns a `Conditional` instance. This class provides the method `evaluate()` which returns a boolean value. This implementation returns true if the targeted node is valid and false otherwise. Listing 4.7 shows the implementation of the *evaluate()* method. This conditional first checks in which state the tool is, i.e. wether the source or target must be selected next (lines 3 and 8). Then it checks if the target or source node currently pointed at is of the correct type, according to the source and target of the type edge (lines 4–6 and

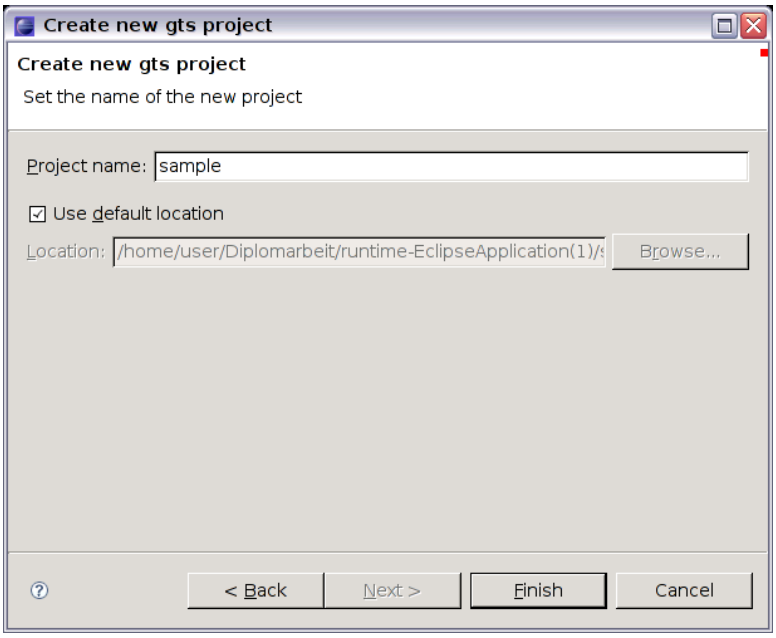


Figure 4.19: Wizard for creating a new GTS project.

9–11). If the node is valid, then `true` is returned, `false` otherwise. With this mechanism, no invalid model states in regard to the type graph can be produced.

For the rule graphs, there must exist a possibility to set the graph of the nodes and edges to R', L', and K depending on whether they are removed, added, or if they stay during the rule application. Therefore, the actions `TransformTypeToKSelectAction`, `TransformTypeToLSelectAction`, and `TransformTypeToRSelectAction` are added to the context menu of the typed graph editor. The entries in the context menu are only available if the `TransformTypeEditPolicy` is added to the node or edge, i.e. the entries are only available in the rule graph editor. By opening the context menu of a node or edge and selecting one of the actions, the selected node or edge is moved to graph L', R', or K (depending on the selected entry). As a result, the action creates a corresponding command and adds it to the command stack.

4.3.3 Sample computation

In this section, I want to show how the GTS version of the dining philosophers problem can be created with the graphical editor. First, a new GTS project must be started called *sample* (figure 4.19). Then a new GTS file is created by launching the according wizard (figure 4.20). As a first step in creating the GTS for the dining philosophers, the type graph must be created, by selecting the according tools to create nodes and edges. The user is prompted to enter a unique identifier for each edge and node. Figure 4.21 shows the complete type graph for the dining philosophers problem.

In the next step, some rules are added. After switching to the rule graph editor page, the button in the bar is clicked to create a new rule (the rule *thinkToWait* is created), see figure 4.22. Now the type graph viewer can be used to select the *philosopher* node and click on the rule graph editor to add this node to the graph. The new node is called *P*. Now, two edges, one of type *think* and one of type *wait*, must be added as loops to the philosopher node *P*. The resulting graph is depicted in figure 4.23. As the *think* edge is

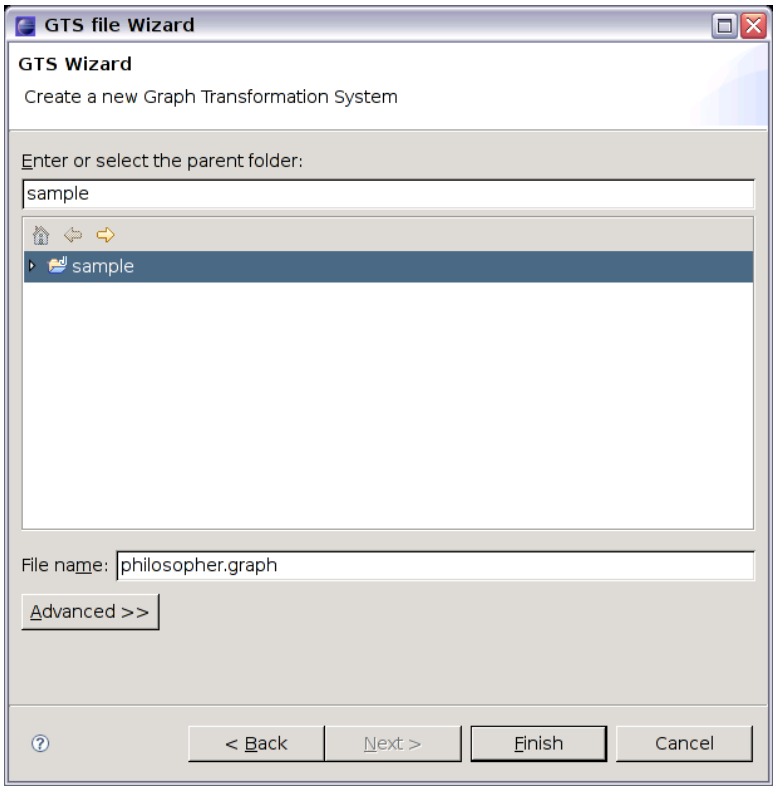


Figure 4.20: Wizard for creating a new GTS file.

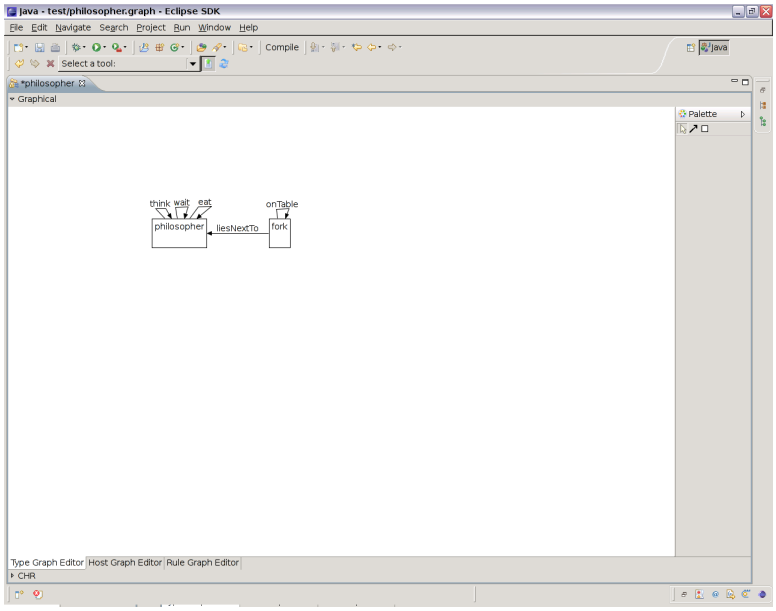


Figure 4.21: The type graph for the dinging philosophers problem.

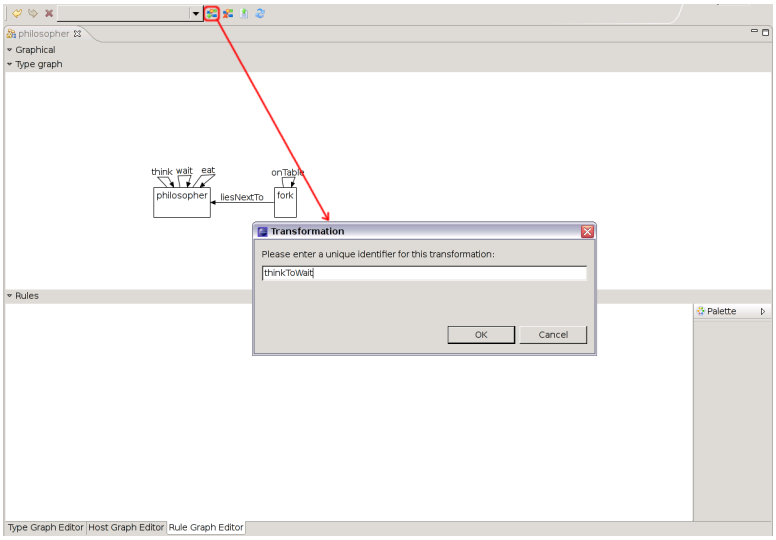


Figure 4.22: Create a new rule for the GTS by clicking on the according icon.

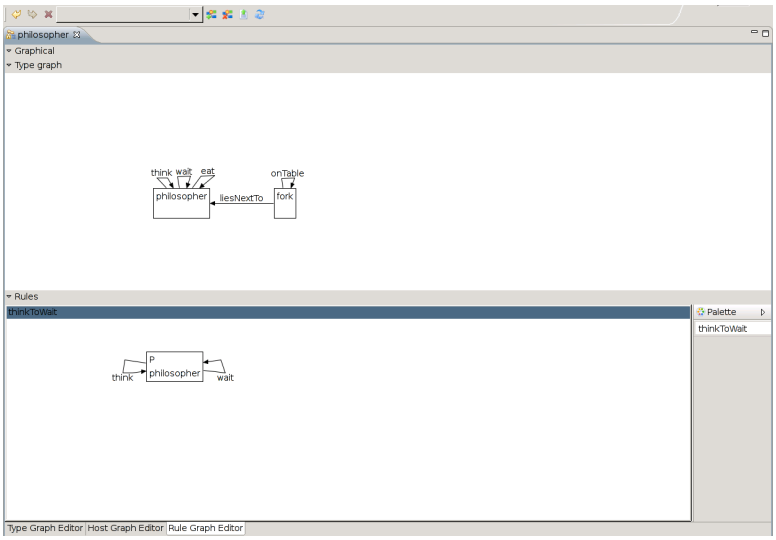


Figure 4.23: Node and two edges added to the rule *thinkToWait*.

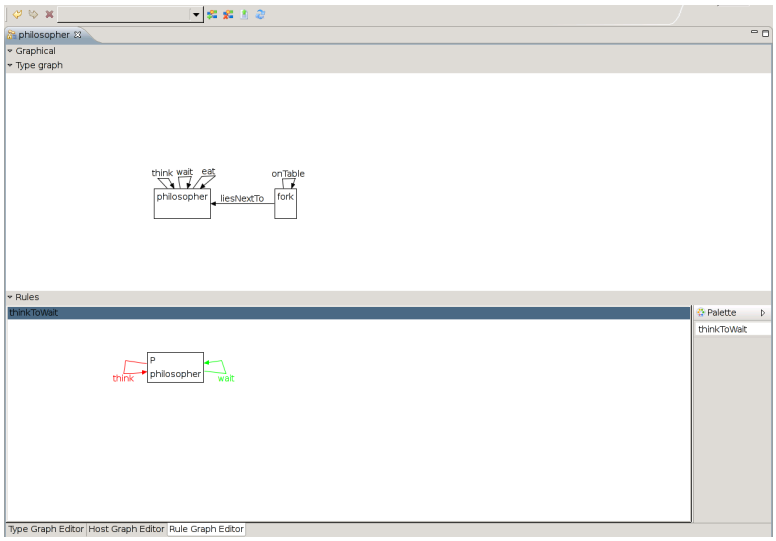


Figure 4.24: Finished rule *thinkToWait*, the *think* edge is removed and the *wait* edge is added by this rule.

removed during rule application, it is selected and the context menu is brought up. When the entry "*Element is removed*" is selected, the edge turns red symbolizing that it is removed during rule application. The *wait* edge is modified by selecting the "*Element is added*" entry from the context menu resulting in the graph shown in figure 4.24. Further rule graphs can be created in the same manner.

Creating a host graph is done in a similar fashion by switching to host graph page and hitting the button to create a new host graph (figure 4.25). Adding nodes and edges is the same as for rule graphs. Figure 4.26 shows a graph with four philosophers where one is eating.

4.4 The textual editor

Originally, the textual editor for CHR was planned as a lab course at the institute where this thesis has been created and should be integrated into the platform. But there were no results available at the time of implementation, so I have decided to provide an own prototypical implementation. Besides creating an editor, an algorithm must be designed to check if the CHR source encodes a valid GTS and apply the according changes to the GTS model. The CHR source code must of course be dynamically updated when the model is edited in the graphical editors or with other analysis tools. Another feature that is provided for rapidly creating rules and host graphs is the "*easy editing mode*" which offers a simpler encoding of host and rule graphs, which is suitable for editing purposes.

4.4.1 Design

The design of the editor is divided into three parts: The design of the GUI elements of the editor itself, the design of the algorithms that check if the CHR source encodes a valid GTS and update the GTS model accordingly, and, as the third part, the notation which used to display an encoded GTS is described. The notation is shown first.

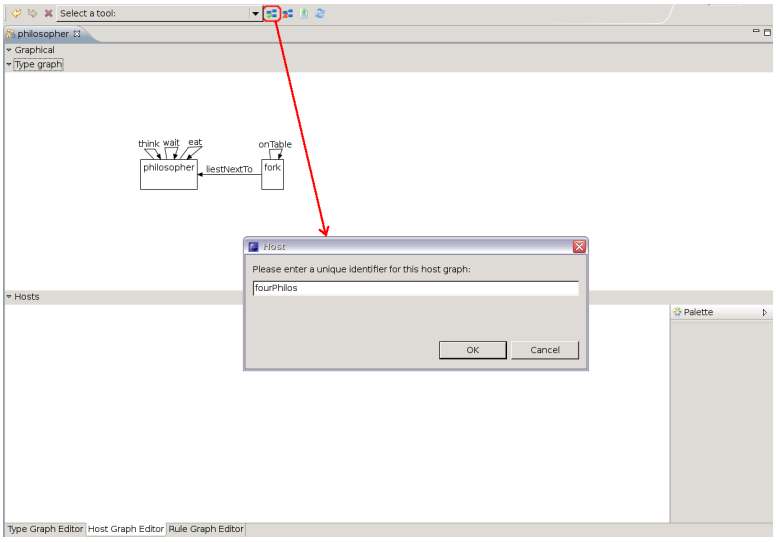


Figure 4.25: Create a new host graph for the GTS by clicking on the according icon.

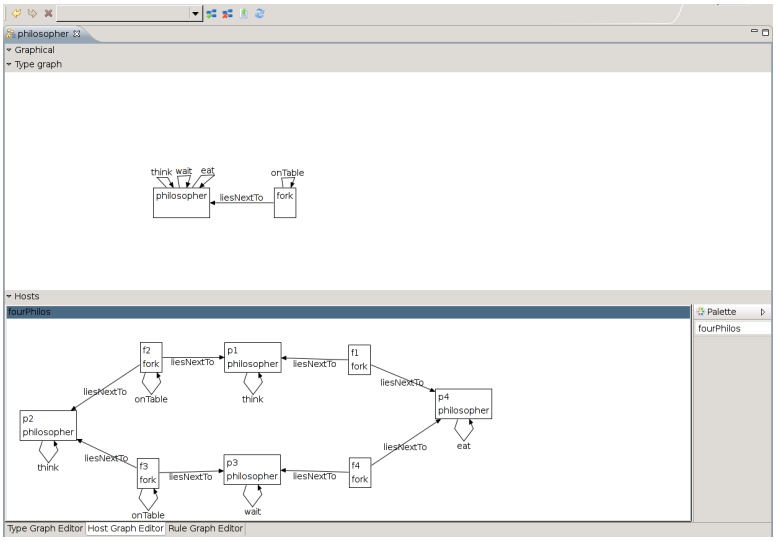
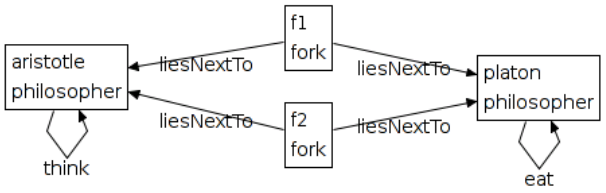


Figure 4.26: Host graph for the dining philosophers problem with four philosophers.




```
:-philosopher(platon,4), fork(f1,2), philosopher(aristotle,4), fork(f2,2),
liesNextTo(del,f1,platon), eat(del,platon,platon), liesNextTo(del,f1,aristotle),
think(del,aristotle,aristotle), liesNextTo(del,f2,platon), liesNextTo(del,f2,aristotle).
```

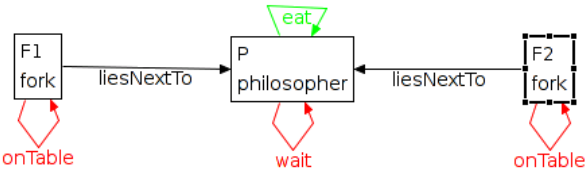
Figure 4.27: Encoding of a host graph.

4.4.1.1 Notation

The textual editor displays the encoding of two graphs at a time, either the type graph or a typed graph together with its type graph. This restriction is set, because the graphical editor described in section 4.3 shows at most two graphs of the GTS: The type graph and eventually a host or a rule graph. Three types of graphs can be displayed, so three modes of displaying the graph in the CHR editor are needed. A type graph is encoded as a list of constraint definitions (see section 2.3). The syntax of the CHR code is kept similar to the one of K.U. Leuven JCHR, because this is the language GTS are exported to for CHR analysis tools (section 4.6). The type graph of the dining philosophers problem, which is shown in figure 2.7, is encoded as follows:

```
1 public Constraint philosopher(Logical,int), fork(Logical,int),
2 think(Logical,Logical,Logical), wait(Logical,Logical,Logical),
3 eat(Logical,Logical,Logical), onTable(Logical,Logical,Logical),
4 liesNextTo(Logical,Logical,Logical);
```

As JCHR is a language with type support, types have to be defined for the constraints. The type `Logical` is a generic type that describes a logical value to which arbitrary values can be assigned. The type `int` stands for an integer value. In the example above, every node and edge type is encoded as a constraint definition. In section 2.3, host graphs are encoded as lists of goal constraints. However, JCHR does not have a notation for defining goals in its source files. Goals can only be given in a java program that uses the JCHR handler. This notation is not very suitable, because it would require a lot of writing for the user. That is why a mixture of JCHR and Prolog based CHR syntax is chosen: The constraint definitions are given at the beginning, according to the encoding of the type graph, then a line starting with "`:-`" followed by a comma-separated list of the encodings of the host graph's nodes and edges which is terminated by a colon ("`.`") is given. Figure 4.27 shows an encoding of a host graph for the example of the dining philosophers problem (section 1.3.2). The encoding of the according type graph can be found above. For the CHR encoding, every node needs a unique identifier, so they are labeled *platon* and *aristotle* for the philosopher nodes and *f1* and *f2* for the fork nodes. The syntax of the encoding of single constraints is equivalent to the one described in section 2.3. The encoding of rule graphs works in a similar way: At the beginning, there are the constraint definitions from the rule graph followed by the encoding of the rules according to the description in section 2.3. The encoding of the rule is kept tight to the syntax of JCHR where all rules are enclosed by a `rules{...}` block and the empty body of a rule is symbolized by the java keyword `true`. Figure 4.28 shows the encoding of the rule *waitToEat* from the dining philosophers example (section 1.3.2), as described in section 2.2. Again, the type graph is not displayed.




```
rules{
philosopher(P,NC_0), fork(F1,NC_1), fork(F2,NC_2), liesNextTo(DN_0,F2,P),
liesNextTo(DN_1,F1,P), onTable(del,F1,F1), onTable(del,F2,F2), wait(del,P,P)
<=>
philosopher(P,NC_0), fork(F1,NC_1-2), fork(F2,NC_2-2),
liesNextTo(DN_0,F2,P), liesNextTo(DN_1,F1,P), eat(del,P,P).
}
```

Figure 4.28: Encoding of a rule graph.

Additionally, another encoding for constraints is supported than the one described in section 2.3. In this encoding, the constraints that represent nodes are unary and they contain only the identifier of the node. The constraints that represent edges are binary constraints and contain the identifiers of the node constraints for the source and target. For the two types of constraints, the degree and the deletion attribute is left away. This can be done for editing, because the degree attribute is only needed for rule application in a CHR program to prevent dangling edges and the deletion attribute is only needed for confluence analysis of CHR programs. The encoding of the host graph from figure 4.27 results in:

```
1 :-philosopher(platon), philosopher(aristotle), fork(f1), fork(f2),
2 liesNextTo(f1,platon), liesNextTo(f1,aristotle),
3 liesNextTo(f2,platon), liesNextTo(f2,aristotle),
4 think(aristotle,aristotle), eat(platon,platon).
```

The rule graph from figure 4.28 results in:

```
1 rules{
2   philosopher(P), fork(F1), fork(F2),
3   liesNextTo(F2,P), liesNextTo(F1,P),
4   onTable(F1,F1), onTable(F2,F2), wait(P,P)
5   <=>
6   philosopher(P), fork(F1), fork(F2),
7   liesNextTo(F2,P), liesNextTo(F1,P), eat(P,P).
8 }
```

4.4.1.2 Encoding/decoding algorithms

In this section, I describe the algorithms for the encoding and decoding of the GTS to and from CHR. Encoding is a simple algorithm for all types of graphs, because each graph can be encoded into a correct CHR program, just as described in section 2.3. However, not every CHR program encodes a correct GTS. This means, if the CHR source is syntactically correct, the CHR rule or goal can still represent an invalid graph, e.g. the graph could have dangling edges. See also section 4.1 for more information about incorrect GTS model states. Therefore, an algorithm must be developed to verify CHR rules and goals, whether they represent a valid host or rule graph. In the previous section, another, simpler encoding for constraints is introduced. However, the encoding and decoding algorithms are not described explicitly for this simpler encoding, because the parts that encode and decode the degree and deletion attributes of the constraints can be omitted in this case.

The verification of type graphs is uncomplicated: if there are only constraint definitions with arity two or three and the correct types as described in the previous section, the encoding represents a valid type graph.

```
1: Input: TypeGraph model, HostGraph model, CHR model
2: separate constraints of CHR model in node and edge constraints
3: if there exist invalid constraints then
4:     model is invalid
5: end if
6: for all edge constraints do
7:     search node constraints with the same identifier as source and target identifier, called source
       and target constraints
8:     if node constraints could not be found then
9:         model is invalid
10:    end if
11:    compare names of source and target constraints and the name of the edge constraints to the
       identifiers of edge and its source and target node in the type graph
12:    if constraint names and node identifiers are unequal then
13:        model is invalid
14:    end if
15: end for
16: for all node constraints do
17:    if identifier of node constraint is duplicate then
18:        model is invalid
19:    end if
20:    count number of edge constraints with the same source or target identifier as the node identifier

21:    if degree attribute of node constraint is unequal counted number then
22:        model is invalid
23:    end if
24: end for
```

Figure 4.29: Pseudo code for verifying a CHR model encoding a host graph.

The verification of a host graph is a bit more complex. An algorithm for verification is given in figure 4.29. For every constraint, one has to check if the constraint name is defined in the constraint definitions and determine if it represents an edge or a node of the GTS. These constraints are called edge constraints and node constraints, respectively. Then, for all edge constraints' target and source terms, a node constraint containing the syntactically same identifier term must be searched. The encoding has dangling edges if an according node constraint cannot be found. If the node constraints are found, their names have to be compared to the identifiers of the source and target type node of the corresponding type edge in the type graph model of the GTS. The type graph model must be used for this, because the encoding of a type graph in CHR does not provide the information to which type nodes the type edges are connected. If constraint names do not match the identifiers of the type nodes, the typing morphism is invalid. The node constraints' second attribute is the degree of this node. Because of that, for every node constraint the edge constraints must be counted that contain the same source or target attribute as the node constraint's identifier attribute. If the calculated number is not equal to the number found in the node constraint, the model is invalid. Furthermore, if two node constraints have the same identifier attribute the model is invalid, too, because a node identifier would have been used twice, which is not allowed.

