Grez – User Manual

H.J. Sander Bruggink

Contents

Contents **i**

1 Introduction **1**
- 1.1 Introduction to *Grez* . . . 1
- 1.2 Name . . . 1
- 1.3 People . . . 2
- 1.4 System requirements . . . 2
- 1.5 Installation . . . 2
- 1.6 Feedback and bug reports . . . 2
- 1.7 How to read this manual . . . 3
- 1.8 License . . . 3

2 The Graphical User Interface **5**
- 2.1 Running *Grez* . . . 5
- 2.2 The main window . . . 5
- 2.3 The main menu . . . 6
- 2.4 Opening graph transformation systems . . . 7
  - 2.4.1 Reading a system from a file . . . 7
  - 2.4.2 Generating random systems . . . 7
- 2.5 Actions . . . 8
  - 2.5.1 Proving termination . . . 8
  - 2.5.2 Comparing algorithms . . . 9
  - 2.5.3 Generating traces . . . 9
- 2.6 Specifying algorithms . . . 9
  - 2.6.1 The algorithm list editor . . . 9
  - 2.6.2 The algorithm editor . . . 9
- 2.7 Configuring external tools . . . 10
  - 2.7.1 Configuring SMT solvers . . . 10
  - 2.7.2 Configuring termination tools . . . 11
- 2.8 Changing user preferences . . . 12
- 2.9 Getting help . . . 12

3 The Command-Line Interface **13**

3.1 Using *Grez* from the command-line . . . 13
3.2 Command-line options . . . 13
3.2.1 General options . . . 13
3.2.2 Action type and system selection . . . 14
3.2.3 Reporting and proof generation . . . 14
3.2.4 Random system generation . . . 14
3.3 Specifying algorithms . . . 15

4 Specifying Graph Transformation Systems: The Simple Graph Format 17
4.1 A first tour . . . 17
4.1.1 Defining rules . . . 17
4.1.2 Defining graph transformation systems . . . 18
4.2 Grammar . . . 19
4.2.1 Tokens . . . 19
4.2.2 Object definitions . . . 19
4.2.3 Graphs . . . 20
4.2.4 Morphisms . . . 20
4.2.5 Transformation rules . . . 21
4.2.6 Transformation systems . . . 21
4.3 Examples . . . 22
4.3.1 Two systems in one file . . . 22
4.3.2 In-place definition . . . 22
4.3.3 Ad-hoc routing . . . 23

5 Theoretical Foundations 25
5.1 Graph transformation . . . 25
5.2 Termination . . . 26
5.3 Satisfiability modulo theories . . . 27
5.4 Algorithms for proving termination and non-termination . . . 27
5.4.1 Finding cycles . . . 27
5.4.2 Node counting . . . 27
5.4.3 Edge counting . . . 28
5.4.4 Petri-net approximation . . . 28
5.4.5 Type graph techniques . . . 29
5.4.6 Match bounds . . . 30

A License texts 33
A.1 BSD License . . . 33
A.2 Apache License, version 2.0 . . . 33
A.3 Creative Commons Attribution 4.0 International Public License . . . 36

Bibliography 41

Index 43

One

Introduction

1.1 Introduction to *Grez*

Termination, the absence of infinite computations, is a property that is required in many applications, in particular in model transformation, algorithms and protocol specifications. Many of these applications, such as graphical model transformation (for example, of UML models) and algorithms that manipulate data structures on the heap, are naturally modeled by graph transformation systems. *Grez* is a software tool that, given a graph transformation system, automatically tries to find a proof that the graph transformation system terminates or a proof that it does not. It does this by running a number of algorithms concurrently and reporting the result of the first of them that successfully finishes. Since the problem is undecidable in general [7], in many cases none of the algorithms will find a termination or non-termination proof.

Grez provides both a graphical user interface and a command-line interface. Configuring *Grez*, in particular the paths and parameters of the external tools called by it, is only possible with the graphical user interface, but other than that all *Grez* functionality can be accessed through both interfaces.

Grez is written in Java 8 and uses the standard Java libraries, including Swing and concurrency, JavaCC, ANTLR and Google Guava, as well as a number of in-house libraries for graph transformation and visualization. For some (optional) functionality it needs an external tool, in particular an SMT solver to find solutions for sets of constraints and a LaTeX distribution to generate PDF reports.

1.2 Name

People have wondered where the name *Grez* comes from. There are two slightly different things that people want to know when they ask this question.

Some people who ask this question are curious about what "Grez" *means*. An implicit assumption here is that every name must mean something. *Amsterdam* is a city which grew around a dam built in the river Amstel; *England* is the land of the angles, an ancient Germanic tribe; *Columbia* is named after Christopher Columbus. But that is not the case here. There is no language I know of, current or historic, where the letter sequence G, R, E, Z has any sensible meaning other than "some word which starts with a G, and ends with a Z."