The procedure for rules is more complex and pseudocode can be found in figure 4.30. The main problem is that the left and right side of a CHR rule partially encode the same graph of a rule. As a consequence, constraints from the head and the body of the rule have to be compared. In a first step, the constraints have to be separated whether they are only on the left, only on the right, or on both sides. Six sets are defined, one for every graph of a GTS rule (R',K, and L') for node and edge constraints. At first, all constraints from the head of the CHR rule are separated into the L'-sets (here is also checked if node constraints with

```
1: Input: type graph model, rule graph model, CHR model
2: separate constraints of CHR rule in node and edge constraints of the graphs R', L' and K.
3: if there exist invalid constraints then
4:     model is invalid
5: end if
6: for all edge constraints do
7:     search node constraints with the same identifier as source and target identifier
8:     if node constraints could not be found then
9:         model is invalid
10:    end if
11:    compare names of source and target constraints and the name of the edge constraint to the
       identifiers of the nodes and edges in the type graph
12:    if constraint names and type node identifiers are unequal then
13:        model is invalid
14:    end if
15:    if the edge constraint's source and target constraint are not in the correct list, according to
       table 4.1 then
16:        model is invalid
17:    end if
18:    if edge is in K and deletion term is ground then
19:        model is invalid
20:    else if edge is in R' or L' and the deletion term is not ground then
21:        model is invalid
22:    end if
23: end for
24: for all node constraints do
25:    if identifier of node constraint is duplicate then
26:        model is invalid
27:    end if
28:    if node constraint is in L' (or R') then
29:        count number of edge constraints in L' (or R') with the same source or target identifier as
           the node identifier
30:        if degree attribute of node constraint is unequal counted number then
31:            model is invalid
32:        end if
33:    else if node constraint is in K then
34:        count number of edge constraints in L' and R' with the same source or target identifier as
           the node identifier and calculate their difference.
35:        calculate the difference of the degree expressions from the head and body constraint
36:        if the differences are unequal then
37:            model is invalid
38:        end if
39:    end if
40: end for
```

Figure 4.30: Pseudo code for verifying a CHR model encoding a rule graph.

the same identifier have already been saved before). Then all constraints from the body are processed: For every constraint, the L'-list it is checked whether it already contains the *same* constraint. Same in this sense means, for node constraints, that the identifier term and the constraint name must be identical. Furthermore, two node constraints must share the same variable in the degree attribute (although the expression can be different on both sides). For edge constraints, the node and target terms must be the same, as well as the name for the deletion variable. If the same constraint is found in the L'-list, it is removed from there and added to the corresponding K list. If it is not available in the L' list, it is added to the R'-list. When the constraints are separated, the validity of the edges is checked. For that, the types of target and source nodes of the corresponding type edge (the type graph model is used here, because the CHR encoding does not provide the typing information) are compared to the names of the source and target node constraints. Additionally, it must be verified that the edges are in the correct list according to their source and target node constraints (see table 4.1 for the allowed combination of edges and nodes). The deletion attributes of the edge constraints have to be checked, as well. For edges in L' and R', the deletion attribute must be ground, i.e. it is a lower case identifier or a number. For edges in K, the deletion attribute must be a variable (i.e. it starts with a capital letter). This variable must be the same for the two constraints in the head and body of the rule. For the node constraints, the degree variable has to be verified. For node constraints in L' and R', the degree attribute must be an integer representing the number of edge constraints associated to it (just as for host graphs). The degree attribute for node constraints in K must be verified in the following way: counting the number of edge constraints in the L' list and the number of edge constraints in the R' list that contain the same source or target attribute as the node constraint, calculating the difference, and comparing it to the difference of the expressions in the head and body constraints of the node constraint. If the differences are unequal then the encoding is invalid.

Although the algorithms are similar in some parts, they still differ too much to reuse the algorithm for host graphs in the algorithm for rule graphs. For example, the test whether the degree attribute is correct for host graphs cannot be used for rule graphs, because rule graphs can also contain variables as degree attributes.

If the encoding of a rule or host graph is valid, it has to be compared to the graph model of the GTS. Changes that result from the CHR encoding must be applied to the graph model. This algorithm is a little bit different for host and rule graphs, but both use the data generated by the previous algorithm, i.e. the separation of the constraints. As already mentioned above, encoded type graphs cannot be decoded to a GTS, because they miss the information how the edges are connected to the nodes. That is why type graphs cannot be edited via CHR. Because of that, there is no need for an update of the GTS model. The update of a host graph model works as follows: for each edge constraint, try to find a corresponding edge in the graph model. Sort the edge constraints into two groups: The group of edge constraints that cannot be found in the model and the group of edge constraints that are already available in the graph. The edges in the model that could not be associated to a constraint are marked for removal. The same is done for node constraints. At first, all edges and nodes that are marked for removal are removed from the graph model. Then, for each constraint that is in the set of node (edge) constraints and has no representation in the model, a new node (edge) is added to the model.

The algorithm for updating a rule graph is similar, except that it has to check if the nodes and edges have been moved between the graphs they belong to (L',R', or K). In the case of a move, commands to modify the corresponding edge or node must be created. In this algorithm edges and nodes are first removed, then nodes and edges are modified, and at last, new nodes and edges are added. It is important that elements are first removed, then modified, and then added, because otherwise invalid model states can occur.

4.4.1.3 Editor

In this section, I want to describe the design of the text editor itself. The editor must give the possibility to display a given type, host, or rule graph (encoded in CHR) and to allow editing of these graphs. The algorithms in the previous section provide this functionality, they only need a representation of the CHR source as a model that contains rules and goals consisting of constraints. For generating a model from text, a parser is needed which reads out the source, checks it for syntactical correctness, and generates a model from it. A framework for generating models from a textual representation is TEF ([35]). For TEF, a grammar must be created which describes the syntax of the text and associates elements of the model to parts of the input text. TEF can then parse the text and return an EMF model of the textual representation. Furthermore, TEF can create a textual representation from a given model, what is called *pretty printing*. In the following, I describe the model used to represent the CHR rules, goals, and constraints. After that, further features of the editor are explained.

I do not model all the aspects of CHR, because a full featured CHR editor is not desired. Only the aspects that are needed for representing constraint definitions, rules, and goals for the encoding of a GTS are regarded. The model is shown in the UML class diagram in figure 4.31. Constraint definitions consist of a name and a list of types for its terms. Constraints consist of a reference to a constraint definition and a list for the terms where each term can be an expression. For the expressions, only addition and subtraction of numbers and literals is modeled, because multiplication is not needed for encoding GTS. Numbers are arbitrary integers while literals are strings. I do not make a distinction of ground and variable terms in the model, because these properties are checked by the algorithms. Inputs are a list of constraints while rules are modeled as two lists of constraints, one for the head and one for the body of a rule. The corresponding grammar is a context free grammar that combines the CHR model with the notation described in the beginning of section 4.4.1. This grammar is further described in the implementation section.

The editor itself must be synchronized with the graphical editor. The model can be used for this purpose. The CHR editor observes the model and updates the displayed text when changes occur. When the currently displayed graph is changed in the graphical editor, it notifies the platform about it. The platform then sends this notification to the textual editor. After that, the editor creates a new CHR encoding of the new graph model and displays it. This flow of information is visualized in figure 4.9. A possibility to select another graph in the CHR editor is currently not realized, because the graphical editor can be used for this. This could be realized in form of a context menu that shows all the available graphs. However, the textual editor would then have to notify the platform about the change of the active graph, so that the graphical editor can be synchronized accordingly.

The editor must provide information about errors in the CHR source. Syntactical errors are automatically detected by the parsing functionality of TEF, as well as referencing errors from constraints to constraint definitions. Other encoding errors which are discovered by the algorithms described above must also be shown as error annotations in the text editor, together with a message that describes the error. The screen-shot in figure 4.32 shows how this annotation mechanism should look like. This example encodes a host graph for the dining philosophers problem with two philosophers where one is eating and the other is thinking. The degree attribute of one philosopher contains the wrong number. This information is also shown in the pop-up.

In section 4.4.1.1, another simpler encoding is introduced, so an action is needed that switches between the encodings. Figure 4.33 shows actions of the CHR editor embedded in the action bar. The second action is for refreshing the encoding from the current graph model. This is needed, when too many errors occur while modifying the CHR source and when the user wants to start again with the original encoding.

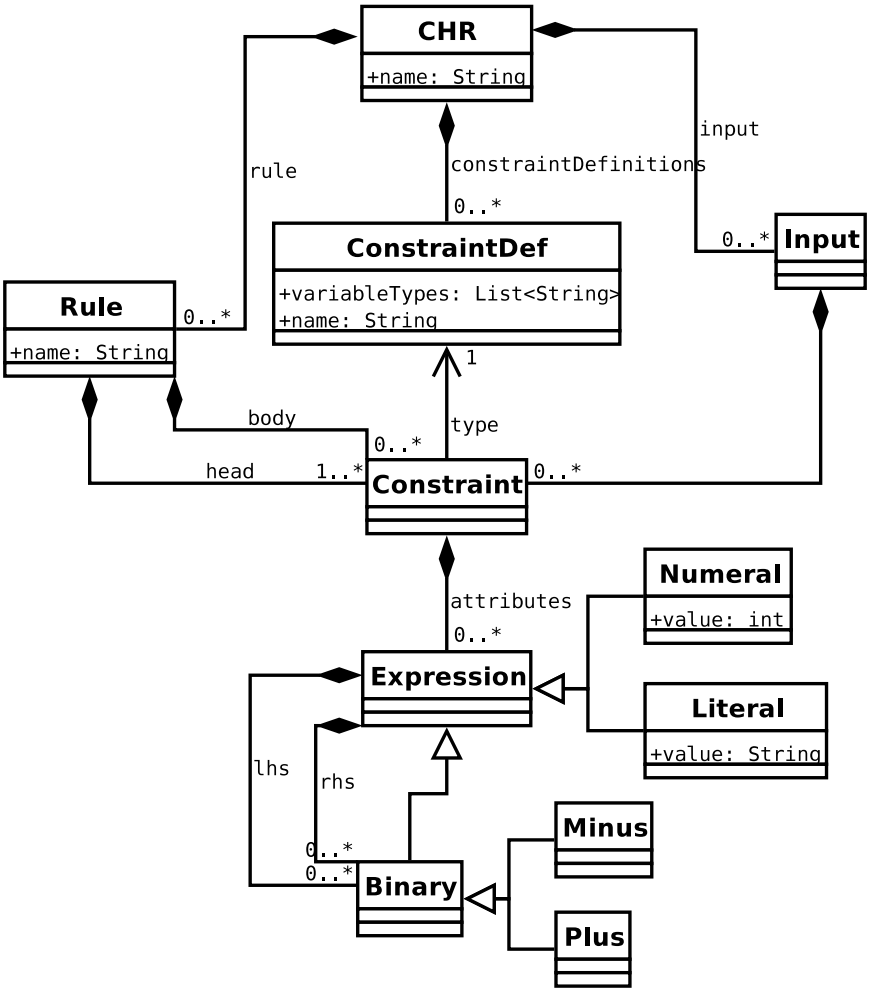


Figure 4.31: UML class diagram of the model for the CHR editor.

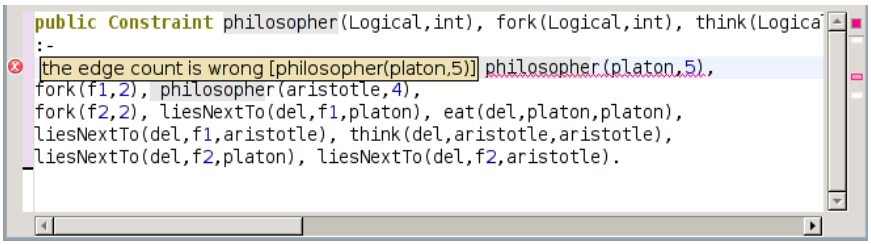


Figure 4.32: CHR editor error annotation mechanism.

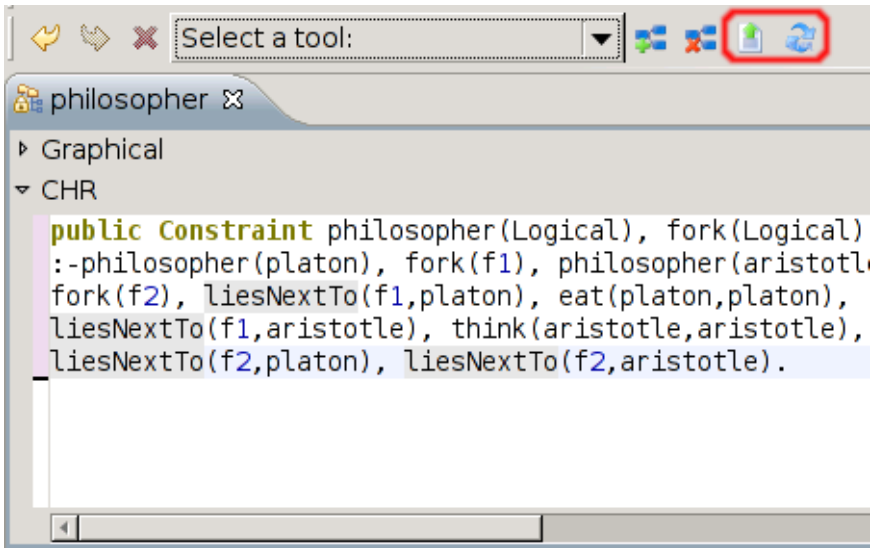


Figure 4.33: Actions for refreshing and switching between the encodings in the CHR editor.

4.4.2 Implementation

In this section, I give implementation details for the CHR model, its corresponding grammar, the algorithms for checking validity and updating the model, and the implementation of the editor itself. The editor itself can be found in the project `org.uniulm.gts.chrtexteditor` and the CHR model can be found in the project `org.uniulm.gts.chrtexteditor.model`.

4.4.2.1 Grammar and model

The CHR model is created using the EMF, because TEF supports only EMF models. The UML class diagram from figure 4.31 can be translated to an ecore model from which source code for the model is produced. See section 4.1.2 for details about EMF and model generation.

The grammar that creates the model from a text is a context free grammar annotated with information from the EMF model. Listing 4.8 shows a part of the grammar that creates a `CHR` model from the syntax described in section 4.4.1.1. The grammar describes the syntax of the constraints and constraint definitions. Each left side of a rule is a non-terminal. They can be followed by an *element* annotation, that contains a class name from the EMF ecore model. This means that this rule describes the content of an instance of the given class. For example, the rule `CHR` (line 1) is annotated with the class of the same name, therefore, this rule describes a CHR object. The right side may contain annotated non-terminals and terminals. The keywords or non-terminals can be followed by a *composite* annotation. The attribute of this annotation is a property of the class which is described by the keyword or non-terminal. The right side of the rule `CHR` contains a non-terminal for constraint definitions, one for the rule block, and an annotated non-terminal which describes an objects of the class `Input`. For each non-terminal which is annotated with a *composite*, a rule must be provided which describes the content of the annotated property. Non-terminals can also be followed by a *reference* annotation. These annotations have also a property of the described object as an argument. The rule for such a non non-terminal describes an object

```
1  CHR:element(CHR) -> (ConstraintDefs (RuleBlock)? (Input:composite(Inputs))?)?;
2  RuleBlock -> "rules{" Rule:composite(Rule) "}";
3  Rule:element(PropRule) -> (PropHead)? "<=>"
4                            PropBody ".";
5  PropHead -> Constraint:composite(head)
6        ("," Constraint:composite(head))*;
7  PropBody -> Constraint:composite(body)
8        ("," Constraint:composite(body))*;
9  PropBody -> "true";
10 ConstraintDefs -> ("public")? "Constraint"
11       ConstraintDef:composite(constraintDef)
12       ("," ConstraintDef:composite(constraintDef))* ";";
13 Input:element(Input) -> ":-" Constraints "."
14 Constraints -> Constraint:composite(constraints)
15       ("," Constraint:composite(constraints))*;
16 ConstraintDef:element(ConstraintDef) ->
17  IDENTIFIER:composite(name) ("("
18  IDENTIFIER:composite(variableTypes)
19  ("," IDENTIFIER:composite(variableTypes))*
20  ")")?;
21 Constraint:element(Constraint) ->
22  ConstraintDefRef:reference(type) ("("
23  Expression:composite(variables)
24  ("," Expression:composite(variables))*
25  ")")?;
26 ConstraintDefRef:element(ConstraintDef) ->
27  IDENTIFIER:composite(name);
```

Listing 4.8: Main part of the grammar for the CHR model and the constraints.

of the corresponding class, but only enough information has to be provided in the text to identify one of the objects that has already been created of the same type. This object is then referenced. For example, the rule `Constraint` (line 21) contains the non-terminal `ConstraintDefRef` (line 22) which has a reference annotation. The rule for this non-terminal describes an instance of the class `ConstraintDef` (compare line 16 ff.), but only the `name` property of the constraint definition is parsed. This object is then compared to the other `ConstraintDef` instances in order to find an object that contains the same `name` property. Terminals are either strings enclosed by quotes or one of several keywords written completely in capital letters (e.g. `INTEGER` or `IDENTIFIER`) which represent a string or number in the parsed string (lines 17–19). If one of the keywords is followed by a composite annotation, the described property is in general a string or an integer. This property is then set to the value found in the input string. The rule `ConstraintDef` (line 16) has on its right side the keyword `IDENTIFIER` followed by a composite annotation (line 17). For that reason, the next character sequence (consisting of alphanumeric symbols) in the input string is used as value for the property `name` (which is of the type string).

The grammar in listing 4.8 shows the structure of the encoding for type, host, and rule graphs. The type graph is always printed while the rule or the input can also be omitted (line 1). The rules for the syntax of the input and the CHR rules correspond to the notation described in section 4.4.1.1. The grammar shows the syntax of the constraints and constraint definitions, as well. Constraints can contain expressions as its terms. The grammar for terms is not further described here. The complete grammar can be found in appendix C.6. The grammar in the appendix also contains *white space tags*, that are only used for pretty printing, but they are left away here to improve the readability of the grammar.

4.4.2.2 Encoding/decoding algorithms

This section describes the implementation of the algorithms for encoding a graph of a GTS to the CHR model and for the validation of a CHR model. The implementation of the algorithm to update the graph that corresponds to a valid CHR model is described, too.

The implementation for the model transformation from the GTS to the CHR model works as follows: The algorithm takes each node and edge from the graph and creates a `Constraint` object (for typed graphs) or a `ConstraintDef` object (for the type graph) for it. The attribute lists of the `ConstraintDef` instances are filled with the according names for the types where the `name` attribute is the identifier of the node or edge. The `type` attribute of a `Constraint` instance references the according `ConstraintDef` instance. Its attribute list is filled with the values described in section 2.3. Values that represent identifiers, i.e. the first argument for nodes and the second and third argument for edges, are ground terms or variables. The second term of a node constraint is a `Numeral` instance containing a number (for host encoded graphs) or a `Binary` instance (`Minus` or `Plus`) which describes an expressions consisting of a `Literal` and a `Numeral`. These model transformation algorithms are provided by the method `getCHRModel(...)` in the class `GTSTextEditor`. This method has the corresponding graph model as attribute. It is overloaded for type, host, and rule graphs.

The implementation of the algorithms for checking the validity of CHR models and updating the corresponding graph model are tied together for technical reasons, because much of the information for the model update can be gathered during the validity check. If there are errors in the CHR encoding, the graph is not updated, but error annotations are displayed in the editor. The key part of the algorithm is to transform both, the GTS model and the CHR model, to a similar form in order to compare them. Consequently, the nodes of the graph and the node constraints of the CHR model are saved in java maps to compare them by their node identifier. For edges, a `MultiHashMap` that allows multiple keys has been created, because an edge is identified by its source node identifier, its target node identifier, and its type edge's identifier. Both, the GTS model and the generated CHR model, are saved in these maps. They are called CHR maps and GTS maps, respectively. These maps are the lists needed by the algorithms described in figures 4.29 and 4.30. Maps were used, because they allow a faster access to their elements when the identifier is known. The validity algorithm is then applied to the CHR maps. If the model is valid, then GEF commands (see table 4.2) are created to update the corresponding graph model. The commands can be created by comparing the GTS and CHR maps. In case of a rule graph, multiple maps are needed to represent the three graphs L', R', and K of a GTS rule. The algorithms to verify and update rules can be found in the method `updateRuleModel()`. The algorithm to verify and update host graphs can be found in the method `updateHostmodel()`. Both methods receive the created CHR model as input and are located in the class `GTSTextEditor`.

The simpler encoding of constraints described in section 4.4.1.1 can also be encoded and decoded with the given algorithm. The execution of the algorithms is controlled by a boolean variable which skips the parts of the algorithms that are not needed by the simpler encoding when set to true. When separating the constraints in the validation algorithm, this boolean variable is also queried, because the sorting depends on the terms of the constraints. The full source code of the method `updateRuleModel()` can be found in appendix C.5.

4.4.2.3 Editor

The implementation of the CHR editor itself is bundled in the class `GTSTextEditor` which is extended from the `TextEditor` base class provided by TEF. To synchronize the editor with the model, the editor registers itself as listener to all elements of the currently displayed graph. This is done as soon as the currently displayed graph model is set by the platform. When the `notifyChanged()` method is called by the model, the method `printNewCHRModel()` is called which transforms the currently active graph model to a CHR model (see section 4.4.2.2) and uses the pretty printing function of TEF to display the new encoding. The Editor registers itself as a listener to the TEF parsing

mechanism. When a new model is parsed from the contents of the editor, the method `errorStatusChanged()` is called. From this method, the algorithms for validating the model are called (section 4.4.2.2). These algorithms also perform the update of the currently encoded graph model. However, if the model is invalid, error annotations are displayed in the text editor. Error annotations contain a message that describes the error. These annotations are then added to the `AnnotationManager` of the text editor together with the position of the text part that is responsible for the error. For finding out the position of the faulty constraints in the text, TEF saves the parsing tree of the last model creation. This tree can be queried with model objects to receive the text parts corresponding to it. From these text parts, the position in the editor can be determined.

Another important point is, that when updating the graph model from the CHR model, notifications from the graph model must be ignored. Otherwise the text would be updated continuously while typing, resulting in a rearrangement of the constraints. Furthermore, the validation and update algorithms must be deactivated when the model has just been pretty printed. The problem that would otherwise occur is, when removing multiple elements in the graphical editor at once, the model gets pretty printed multiple times in the textual editor. After each pretty printing, the model is parsed asynchronously in the background, resulting in a new CHR model. As the parsing runs asynchronously, the generated model might encode an older version of the graph model. Therefore, the validity and update algorithms would produce further commands to change the model which is wrong in this case.

The actions are added to the action bar of the platform as described in section 4.2.2.5. The `RefreshCHRAction` just calls the method `refreshCHR()` of the editor it is associated with. This method resets the current graph model in the editor to force a refresh of the encoding. The class `SwitchCHRModeAction` is the action for switching between the encodings. It calls the method `switchCHRMode()` of the editor. This method switches the value of the boolean variable that controls the execution of the encoding and decoding algorithms described in section 4.4.2.2.

4.4.3 Sample computation

In this section, I describe how to use the textual editor to edit a rule, show how the validity check works, and how the graph model is updated. Figure 4.34 shows the rule *thinkToWait*, created in section 4.3.3, but with the green *wait* edge missing on the philosopher. The encoding is displayed in the more complex version with deletion and degree attributes. To add the wait edge, a `wait(del,P,P)` constraint must be added to the body of the displayed rule. This constraint symbolizes a loop on the *philosopher* node, identified by `P`. As the constraint is only in the body of the CHR rule, it is automatically in the graph R' of the GTS rule. Figure 4.35 shows that the encoding is invalid by highlighting the philosopher constraint in the body of the rule. This is because the validation algorithm found an error when it checked the degree attribute of the philosopher constraint. In this example, the term has the value `NC_0-2`. That means the degree is smaller by two after the rule application. However, because a new edge is added, the degree is unmodified. For that reason, the subtraction of 2 must be removed in the degree attribute of the philosopher constraint. Figure 4.36 shows the final rule with the green *wait* edge added.

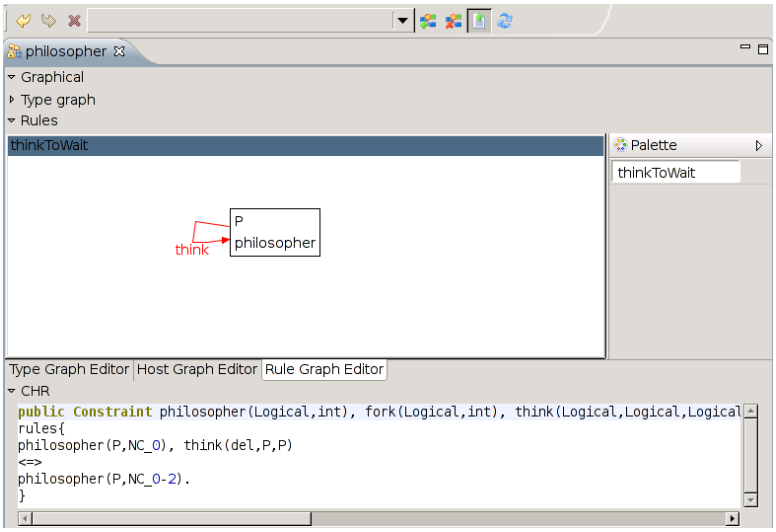


Figure 4.34: *thinkToWait* rule displayed by both editors. The green *wait* edge is still missing.

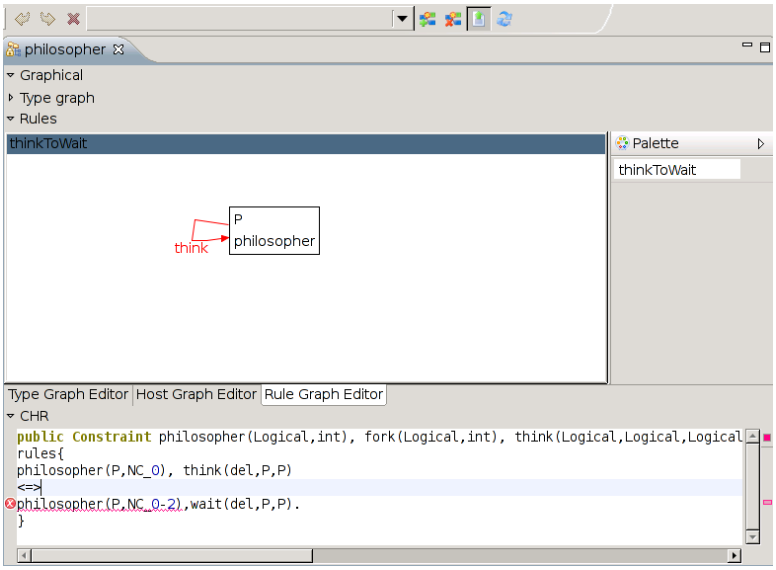


Figure 4.35: Invalid encoding, the degree attribute of the philosopher constraint must be updated, because of the added wait constraint.

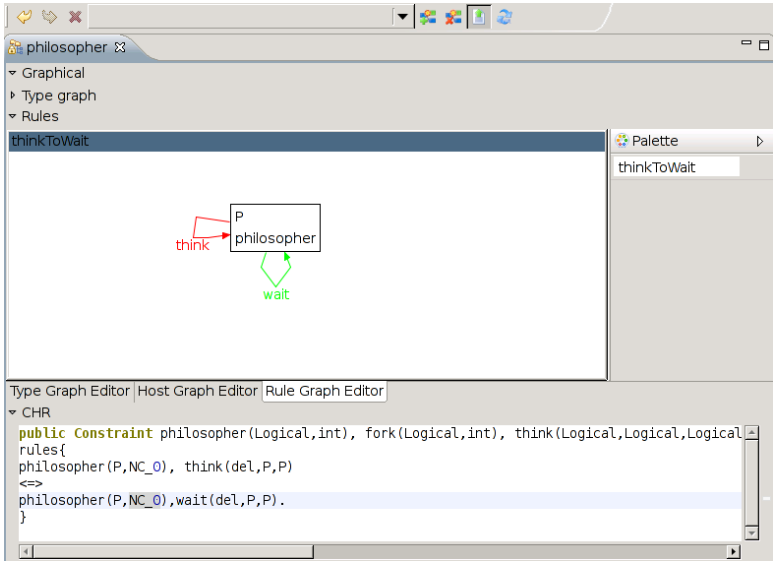


Figure 4.36: Final rule with the correct encoding.

4.5 How to create a tool: The termination analysis tool

In this section, I describe explicitly the steps that are needed to create a GTS analysis tool. The creation is shown with the help of an example tool that checks whether a GTS terminates. A GTS terminates when a graph is reached during rule applications to which no rules can be applied anymore. Of course, termination is an undecidable problem, because of that, the method can only prove termination, but not non-termination. First, I describe the theoretical background of this analysis and how the graphical user interface for this tool can be designed. Then, I describe in detail how to implement this tool.

4.5.1 Design

The analysis method used here is based on a ranking function. Section 4.5.1.1 provides an explanation of ranking functions. The GUI for entering the ranking function and displaying the rules that might not lead to a terminating GTS is described in section 4.5.1.2

4.5.1.1 Ranking functions

Termination of CHR programs can be analyzed with a ranking function ([20]). A ranking function $f_R : Constraints \mapsto \mathbf{R}^+$ maps the set of constraints to the set of positive real numbers, i.e. each constraint is assigned a value called its rank. These functions can be defined for each type of constraint, e.g. it can be a constant or it may depend on the length of a list in the constraint. The ranking function can be applied to a conjunction of constraints $f_R(\bigwedge_{c \in Constraints} c) = \sum_{c \in Constraints} f_R(c)$, then the value is the sum of the ranks of the constraints. By choosing a good ranking function and calculating the rank of the rule's head and body, one can state if a rule leads to a terminating system. If the rank of the head is higher than the rank of the body and this is valid for all rules, the CHR program terminates, because the rank of the constraint store strictly decreases and reaches

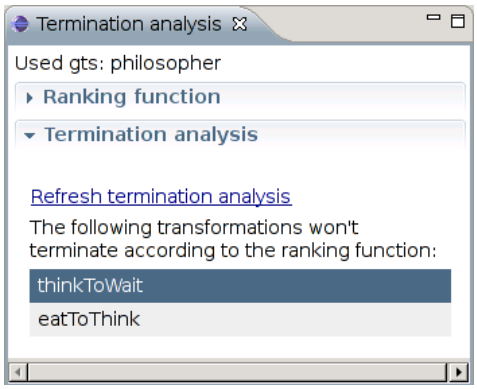


Figure 4.37: Termination analysis of the dining philosophers problem, the ranking function is collapsed.

a minimal value when no rule is applicable anymore. However, if for only one rule the rank of the head is less than or equal to the rank of the body, no statement can be raised whether the program is terminating or not.

The encoding of the examples for finding circular lists shown in section 2.3 can be analyzed with a ranking function $f_R(c) = 1$ for every constraint c. This function can be used, because edge constraints are never modified, but only removed. The same statement holds for node constraints, although they contain the number of connected edges which is modified by rules. This number depends on the number of edge constraints. Therefore, the rank of a node constraint can be set to one. The encoding for the circular list example is terminating, because each rule removes more node or edge constraints than it adds. For the dining philosophers problem's encoding, no reasonable ranking function can be found, because it is obviously not a terminating system.

This method can be adapted to rule graphs of a GTS: A ranking function is assigned to the nodes and edges. Therefore, the ranking of a graph is the sum of the ranks of its nodes and edges. The ranks of the R and L graph of a GTS rule $p = (L \xleftarrow{l} K \xrightarrow{r} R)$ are calculated. If the L graph has a higher rank than the R graph and this holds for all rules, then the GTS is terminating.

As seen above, encoded GTS rules can be ranked by a function $f_R(c) = a_c$, where a_c is a constant value for each constraint of type c. Applied to a GTS, a constant value is associated with each type node and each type edge. As a consequence, the rule nodes and rule edges are ranked by their type.

4.5.1.2 Graphical user interface

The GUI must show a table to enter the values a_c for each type node and edge. Besides that, a list must be displayed that shows the rules that might lead to a non-terminating system. Figure 4.37 shows the analysis of the dining philosophers GTS. In this example, the ranks for each type node and edge is set to one (the table to enter these values is collapsed in the figure). The entered values can be arbitrary in this case, because there exists no ranking function that can prove termination of this system, because it is non-terminating. The example shows that the rules *thinkToWait* and *eatToWait* might lead to a non-terminating system. When rules are selected from the list of the non-terminating rules, they are displayed in the editors. Figure 4.38 shows the termination analysis of the circular list example. This sys-

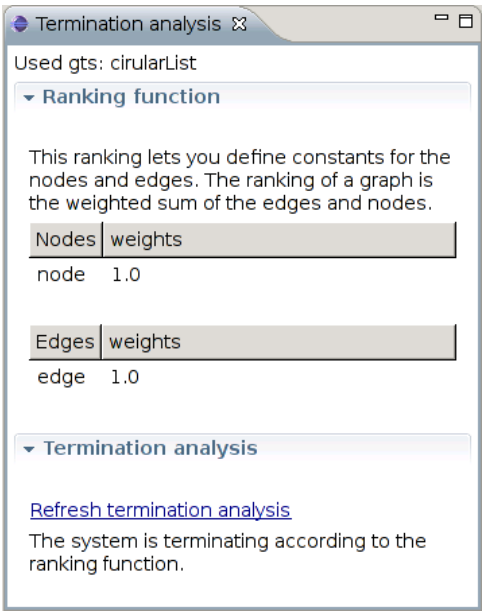


Figure 4.38: Termination analysis of the circular list detection GTS.

```
<plugin>
 <extension   ...>
  <gtsanalysistool
    class="org.uniulm.gts.analysistools.termination.TerminationAnalysisTool"
    name="Termination analysis">
  </gtsanalysistool>
 </extension>
  ...
</plugin>
```

Listing 4.9: snippet of the `plugin.xml` file for the termination analysis tool.

tem terminates according to the ranking function. As a result, no rules are displayed in the list of rules, but a message is shown that all rules have strictly decreasing rank. When selecting the node or edge identifiers in the table, the type graph must be displayed in the editor with the according node or edge highlighted.

4.5.2 Implementation

In this section, I demonstrate in detail how an analysis tool is created in general by using the termination analysis tool as an example. Creating a new analysis tool is usually started by creating a new eclipse plug-in project. The name of this project is `org.uniulm.gts.analysistools.termination`. The first thing is to add an extension declaration to the `plugin.xml` file in the project. Listing 4.9 shows the content of this file for this tool. An analysis tool extension is defined by a `gtsanalysistool` tag that contains the attributes `name`, which is the human-readable name of the tool, and `class`, which gives the location of the class that implements the interface `IGTSAnalysisTool`. This class is loaded when the tool is started. Now, the class `TerminationAnalysisTool` has to be created. First, I describe the methods that must be implemented for the `IGTSAnalysisTool` interface. The methods for adding and removing listeners are implemented in a standard way by using a list that saves the listener objects. Furthermore, the methods shown in listing 4.10 are implemented to simplify

```
1  public void selectedTransformation(String graph) {
2   for (IGraphTransformation gt : this.m.getTransformations())
3    if (gt.getId().equals(graph)) {
4     notifyAll(gt);
5     break;
6    }
7  }
8  public void selectedEdge(String edge) {
9   GTSElement el=new GTSElement(GTSElement.EDGE);
10  for(IAbstractEdge e:m.getTypeGraph().getEdges()){
11   if(((ITypeEdge)e).getID().equals(edge)){
12    el.setEdge(e);
13    notifyAll(el);
14    break;
15   }
16  }
17 }
18 public void selectedNode(String node) {
19  GTSElement el=new GTSElement(GTSElement.NODE);
20  for(IAbstractNode e:m.getTypeGraph().getNodes()){
21   if(((ITypeNode)e).getID().equals(node)){
22    el.setNode(e);
23    notifyAll(el);
24    break;
25   }
26  }
27 }
```

Listing 4.10: Helper methods for notifying listeners.

```
1  public void runAnalysis(){
2  IViewPart viewer = getPage()
3    .getActivePage()
4    .showView("org.uniulm.gts.analysistools.termination.TerminationView");
5   if (viewer instanceof TerminationView) {
6    content = ((TerminationView) viewer).getContent();
7    content.setTool(this);
8   }
9  }
```

Listing 4.11: Implementation of the `runAnalysis()` method.

the communication with the listeners. These methods receive the identifier of the node, edge, or rule graph that should be displayed by the editors, search the according object in the model, and call a `notifyAll()` method. The `notifyAll()` method iterates over the listeners and notifies them. The methods for setting and getting the currently displayed graph, nodes, or edges are not implemented, because this tool has no use for these methods. The methods for setting the file and the command-stack also have no implementation, because they are not needed. The method `setModel()` saves the given model to a variable and registers itself to it as a listener (the EMF `Adapter` interface has to be implemented for this). If a model has been set before, the tool removes itself as listener from the old model. The method `runAnalysis()` is implemented as shown in listing 4.11. The method opens a view, receives its content, and adds itself to the content by using the `setTool()` method. The content is an instance of `TerminationViewContent`, which is a class that contains all the GUI elements and provides methods to set ranking function entries and the list of rules that might lead to a non terminating system. This class is not described in detail here. The important point is that this GUI class calls the methods of the tool, shown in listing 4.10, when new elements in its lists are selected. Because of that, the type edge, type node, or rule graph is displayed in the editor. The method `updatedWeights()` is called by the content object when the user edits the table of the ranking function values. When the `updatedWeights()` method is called by the content object or the `notifyChanged()` method is called upon a model change, the calculations according to section 4.5.1.1 are computed and the content object is updated accordingly. Additionally, when `notifyChanged()` is called, the entries of the ranking function ta-

ble are updated, because a type node or type edge could have been added, which changes the ranking function.

To conclude, the main points in creating an analysis tool are:

- Create a new eclipse plug-in project
- Implement the listener methods of the `IGTSAnalysis` analysis tool in order to display nodes, edges, and graphs in the editors.
- Add the tool itself as a listener to the parts of the model that are analyzed to dynamically update the analysis.
- The `setModel()`, `setFile()`, and `setCommandStack()` methods should only save values, and not start the analysis itself. This should be done when the `runAnalysis()` method is called.
- When a GUI is involved, start it in the `runAnalysis()` method.

4.6 The CHR based analysis tool

One of the main goals (section 3.2) is to develop a platform for the CHR-based analysis of GTS. Therefore, a tool has to be created that allows the simple embedding of analysis methods for CHR. This tool is an intermediate tool which itself defines an extension point that offers the possibility to integrate available CHR analysis methods. This tool is called *CHR-based analysis tool* or *CHR-based tool* in the remainder. The tools that are used by the CHR-based tool are called *CHR analysis tools* or *CHR tools* if the context is clear. One such CHR tool is given in the form of a compiler for the CHR source in order to create an executable environment. The description of this tool is a tutorial for embedding other CHR analysis methods and can be found in section 4.6.2.

No tools for analyzing JCHR programs have been available at the time of implementation. For that reason, no further CHR tools are provided. Implementing analysis methods for JCHR would have exceed the scope of this thesis.

4.6.1 Design

This analysis tool basically gives the possibility to launch tools for analyzing CHR programs. Therefore, a simple GUI must be created, as well as another extension point specifically designed for CHR analysis tools. To create CHR source code files from the GTS model, a method for code generation is required.

4.6.1.1 CHR analysis tools extension point

The extension point and its interface is designed similarly to the extension point for GTS analysis tool defined in section 4.2.1.2. The `plugin.xml` contains the same entries: one to define a class that should be loaded, one for setting the name of the tool and the identifier. The interface is also designed similar. However, for this interface, no command stack is needed. Furthermore, there is no model of the GTS but a file containing the corresponding CHR program. JCHR is used as CHR implementation. That is why analysis tools must also be based on JCHR. JCHR uses `.jchr` source files for its programs. For more information on JCHR, see [43]. One problem with the JCHR source code files is, that goals are not included in them. Therefore additional methods have to be provided by the interface in

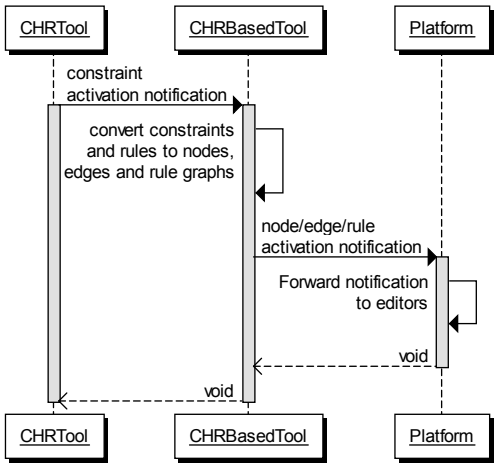


Figure 4.39: UML sequence diagram of the communication of CHR tools with the platform.

order to add goals that should be analyzed. Just like the interface for the GTS analysis tools, this interface has a method for starting the tool itself after setting the file and adding the goals.

The interface must also contain methods for the listener pattern, so that a bidirectional communication is possible between the CHR tools and the CHR-based tool. The bidirectional communication informs the CHR-based tool which rules, goals, or constraints should be displayed. Consequently, the CHR-based tool must transform these notifications for CHR rules, goals, or constraints to notifications for rule, host, or type graphs and their nodes and edges, respectively. These new notifications are then sent to the platform. Figure 4.39 shows a sequence diagram describing the flow of information between CHR tools and the platform.

4.6.1.2 Graphical user interface

The GUI must display a list to choose which rules should be analyzed and to set their order. There has to be a list to choose the CHR goals, i.e. the encoded host graphs that should be analyzed. When selecting the elements of these lists, the according graphs are displayed in the editor. Finally, a drop down menu has to be provided to select the CHR-based analysis tool that should be started. The screen-shot in figure 4.40 shows how the GUI should look like. This screen-shot shows the dining philosophers GTS. On the right, the CHR based analysis tool is opened and the host graph with two philosophers is currently selected. Because of that, it is displayed in the editor on the left. Also notice the two text fields on the top of the tool in which the name of the generated handler and the package used for it can be entered. The contents of these text fields are needed for the code generation as described in section 4.6.1.3. Below the combo box for the selection of the CHR tool is a hyperlink that must be clicked to start the tool that is currently shown in the combo box.

4.6.1.3 Code generation

CHR analysis tools need syntactically correct CHR source code for the analysis, therefore, code generation mechanisms are needed. This generation is done by JET, the java emit-

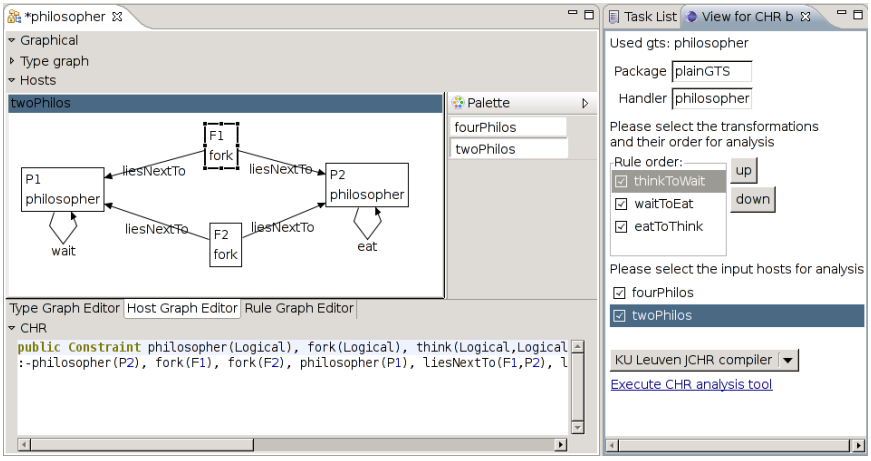


Figure 4.40: The GUI of the CHR based analysis tool together with the editor.

```
1  package examples.fib;
2  import java.math.BigInteger;
3  import runtime.Logical;
4  import util.arithmetics.primitives.intUtil;
5  public handler fib {
6      solver runtime.EqualitySolver<BigInteger>;
7      public constraint fib(int N, Logical<BigInteger> M);
8      rules {
9          local Logical<BigInteger> M1, M2;
10         fib(0,M) <=> M = 1;
11         fib(1,M) <=> M = 1;
12         fib(N,M) <=>
13             fib(intUtil.dec(N), M1),
14             fib(intUtil.sub(N, 2), M2),
15             M = M1.add(M2);
16     }
17 }
18 }
```

Listing 4.12: JCHR program for calculating the Fibonacci numbers.

ter templates. Details about the code generation are given in the implementation section. The generated code uses the syntax of JCHR. The syntax has some similarities to the java syntax. Listing 4.12 shows a JCHR program for the calculation of the Fibonacci numbers. At the beginning of a JCHR file, the package name, as well as imports are listed, which describe the packages that are needed. JCHR needs special methods for doing basic numerical operations. Therefore, the import of the `intUtil` class is listed in the example above. The main part of the program is enclosed by a `handler` statement (lines 5 ff.) that is similar to a `class` statement in java. The java class that is generated from this example will have the name `FibHandler`. In the handler, the constraints are defined by giving the keywords(s) (`public`/`private`) `constraint`, followed by the names of the constraints together with the types of their attributes. Furthermore, a `solver` is defined (line 6). Solvers are used to solve built-in constraints, i.e. they provide the functionality of a *Constraint Theory* (section 2.1). In this example, a solver for the `BigInteger` class is created. The next part is enclosed by a `rules{...}` block (lines 8–16) in which all rules are listed. The rules have a syntax similar to that described in section 2.1. However, variables that are only in the body of a rule but not in the head must be declared first by giving the `local` keyword, followed by the type and the name of the variable. In this example, two variables of the type `Logical<BigInteger>` are defined (line 9). `Logical` is a

special JCHR type that defines an arbitrary term of the given type (`BigInteger` in this case) which can be compared and set equal to other terms without having a value.

As a result, the code generator must query the rules of the GTS and encode their nodes an edges as constraints of a rule, which is described in section 2.3. In addition to that, all variables that are in the body, but not in the head of a rule, must be declared at the beginning by using a `local` statement. Variables are introduced when new nodes are added in a GTS rule, because each added node contains a new unique identifier. The names for the package and the handler must be provided for code generation. These values are queried in the GUI (see figure 4.40).

4.6.1.4 Compiler tool

One vital part of several analysis methods for CHR is that an executable environment must be present to test generated inputs (e.g. confluence analysis ([20])). Therefore, I provide a CHR tool that calls the K.U. Leuven JCHR compiler that compiles the JCHR source to java source which is then compiled to java byte-code by the eclipse platform. This CHR analysis tool has no graphical user interface. Only when an error occurs, an error dialog is shown that informs the user about the error.

4.6.2 Implementation

First, I describe the implementation of the extension point for CHR analysis tools, then the intermediate analysis tool and the code generation using JET. At the end, I describe the implementation of a CHR analysis tool in detail. I use the example of embedding the K.U. Leuven compiler for this purpose. The extension point definition can be found in the project `org.uniulm.gts.chranalysistool`, the CHR based tool in `org.uniulm.gts.analysistools.chrbasedanalysis`, the compiler CHR tool in `org.uniulm.gts.chranalysistools.chrcompiler`, and the JET template for generating the source code is located in the project `org.uniulm.gts.chrexporter.plainchrexporter`.

4.6.2.1 Extension point

The extension point definition is similar to the one from the GTS analysis tool. It basically consists of two interfaces, `ICHRAnalyzer` and `ICHRListener`, as well as a class `ConstraintNotification`. The interface `ICHRAnalyzer` contains methods for setting the file with the JCHR source (`setFile(IFile)`) and adding goals to the CHR analysis tool (`addInput(List<ConstraintNotification>)`). A goal is represented as a list of `ConstraintNotification` instances. This class consists of the constraint's name and a list of strings representing the attributes of the constraint. Furthermore, methods for adding and removing listeners are provided. The method `runChrAnalysis()` starts the according tool. The `ICHRListener` interface contains methods to notify the listener about activated rules, constraints and constraint definitions. The methods contain either the rule name as string or a list of instances of the `ConstraintNotification` class as its attribute.

4.6.2.2 Tool

The tool itself is realized by two classes, one providing the graphical front-end with methods to set and receive information and the other one providing the functionality of the tool.

```
1 public constraint
2 <c:iterate select="$gts/typeGraph/nodes" var="typenode" delimiter=", ">
3  n_<c:get select="translate($typenode/@iD,' ','_')"/>(LogicalInt,int)
4 </c:iterate>
5 <c:if test="count($gts/typeGraph/nodes)>0">,</c:if>
6 <c:iterate select="$gts/typeGraph/edges" var="typeedge" delimiter=", ">
7  e_<c:get select="translate($typeedge/@iD,' ','_')"/>(
8   int,LogicalInt,LogicalInt)
9 </c:iterate>;
```

Listing 4.13: Snippet of a template for creating the constraint definitions from a type graph model.

The tool implements the interface for GTS analysis tools and the listener interface for CHR analysis tools. First, I describe the interface for the GTS analysis tool. The `setModel()` method saves the given model as its current model and adds itself as listener to it, so that the GUI can be updated when graphs are removed or added. The `setFile()` method is used to set the name of the JCHR handler in the GUI (because of that, the default name for the generated file is identical to the file name from which the model was loaded). When `runAnalysis()` is called, a view is opened. The contents of this view are initialized by filling the lists of rule and host graphs with the corresponding identifiers and registering the tool itself as selection listeners to these lists. Furthermore, all CHR analysis tools extensions are searched, their names are saved in a combo box in the GUI, and a selection listener is added to a hyperlink that is used to start the currently selected tool. The selection listeners for the lists check the currently selected element of the according list, search the graph with the same identifier, and notify the listeners of this tool (i.e. the platform) that this graph is currently selected. The listener for the hyperlink checks which tool is selected, generates the source (which is described later), and calls the `setFile()` method of the according tool followed by several calls of the `addGoal()` method for every activated host graph in the list of host graphs. When all goals are added, `runChrAnalysis()` is called which starts the functionality of the CHR analysis tool.

4.6.2.3 Code generation

The creation of the JCHR source is done with JET. A JET code generator is described by a template file that contains the static parts of the target document together with XML tags that describe the dynamic content. Listing 4.13 shows a snippet from the template to create CHR source from a GTS model. The snippet shows how to generate the list of constraint definitions at the beginning of the handler. It contains the `<c:iterate ...>` tag (lines 2–4) which iterates over a number of objects defined by its `select` attribute. The objects are selected by using *XPath*. XPath is a query language designed to select parts of XML documents, but can also be used to query EMF models. For more information about XPath, see [4]. Basically, the XPath expression provides a path to reach the desired objects in the model. In the first iterate tag stated above, all nodes in the type graph model are selected by giving the path `$gts/typeGraph/nodes`, which means that `gts` is the root of the model from where the `typeGraph` attribute is selected. In there, all the `node` attributes are chosen (also compare to the definition of the GTS model described in section 4.2). The iterate tag iterates over all nodes. These nodes are represented by the variable `$typenode`. The `delimiter` attribute of the tag defines the string that separates the elements of the iteration from each other. Values can be printed from the model with the `<c:get select=.../>` tag. This tag also uses XPath to select a value from the model. `$typenode/@iD`, for example, selects the `iD` attribute of the object `typenode`. Objects in the model that represent final values, i.e. strings, numbers, and boolean values, are addressed in XPath with an `@` sign. The function `translate` is an XPath function

```
1  props=new HashMap<String,Object>();
2  props.put("org.eclipse.jet.resource.project.name", file.getProject().getFullPath().
       toOSString());
3  props.put("jchrpackage",pkg);
4  props.put("jchrhandler",hndl);
5  props.put("order",content.getOrder());
6  IStatus res = JET2Platform.runTransformOnObject(
7   "org.uniulm.gts.chrexporter.plainchrexporter",
8   model,
9   props,
10  new NullProgressMonitor());
```

Listing 4.14: Code snippet for starting a JET transformation with additional values from a java program.

and replaces blanks by underscores in the first attribute. Another tag that is used here, is the `<c:if test=...>` tag (line 5) which can be used to conditionally print values to the target document. The `test` attribute contains a conditional in which comparison operators (`<, >, =, <=, >=`) can be used together with XPath expressions. The conditional `count($gts/typeGraph/nodes)>0` uses the XPath function `count` that returns the number of elements in the given XPath expression. There are several XPath expressions already available and JET provides the possibility to define new functions by creating an extension to JET. These functions are implemented as java classes. The example listing first prints the node and then the edge constraints, followed by a semi-colon.

The rules are printed by using an iterate tag. However, the iterate tag prints the rules in the order they appear in the model. Therefore, an XPath function has been designed as a JET extension that receives a list of objects and a list of integers. The list represents a permutation of the elements in the list of objects. This function reorders the list of objects and returns the new list. The function is called `reorder`. Certain elements can also be left away by removing the according number from the list of integers. At the beginning of the rules block in a JCHR file, all variables must be listed that appear in the body but not in the head of the rules. To realize this, a second XPath function has been created that receives a list of objects and returns a list of its distinct values, i.e. it removes duplicate entries. This function is called `distinct-values`. To print all variable names contained in the bodies of rules, all these variables are selected with an XPath expression and handed to the new defined function. Generating the rules requires to introduce new variable names for the deletion and degree attributes of the edge and node constraints. These variables are named `DEL_` or `NC_`, followed by an increasing number to uniquely identify them within a rule. To avoid name clashes between the generated variable names and the variable names introduced by the model, the names of the identifier variable in node constraints are preceded by `NN_` for variables introduced in the body of a rules or `ON_` for variables that appear in the head of a rule, too.

Furthermore, the template needs the information about the handler and the package name. These values can be given to JET in form of a java map as shown in the code snippet in listing 4.14. The snippet shows how a map is filled with values. The variables `pkg` and `hndl` contain the name of the package and the handler. The property `order` that is put into the map, is the list with integers that represents the permutation of the rules. The GTS model itself is saved in the variable `model` and is handed to the `runTransformationOnObject()` method of the JET platform. This method receives the identifier of the transformation, the EMF model object, the properties, and an instance of a progress monitor. Each JET transformation is an extension to JET which contains the location of the template file. JET also provides tags to create files, folders, and projects in the eclipse workspace. The complete JET template file can be found in appendix C.4. The source in the appendix is trimmed for readability. That is why the generated code is not formatted very nice in this form.

```
1 <plugin>
2  <extension
3    point="org.uniulm.gts.chranalysistool">
4    <chranalysistool
5      class="org.uniulm.gts.chranalysistools.chrcompiler.ChrCompiler"
6      name="KU Leuven JCHR compiler">
7    </chranalysistool>
8  </extension>
9 </plugin>
```

Listing 4.15: `plugin.xml` file for the CHR compiler tool.

4.6.2.4 How to create a CHR tool: Compiler tool

In this section, I show a guide for the creation of a CHR analysis tool by using the example of integrating the K.U. Leuven JCHR compiler as a CHR tool. This tool calls the K.U. Leuven JCHR compiler with the generated JCHR file as input and raises an error message when errors occur during execution.

At first, the project `org.uniulm.gts.chranalysistools.chrcompiler` must be created as a plug-in project. In this project, an extension to the CHR analysis tool extension point must be created in the `plugin.xml` file of the project. Listing 4.15 shows the file. The `chranalysistool` tag defines the name of the tool and the class where it is implemented. In a next step, the java class referenced in the extension must be created. This class must implement the `IChrAnalysisTool` interface described in section 4.6.2.1. The interface is not directly implemented by the `CHRCompiler` class, but it extends the class `AbstractCHRAnalysisTool`. This class basically implements the methods for the listener pattern and provides two methods, `notifyRule(...)` and `notifyConstraintDef(...)`, in order to notify the listeners about a rule or a constraint that should be displayed. The `ChrCompiler` class must still implement the remaining methods `setFile(...)` (file is copied to a variable `file`), `addGoal(...)`, and `startAnalysis()`. The method for adding goals can be left unimplemented, because the goals are not needed by the compiler. `runChrAnalysis()` is the interesting method. CHR analysis tools in general start their calculations or display further dialogs, like views or wizards, in this method. In this example, the K.U. Leuven JCHR compiler is started. Listing 4.16 shows the source code of the method. This method checks whether the source file exists and creates a new `Job` object for compiling the source code (lines 2–5). The tool adds itself as listener to the job and schedules it for execution (lines 17 and 18). If the source file does not exist, an error message is displayed. When the job is finished, it returns a status code to the tool by calling one of the methods from the `IJobChangeListener` interface (this is not shown in the listing). If an error is returned, an error dialog is displayed. For the Job itself, the `run()` method must be implemented. In this method, a `Compiler` instance is created (line 7). This class also provides a method `run()` that starts the K.U. Leuven JCHR compiler. The K.U. Leuven JCHR compiler does not provide a documented method to call it from other java classes. Therefore, the `Compiler` class creates a new process for launching the K.U. Leuven compiler. The compiler can only be started when the binaries of the java run-time environment are in a folder that is searched by the system. The jar file of the K.U. Leuven JCHR compiler must be in the java class-path, so that it can be found by the run-time environment. The standard and error output is returned as a string. These functions are not described any further, because they are only a work-around for a lack of functionality of the JCHR compiler. The `Compiler` class returns a status code that signals whether the execution was successful. Additionally, the error output is checked for errors of the JCHR compiler. Then a corresponding status code is returned.

Now the new CHR tool is listed in the combo box of the CHR based analysis tool with the name specified in the `plugin.xml` (see figure 4.41). When the link under it is clicked,

```
1  public void runChrAnalysis() {
2   boolean success=true;
3   status=Status.OK_STATUS;
4   if(file!=null&& file.exists()){
5    Job generateCode=new Job("Generate Code"){
6     protected IStatus run(IProgressMonitor monitor) {
7      cmp=new Compile(file);
8      cmp.run(null);
9           if(cmp.getStatus()!=Status.OK_STATUS)
10           return cmp.getStatus();
11          if(!cmp.getErrorOutput().equals(""))
12           return new Status(Status.ERROR,
13              "org.uniulm.gts.chranalysistools",
14              "KULeuven JCHR Compiler produced errors");
15          return Status.OK_STATUS;
16    }};
17   generateCode.addJobChangeListener(this);
18   generateCode.schedule();
19  }else{
20   success=false;
21   status = new Status(Status.ERROR,
22      "org.uniulm.gts.analysistools.chranalyzer",
23      "Source file does not exist");
24  }
25  if(!success){
26   ErrorDialog err=new ErrorDialog(null,
27      "Error compiling file",
28      "The jchr file could not be opened",status,0);
29   err.open();
30  }
31 }
```

Listing 4.16: `startAnalysis()` method of the `ChrCompiler` class.

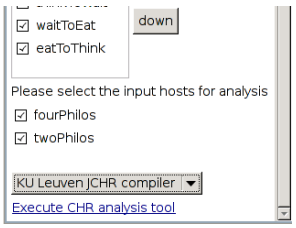


Figure 4.41: CHR compile tool, listed in the combo box of the CHR based tool.

the generated code will be compiled to a java source file that can be embedded into other projects or used by further analysis tools.

4.7 The graphical analysis tool

When analyzing a GTS, an important requirement is to have the possibility to apply the rules to a host graph. The computation of the transformations can be done by CHR (section 2.3). However, methods must be developed to apply rules step by step and to manually define to which nodes and edges the rule should be applied.

4.7.1 Design

In this section, I develop a method that realizes a step-by-step-application of CHR rules and describe how to define the next match of a CHR rule manually. Furthermore, I design the graphical user interface of the tool. To create an executable environment for a GTS, JCHR code has to be generated similar to the method described in the section 4.6.1.3. Additionally, a possibility is required to load and access the generated JCHR handler.

4.7.1.1 Interactive CHR environment

Usually, CHR does not provide the possibility to activate and deactivate rules, apply them step by step, or manually select to which constraints a rule is applied. In the following, I describe approaches to solve each of the three problems.

I found two approaches for the step-by-step-application of rules. The first approach is very specific to JCHR handlers: They offer the possibility to register a listener to them, so that the handler notifies its listeners before a rule application. To use this feature, the debug mode has to be activated in the compiler. However, the listener only gets notified that a rule is going to be applied, but this cannot be canceled. The other approach is to modify the rules of the original CHR program by adding a new constraint to each rule. Therefore, the rule

$$r@H_{kep} \backslash H_{rem} \Leftrightarrow G|B.$$

is transformed to the rule

$$r^1@H_{kep} \backslash start, H_{rem} \Leftrightarrow G|B.$$

The $start/0$ constraint is a special constraint that is added to the constraint definitions of the program. This constraint must be added to the CHR goal store if a rule should be applied. When the $start$ constraint is added, the goal stack of the current state is empty, because all its constraints are already in the CHR store. When the new constraint is added to the goal stack, it becomes an active constraint and is associated with its occurrences. When an occurrence is found, all numbered constraints are searched to find a set of constraints that match to the head of this rule (see section 2.1 for a description of the operational semantics of CHR). Take, for example, the greatest common divisor program from section 2.1 with the modified rules

$$gcd1^1 \ @ \ start, gcd(0) \Leftrightarrow true.$$

$$gcd2^1 \ @ \ gcd(I) \backslash start, gcd(J) \Leftrightarrow J \geq I, I > 0 | K \text{ is } J - I, gcd(K).$$

When adding the constraints $gcd(6)$ and $gcd(9)$, no rules can be applied and the constraints are moved to the CHR store. The according state is:

$$\langle [], \{gcd(6)\#1, gcd(9)\#2\} \rangle_3$$

When a $start$ constraint is added

$$\langle [start], \{gcd(6)\#1, gcd(9)\#2\}\rangle_3$$

it becomes the active constraint resulting in the state:

$$\langle [start\#3:1], start\#3, gcd(6)\#1, gcd(9)\#2\}\rangle_4$$

When the occurrence matches the $start$ constraint of the second rule

$$\langle [start\#3:2], \{start\#3, gcd(6)\#1, gcd(9)\#2\}\rangle_4$$

a match is found and the rule is applied resulting in the state

$$\langle [gcd(3)], \{gcd(6)\#1\}\rangle_4$$

No rule of the CHR program is applied and the $gcd(3)$ constraint is moved into the CHR store, therefore, the final state is

$$\langle [], \{gcd(6)\#1, gcd(3)\#4\}\rangle_5$$

where no transition rule can be applied anymore. Furthermore, a $noprocess$ constraint is added to the program together with the rule

$$start \Leftrightarrow noprocess.$$

at the very end of the CHR program, so that $start$ constraints are replaced by $noprocess$ constraints when no rule has been applied. This constraint can be looked up by the user to check whether a rule could be applied or not. However, these constraints have to be removed from the CHR store when a new $start$ constraint is added, because otherwise, it cannot be determined whether a rule has been applied or not. For this reason, the rule

$$start \backslash noprocess \Leftrightarrow true.$$

is added at the beginning of the program. This rule removes all $noprocess$ constraints that are currently in the store before attempting to apply any of the other rules when a new $start$ constraint is added.

The second approach is used, because it is portable to other CHR implementations and the listener feature of JCHR is still in experimental state and might be removed in further versions ([43]).

Activating and deactivating rules is also done by modifying a given CHR program. Each rule

$$r_i \ @ \ H_{kep} \backslash H_{rem} \Leftrightarrow G|B.$$

has to be modified in the following form:

$$r_i^2 \ @ \ rule\_activate(i), H_{kep} \backslash H_{rem} \Leftrightarrow G|B.$$

$rule\_activate/1$ is here a new constraint and i is a unique number which is assigned to each rule. As a consequence, only when a $rule\_activate(i)$ constraint is in the CHR store, the rule can be applied. The greatest common divisor program is used as an example. The rules must be modified in the following form:

$$gcd1^2 \ @ \ rule\_activate(1) \backslash gcd(0) \Leftrightarrow true.$$

$$gcd2^2 \ @ \ rule\_activate(2), gcd(I) \backslash gcd(J) \Leftrightarrow J \geq I, I > 0 | K \text{ is } J - I, gcd(K).$$

When adding the constraint $gcd(9)$ and $gcd(6)$ no rule of the CHR program can be applied resulting in the state

$$\langle [], \{gcd(6)\#1, gcd(9)\#2\}\rangle_3$$

When a $rule\_activate(2)$ constraint is added

$$\langle [rule\_activate(2)], \{gcd(6)\#1, gcd(9)\#2\}\rangle_3$$

it becomes the active constraint, resulting in the state:

$$\langle [rule\_activate(2)\#3 : 1], \{rule\_activate(2)\#3, gcd(6)\#1, gcd(9)\#2\}\rangle_4$$

When the occurrence matches the $rule\_activate(2)$ constraint of the second rule

$$\langle [rule\_activate(2)\#3 : 2], \{rule\_activate(2)\#3, gcd(6)\#1, gcd(9)\#2\}\rangle_3$$

a match can be found and the second rule is applied, resulting in the state

$$\langle [gcd(3)], \{rule\_activate(2)\#3, gcd(6)\#1\}\rangle_3$$

The rule can be applied two further times, to reach the state

$$\langle [], \{rule\_activate(2)\#3, gcd(3)\#5, gcd(0)\#6\}\rangle_7$$

in which no transition rule can be applied anymore. In order to apply the first CHR rule to remove the $gcd(0)$ constraint, a $rule\_activate(1)$ constraint must be added. To deactivate an activated rule, another rule is added:

$$rule\_activate(N), rule\_deactivate(N) \Leftrightarrow true.$$

This rule removes pairs of $rule\_activate(N)$ and $rule\_deactivate(N)$ constraints. In order to deactivate a rule, the user has to add a $rule\_deactivate$ constraint with the according rule number to the store. In order to activate a rule a $rule\_activate$ constraint with the according rule number must be added. This is especially useful if the user wants to see the intermediate state that is reached when only a set of rules is applied. If a final state is reached the rules can be deactivated and other rules can be activated to reach another intermediate state.

There are also two approaches for the manual selection of CHR matches. The first approach is specific to the JCHR K.U. Leuven compiler: Each constraint's arity is increased by one to hold a boolean value which is true when a constraint can be used for a rule application, and false otherwise. The user has to manage these variables and set them true or false, depending on whether the constraint should be used in a match, or not. The rules must be changed accordingly, so that only *selected* constraints are used. The problem arising from this is the *modified problem* ([43]), i.e. how does the JCHR handler know when a constraint's term is changed and must be revised for rule application. The K.U. Leuven compiler provides a solution in the form of observable variables which notify the JCHR handler when their value changes. Then JCHR reconsiders the constraints of these variables for rule application. However, this feature is work in progress, has changed in the newer versions of JCHR, and is not documented very well in the current version. The second approach is to modify the CHR program itself: different CHR constraints are used to represent *selected* and *deselected* states of constraints. For every constraint type c, two other constraint types are defined: c_sel and c_desel with the same signature as c. For each constraint c, two new rules of the form

$$c_{sel} \ @ \ c\_sel(N) \backslash c(N) \Leftrightarrow true.$$

and

$$c_{desel} \ @ \ c\_sel(N), c\_desel(N) \Leftrightarrow c(N).$$

are introduced where the N symbolizes that all terms of the constraints must be identical. These rules are added at the top of the CHR program. Now, the user can add c_sel constraints with the attributes of the according c constraint to select a certain constraint. A c_desel constraint is added to the goal stack in order to revert the constraint to the normal, deselected status. The $gcd/1$ constraint of the gcd example, therefore, results in the constraint definitions

$$gcd/1, gcd\_sel/1, gcd\_desel/1$$

the rules

$$gcd_{sel} @ gcd\_sel(N) \backslash gcd(N) \Leftrightarrow true.$$

and

$$gcd_{desel} @ gcd\_sel(N), gcd\_desel(N) \Leftrightarrow gcd(N).$$

To make sure that the rules only work on selected constraints, all constraints in a rule's head have to be replaced by the corresponding sel version of the constraint. So, every rule

$$r \ @ \ H_{kep} \backslash H_{rem} \Leftrightarrow G|B$$

is replaced by a rule

$$r^3 \ @ \ H' \Leftrightarrow G|B, H_{kep}$$

where for all constraints $c \in H_{kep} \cup H_{rem}$ a c_sel constraint in H' is added. As you can see, simpagation rules are replaced by simplification rules, because the constraints are deselected after rule application. The main problem with this approach is that CHR cannot find matches on its own anymore. Because of that, another modification to the CHR program is added. For each rule

$$r \ @ \ H_{kep} \backslash H_{rem} \Leftrightarrow G|B$$

a new rule

$$r^3_{find} \ @ \ find, H_{kep}, H_{rem} \Leftrightarrow G|H'$$

is added which replaces the constraints from an original rule's head. That is done by the corresponding selected constraints that are needed by the modified rule r^3. $find/0$ is a new constraint, which must be added manually. This constraint signals the handler to search for a new match. The rules from the gcd example are modified to

$$gcd1^3_{find} \ @ \ find, gcd(0) \Leftrightarrow gcd\_sel(0).$$

$$gcd1^3 \ @ \ gcd\_sel(0) \Leftrightarrow true.$$

$$gcd2^3_{find} \ @ \ find, gcd(I), gcd(J) \Leftrightarrow J \geq I, I > 0 | gcd\_sel(I), gcd\_sel(J).$$

$$gcd2^3 \ @ \ gcd\_sel(I), gcd\_sel(J) \Leftrightarrow J \geq I, I > 0 | K \text{ is } J - I, gcd(K), gcd(I).$$

A further rule

$$find \Leftrightarrow nofind.$$

is added to the end of the CHR program and

$$find \backslash nofind \Leftrightarrow true.$$

to its beginning. The first rule gives the user the possibility to see that no match was found and the second one removes the $nofind$ constraints when a new $find$ constraint is added, so that old $nofind$ constraints cannot be looked up when a new $find$ constraint is added. I present the gcd example and start with a state where the two constraints $gcd(9)$ and $gcd(6)$

are in the CHR store:

$$\langle [], \{gcd(6)\#1, gcd(9)\#2\}\rangle_3$$

Now the constraint $gcd\_sel(6)$ is added to the goal stack. Therefore, the rule gcd_{sel} is triggered, resulting in the state

$$\langle [], \{gcd\_sel(6)\#3, gcd(9)\#2\}\rangle_4$$

No rule can be applied from this state, but when adding another $gcd\_sel(9)$ constraint, the rule gcd_{sel} is applied again to form the state

$$\langle [gcd\_sel(9)\#4], \{gcd\_sel(6)\#3\}\rangle_5$$

The $gcd\_sel(9)$ constraint in the goal stack is compared to the occurrences of the remaining rules. Finally, the rule $gcd2^3$ is applied to form the final state

$$\langle [], \{gcd(3)\#5, gcd(6)\#6\}\rangle_7$$

When a $find$ constraint is added

$$\langle [find], \{gcd(3)\#5, gcd(6)\#6\}\rangle_7$$

a match is found with the occurrence of $find$ in rule $gcd2^3_{find}$, which results in the state

$$\langle [gcd\_sel(3), gcd\_sel(6)], \{\}\rangle_8$$

The $gcd\_sel(3)$ constraint is moved to the CHR store, because no rule can be applied. Then the rule $gcd2^3$ can be applied which yields the final state

$$\langle [], \{gcd(3)\#11, gcd(3)\#10\}\rangle_{12}$$

After adding two more $find$ constraints, the goal store only contains one $gcd(3)$ constraint.

The second approach is chosen with the same arguments mentioned before: It is portable to other CHR implementations and the documentation of the K.U. Leuven system is not very detailed on this topic and the features required could also be removed or changed in future versions.

By using the modifications described above, a CHR program can be created that can be executed step by step with selected rules and only with the selected constraints. Given a CHR program $p = (C, R)$ consisting of a set of constraint definitions C and a list of rules R, the new CHR program $p_{interactive} = (C', R')$ can be created as follows:

$$C' = \{find, nofind, start, noprocess, rule\_activate(int), rule\_deactivate(int)\} \cup \{\{c\_sel, c\_desel, c\} | c \in C\}$$

$$\begin{array}{lll} R' = & [(find \backslash nofind \Leftrightarrow true), (start \backslash noprocess \Leftrightarrow true)] & \\ & ||[(c\_sel(N) \backslash c(N) \Leftrightarrow true), (c\_sel(N), c\_desel(N) \Leftrightarrow c)] & \forall c \in C \\ & ||[r^4_{find}, r^4] & \forall r \in R \\ & ||[(find \Leftrightarrow nofind), (start \Leftrightarrow noprocess)] & \end{array}$$

where r^4_{find} is a combination of the rule r^3_{find} and r^2 where an additional $rule\_activate(i)$ constraint is added to H_{kep} of r^3_{find} and where i is the number of rule r of the original program p. r^4 is a combination of the rules r^1, r^2, and r^3 where an additional $start$ constraint is added to H_{rem} of r^3, so that the rule is applied only when a $start$ constraint is added by the user. Furthermore, an additional $rule\_activate(i)$ is added to H_{kep} where i is the number of the rule r of the original program p. The operator $||$ is the concatenation

operator for lists. The order of the rules is important, because of the operational semantics described in section 2.1.

The rules of the CHR program for the gcd example would, therefore, be

$$find \backslash nofind \Leftrightarrow true.$$

$$start \backslash noprocess \Leftrightarrow true$$

$$gcd_{sel} \ @ \ gcd\_sel(N) \backslash c(N) \Leftrightarrow true$$

$$gcd_{desel} \ @ \ gcd\_desel(N), c\_sel(N) \Leftrightarrow c(N).$$

$$gcd1^4_{find} \ @ \ rule\_activate(1) \backslash find, gcd(0) \Leftrightarrow gcd\_sel(0).$$

$$gcd1^4 \ @ \ rule\_activate(1) \backslash start, gcd\_sel(0) \Leftrightarrow true.$$

$$gcd2^4_{find} \ @ \ rule\_activate(2) \backslash find, gcd(I), gcd(J) \Leftrightarrow J \geq I, I > 0 | gcd\_sel(I), gcd\_sel(J).$$

$$gcd2^4 \ @ \ rule\_activate(2) \backslash start, gcd\_sel(I), gcd\_sel(J) \Leftrightarrow J \geq I, I > 0 | K \text{ is } J-I, gcd(K), gcd(I).$$

$$find \Leftrightarrow nofind.$$

$$start \Leftrightarrow noprocess.$$

For a CHR program of this form, a rule can only be applied if it is activated and a $start$ constraint is added. Furthermore, the activated rules are only applied to the selected constraints. These constraints can be manually selected or by adding a $find$ constraint. When adding a $find$ constraint, only the rules are used that have been activated by the user. The rules can be activated and deactivated by adding $rule\_activate$ and $rule\_deactivate$ constraints.

The following list provides the instructions on how to use a program of the form above:

activation and deactivation of rules Add the according $rule\_activate(i)$ or $rule\_deactivate(i)$ constraint to the store. Only the activated rules are used for rule application or for automatically selecting constraints.

find constraints that match to the activated rules Add a c_desel constraint for every c_sel constraint in the CHR store to remove the current selection. Then add a $find$ constraint. The activated rules are then used to find constraints that match to a head. These constraints are then present as selected constraints in the CHR store. The according rule can be applied to them by adding a $start$ constraint. Before constraints can be selected manually, rules have to be activated.

automatic rule application First, add a $find$ and then a $start$ constraint to the CHR store until a $noprocess$ constraint can be looked up in the CHR store. When only a fixed number of steps should be executed, `find` and `noprocess` constraints are added alternately to the CHR store the according number of times. This procedure can also be used to execute one step of the program automatically. Only the activated rules are used for finding and applying constraints. Therefore, these rules must be activated prior to adding $find$ and $start$ constraints.

manual step by step rule application To select constraints, the available constraints must be visible to the user. New c_sel constraints must be added with the terms from the chosen c constraint. For deactivation, c_desel constraints with the same terms as the c_sel constraints are added to the store. To apply one of the activated rules to the selection, a $start$ constraint must be added. When no rule could be applied, a $noprocess$ constraint is found in the CHR store. The rules have to be activated by the user prior to adding a $start$ constraint.

4.7.1.2 Code generation

With the methods described in the previous section, the CHR encoding of a GTS can be modified in a way that it can be executed step by step and that the nodes and edges can be selected which are matched by the next rule application. The code for realizing this will be generated in the JCHR syntax, just as described in section 4.6.1.3. But in order to use the generated handler, it has to be compiled to a java class (section 4.6.2.4). This class contains methods that depend on the constraint definitions in the JCHR program. Two methods are provided for each constraint type: one to receive a list of the constraints in the CHR store and one to add the constraint to the CHR store. Therefore, loading the generated class dynamically is difficult, because the methods would have to be determined dynamically. A solution to this problem is to generate another java source file which implements a java interface that provides methods to access node or edge constraints of a JCHR handler more generically.

In more detail, this interface must provide methods to return all the information contained in the JCHR program plus the information about host graphs which are not encoded in the JCHR program. For that reason, the interface provides methods to return all rule and host graphs' names, to activate and deactivate rules by giving their name, to restart the CHR program with a new host graph (by giving its name), to read all node and edge constraints, to (de)select node and edge constraints, to add $start$ and $find$ constraints, and to check whether a rule has been applied or if a match was found. Node and edge constraints need a representation that is independent from the underlying CHR handler, hiding the different constraints for nodes and edges. Therefore, two wrapper classes for these constraint types must be provided. The methods for activating and deactivating constraints receive an instance of one of the wrapper classes and add new constraints to the CHR handler. This generated java class can then be dynamically loaded from the tool to access the JCHR handler in a generic way.

4.7.1.3 Graphical user interface

The graphical user interface consists of two parts: The part where settings are made for generating the CHR code and the part where the generated handler is displayed. Figure 4.43 shows the first part. The settings that can be made here are similar to the ones described in section 4.6.1.2. Only the list for selecting host graphs for export is missing, because they are all available for selection. When activating and deactivating rules, they should be displayed in the editors. When the CHR handler is created and loaded, the second part of the GUI becomes visible which is shown in figure 4.44. A list for activating and deactivating rules is provided together with a combo box that lists the available host graphs. When rules are (de)activated they should also be displayed in the editors of the platform. The main element is, however, the graphical representation of the current host graph. As the rules of the GTS are applied to a host graph by using CHR, automated layout for the graph is needed, because the constraints do not contain information about their position. The nodes and edges can be selected by double clicking on them, so that rules can be applied to them. For that reason, the displayed graph must highlight which nodes and edges are currently selected. Additionally, hyperlinks are available that enable the user to apply the currently activated rules to the currently selected constraints, to select constraints that can be matched by one of the activated rules, and to apply the rules several times in a row or until a final state can be reached. When running the GTS to a final state, another button must be displayed to cancel the calculations in case a non-terminating GTS was designed. Furthermore, a hyperlink is provided to deselect all nodes and edges. The use case diagram in figure 4.42 shows the interaction between the user and the tool.

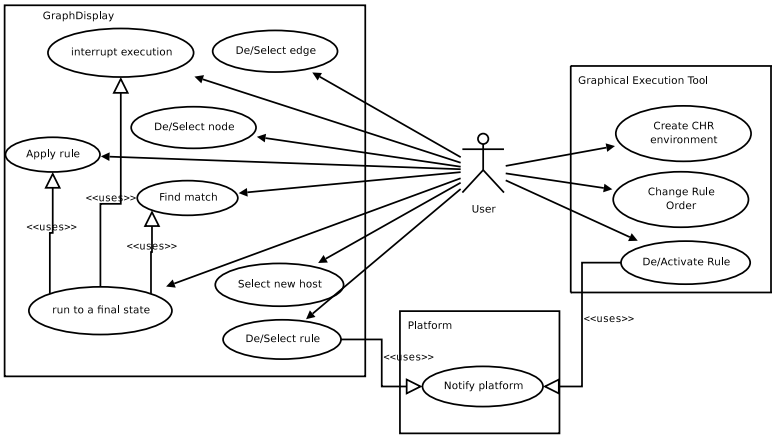


Figure 4.42: UML use case diagram for the interaction between the user and the graphical simulation tool.

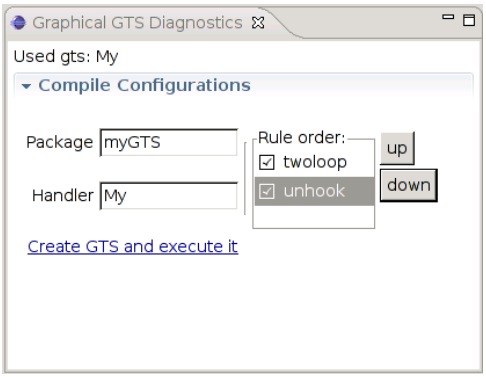


Figure 4.43: Selection and order of the rules for graphical simulation.

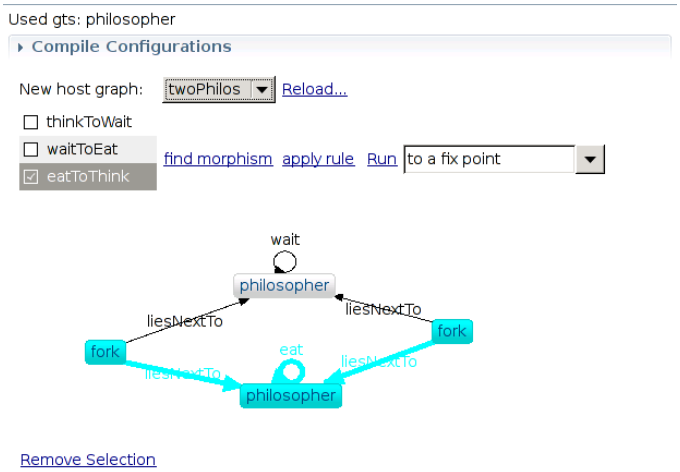


Figure 4.44: Running simulation of a GTS with selected edges and nodes.

4.7.1.4 Synchronization of display and handler

To represent the graph in a diagram, it has to be synchronized with the constraint store of the underlying CHR handler. An algorithm is provided that checks for removal and addition of constraints. The CHR rules that are generated for the selection and deselection of constraints in fact remove and add constraints to the store. The graphical representation, however, should not remove and re-add the according nodes, because this would result in laying out the graph again. The algorithm maintains mappings from node constraints to the displayed nodes (*node mapping*) and from edge constraints to the displayed edges (*edge mapping*). In fact, the node mapping does not map the constraints to the displayed nodes, but the identifier attribute (that is the first attribute of the constraint, see section 2.3) is mapped to the displayed node. That is done to overcome the problem that two different constraint types encode the same node which would otherwise lead to a re-layout of the displayed graph when a node becomes (de)selected. When the view of the displayed graph is updated, the node constraints are all stored in a new mapping that also links the first identifier attribute of the node constraints to nodes in the diagram. If the node attribute is not available in the node map, a new node for the diagram is added otherwise the old node from the diagram is taken from the old node mapping. The same procedure is repeated for the selected nodes. After that, the identifier attributes that are in the old node map, but not in the new node map, are searched and their nodes in the graph display are disposed. The same procedure is repeated for edge constraints, as well. The difference is that the mapping goes directly from the constraints to the displayed edges. Because of that, edges that are (de)selected are removed and added again, but this does not impose any problems. This will not lead to a re-layout, because the position of the edge is determined by the positions of the source and target node. This synchronization has to be done each time the CHR store is changed in the handler.

4.7.2 Implementation

In this section, I give some details about the generation of the JCHR handler and the corresponding interface to access it. Then I describe the implementation of the GUI where I pay special attention to the graphical representation of the constraints as a graph. This tool can be found in the project `org.uniulm.gts.analystools.graphicalsimulation`.

4.7.2.1 Generation of the JCHR handler and its interface

The generation of the CHR source is similar to the method described in 4.6, but the already developed template is not reused, because the generated code is no regular CHR encoding (section 2.3). The CHR encoding is modified according to the rules in section 4.7.1.1 and this modified encoding is generated directly from the GTS model without generating the regular encoding first. However, additionally to the CHR source, a java source file has to be generated which implements the interface `IGTSFacade` which contains methods to generically access the JCHR handler. For a list of the provided methods, please see table 4.5. The generated code for the java interface does not implement the interface directly, but an abstract base class `AbstractGtsFacade`. This class implements some methods of the interface and provides further methods to simplify generated code. Table 4.6 shows the methods of the abstract base class, that need to be implemented. All these methods are implemented in a way that they query the CHR store or add a CHR constraint to it. The code generation is done by JET, just like for the JCHR file. The activation of rules, for example, is generated by the template in listing 4.17. This template iterates over the rules and prints *if statements* that add `rule_activate` constraints to the handler. The dining philosophers problem would, consequently, produce the code shown in listing 4.18. The

Read Handler	returns...
`List<String> getHosts()`	list of host identifiers
`List<String> getRules()`	list of rule identifiers
`List<String> getTypeNodes()`	list of type node identifiers
`List<String> getTypeEdges()`	list of type edge identifiers
`List<GtsChrNode> getNodes()`	list of nodes
`List<GtsChrEdge> getEdges()`	list of edges
`List<GtsChrNode> getSelectedNodes()`	list of selected nodes
`List<GtsChrEdge> getSelectedEdges()`	list of selected edges
`boolean nothingFound()`	whether match was found
`boolean nothingProcessed()`	whether rule was applied
Modify Handler	
`void putStart()`	apply rule
`void findMatchMorphism()`	find match
`void activateRule(String rule)`	activate rule
`void deactivateRule(String rule)`	deactivate rule
`void activateAllRules()`	activate all rules
`void deactivateAllRules()`	deactivate all rules
`void setHost(String host)`	put new host encoding in goal store
`void select(GtsChrNode n)`	select node
`void select(GtsChrEdge e)`	select edge
`void deselect(GtsChrNode n)`	deselect node
`void deselect(GtsChrEdge e)`	deselect edge

Table 4.5: List of methods provided by the `IGTSFacade` interface.

Method	Description
`activateRule(String rule)`	Activate the rule with the given id
`deactivateRule(String rule)`a	Deactivate the rule with the given id
`deselect(GtsChrNode n)`	Deselect the given node constraint
`deselect(GtsChrEdge e)`	Deselect the given edge constraint
`select(GtsChrNode n)`	Select the given node constraint
`void select(GtsChrEdge e)`	Select the given edge constraint
`putStart()`	Do a transformation step
`findMatchMorphism()`	Find a match morphism
`setHost(String host)`	Start program with the host of the given id
`getNodes(String type)`	Return all node constraints of the given type
`getEdges(String type)`	Return edge constraints of the given name
`getSelectedEdges(String s)`	Return selected edge constraints of the given name
`getSelectedNodes(String s)`	Return selected nodeconstraints of the given name
`boolean nothingFound()`	Returns if a rule could be applied
`boolean nothingProcessed()`	Returns if a match could be found

Table 4.6: List of methods that need to be implemented by classes implementing the base class `AbstractGTSFacade`.

```
1 public void activateRule(String rule) {
2  <c:setVariable var="ruleCnt" select="0"/>
3  <c:iterate select="reorder($gts/transformations/@id,$order)" var="rule" delimiter="
       else ">
4  <c:setVariable var="ruleCnt" select="$ruleCnt+1"/>
5  if(rule.equals("<c:get select="$rule"/>")){
6   handler.tellRule_activate(<c:get select="$ruleCnt"/>);
7  }</c:iterate>
8 }
```

Listing 4.17: JET template for generating the `activateRule()` method.

```
public void activateRule(String rule) {
 if(rule.equals("thinkToWait")){
  handler.tellRule_deactivate(1);
 }else if(rule.equals("waitToEat")){
  handler.tellRule_deactivate(2);
 }else if(rule.equals("eatToThink")){
  handler.tellRule_deactivate(3);
}
}
```

Listing 4.18: Generated code from the template shown in listing 4.17.

```
if(host.equals("twoPhilos")){
 handler=new PhilosopherHandler();
 HashMap<String,LogicalInt> nodes=new HashMap<String,LogicalInt>();
 nodes.put("P2",new LogicalInt());
 handler.tellN_philosopher(nodes.get("P2"),3);
 nodes.put("F1",new LogicalInt());
 handler.tellN_fork(nodes.get("F1"),2);
 nodes.put("F2",new LogicalInt());
 handler.tellN_fork(nodes.get("F2"),2);
 nodes.put("P1",new LogicalInt());
 handler.tellN_philosopher(nodes.get("P1"),3);
 handler.tellE_liestNextTo(nodes.get("F1"),nodes.get("P2"));
 handler.tellE_liestNextTo(nodes.get("F2"),nodes.get("P2"));
 handler.tellE_eat(nodes.get("P2"),nodes.get("P2"));
 handler.tellE_wait(nodes.get("P1"),nodes.get("P1"));
 handler.tellE_liestNextTo(nodes.get("F1"),nodes.get("P1"));
 handler.tellE_liestNextTo(nodes.get("F2"),nodes.get("P1"));
}
```

Listing 4.19: Generated code to add the constraints of the encoded host graph *twoPhilos* of the dining philosophers example.

other methods for (de)selecting nodes and edges are implemented similarly. The method `setHost(String)` creates a new handler and adds the constraints of the encoding of the host graph which is identified by the given string. Listing 4.19 shows the code that is generated for the *twoPhilos* host graph of the dining philosophers example. First, new `LogicalInt` values are created and saved in a map, identified by the node identifier of the host. These values are then added to the handler as new fork or philosopher constraints together with the number of adjacent edges.

Both files are generated when the user has set the order of the rules and given the name for the package and the handler in part one of the GUI described in section 4.6.1.2. The JCHR program must then be compiled by the K.U. Leuven JCHR compiler, just as described in section 4.6.2.4.

4.7.2.2 Loading of the generated files

The result of the code generation and JCHR compilation are two java classes that are compiled by the eclipse platform to java byte-code. These classes can be loaded by the java class loader. The problem is that the generated implementation of the interface depends on the plug-in created for this GTS analysis tool, because it contains the interface definition. Therefore, the tool must register itself on start-up as a dependency in the project where the generated files are placed in. That is why new GTS projects are eclipse plug-in projects (see section 4.2.1.1), because plug-ins can only be added as dependencies to plug-in projects and not to common java projects. Listing 4.20 shows how a plug-in is programmatically added as a dependency. As described in section 2.4, the dependencies of a project are saved in the `MANIFEST.MF` file of the project. Eclipse offers a set of classes to query and edit elements of the current plug-in project. The `BundlePluginModelBase`

```
1  //the current project is saved in curProject
2  BundlePluginModelBase pm=(BundlePluginModelBase)PluginRegistry.findModel(curProject);
3  IPluginBase pb=pm.getPluginBase();
4  String pluginId = GraphicalExecutionDiagnosticToolPlugin.PLUGIN_ID;
5  PluginImport plugin = new PluginImport();
6  ManifestElement element =
7     ManifestElement.parseHeader(Constants.REQUIRE_BUNDLE, pluginId);
8  plugin.load(element, 1);
9  plugin.setModel(pb.getModel());
10 //set the model editable
11 ((WorkspaceBundleModel)((BundlePluginModelBase)pb.getModel()).
12    getBundleModel()).setEditable(true);
13 //add the plugin as a dependency
14 pb.add(plugin);
15 //save the made changes
16 pm.save();
```

Listing 4.20: Adding a plug-in as dependency of a project.

can be received from the current project to get a `IPluginBase` instance which manages the dependencies of a project. A new `PluginImport` instance can be added to it that contains the plug-in id of the graphical analysis tool plug-in (lines 2–9). In order to save these changes to the `MANIFEST.MF` file, the model has to be set editable (lines 11,12). When the tool has added itself as dependency to the current project, the interface and the JCHR can be loaded dynamically by the class loader.

4.7.2.3 Using the generated handler

The JCHR handler is not used directly, but via the interface that is also generated. To represent the constraints contained in the handler, *Zest* is used. Zest is a tool for displaying and laying out graphs. It is based on draw2d (see section 4.4) for drawing the nodes and edges. Zest provides a `Graph` widget to which `GraphNode` and `GraphConnection` widgets can be added. Widgets are elements of the GUI provided by the SWT. The elements can then be laid out automatically. The graph widget is updated every time a constraint is added to the JCHR handler. Therefore, edge constraints are represented by `GraphConnection` widgets and node constraints are represented as `GraphNode` widgets. The algorithm described in section 4.7.1.4 is used to synchronize the contents of the JCHR handler with the graph widget. To allow the (de)selection of constraints, a `MouseListener` is added to the `Graph` instance. When a double click occurs, the constraint linked to the currently selected `GraphNode` or `GraphConnection` will be selected in the handler by calling the `select(GtsChrNode)` method of the generated interface class. All the communication with the handler is done via the generated implementation of the `IGTSFacade`, as described in the sequence diagrams in figure 4.45. When the user clicks on a rule in the list of rules or double clicks an edge or node, an according method in the interface is activated which adds a new constraint to the CHR handler.

The option to simulate the GTS until a final state is reached must run in an own thread, because this can be a very long running task which would otherwise block the whole system. Eclipse provides this functionality in the form of a `Job` object which has further benefits: It is shown in the eclipse job view and can also be stopped from there. The Job itself repeatedly calls the methods `putStart()` and `findMatchMorphism()` of the interface (see also section 4.7.1.1) and stops when no rule can be applied anymore (done by calling the method `nothingProcessed()`) or when the user requests to stop the job. The tool adds itself as listener to the Job, so that it gets notified when the CHR program has finished to synchronize the graphical representation with the state of the CHR handler. Figure 4.46 shows how morphisms are found and how rules are applied by using the generated implementation of the interface. Finding a new match morphism is done by adding a $find$ constraint which selects the constraints that can be matched by a rule. But before

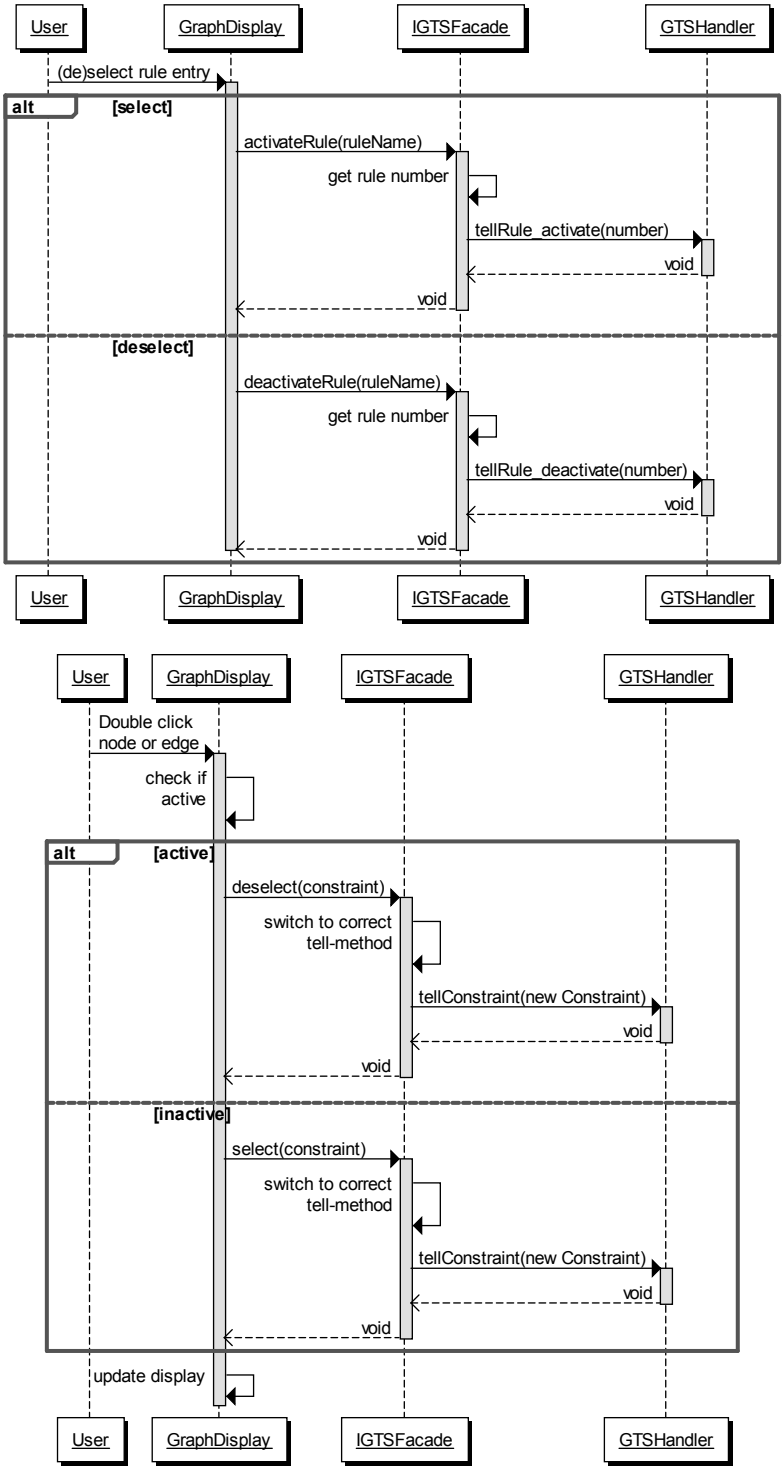


Figure 4.45: UML sequence diagram for the activation or deactivation of rules (upper) and node or edge constraints (lower).

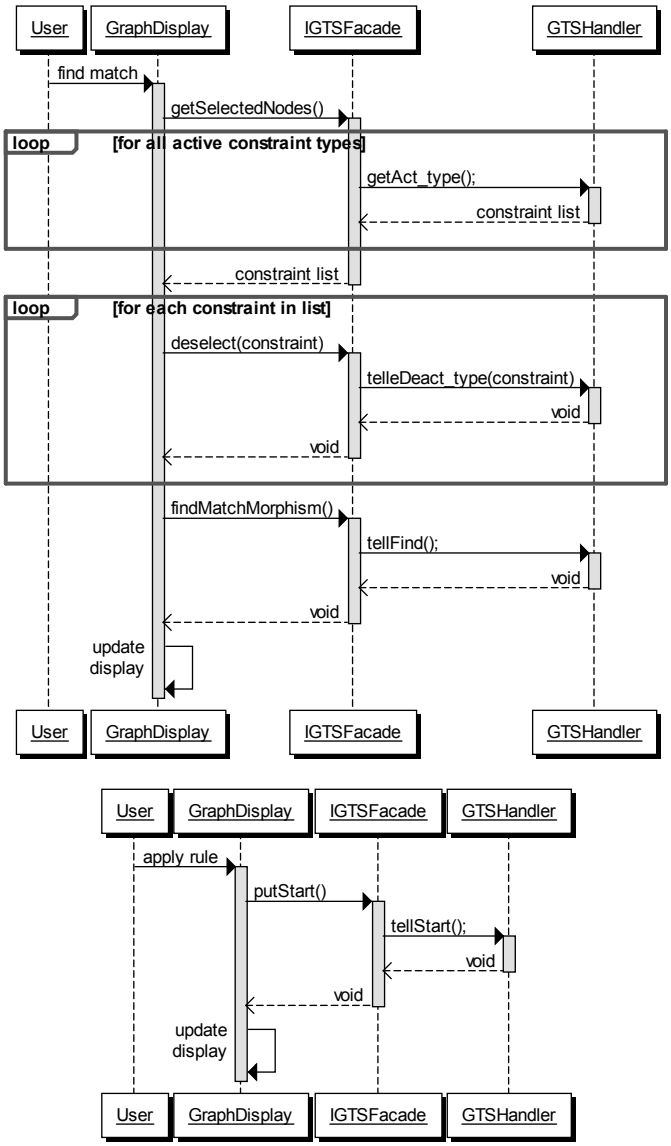


Figure 4.46: UML sequence diagram for finding matches (upper) and applying rules (lower).

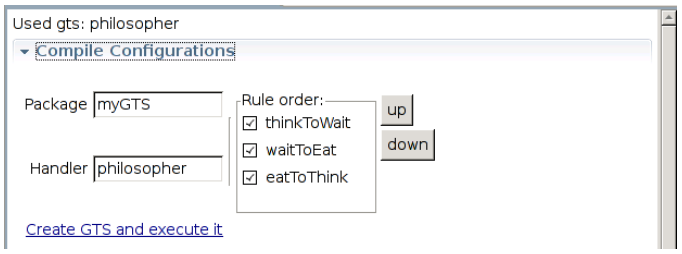


Figure 4.47: Dialog to adjust code generation.

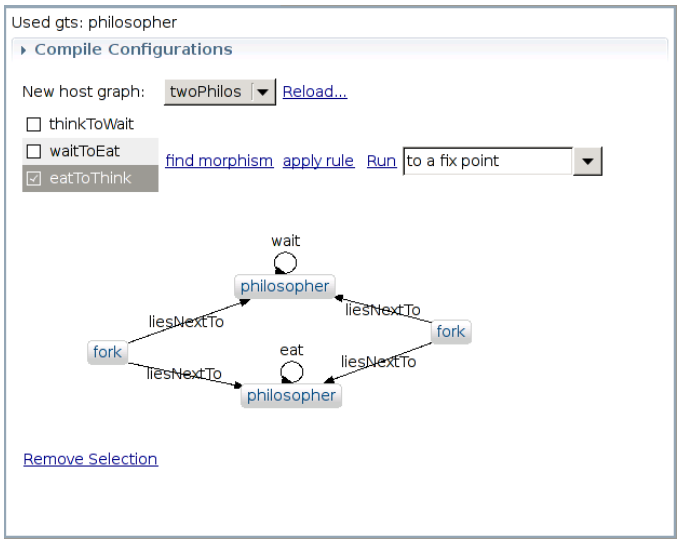


Figure 4.48: Interface that shows a host graph after the generation of the handler.

adding the $find$ constraint, all selected constraints are deselected first, so that the rule is applied to the constraints that are selected by the $find$ constraint. Applying a rule is done by adding a $start$ constraint to the store, so that rules can be applied.

4.7.3 Sample computation

In this section, I want to present the simulation of the dining philosophers example. Figure 4.47 shows the dialog which allows to select the rules and set their order. Furthermore, a package and a handler name can be entered, but the default values are used just as the default ordering of the rules. After a click on the *Create GTS and execute it* hyperlink, a message will appear at the bottom and the scheduled job will be shown in the bottom right corner of the workbench. After the compilation of the JCHR program is finished (this takes about five to ten seconds) the second part of the GUI will open. Figure 4.48 shows this GUI with the upper dialog collapsed. The host graph with the two philosophers is selected showing the corresponding host graph. When activating constraints in a way that none of the selected rules can match it like in figure 4.49 and the *apply rule* hyperlink is clicked, a message will be shown which informs the user that no rule could be applied. With a click on *find morphisms* and only the rule *eating* activated, the next match will be highlighted in the graph as shown in figure 4.50. After clicking on *apply rule*, the rule is applied, resulting in the graph shown in figure 4.51. Every time when a rule is selected in the list, it is also

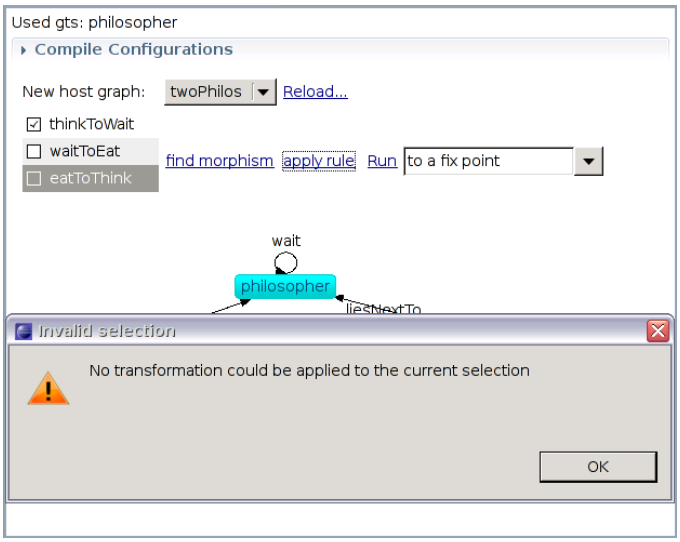


Figure 4.49: Message that is displayed when no rule can be applied.

displayed in the editor.

4.8 Further analysis tools

In this section, I present another analysis tool and give an idea how a CHR analysis tool could be realized that checks confluence of GTS.

4.8.1 Random host generation tool

An analysis tool has been developed for the easy generation of random host graphs. This tool provides a view where the numbers of type nodes and type edges can be entered. When all numbers are entered, a graph with the selected amount of nodes of each type is created. The nodes are randomly connected by the selected number of edges. It is taken care of, that the edges are connected to nodes of the correct type. Furthermore, if there are no nodes of the correct type for a given edge, the edges are not added, of course. This tool implements the analysis tool interface, so that the model and the command-stack can be provided. When the tool is started, it launches an eclipse wizard that contains two tables which let you enter the numbers of the type nodes and type edges. The names of the nodes in the host graph are increasing numbers. The graph is created using the command stack, but the nodes and edges are created without the command-stack, so that undoing the creation directly removes the graph and not every node or edge separately. Figure 4.52 shows the GUI of the graph generation wizard. The default values are five nodes and edges per type.

4.8.2 Confluence analysis

As already mentioned in section 4.6, there is no tool available for analyzing confluence of JCHR programs. Confluence means that the order of the rule application a CHR program does not influence its final state. Confluence for CHR programs is covered in [20]. Basically, a confluence tester creates an input I from the heads of two rules r_1 and r_2, applies

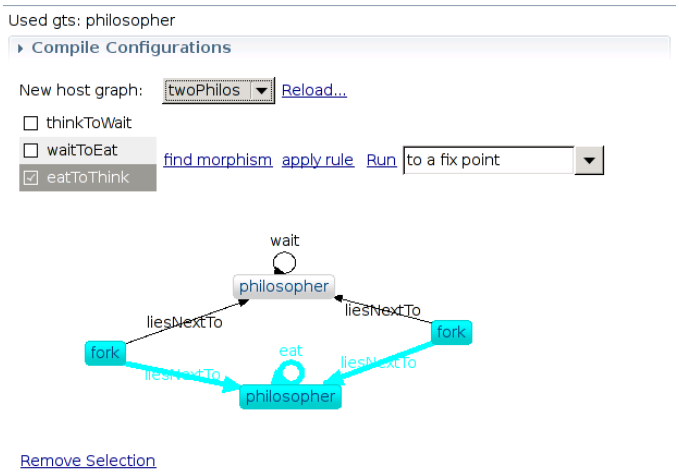


Figure 4.50: Highlighted match morphism for the activated rules.

r_1 to I, resulting in I_1 and applies r_2 to I, resulting in I_2. Then, two runs of the CHR program are done, one with input I_1 and one with input I_2. If the constraint and the built-in stores of the two final states are equal[1] and this holds for all pairs of rules, then the CHR program is confluent.

This test can also be used to test confluence for CHR encoded GTS ([32]). However, when creating Inputs from heads of the rules, further attention must be paid. The inputs might not always be an encoding of a correct graph. Therefore, these inputs do not need to be tested.

A CHR tool would need to create an executable version of the CHR program by using the CHR tool from section 4.6.2.4 and create inputs according to the description above. These inputs are then used as goals for the generated JCHR handler. The resulting final states of the JCHR handler are then compared to decide whether the CHR program is confluent. Pairs of rules that might lead to a non-confluent system can be listed in the tool and displayed in the editors.

[1] In fact they do not have to be equal, but they must be *variants* of each other. This described in [20].

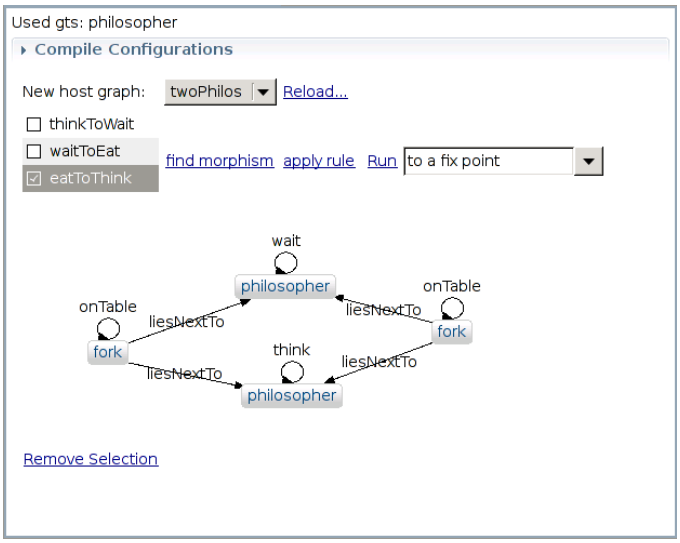


Figure 4.51: New host graph after a rule has been applied.

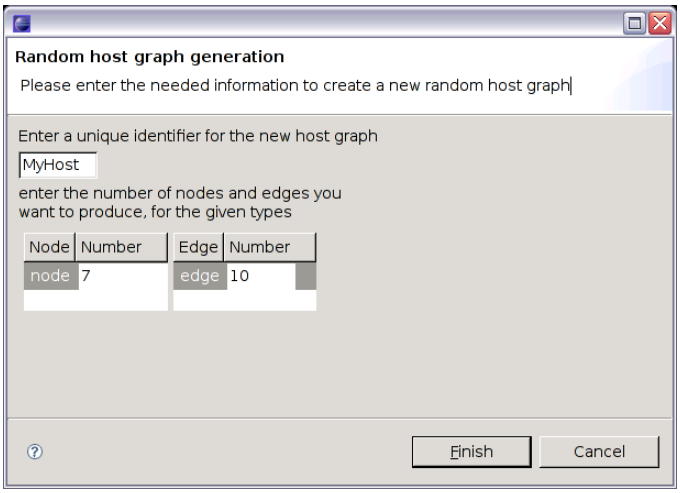


Figure 4.52: GUI of the wizard for creating a random host graph.

5 Conclusion

In this final chapter, the results of the thesis are summarized and possible future projects based on it are presented. Furthermore, some application areas of the developed platform are listed.

5.1 Summary of results

During this thesis, an extensible platform, based on eclipse, for editing and analyzing *Graph Transformation Systems (GTS)* has been developed. This platform allows the editing of GTS not only with a graphical, but also with a textual editor. The textual representation is based on *Constraint Handling Rules (CHR)*. Raiser describes in [32] how GTS can be encoded in a CHR program. The editing can be done simultaneously by the two editors. The benefit of encoding GTS in CHR is that the rule applications in GTS correspond to rule applications in CHR. Therefore, CHR can be used to simulate GTS. Another advantage of using the CHR encoding is, that analysis methods that can be applied to CHR can now also be applied to GTS, e.g. confluence analysis and operational equivalence analysis.

The platform's functionality can be extended by further tools. For that reason, an eclipse extension point has been provided that can be used to add analysis and editing features to the platform. A few prototypical tools are already provided. One tool offers the possibility to apply the rules of a GTS to a host graph interactively by choosing the nodes and edges to which a selected rule can be applied. This tool is based on the encoding of GTS in CHR and offers a graphical front-end in order to display the modified graph. A similar feature can be found in the already available programs for GTS, AGG (section 3.1.1) and Groove (section 3.1.2). The other two exemplary tools are for creating random host graphs and termination analysis based on a ranking function. More sophisticated tools like the critical pair analysis that can be found in AGG would not have been in the scope of this thesis and are not provided.

To give the possibility to add analysis methods that are available for K.U. Leuven JCHR (the CHR implementation based on java, used by the platform), another tool has been provided. This tool defines another eclipse extension point, so that analysis methods for JCHR can easily be added to the platform. One such *CHR tool* is provided in form of the K.U. Leuven JCHR compiler to create an executable environment of a GTS encoded in CHR. More sophisticated CHR analysis methods like confluence analysis are not provided, because no implementations have been available and it would have been not in the scope of this thesis to implement them.

The platform can be used for various purposes. It provides the possibility to create executable JCHR handlers based on the GTS, so that they can be used in other projects. Furthermore, the platform can be used in lectures about CHR, GTS, or rule based programming languages in general to demonstrate application areas and show how rule application is done in CHR and GTS. Rudimentary analysis methods are already provided to make this platform an alternative to the already available programs (AGG and Groove) for designing GTS.

To conclude, the platform provides new and interesting features. Two editing possibilities for GTS are available, that can be used in parallel. The platform is not a monolithic block

of functionality, but it can be extended by further tools to analyze and edit GTS. Due to the possibilities of encoding a GTS in CHR, analysis methods developed for CHR can be applied to GTS. These analysis methods can easily be embedded in the platform as further tools. Guides on how to create those tools are given in section 4.5.2 and section 4.6.2.4. An installation guide for the platform can be found in appendix A.

5.2 Future work

The developed platform is a basis for future work, because new analysis tools for GTS and for CHR can be added to it. Projects for lab courses could include analysis tools for the calculation of metrics for the GTS. Other interesting features to raise the versatility of the platform would be import tools that give the possibility to import GTS created in Groove or AGG. Projects for further master or bachelor thesis could include the realization of (semi-) automatic analysis methods for JCHR that can be integrated into the platform, e.g. the already described confluence test (section 4.8.2). Another improvement of the platform would be to add support for attributed GTS ([18]). This additional feature could either be implemented as a further tool or by modifying the editors themselves to integrate it more tightly into the platform.

Bibliography

[1] ABDENNADHER, S., AND FAWZY, S. Jchride: An integrated development environment for jchr. *22nd Workshop on (Constraint) Logic Programming* (2008).

[2] ABDENNADHER, S., KRAMER, E., SAFT, M., AND SCHMAUSS, M. Jack: A java constraint kit. *Electronic Notes in Theoretical Computer Science 64* (September 2002), 1–17.

[3] ANISZCZYK, C., AND HUDSON, R. Create an eclipse-based application using the graphical editing framework. World Wide Web electronic publication, 2007. `http://www.ibm.com/developerworks/library/os-eclipse-gef11/`.

[4] BOAG, S., BERGLUND, A., CHAMBERLIN, D., SIMÉON, J., KAY, M., ROBIE, J., AND FERNÁNDEZ, M. F. XML path language (XPath) 2.0. W3C Recommendation xpath20-20070123, W3C, January 2007. `http://www.w3.org/TR/2007/REC-xpath20-20070123/`.

[5] BOESPFLUG, M. TaiChi:how to check your types with serenity. *The Monad.Reader 9* (Nov. 2007), 17–31.

[6] BUDINSKY, F., MERKS, E., AND STEINBERG, D. *Eclipse Modeling Framework 2.0 (2nd Edition)*. Addison-Wesley Professional, 2006.

[7] CORRADINI, A., EHRIG, H., MONTANARI, U., RIBEIRO, L., AND ROZENBERG, G., Eds. *Graph Transformations, Third International Conference, ICGT 2006, Natal, Rio Grande do Norte, Brazil, September 17-23, 2006, Proceedings* (2006), vol. 4178 of *Lecture Notes in Computer Science*, Springer.

[8] DIJKSTRA, E. W. Hierarchical ordering of sequential processes. *Acta Informatica 1* (1971), 115–138.

[9] DJELLOUL, K., DUCK, G. J., AND SULZMANN, M., Eds. *CHR '07: Proc. 4th Workshop on Constraint Handling Rules* (Porto, Portugal, Sept. 2007).

[10] ECLIPSE FOUNDATION, INC. Eclipse modeling framework overview. World Wide Web electronic publication, 2008. `http://help.eclipse.org/stable/topic/org.eclipse.emf.doc/references/overview/EMF.html`.

[11] ECLIPSE FOUNDATION, INC. Java emitter templates (jet). World Wide Web electronic publication, 2008. `http://www.eclipse.org/modeling/m2t/?project=jet`.

[12] ECLIPSE FOUNDATION, INC. Jet developer guide. World Wide Web electronic publication, 2008. `http://help.eclipse.org/stable/index.jsp?nav=/27`.

[13] ECLIPSE FOUNDATION, INC. Textual modeling framework (tmf): Xtext. World Wide Web electronic publication, 2008. `http://www.eclipse.org/modeling/tmf/`.

[14] ECLIPSE FOUNDATION, INC. Zest: The eclipse visualization toolkit. World Wide Web electronic publication, 2008. `http://www.eclipse.org/gef/zest/`.

[15] ECLIPSE FOUNDATION, INC. Eclipse modeling framework (emf). World Wide Web electronic publication, 2009. `http://www.eclipse.org/modeling/emf/`.

[16] ECLIPSE FOUNDATION, INC. Graphical editing framework (gef). World Wide Web electronic publication, 2009. `http://www.eclipse.org/gef/`.

[17] ECLIPSE FOUNDATION, INC. Graphical modeling framework (gmf). World Wide Web electronic publication, 2009. `http://www.eclipse.org/gmf/`.

[18] EHRIG, H., EHRIG, K., PRANGE, U., AND TAENTZER, G. *Fundamentals of Algebraic Graph Transformation (Monographs in Theoretical Computer Science. An EATCS Series)*. Springer, March 2006.

[19] EHRIG, H., HECKEL, R., KORFF, M., LÖWE, M., RIBEIRO, L., WAGNER, A., AND CORRADINI, A. Algebraic approaches to graph transformation. part ii: single pushout approach and comparison with double pushout approach. 247–312.

[20] FRÜHWIRTH, T. *Constraint Handling Rules*. Cambridge University Press, June 2009. to appear.

[21] FRÜHWIRTH, T., AND ABDENNADHER, S. *Essentials of Constraint Programming*. Springer, 2003.

[22] GAMMA, E., AND BECK, K. *Contributing to Eclipse: Principles, Patterns, and Plugins*. Addison-Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 2003.

[23] GAMMA, E., HELM, R., JOHNSON, R., AND VLISSIDES, J. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Professional, 1995.

[24] HOLZBAUR, C., AND FRÜHWIRTH, T. A Prolog Constraint Handling Rules compiler and runtime system. [25], pp. 369–388.

[25] HOLZBAUR, C., AND FRÜHWIRTH, T., Eds. *Special Issue on Constraint Handling Rules*, vol. 14(4) of *Journal of Applied Artificial Intelligence*. Taylor & Francis, Apr. 2000.

[26] KAY, M. Xsl transformations (xslt) version 2.0. Tech. Rep. xslt20-20070123, World Wide Web Consortium, 2007. `http://www.w3.org/TR/2007/REC-xslt20-20070123/`.

[27] KRASNER, G. E., AND POPE, S. T. A cookbook for using the model-view controller user interface paradigm in smalltalk-80. *J. Object Oriented Program. 1*, 3 (1988), 26–49.

[28] LAM, E. S. L., AND SULZMANN, M. A concurrent constraint handling rules implementation in haskell with software transactional memory. In *DAMP '07: Proceedings of the 2007 workshop on Declarative aspects of multicore programming* (New York, NY, USA, 2007), ACM Press, pp. 19–24.

[29] MOORE, W., DEAN, D., GERBER, A., WAGENKNECHT, G., AND VANDERHEYDEN, P. *Eclipse Development using the Graphical Editing Framework and the Eclipse Modeling Framework*. IBM redbooks. IBM International Technical Support Organization, Boca Raton, FL, USA, 2004.

[30] OPENARCHITECTUREWARE.ORG. openarchitectureware (oaw) homepage. World Wide Web electronic publication, 2008. `http://www.openarchitectureware.org/`.

[31] RAISER, F. Graph Transformation Systems in CHR. In *Logic Programming, 23rd International Conference, ICLP 2007* (Porto, Portugal, September 2007), V. Dahl and I. Niemelä, Eds., vol. 4670 of *Lecture Notes in Computer Science*, Springer, pp. 240–254.

[32] RAISER, F., AND FRÜHWIRTH, T. Strong joinability analysis for graph transformation systems in CHR. In *5th International Workshop on Computing with Terms and Graphs, TERMGRAPH 2009* (2009). accepted.

[33] RENSINK, A. Graphs for object-oriented verification (groove). World Wide Web electronic publication, 2003. `http://groove.cs.utwente.nl/`.

[34] RENSINK, A. The GROOVE simulator: A tool for state space generation. In *Applications of Graph Transformations with Industrial Relevance (AGTIVE)* (2004), J. Pfalz, M. Nagl, and B. Böhlen, Eds., vol. 3062 of *Lecture Notes in Computer Science*, Springer, pp. 479–485.

[35] SCHIEDGEN, M. Textual editing framework (tef). World Wide Web electronic publication, 2008. `http://www2.informatik.hu-berlin.de/sam/meta-tools/tef/index.html`.

[36] SCHRIJVERS, T., AND DEMOEN, B. The K.U.Leuven CHR system: Implementation and application. In *First workshop on constraint handling rules: selected contributions* (2004), pp. 1–5. Published as technical report: Ulmer Informatik-Berichte Nr. 2004-01, `http://www/informatik.uni-ulm.de/epin/pw/10481`.

[37] SCHRIJVERS, T., AND FRÜHWIRTH, T., Eds. *CHR '05: Proc. 2nd Workshop on Constraint Handling Rules* (Sitges, Spain, 2005), K.U.Leuven, Dept. Comp. Sc., Technical report CW 421.

[38] SCHRIJVERS, T., WIELEMAKER, J., AND DEMOEN, B. Poster: Constraint Handling Rules for SWI-Prolog. In Wolf et al. [44]. `http://www.informatik.uni-ulm.de/epin/pw/11541`.

[39] SNEYERS, J., WEERT, P. V., SCHRIJVERS, T., AND KONINCK, L. D. As time goes by: Constraint Handling Rules – A survey of CHR research between 1998 and 2007. submitted to *Journal of Theory and Practice of Logic Programming*, 2009.

[40] VAN WEERT, P., SCHRIJVERS, T., AND DEMOEN, B. K.U.Leuven JCHR: a user-friendly, flexible and efficient CHR system for Java. In Schrijvers and Frühwirth [37], pp. 47–62. `http://www.cs.kuleuven.be/~petervw/JCHR/`.

[41] VELASCO, P. P. P., AND DE LARA, J. Matrix approach to graph transformation: Matching and sequences. In Corradini et al. [7], pp. 122–137.

[42] WEERT, P. V. The k.u.leuven jchr system homepage. World Wide Web electronic publication, 2008. `http://www.cs.kuleuven.be/~petervw/JCHR/`.

[43] WEERT, P. V. K.u.leuven jchr user's manual. World Wide Web electronic publication, 2008. Available at [42].

[44] WOLF, A., FRÜHWIRTH, T., AND MEISTER, M., Eds. *W(C)LP '05: Proc. 19th Workshop on (Constraint) Logic Programming* (Universität Ulm, Germany, Feb. 2005), vol. 2005-01 of *Ulmer Informatik-Berichte*. `http://www.informatik.uni-ulm.de/epin/pw/11541`.

[45] WUILLE, P., SCHRIJVERS, T., AND DEMOEN, B. CCHR: the fastest CHR implementation, in C. In Djelloul et al. [9], pp. 123–137. `http://www.cs.kuleuven.be/~pieterw/CCHR/`.

A Installation guide and CD content

This chapter explains how to install the GTS Editor and its tools in the eclipse platform. A list of the files and folders on the accompanying *Compact Disc (CD)* is provided, as well.

A.1 CD content

- `src/` – contains the source files of the platform and the plug-ins
- `bin/` – contains the binary distribution of the projects that can be installed to eclipse
- `lib/` – contains the libraries that are needed to run the K.U. Leuven JCHR compiler
- `doc/` – contains the JavaDoc of all the projects
- `dist/` – contains a complete eclipse distribution with the platform already integrated
- `thesis/` – contains this thesis in *Portable Document Format (PDF)*

A.2 Installation Guide

The installation guide will cover the installation on Linux systems. The installation for Microsoft Windows based systems is similar, differences are mentioned. Several libraries and plug-ins must be available on the system and in eclipse in order to run the platform. The files that are found in the `lib/` folder on the CD must be added to the `CLASSPATH` system variable. Copy all the files to the directory `~/jchrlib/`. Now add the command

```
export CLASSPATH=~/jchrlib/antlr-2.7.5.jar:\
~/jchrlib/args4j-2.0.5.jar:\
~/jchrlib/KULeuven_JCHR.jar
```

to your `.bashrc` and log in to the system again. The libraries can now be accessed by java. On Microsoft Windows-based systems, the `CLASSPATH` can be changed in the system properties. Now add the `KULeuven_JCHR.jar` file to the root of the eclipse folder, so that it is accessible by the eclipse platform.

The dependencies for the GTS platform are GEF, JET, EMF, and TEF. The GEF, EMF, and JET run-time plug-ins can be installed via the update mechanism of eclipse. TEF can be obtained at [35]. Furthermore, the *Plug-in Development Environment (PDE)* is needed to create GTS projects. To install the platform with all its tools, copy all *Java Archive (JAR)* files from the `bin/` directory of the CD into the `plugins` folder of your eclipse installation.

When starting eclipse, the new project and file wizards for GTS are available and you can start creating GTS. Have fun!

For those who do not want to integrate the platform in their eclipse distribution, a distribution for Linux containing all the dependencies can be found in the `dist/` folder. Unpack the file `dist.tar` to your hard disk and start the `eclipse` executable. The K.U. Leuven JCHR compiler and its dependencies must, of course, still be added to the systems classpath.

B Introduction to category Theory

In this appendix, I want to give a short introduction to category theory, because it provides the theoretical basis for algebraic graph transformation systems. I want to present the basic definitions of categories and pushouts and how to construct them. Furthermore, I give some examples to ease the understanding of category theory when applied to GTS. This appendix is based on [18] which also gives a more detailed explanation of the topic.

B.1 Categories

A category $C = (Obj_C, Mor_C, \circ, id)$ is a tuple and its elements have the following meaning: Obj_C is a class of objects. For each tuple $A, B \in Mor_C$ there is a set $Mor_C(A, B)$ of morphisms. $\circ$ is the concatenation of morphisms in Mor_C. It has the form $Mor_C(B, D) \times Mor_C(A, B) \to Mor_C(A, D)$. $id_A \in Mor_C(A, A)$ is the identity morphism for every $A \in Obj_C$.

For morphisms $f \in Mor_C(A, B)$ the notation $f : A \to B$ is used. Furthermore, A is called the domain and B the codomain of f. An example of a category would be **Sets**. The objects are sets and functions $f : A \to B$ are morphisms. The concatenation $(g \circ f)(x)$ is given by $g(f(x))$ and the identity $id_A : A \to A : x \mapsto x$. The category **Graphs** can be constructed component-wise from **Sets** categories for the nodes and edges of a graph. The morphisms are graph morphisms (see 2.2). The concatenation is the concatenation of graph morphisms. The identities are the pairwise identities for nodes and edges.

In addition to that, I want to describe the construction of categories from already existing ones. The *product category* $C \times D$ from two categories C and D is given by

- $Obj_{C\times D} = Obj_C \times Obj_D$
- $Mor_{C\times D}((A, A'), (B, B')) = Mor_C(A, B) \times Mor_D(A', B)$
- for morphisms $f : A \to B$, $g : B \to C$, and $f' : A' \to B'$, $g' : B' \to C'$ we define $(g, g') \circ (f, f') = (g \circ f, g' \circ f')$
- $id_{(A,A')} = (id_A, id_{A'})$

Another way of creating a category is *(co)slicing*. A (co)slice category $C\backslash X$ ($X\backslash C$) of a given category C consists of the morphisms to (from) a distinguished object $X \in Obj_C$ as the Objects. The morphisms in the slice category are morphisms that connect the object morphisms. In more detail for slice categories:

- $Obj_{C\backslash X} = \{f : A \to X | A \in Obj_C, f \in Mor_C(A, X)\}$
- $Mor_{C\backslash X}(f : A \to X, g : B \to X) = m : A \to B | g \circ m = f$
- The concatenation for two morphisms is defined as in the category C
- $id_{f:A\to X} = id_A \in Mor_C$

Similarly for coslice categories:

- $Obj_{X\backslash C} = \{f : X \to A | A \in Obj_C, f \in Mor_C(X, A)\}$
- $Mor_{X\backslash C}(f : X \to A, g : X \to B) = m : A \to B | g \circ m = f$
- The concatenation for two morphisms for three objects is defined as in C
- $id_{f:A\to X} = id_A \in Mor_C$

An example for a slice category is **GraphsTG** which can be seen as **Graphs**$\backslash TG$. Each typed graph is presented by its typing morphism and the typed graph morphisms (section 2.2) are exactly the morphisms in the slice category.

The last category I would like to describe is the *dual category* C^{op}. It is given by reversing all morphisms, so $Mor_C(A, B) = Mor_{C^{op}}(B, A)$. For the concatenation, the order of execution is reversed in the dual category (because the morphisms are reversed, too). The identity morphisms are the same as in C.

B.2 Morphisms

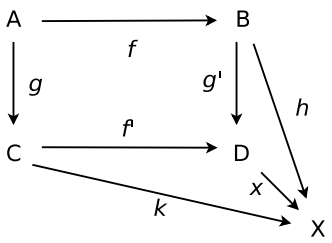
Morphisms in a category C can be of three types: mono-, epi-, and isomorphisms. A morphism $m : B \to C$ is a *monomorphism*, if for morphisms $f, g : A \to B \in Mor_C$, it holds that $m \circ f = m \circ g \Leftrightarrow f = g$. A morphism $e : A \to B$ is an *epimorphism*, if for morphisms $f, g : B \to C \in Mor_C$, it holds that $f \circ e = g \circ e \Leftrightarrow f = g$. Epi- and monomorphisms are dual notions in the respective dual category of each other. A morphism $i : A \to B$ is called an *isomorphism* if there exists a morphism $i^{-1} : B \to A$ such that $i^{-1} \circ i = id_A$. Two objects are called *isomorphic* to each other ($A \cong B$), if there exists an isomorphism between them. If i is an isomorphism, then it is also a mono- and an epimorphism and the inverse morphism of i is unique.

For example in the category **Sets** the mono- and epimorphisms are the injective and surjective mappings. Therefore, in **Graphs** and **GraphsTG** mono- and epimorphisms are those morphisms that are injective and surjective, respectively.

B.3 Pushouts

Intuitively, a pushout in category theory is the gluing of two objects along a common object. Here, I want to present the definitions and construction of such pushouts (POs) in specific categories.

Given $f : A \to B$ and $g : A \to C \in Mor_C$, then a *pushout* (D, f', g') is defined by a *pushout object* D and morphisms $f' : C \to D$, $g' : B \to D$ with $f' \circ g = g' \circ f$ such that the following universal property is fulfilled: For all objects $X \in Obj_C$ with morphisms $h : B \to X$ and $k : C \to X$ with $k \circ g = h \circ f$ there is a unique morphism $x : D \to X$ such that $x \circ g' = h$ and $x \circ f' = k$.



In **Sets**, the pushout over the moprhisms $f : A \to B$ and $g : A \to C$ can be constructed as follows: Define the relation $f(a) \sim g(a)$ for all $a \in A$ and let $\equiv$ be the reflexive, symmetric and transitive closure of $\sim$. $[x] = \{y \in B \uplus C | x \equiv y\}$ are the equivalence classes of the elements in the disjoint union of B and C. Define $D = \{[x] | x \in B \uplus C\}$ as the set of all equivalence classes of $B \uplus C$. Now the pushout can be constructed with $f'(c) = [c]$ for all $c \in C$ and $g'(b) = [b]$ for all $b \in B$. The proof that this is a valid pushout can be found in [18]. This pushout construction works analogously for **Graphs** and **GraphsTG** for the sets of nodes and edges of the pushout graph. The source and target functions of the pushout graph are uniquely determined by the pushout properties of the node set. To ease the understanding, I give a short example for the construction of a pushout in **Sets**. Given $A = \{a, b, c, d\}$, $B = \{1, 2, 3, 4\}$, and $C = \{5, 6, 7, 8\}$ with morphisms $f : A \to B, f(a) = 1, f(b) = f(c) = 2, f(d) = 3$ and $g : A \to C, g(a) = g(b) = 5, g(c) = 6, g(d) = 7$, the relation $\sim$ yields $1 \sim 5$, $2 \sim 5$, $2 \sim 6$ and $3 \sim 7$. Creating the transitive closure leads to $1 \equiv 2 \equiv 5 \equiv 6$ and $3 \equiv 7$. Therefore, $[1] = [2] = [5] = 6 = \{1, 2, 5, 6\}$, $[3] = [7] = \{3, 7\}$, $[4] = \{4\}$ and $[8] = \{8\}$. The resulting pushout object is then $D = \{[1], [3], [4], [8]\}$.

C Source Code

C.1 Example commands

This class implements the command for removing a node from a graph, while keeping the model consistent.

```
1  package org.uniulm.gts.model.commands;
2  public class NodeDeleteCommand extends Command {
3   private IGraphModel parent;
4   private IAbstractNode child;
5   private List<EdgeDeleteCommand> removedEdges;
6   private List<NodeDeleteCommand> removedTypedNodes;
7   private ITypeNode typeNode;
8
9   public NodeDeleteCommand(IAbstractNode child){
10   this.parent=child.getGraph();
11   this.child=child;
12   setLabel("Delete node");
13  }
14  public boolean canExecute() {
15   return parent != null && child!= null ;
16  }
17
18  //further commands are generated for deleting adjacent edges
19  public void execute() {
20   removedEdges=new LinkedList<EdgeDeleteCommand>();
21   removedTypedNodes=new LinkedList<NodeDeleteCommand>();
22   //if the deleted node is a type node,
23   //create delete commands for all typed nodes from it
24   if(child instanceof ITypeNode){
25    for(IAbstractNode n:((ITypeNode)child).getTypes()){
26     removedTypedNodes.add(new NodeDeleteCommand(n));
27    }
28   }
29   if(child instanceof INode){
30    typeNode = ((INode)child).getType();
31   }
32   //create delete commands for the edges connected to the node
33   for(IAbstractEdge e:child.getSrcEdg()){
34    removedEdges.add(new EdgeDeleteCommand(e));
35
36   }
37   for(IAbstractEdge e:child.getTgtEdg()){
38    removedEdges.add(new EdgeDeleteCommand(e));
39   }
40   //execute the commands
41   redo();
42  }
43
44  public void redo() {
45   //first the adjacent edges are removed
46   for(EdgeDeleteCommand e:removedEdges){
47    if(e.canExecute()) e.execute();
48   }
49   //then the typed nodes (which are now not connected
50   //to edges anymore)
51   if(child instanceof ITypeNode){
52    for(NodeDeleteCommand n:removedTypedNodes){
```

```
53        if(n.canExecute()) n.execute();
54       }
55      }
56      //If it is a typed node, remove its connection to
57      //its type node
58      if(child instanceof INode){
59       ((INode)child).setType(null);
60      }
61      //remove it from the graph
62      child.setGraph(null);
63     }
64
65     public void undo() {
66      //all the commands from redo are undone in reverse
67      //order
68      child.setGraph(parent);
69      if(child instanceof INode){
70       ((INode)child).setType(typeNode);
71      }
72      if(child instanceof ITypeNode){
73       for(NodeDeleteCommand n:removedTypedNodes){
74        n.undo();
75       }
76      }
77      for(EdgeDeleteCommand e:removedEdges){
78       e.undo();
79      }
80     }
81    }
```

Listing C.1: The NodeDeleteCommand class

C.2 GTS analysis tool extension point

This appendix contains the definition of the extension point and its interface for GTS analysis tools.

Listing C.2: Definition for the entries in the plugin.xml

```
1  <!ELEMENT extension (gtsanalysistool)>
2  <!ATTLIST extension
3   point CDATA #REQUIRED
4   id    CDATA #IMPLIED
5   name  CDATA #IMPLIED>
6  <!ELEMENT gtsanalysistool EMPTY>
7  <!ATTLIST gtsanalysistool
8   class CDATA #IMPLIED
9   name  CDATA #REQUIRED>
10   * class - The class that is loaded when invoking
11      the plug-in, should implement IGTSAnalysisTool
12   * name - the name displayed by plug-in selectors
```

Listing C.3: Source code of the interface

```
1  package org.uniulm.gts.diagnostictools;
2  /**
3   * This interface is used for tools that analyze
4   * graph transformation systems. It lets set model,
5   * command stack and file being worked on, as well
6   * as add listeners. Furthermore there are methods
7   * that lets you set set active graphs or graph.
8   * elements for display purposes
9   */
10 public interface IGTSAnalysisTool{
11  /**
```

```
12     * sets the IGraphTransformationSystem model for this
13     * diagnostictool
14     * @param model the GTS model used
15     */
16    public void setModel(IGraphTranformationSystem model);
17    /**
18     * sets the file which contains the model that is
19     * used by the editor
20     * @param file the file used
21     */
22    public void setFile(IFile file);
23    /**
24     * add a listener for this tool
25     * @param a the listener to be added
26     */
27    public void addChangeListener(IGTSAnalysisToolListener a);
28    /**
29     * removes the given listener from this tool
30     * @param a the listener to be removed
31     */
32    public void removeChangeListener(IGTSAnalysisToolListener a);
33    /**
34     * remove all listeners
35     */
36    public void removeAllListeners();
37    /**
38     * starts the tool, this method should start views,
39     * editors and so on. When calling a tool, make sure
40     * to call all the set methods first
41     */
42    public void runAnalysis();
43    /**
44     * thats the commandstack that should be used when
45     * modifying the model
46     * @param cs the commandstack to be used
47     */
48    public void setCommandStack(CommandStack cs);
49    /**
50     * returns the currently selected Graph
51     * @return currently selected graph
52     */
53    public IGraphModel getActiveGraph();
54    /**
55     * return the currently selected elements of the
56     * currently active graph in this tool
57     * @return currently selected elements
58     */
59    public List<GTSElement> getActiveSelection();
60    /**
61     * Set the selected elements in the given graph
62     * @return currently selected elements
63     */
64    public void setActiveSelection(IGraphModel m, List<GTSElement> l);
65    /**
66     * activate the given graph in this tool
67     * @param m the graph to be shown
68     */
69    public void setActiveGraph(IGraphModel m);
70  }
```

C.3 CHR analysis tool extension point

This appendix contains the definition of the extension point and its interface for CHR analysis tools.

Listing C.4: Definition for the entries in the plugin.xml

```
<!ELEMENT extension (diagnostictool)>
<!ATTLIST extension
 point CDATA #REQUIRED
 id    CDATA #IMPLIED
 name  CDATA #IMPLIED>
<!ELEMENT chranalysistool EMPTY>
<!ATTLIST chranalysistool
 class CDATA #IMPLIED
 name  CDATA #REQUIRED>
  * class - The class that is loaded when invoking
    the plugin, should implement IChrAnalysisTool
  * name - the name displayed by plugin selectors
```

Listing C.5: Source code of the interface

```
import java.util.List;
import org.eclipse.core.resources.IFile;
/**
 * IChrAnalyzer lets you start analysis of jchr files. It provides the
     additional capability of adding inputs to the
 * file for additional analysis. Furthermore listeners can be added.
 * @author Mathias Wasserthal
 *
 */
public interface IChrAnalysisTool {
        /**
         * Set the file which contains the jchr source
         * @param file
         */
        public void setFile(IFile file);
        /**
         * Command used to start the analysis.
         */
        public void runChrAnalysis();
        /**
         * Add a listener that is notified about selected constraints
         * @param listener
         */
        public void addChrListener(IChrListener listener);
        /**
         * Removes the given listener
         * @param listener
         */
        public void removeChrListener(IChrListener listener);
        /**
         * method to add input to the chr analysis. Input is given as alist
              of {@link ConstraintNotification}s,
         * containing the constraint name, its arity and the contents of
             the variables.
         * @param list
         */
        public void addInput(List<ConstraintNotification> list);
}
```

C.4 JET Template for JCHR code generation

This appendix lists the JET template for generating JCHR source files from a GTS model. Listing C.6 shows the generation of plain JCHR source. This is used by the CHR based analysis tool (section 4.6). This template is trimmed for readability. Therefore, the generated source code is not formattet very nice.

Listing C.6: JET temmplate for JCHR code generation.

```
package <c:get select="$jchrpackage"/>;
```

```
2  import runtime.*;
3  import runtime.primitive.*;
4  import util.ArithmeticsUtils;
5  public handler <c:get select="lowercaseFirst(removeWhitespace($jchrhandler))"/> {
6  <f:indent text="        ">
7  public constraint
8   <c:iterate select="$gts/typeGraph/nodes" var="typenode" delimiter=", ">
9    n_<c:get select="$typenode/@iD"/>
10   (<c:get select="$variabletype"/>,int)
11  </c:iterate>
12  <c:if test="count($gts/typeGraph/nodes)>0">,</c:if>
13  <c:iterate select="$gts/typeGraph/edges" var="typeedge" delimiter=", ">
14   e_<c:get select="$typeedge/@iD"/>
15   (int,<c:get select="$variabletype"/>,<c:get select="$variabletype"/>)
16  </c:iterate>
17  <c:if test="count($gts/typeGraph/nodes)>0">,</c:if>
18  empty_head();
19  rules {
20  <f:indent text="        ">
21   <c:if test="count($gts/transformations/nodes[@trans='R']/@iD)>0">
22    local <c:get select="$variabletype"/>
23    <c:iterate select="distinct-values($gts/transformations/nodes[@trans='R']/@iD)" var
          ="ids" delimiter=", ">
24     NN_<c:get select="$ids"/>
25    </c:iterate>;
26   </c:if>
27   <c:iterate select="reorder($gts/transformations,$order)" var="trans">
28    rule_<c:get select="translate($trans/@id,' ','_')"/> @
29    <c:if test="count($trans/edges[@trans='L' or @trans='K']|$trans/nodes[@trans='L' or
          @trans='K'])=0">
30     empty_head()
31    </c:if>
32    <c:iterate select="$trans/nodes[@trans='L']" var="node" delimiter=",">
33     n_<c:get select="$node/type/@iD"/>
34     ( ON_<c:get select="translate($node/@iD,' ','_')"/>
35     , <c:get select="count($node/srcEdg[@trans='L'])+count($node/tgtEdg[@trans='L'])"
          />)
36    </c:iterate>
37    <c:if test="count($trans/nodes[@trans='L'])>0 and count($trans/nodes[@trans='K'])>0
          ">
38     ,
39    </c:if>
40    <c:iterate select="$trans/nodes[@trans='K']" var="node"      delimiter=",">
41     n_<c:get select="$node/type/@iD"/>
42     ( ON_<c:get select="translate($node/@iD,' ','_')"/>
43     , NC_<c:get select="$node/@iD"/> )
44    </c:iterate>
45    <c:if test="count($trans/nodes[@trans='K']|$trans/nodes[@trans='L'])>0 and count(
          $trans/edges[@trans='L'])>0">
46     ,
47    </c:if>
48    <c:iterate select="$trans/edges[@trans='L']" var="edge" delimiter=",">
49     e_<c:get select="$edge/type/@iD"/>
50     (0, ON_<c:get select="translate($edge/src/@iD,' ','_')"/>
51     , ON_<c:get select="translate($edge/tgt/@iD,' ','_')"/>)
52    </c:iterate>
53    <c:if test="count($trans/nodes[@trans='K']|$trans/nodes[@trans='L']|$trans/edges[
          @trans='L'])>0 and count($trans/edges[@trans='K'])>0">
54     ,
55    </c:if>
56    <c:setVariable var="delCnt" select="0"/>
57    <c:iterate select="$trans/edges[@trans='K']" var="edge" delimiter=",">
58     e_<c:get select="$edge/type/@iD"/>
59     (DEL_<c:get select="$delCnt"/>
60     , ON_<c:get select="translate($edge/src/@iD,' ','_')"/>
61     , ON_<c:get select="translate($edge/tgt/@iD,' ','_')"/>)
62    </c:iterate>
63    <=>
64    <c:iterate select="$trans/nodes[@trans='K']" var="node" delimiter=",">
65     <c:setVariable var="diff"
66      select="0-(
67         count($node/srcEdg[@trans='L']|$node/srcEdg[@trans='K'])+
68                 count($node/tgtEdg[@trans='L']|$node/tgtEdg[@trans='K']))+
69                 (count($node/srcEdg[@trans='R']|$node/srcEdg[@trans='K'])+
70                 count($node/tgtEdg[@trans='R']|$node/tgtEdg[@trans='K']))"/>
71     <c:if test="$diff<0">
72      n_<c:get select="$node/type/@iD"/>
73      ( ON_<c:get select="translate($node/@iD,' ','_')"/>
74      , ArithmeticsUtils.sub(NC_<c:get select="$node/@iD"/>,<c:get select="0-$diff"/>))
75     </c:if>
```

```
76      <c:if test="$diff>0">
77       n_<c:get select="$node/type/@iD"/>
78       ( ON_<c:get select="translate($node/@iD,' ','_')"/>
79       , ArithmeticsUtils.add(NC_<c:get select="$node/@iD"/>,<c:get select="$diff"/>))
80      </c:if>
81      <c:if test="$diff=0">
82       n_<c:get select="$node/type/@iD"/>
83       ( ON_<c:get select="translate($node/@iD,' ','_')"/>
84       , NC_<c:get select="$node/@iD"/>)
85      </c:if>
86     </c:iterate>
87     <c:if test="count($trans/nodes[@trans='K'])>0 and count($trans/nodes[@trans='R'])>0
           ">
88      ,
89     </c:if>
90     <c:iterate select="$trans/nodes[@trans='R']" var="node" delimiter=",">
91      n_<c:get select="$node/type/@iD"/>
92      ( NN_<c:get select="translate($node/@iD,' ','_')"/>
93      ,<c:get select="count($node/srcEdg[@trans='R'])+count($node/tgtEdg[@trans='R'])"
            />)
94     </c:iterate>
95     <c:if test="count($trans/nodes[@trans='R' or @trans='K'])>0 and count($trans/edges[
           @trans='R'])>0">
96      ,
97     </c:if>
98     <c:iterate select="$trans/edges[@trans='R']" var="edge" delimiter=",">
99      e_<c:get select="$edge/type/@iD"/>
100     (0, <c:if test="$edge/src/@trans='R'">NN_</c:if>
101         <c:if test="$edge/src/@trans='K'">ON_</c:if>
102         <c:get select="translate($edge/src/@iD,' ','_')"/>
103      , <c:if test="$edge/tgt/@trans='R'">NN_</c:if>
104        <c:if test="$edge/tgt/@trans='K'">ON_</c:if>
105        <c:get select="translate($edge/tgt/@iD,' ','_')"/>)
106    </c:iterate>
107    <c:if test="count($trans/nodes[@trans='K']|$trans/nodes[@trans='R']|$trans/edges[
           @trans='R'])>0 and count($trans/edges[@trans='K'])>0">
108     ,
109    </c:if>
110    <c:setVariable var="delCnt" select="0"/>
111    <c:iterate select="$trans/edges[@trans='K']" var="edge" delimiter=",">
112     e_<c:get select="$edge/type/@iD"/>
113     (DEL_<c:get select="$delCnt"/>
114      , ON_<c:get select="translate($edge/src/@iD,' ','_')"/>
115      , ON_<c:get select="translate($edge/tgt/@iD,' ','_')"/>)
116    </c:iterate>
117    <c:if test="count($trans/edges[@trans='R' or @trans='K']|
118                      $trans/nodes[@trans='R' or @trans='K'])>0
119           and count($trans/edges[@trans='L' or @trans='K']|
120                      $trans/nodes[@trans='L' or @trans='K'])=0">
121     ,
122    </c:if>
123    <c:if test="count($trans/edges[@trans='L' or @trans='K']|
124                      $trans/nodes[@trans='L' or @trans='K'])=0">
125     empty_head()
126    </c:if>
127    <c:if test="count($trans/edges[@trans='L' or @trans='K']|
128                      $trans/nodes[@trans='L' or @trans='K'])>0
129           and count($trans/edges[@trans='R' or @trans='K']|
130                      $trans/nodes[@trans='R' or @trans='K'])=0">
131     true
132    </c:if>
133    .
134   </c:iterate>
135  </f:indent>
136  }
137 </f:indent>
138 }
```

C.5 CHR editor validity and update algorithm

This Appendix lists the algorithm for updating a rule graph from a given CHR model. The first part checks the validity of the encoding, the second part updates the rule graph model.

Listing C.7: Algorithm for checking the validity of rules and applying the corresponding updates to the rule graph.

```
protected boolean updateTransformationModel(CHR chr){
  notify=false;
  //first the type graph and rule graph model are put into hash maps
  //hashmaps for typenodes and typeedges
  HashMap<String,IAbstractNode> typenodes =
   new HashMap<String,IAbstractNode>();
  HashMap<String,IAbstractEdge> typeedges =
   new HashMap<String,IAbstractEdge>();
  ITypeGraphModel tg=((ITypedGraphModel)model).getTypeGraph();
  for(IAbstractEdge e:tg.getEdges())
   if(e instanceof ITypeEdge)
    typeedges.put(((ITypeEdge)e).getID(), e);
  for(IAbstractNode n:tg.getNodes())
   if(n instanceof ITypeNode)
    typenodes.put(((ITypeNode)n).getID(), n);

  IGraphTransformation gt = (IGraphTransformation)model;
  //hashmaps for nodes and egges of the model
  //nodes
  HashMap<String,ITransformNode> nodesL = new HashMap<String,ITransformNode>();
  HashMap<String,ITransformNode> nodesK = new HashMap<String,ITransformNode>();
  HashMap<String,ITransformNode> nodesR = new HashMap<String,ITransformNode>();
  HashMap<String,ITypeNode> nodeTypes = new HashMap<String,ITypeNode>();
  HashMap<String,TransformElementType> nodeTrans = new HashMap<String,
      TransformElementType>();
  HashMap<String,Integer> nodeDegreeExpr=new HashMap<String,Integer>();
  for(IAbstractNode tn:gt.getNodes()){
   if(tn instanceof ITransformNode)
    switch(((ITransformNode)tn).getTrans().getValue()){
    case TransformElementType.K_VALUE:nodesK.put(tn.getID(),(ITransformNode)tn); break
        ;
    case TransformElementType.L_VALUE:nodesL.put(tn.getID(),(ITransformNode)tn); break
        ;
    case TransformElementType.R_VALUE:nodesR.put(tn.getID(),(ITransformNode)tn); break
        ;
    }
    nodeTypes.put(((ITransformNode)tn).getID(),((ITransformNode)tn).getType());
    nodeTrans.put(((ITransformNode)tn).getID(),((ITransformNode)tn).getTrans());
  }
  //edges
  MultiHashMap3<ITransformNode,ITransformNode,ITypeEdge,LinkedList<ITransformEdge>>
      edgesL = new MultiHashMap3<ITransformNode,ITransformNode,ITypeEdge,LinkedList<
      ITransformEdge>>();
  MultiHashMap3<ITransformNode,ITransformNode,ITypeEdge,LinkedList<ITransformEdge>>
      edgesK = new MultiHashMap3<ITransformNode,ITransformNode,ITypeEdge,LinkedList<
      ITransformEdge>>();
  MultiHashMap3<ITransformNode,ITransformNode,ITypeEdge,LinkedList<ITransformEdge>>
      edgesR = new MultiHashMap3<ITransformNode,ITransformNode,ITypeEdge,LinkedList<
      ITransformEdge>>();

  HashMap<ITransformEdge,ITypeEdge> edgeTypes = new HashMap<ITransformEdge,ITypeEdge
      >();
  HashMap<ITransformEdge,String> edgeSrc = new HashMap<ITransformEdge,String>();
  HashMap<ITransformEdge,String> edgeTgt = new HashMap<ITransformEdge,String>();
  HashMap<ITransformEdge,TransformElementType> edgeTrans = new HashMap<ITransformEdge,
      TransformElementType>();

  for(IAbstractEdge e:gt.getEdges()){
   if(e instanceof ITransformEdge){
    ITransformNode src=(ITransformNode)e.getSrc(), tgt=(ITransformNode)e.getTgt();
    ITypeEdge type = ((ITransformEdge)e).getType();
    switch(((ITransformEdge)e).getTrans().getValue()){
    case TransformElementType.K_VALUE:
     if(!edgesK.containsKey(src,tgt,type))
      edgesK.put(src,tgt,type,new LinkedList<ITransformEdge>());
     edgesK.get(src,tgt,type).add((ITransformEdge)e);
     break;
    case TransformElementType.L_VALUE:
     if(!edgesL.containsKey(src,tgt,type))
      edgesL.put(src,tgt,type,new LinkedList<ITransformEdge>());
     edgesL.get(src,tgt,type).add((ITransformEdge)e);
     break;
    case TransformElementType.R_VALUE:
     if(!edgesR.containsKey(src,tgt,type))
      edgesR.put(src,tgt,type,new LinkedList<ITransformEdge>());
     edgesR.get(src,tgt,type).add((ITransformEdge)e);
```

```
65          break;
66         }
67         edgeSrc.put((ITransformEdge)e,e.getSrc().getID());
68         edgeTgt.put((ITransformEdge)e,e.getTgt().getID());
69         edgeTypes.put((ITransformEdge)e,((ITransformEdge)e).getType());
70        }
71       }
72
73       if(chr.getRule().size()!=1) return false;
74       //CHR program is encoded in hashmaps
75       HashMap<Constraint,String> errors=new HashMap<Constraint,String>();
76
77       //save all constraints in a set
78       Set<Constraint> constraintSet = new HashSet<Constraint>();
79       for(Constraint c:((PropRule)chr.getRule().get(0)).getHead()) constraintSet.add(c);
80       for(Constraint c:((PropRule)chr.getRule().get(0)).getBody()) constraintSet.add(c);
81       HashMap<String,Constraint> nodeConstraintsL = new HashMap<String,Constraint>();
82       HashMap<String,Constraint> nodeConstraintsK = new HashMap<String,Constraint>();
83       HashMap<String,Constraint> nodeConstraintsR = new HashMap<String,Constraint>();
84       HashMap<String,Integer> nodeEdgeCount=new HashMap<String,Integer>();
85
86        //MultiHashMaps for the edge constraints
87       MultiHashMap3<String,String,String,List<Constraint>> edgeConstraintsL = new
             MultiHashMap3<String,String,String,List<Constraint>>();
88       MultiHashMap3<String,String,String,List<Constraint>> edgeConstraintsK = new
             MultiHashMap3<String,String,String,List<Constraint>>();
89       MultiHashMap3<String,String,String,List<Constraint>> edgeConstraintsR = new
             MultiHashMap3<String,String,String,List<Constraint>>();
90       Set<String> edgeDelVars=new HashSet<String>();
91       //put the constraints into the groups L,R and K
92       //process headconstraints first
93       for(Constraint c:((PropRule)chr.getRule().get(0)).getHead()){
94        //filter errors in constraints (e.g.  multiple identifiers)
95        //node constraints
96        if((simple && c.getVariables().size()==1
97          && c.getVariables().get(0) instanceof Literal
98          || !simple && c.getVariables().size()==2
99          && c.getVariables().get(0) instanceof Literal)
100         && c.getType()!=null
101         && typenodes.containsKey(c.getType().getName())){
102        String id=((Literal)c.getVariables().get(0)).getValue();
103        if(!nodeConstraintsL.containsKey(id)){
104         nodeConstraintsL.put(id, c);
105         if(!nodeEdgeCount.containsKey(id))
106          nodeEdgeCount.put(id, 0);
107         constraintSet.remove(c);
108        }else{
109         errors.put(c, "duplicate node identifier");
110         continue;
111        }
112       //edge constraints
113       }else if(simple && c.getVariables().size()==2
114         && c.getVariables().get(0) instanceof Literal
115         && c.getVariables().get(1) instanceof Literal
116         && c.getType()!=null
117         && typeedges.containsKey(c.getType().getName())
118         || !simple && c.getVariables().size()==3
119         && c.getVariables().get(0) instanceof Literal
120         && c.getVariables().get(1) instanceof Literal
121         && c.getVariables().get(2) instanceof Literal && c.getType()!=null
122         && typeedges.containsKey(c.getType().getName())){
123        String src,tgt;
124        int shift=simple?0:1; //different positions when in simple mode
125        src=((Literal)c.getVariables().get(0+shift)).getValue();
126        tgt=((Literal)c.getVariables().get(1+shift)).getValue();
127        String type=c.getType().getName();
128        if(!simple && Character.isUpperCase(((Literal)c.getVariables().get(0)).getValue().
               charAt(0))){
129         if(!edgeConstraintsL.containsKey(src,tgt,type))
130          edgeConstraintsL.put(src,tgt,type, new LinkedList<Constraint>());
131         edgeConstraintsL.get(src,tgt,type).add(c);
132        }else{
133         if(!edgeConstraintsL.containsKey(src,tgt,type))
134          edgeConstraintsL.put(src,tgt,type, new LinkedList<Constraint>());
135         edgeConstraintsL.get(src,tgt,type).add(c);
136        }
137        if(nodeEdgeCount.containsKey(src)) nodeEdgeCount.put(src, nodeEdgeCount.get(src)
               -1);
138        else nodeEdgeCount.put(src, -1);
```

```
139       if(nodeEdgeCount.containsKey(tgt)) nodeEdgeCount.put(tgt, nodeEdgeCount.get(tgt)
              -1);
140       else nodeEdgeCount.put(tgt, -1);
141       constraintSet.remove(c);
142      }else{
143       errors.put(c, "Invalid Constraint");
144       continue;
145      }
146     }
147
148     //process body constraints
149     for(Constraint c:((PropRule)chr.getRule().get(0)).getBody()){
150      //filter errors (wrong variable names for attributes, duplicate entries, occurence
             of variables that should not be there,
151      //    multiple deletion variables)
152      //node constraints
153      if(simple && c.getVariables().size()==1
154       && c.getVariables().get(0) instanceof Literal
155       && c.getType()!=null
156       && typenodes.containsKey(c.getType().getName())
157       || !simple && c.getVariables().size()==2
158       && c.getVariables().get(0) instanceof Literal
159       && c.getType()!=null && typenodes.containsKey(c.getType().getName())){
160       String id=((Literal)c.getVariables().get(0)).getValue();
161       if( nodeConstraintsL.containsKey(id)){
162        if(!simple){
163         List<Literal> lL=new LinkedList<Literal>(), lR=new LinkedList<Literal>();
164         getLiteral(nodeConstraintsL.get(id).getVariables().get(1),lL);
165         getLiteral(c.getVariables().get(1),lR);
166         if(lR.size()!=1 || lL.size()!=1 || !lR.get(0).getValue().equals(lL.get(0).
                 getValue())){
167          errors.put(c, "Constraints in head and body must share the same Variable in the
                  second attribute");
168          continue;
169         }
170         nodeDegreeExpr.put(id, evalExpression(c.getVariables().get(1)));
171        }
172        nodeConstraintsK.put(id, nodeConstraintsL.remove(id));
173        constraintSet.remove(c);
174       }else{
175        if(nodeConstraintsR.containsKey(id)){
176         errors.put(c, "Duplicate node entry in body");
177         continue;
178        }
179        if(!simple){
180         List<Literal> lit=new LinkedList<Literal>();
181         getLiteral(c.getVariables().get(1),lit);
182         if(lit.size()!=0){
183          errors.put(c,"Second attribute must not contain variables");
184          continue;
185         }
186        }
187        nodeConstraintsR.put(id, c);
188        if(!nodeEdgeCount.containsKey(id))
189         nodeEdgeCount.put(id, 0);
190        constraintSet.remove(c);
191       }
192      }
193      //edge constraints
194      else if(simple && c.getVariables().size()==2 && c.getVariables().get(0) instanceof
             Literal
195       && c.getVariables().get(1) instanceof Literal && c.getType()!=null
196       && typeedges.containsKey(c.getType().getName())
197       || !simple && c.getVariables().size()==3 && c.getVariables().get(0) instanceof
              Literal
198       && c.getVariables().get(1) instanceof Literal && c.getVariables().get(2)
             instanceof Literal && c.getType()!=null
199       && typeedges.containsKey(c.getType().getName())){
200       int shift=simple?0:1;
201       String src=((Literal)c.getVariables().get(0+shift)).getValue();
202       String tgt=((Literal)c.getVariables().get(1+shift)).getValue();
203       String type=c.getType().getName();
204
205       if(edgeConstraintsL.containsKey(src,tgt,type) && edgeConstraintsL.get(src,tgt,type
             ).size()>0){
206
207        int edg=0;
208        if(!simple){
209
210         String del=((Literal)c.getVariables().get(0)).getValue();
```

```
211         if(edgeDelVars.contains(del)){
212          errors.put(c, "Delete variable was used already");
213          continue;
214         }
215         if(del.length()==0 || Character.isLowerCase(del.charAt(0))){
216          errors.put(c, "First attribute must not be a ground term");
217          continue;
218         }
219         edg=-1;
220         int cnt=-1;
221         for(Constraint ch:edgeConstraintsL.get(src,tgt,type)){
222          cnt++;
223          if(((Literal)ch.getVariables().get(0)).getValue().equals(del)){
224           edg=cnt;
225          }
226         }
227         if(edg==-1){
228          errors.put(c, "There is no edge constraint in the head with the same del
                 variable");
229          continue;
230         }
231         edgeDelVars.add(del);
232        }
233        if(!edgeConstraintsK.containsKey(src,tgt,type)){
234         edgeConstraintsK.put(src,tgt,type,new LinkedList<Constraint>());
235        }
236        edgeConstraintsK.get(src,tgt,type).add(edgeConstraintsL.get(src,tgt,type).remove(
             edg));
237        if(nodeEdgeCount.containsKey(src)) nodeEdgeCount.put(src, nodeEdgeCount.get(src)
             +1);
238        else nodeEdgeCount.put(src, 1);
239        if(nodeEdgeCount.containsKey(tgt)) nodeEdgeCount.put(tgt, nodeEdgeCount.get(tgt)
             +1);
240        else nodeEdgeCount.put(tgt, 1);
241        constraintSet.remove(c);
242       }else{
243        if(!edgeConstraintsR.containsKey(src,tgt,type))
244         edgeConstraintsR.put(src,tgt,type, new LinkedList<Constraint>());
245        edgeConstraintsR.get(src,tgt,type).add(c);
246        if(nodeEdgeCount.containsKey(src)) nodeEdgeCount.put(src, nodeEdgeCount.get(src)
             +1);
247        else nodeEdgeCount.put(src, +1);
248        if(nodeEdgeCount.containsKey(tgt)) nodeEdgeCount.put(tgt, nodeEdgeCount.get(tgt)
             +1);
249        else nodeEdgeCount.put(tgt, +1);
250        constraintSet.remove(c);
251       }
252      }else{
253       errors.put(c, "Invalid Constraint");
254       continue;
255      }
256     }
257
258     //in complex mode edge constraints must not contain variables if they are in L\K or
          R\K
259     if(!simple && constraintSet.size()==0)
260      for(Constraint c:nodeConstraintsL.values()){
261       List<Literal> lit=new LinkedList<Literal>();
262       getLiteral(c.getVariables().get(1),lit);
263       if(lit.size()!=0){
264        errors.put(c,"Second attribute must not contain variable");
265        constraintSet.add(c);
266        continue;
267       }
268      }
269
270     //check if edges are connected to the correct nodes (according to their type edge),
          and the correct group  (L,K or R)
271     for(int i=0;i<3;i++){
272      if(constraintSet.size()>0) break;
273      HashMap<String,HashMap<String,HashMap<String,List<Constraint>>>> map=null;
274      switch(i){
275      case 0: map=edgeConstraintsL.getMap(); break;
276      case 1: map=edgeConstraintsK.getMap();break;
277      case 2: map=edgeConstraintsR.getMap();break;
278      }
279
280      for(String src:map.keySet())
281       for(String tgt:map.get(src).keySet())
282        for(String type:map.get(src).get(tgt).keySet())
```

```
283         for(Constraint c:map.get(src).get(tgt).get(type)){
284          //check to which group it belongs (L,R,K)
285          Constraint srcc=null, tgtc=null;
286          if(i==0 && nodeConstraintsL.containsKey(src))
287           srcc=nodeConstraintsL.get(src);
288          if(nodeConstraintsK.containsKey(src))
289           srcc=nodeConstraintsK.get(src);
290          if(i==2 && nodeConstraintsR.containsKey(src))
291           srcc=nodeConstraintsR.get(src);
292          if(i==0 && nodeConstraintsL.containsKey(tgt))
293           tgtc=nodeConstraintsL.get(tgt);
294          if(nodeConstraintsK.containsKey(tgt))
295           tgtc=nodeConstraintsK.get(tgt);
296          if(i==2 && nodeConstraintsR.containsKey(tgt))
297           tgtc=nodeConstraintsR.get(tgt);
298          if(srcc==null ){
299           errors.put(c,"could not find a valid source node constraint");
300           constraintSet.add(c);
301           continue;
302          }
303          if(tgtc==null ){
304           errors.put(c,"could not find a valid target node constraint");
305           constraintSet.add(c);
306           continue;
307          }
308          ITypeEdge te = (ITypeEdge)typeedges.get(c.getType().getName());
309          ITypeNode tnsrc = (ITypeNode)typenodes.get(srcc.getType().getName());
310          ITypeNode tntgt = (ITypeNode)typenodes.get(tgtc.getType().getName());
311          if(tnsrc!=te.getSrc()){
312           errors.put(c, "Type of source node incorrect");
313           constraintSet.add(c);
314           continue;
315          }
316          if(tntgt!=te.getTgt()){
317           errors.put(c, "Type of target node incorrect");
318           constraintSet.add(c);
319           continue;
320          }
321
322         }
323     }
324     //complex mode: check if the del and count attribute is correctly set ground or
            variable
325     if(!simple && constraintSet.size()==0){
326      //check host nodes (here the attribute must be ground)
327      for(int i=0;i<2;i++){
328       //check edges (del)
329       MultiHashMap3<String,String,String,List<Constraint>> map=null;
330       switch(i){
331       case 0: map=edgeConstraintsL; break;
332       case 1: map=edgeConstraintsR;break;
333       }
334       for(List<Constraint> l:map.values())
335        for(Constraint c:l){
336         String del=((Literal)c.getVariables().get(0)).getValue();
337         if(del.length()==0 || Character.isUpperCase(del.charAt(0))){
338          errors.put(c,"Deletion argument must be ground");
339          constraintSet.add(c);
340          continue;
341         }
342
343        }
344
345       //check nodes (count)
346       HashMap<String,Constraint> nmap=null;
347       switch(i){
348       case 0: nmap=nodeConstraintsL;break;
349       case 1: nmap=nodeConstraintsR;break;
350       }
351       for(Constraint c:nmap.values()){
352        if(c.getVariables().get(1) instanceof Numeral){
353         int cnt=((Numeral)c.getVariables().get(1)).getValue();
354         String id=((Literal)c.getVariables().get(0)).getValue();
355
356         if(!nodeEdgeCount.containsKey(id) || Math.abs(nodeEdgeCount.get(id))!=cnt){
357          errors.put(c, "second attribute must be a number and represent the number of
                 edges on this node");
358          constraintSet.add(c);
359          continue;
360         }
```

```
361         }
362        }
363       }
364       //check the edge constraints that are kept (in group K), here the del attribute
              must be variable
365       for(List<Constraint> l:edgeConstraintsK.values()){
366        for(Constraint c:l){
367         String del=((Literal)c.getVariables().get(0)).getValue();
368         if(del.length()==0 || Character.isLowerCase(del.charAt(0))){
369          errors.put(c,"Deletion argument must be a variable");
370          constraintSet.add(c);
371          continue;
372         }
373        }
374       }
375       //check the node constraints that are kept (in group K), here the count attribute
              must contain a variable
376       for(Constraint c:nodeConstraintsK.values()){
377
378        int cnt=nodeDegreeExpr.get(((Literal)c.getVariables().get(0)).getValue())-
              evalExpression(c.getVariables().get(1));
379        String id=((Literal)c.getVariables().get(0)).getValue();
380        int cnt2=0;//=1000;
381        if(nodeEdgeCount.containsKey(id)) cnt2=nodeEdgeCount.get(id);
382        if(!nodeEdgeCount.containsKey(id) || cnt2!=cnt){
383         errors.put(c, "second attribute must be a Expression with one Variable and
              represent the different number of edges");
384         constraintSet.add(c);
385         continue;
386
387        }
388       }
389
390
391
392      }
393
394      //remove all old annotations
395      IAnnotationModel am=this.getSourceViewer().getAnnotationModel();
396      for(Annotation a:alTrans){
397       am.removeAnnotation(a);
398
399      }
400
401      //if there are still unprocessed Constraints print error annotations
402      if(constraintSet.size()>0){
403       alTrans=new LinkedList<Annotation>();
404       IModelCreatingContext mcc= this.getLastModelCreatingContext();
405
406       for(Constraint c:constraintSet){
407        Position p=mcc.getTreeNodeForObject(c).getPosition();
408        Annotation a;
409        if(errors.containsKey(c))
410         a=new ErrorAnnotation(errors.get(c)+" ["+mcc.getTreeNodeForObject(c).getNodeText
              ()+"]");
411        else
412         a=new ErrorAnnotation("Error in Constraint "+mcc.getTreeNodeForObject(c).
              getNodeText());
413        am.addAnnotation((org.eclipse.jface.text.source.Annotation) a,p);
414        alTrans.add(a);
415       }
416
417       return false;
418      }
419
420
421      //modify, add or delete nodes and edges
422
423      //new and modified nodes
424      HashMap<String,ITransformNode> newNodes = new HashMap<String,ITransformNode>();
425
426      List<ITransformNode> nodeModify = new LinkedList<ITransformNode>();
427      List<ITransformNode> nodeAdd=new LinkedList<ITransformNode>();
428      List<ITransformNode> nodeDelete=new LinkedList<ITransformNode>();
429      List<ITransformEdge> edgeModify = new LinkedList<ITransformEdge>();
430      List<ITransformEdge> edgeAdd=new LinkedList<ITransformEdge>();
431      List<ITransformEdge> edgeDelete=new LinkedList<ITransformEdge>();
432      //adjust the group of the nodes
433      for(TransformElementType i:TransformElementType.values()){
434       HashMap<String,Constraint> cons=null;
```

```
435      switch(i.getValue()){
436      case TransformElementType.K_VALUE:{cons=nodeConstraintsK;break;}
437      case TransformElementType.L_VALUE:{cons=nodeConstraintsL;break;}
438      case TransformElementType.R_VALUE:{cons=nodeConstraintsR;break;}
439      }
440      for(String id: cons.keySet()){
441       if( nodesR.containsKey(id)){
442        if(!i.equals(TransformElementType.R)){
443         nodeModify.add(nodesR.get(id));
444        }
445
446        nodeTrans.put(id, i);
447        newNodes.put(id, nodesR.get(id));
448        nodesR.remove(id);
449       }else
450       if(nodesK.containsKey(id)){
451        if(!i.equals(TransformElementType.K))
452         nodeModify.add(nodesK.get(id));
453        newNodes.put(id, nodesK.get(id));
454        nodeTrans.put(id, i);
455        nodesK.remove(id);
456       }else
457       if(nodesL.containsKey(id)){
458        if(!i.equals(TransformElementType.L))
459         nodeModify.add(nodesL.get(id));
460        newNodes.put(id, nodesL.get(id));
461        nodeTrans.put(id, i);
462        nodesL.remove(id);
463       }else{
464        //new nodes
465        ITransformNode nn=new ITransformNodeImpl();
466        nn.setID(((Literal)cons.get(id).getVariables().get(0)).getValue());
467        newNodes.put(nn.getID(), nn);
468        nodeAdd.add(nn);
469        nodeTypes.put(nn.getID(), (ITypeNode)typenodes.get(cons.get(id).getType().getName
              ()));
470        nodeTrans.put(nn.getID(), i);
471       }
472      }
473     }
474     //removed nodes
475     nodeDelete.addAll(nodesR.values());
476     nodeDelete.addAll(nodesL.values());
477     nodeDelete.addAll(nodesK.values());
478
479     //new and modified edges
480     //adjust group of edge
481     for(TransformElementType i:TransformElementType.values()){
482      MultiHashMap3<String,String,String,List<Constraint>> cons=null;
483      switch(i.getValue()){
484      case TransformElementType.K_VALUE:{cons=edgeConstraintsK;break;}
485      case TransformElementType.L_VALUE:{cons=edgeConstraintsL;break;}
486      case TransformElementType.R_VALUE:{cons=edgeConstraintsR;break;}
487      }
488      HashMap<String,HashMap<String,HashMap<String,List<Constraint>>>> map=cons.getMap();
489      for(String src: map.keySet()){
490       for(String tgt: map.get(src).keySet())
491        for(String type:map.get(src).get(tgt).keySet())
492         for(Constraint c:map.get(src).get(tgt).get(type)){
493          ITransformNode nsrc=newNodes.get(src), ntgt=newNodes.get(tgt);
494          ITypeEdge etype=(ITypeEdge)typeedges.get(type);
495          if( edgesR.containsKey(nsrc,ntgt,etype) &&
496            !edgesR.get(nsrc,ntgt,etype).isEmpty()){
497           ITransformEdge foundEdge=edgesR.get(nsrc,ntgt,etype).remove(0);
498           if(!i.equals(TransformElementType.R)){
499            edgeModify.add(foundEdge);
500           }
501           edgeTrans.put(foundEdge, i);
502
503          }else if( edgesK.containsKey(nsrc,ntgt,etype) &&
504             !edgesK.get(nsrc,ntgt,etype).isEmpty()){
505           ITransformEdge foundEdge=edgesK.get(nsrc,ntgt,etype).remove(0);
506           if(!i.equals(TransformElementType.K)){
507            edgeModify.add(foundEdge);
508           }
509           edgeTrans.put(foundEdge, i);
510
511          }else
512          if( edgesL.containsKey(nsrc,ntgt,etype) &&
513             !edgesL.get(nsrc,ntgt,etype).isEmpty()){
```

```
514             ITransformEdge foundEdge=edgesL.get(nsrc,ntgt,etype).remove(0);
515             if(!i.equals(TransformElementType.L)){
516              edgeModify.add(foundEdge);
517             }
518             edgeTrans.put(foundEdge, i);
519
520            }else{
521             //new edges
522             ITransformEdge ne=new ITransformEdgeImpl();
523             edgeAdd.add(ne);
524             edgeTypes.put(ne, (ITypeEdge)typeedges.get(c.getType().getName()));
525             edgeSrc.put(ne, src);
526             edgeTgt.put(ne, tgt);
527             edgeTrans.put(ne, i);
528            }
529
530           }
531
532      }
533     }
534     //removed edges
535     for(TransformElementType i:TransformElementType.values()){
536      MultiHashMap3<ITransformNode,ITransformNode,ITypeEdge,LinkedList<ITransformEdge>>
            cons=null;
537      switch(i.getValue()){
538      case TransformElementType.K_VALUE:{cons=edgesK;break;}
539      case TransformElementType.L_VALUE:{cons=edgesL;break;}
540      case TransformElementType.R_VALUE:{cons=edgesR;break;}
541      }
542      HashMap<ITransformNode,HashMap<ITransformNode,HashMap<ITypeEdge,LinkedList<
            ITransformEdge>>>> map=cons.getMap();
543      for(ITransformNode src:map.keySet())
544       for(ITransformNode tgt:map.get(src).keySet())
545        for(ITypeEdge type:map.get(src).get(tgt).keySet())
546         edgeDelete.addAll(map.get(src).get(tgt).get(type));
547     }
548
549     //execute the changes: in this order: delete edges, delete nodes, modify nodes,
           modify edges, add nodes, add edges
550     for(ITransformEdge e:edgeDelete){
551      EdgeDeleteCommand cmd=new EdgeDeleteCommand(e);
552      commandStack.execute(cmd);
553     }
554     for(ITransformNode n:nodeDelete){
555      NodeDeleteCommand cmd=new NodeDeleteCommand(n);
556      commandStack.execute(cmd);
557     }
558     for(ITransformEdge e:edgeModify){
559      TransformTypeChangeCommand cmd=new TransformTypeChangeCommand();
560      cmd.setModel(e);
561      cmd.setNewType(edgeTrans.get(e));
562      commandStack.execute(cmd);
563     }
564     for(ITransformNode n:nodeModify){
565      TransformTypeChangeCommand cmd=new TransformTypeChangeCommand();
566      cmd.setModel(n);
567      cmd.setNewType(nodeTrans.get(n.getID()));
568      commandStack.execute(cmd);
569     }
570     for(ITransformNode n:nodeAdd){
571      NodeCreateCommand cmd=new NodeCreateCommand(gt,new Rectangle(0,0,-1,-1),nodeTypes.
            get(n.getID()),n.getID());
572      commandStack.execute(cmd);
573      cmd.getCreatedNode().eAdapters().add(this);
574      newNodes.put(n.getID(), (ITransformNode) cmd.getCreatedNode());
575      TransformTypeChangeCommand cmd2= new TransformTypeChangeCommand();
576      cmd2.setModel(cmd.getCreatedNode());
577      cmd2.setNewType(nodeTrans.get(n.getID()));
578      commandStack.execute(cmd2);
579     }
580     for(ITransformEdge e:edgeAdd){
581      EdgeCreateCommand cmd=new EdgeCreateCommand(newNodes.get(edgeSrc.get(e)),edgeTypes.
            get(e));
582      cmd.setTarget(newNodes.get(edgeTgt.get(e)));
583      commandStack.execute(cmd);
584      cmd.getCreatedEdge().eAdapters().add(this);
585      TransformTypeChangeCommand cmd2= new TransformTypeChangeCommand();
586      cmd2.setModel(cmd.getCreatedEdge());
587
588      cmd2.setNewType(edgeTrans.get(e));
```

```
589       commandStack.execute(cmd2);
590
591     }
592
593     return true;
594   }
```

C.6 Full TEF grammar

This appendix lists the full TEF grammar used for CHR programs.

Listing C.8: TEF grammar for CHR.

```
1  syntax(CHR) "resources/chrtypegraph.ecore" {
2   CHR:element(CHR) -> (ConstraintDefs (RuleBlock)?
3         (Input:composite(Inputs))?)?;
4   RuleBlock -> "rules{" ws(statement)
5         ws(indent) Rule:composite(Rule) ws(statement)
6         //(ws(indent) Rule:composite(Rule))*
7       "}";
8   Rule:element(PropRule) -> (PropHead)? ws(statement)
9         "<=>" ws(statement) PropBody ".";
10  PropHead -> Constraint:composite(head)
11     ("," ws(space) Constraint:composite(head))*;
12  PropBody -> Constraint:composite(body)
13     ("," ws(space) Constraint:composite(body))*;
14  PropBody -> "true";
15  ConstraintDefs -> ("public" ws(space))? "Constraint"
16       ws(space) ConstraintDef:composite(constraintDef)
17       ("," ws(space) ConstraintDef:composite(constraintDef))*
18       ";" ws(statement);
19  Input:element(Input) -> ":-" Constraints "." ws(statement);
20  ConstraintDef:element(ConstraintDef) ->
21       IDENTIFIER:composite(name) ("("
22       IDENTIFIER:composite(variableTypes) ("," 
23       IDENTIFIER:composite(variableTypes))*")")?;
24  Constraints -> Constraint:composite(constraints)
25       ("," ws(space) Constraint:composite(constraints))*;
26  Constraint:element(Constraint) ->
27       ConstraintDefRef:reference(type) ("("
28       Expression:composite(variables) ("," 
29       Expression:composite(variables))*")")?;
30  ConstraintDefRef:element(ConstraintDef) ->
31       IDENTIFIER:composite(name);
32  Expression -> Literal;
33  Expression -> Numeral;
34  Expression -> Minus;
35  Expression -> Plus;
36  Literal:element(Literal) -> IDENTIFIER:composite(value);
37  Numeral:element(Numeral) -> INTEGER:composite(value);
38  Minus:element(Minus) -> Expression:composite(lhs)
39       "-" Expression:composite(rhs);
40  Plus:element(Plus) -> Expression:composite(lhs)
41       "+" Expression:composite(rhs);
42  }
```

Name: Mathias Wasserthal Matrikelnummer: 541501

Erklärung

Ich erkläre, dass ich die Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Ulm, den .....

Mathias Wasserthal