Other people do not want to know what "Grez" means but what the reason is that the name "Grez" was picked for this particular piece of software. This is an entirely different question, one with an answer! I took the letters T, E, R and M, which

are the first four letter of the word "termination", and the applied the well-known ROT-13 cipher to them. The result is "Grez", which, coincidentally, starts with the same two letters as the word "Graph".

1.3 People

The following people (in alphabetical order) are or were involved in the development of *Grez*, either by directly implementing the tool or by providing its theoretical foundation:

- H.J. Sander Bruggink (main developer)
- Barbara König
- Dennis Nolte
- Hans Zantema

1.4 System requirements

Grez is written in Java. To use *Grez*, you need a recent Java virtual machine (Java 8 or newer). Additionally, to export proofs to PDF, you need a LaTeX distribution installed, including common packages and style files (the command `pdflatex` must be in the path).

The performance of some algorithms is greatly improved if an external SMT solver is used (SMT means *satisfiability modulo theories*), and some algorithms even require this. All SMT solvers which understand the SMT-LIB 2 input format are supported.[1] External SMT solvers can be set up using the GUI, and after that they are also available for use with the command-line interface.

The standard *Grez* distribution contains all libraries that *Grez* uses; thus, other than Java itself, you don't have to install additional software to run *Grez*. To compile from source, however, you need the following external Java libraries:

- ANTLR 4 (`www.antlr.org`), used for parsing the output of SMT solvers;
- Google Guava, version 14 or newer (`github.com/google/guava`);

1.5 Installation

To install *Grez* it is only needed to copy the file `grez.jar` to a location of your choice.

If you plan on using *Grez* through the command-line a lot, it is advisable to create a shell script in a directory in the path. In Unix-like operating systems, such as Mac OS X and Linux, this file could be named `grez` and contain:

```
#/bin/sh
java -jar -Dgrez.script=grez
    ➥ /path/to/grez.jar "$@"
```

(the last two lines should be on the same line). In Windows, the file could be named `grez.bat` and look like:

```
java -jar -Dgrez.script=grez
    ➥ C:\path\to\grez.jar %*
```

(the file should consist of a single line). The `-Dgrez.script=grez` option sets a Java property which is read by *Grez*. It has no effect to the working of the program except that the command name is displayed correctly in some user interactions. Creating such a shell script allows you to use `grez` as a command instead of `java -jar grez.jar`.

1.6 Feedback and bug reports

Grez is free software, both as in free speech and as in free beer. This means that you can use the software and inspect and modify its source code without cost, but it also means that there is no warranty, no guarantee and no customer support.

[1] Some SMT solvers that claim to support the SMT-LIB 2 format, actually do not fully do so.

Still, if you have feedback, bug reports and questions you may send them to the main developer by e-mail:

`sanderbruggink@gmail.com`

(Please indicate *Grez* in the subject line.) If time allows for it, you might even get an answer!

1.7 How to read this manual

This manual is not intended to be read sequentially. In particular, the theoretical foundations of the variant of graph transformation that *Grez* uses, as well as of the algorithms that it employs to find termination proofs, are covered in Chapter 5, so if you want to know the theory behind *Grez*, you should start there.

Grez offers to main user interfaces, a graphical user interface and a command-line interface, which are covered in Chapters 2 and 3, respectively. If you are not interested in the graphical user interface you do not need to read Chapter 2, and if you do not want to use the command-line interface, you do not need to read Chapter 3.

Chapter 4 presents the file format that *Grez* uses. Since *Grez* currently lacks a graphical user interface for specifying graph transformation systems, you need to read this chapter if you want to apply it to your own graph transformation systems.

1.8 License

Grez is distributed under the terms of the (three clause) BSD license (see Appendix A.1). The two main third-party libraries used are ANTLR and Google Guava. The first is, like *Grez* itself, distributed under the terms of the (three clause) BSD license, while the second is distributed under the Apache license (see Appendix A.2).

This documentation is distributed under the *Creative Commons 4.0 Attribution* licence, CC BY 4.0 (see Appendix A.3).

Two

The Graphical User Interface

The most common way to interact with *Grez* is the graphical user interface, which is presented in this chapter. The graphical user interface allows the user to open graph transformation systems (and generate random ones), run algorithms which try to prove termination or non-termination, and inspect termination and non-termination proofs produced by these algorithms. Additionally the graphical user interface is used to configure the external tools called by *Grez*.

2.1 Running *Grez*

To run *Grez*, a working Java 8 installation must be present. If this is the case *Grez*'s graphical user interface (GUI) is started on the command-line by changing to the directory where the file `grez.jar` resides and issuing the command

```
java -jar grez.jar
```

Alternatively, in most operating systems one can double-click the `grez.jar` file in the file browser.

When starting *Grez*'s graphical user interface, one can also specify an input file and use the command-line option `--system` to open a specific graph transformation system (see § 3.2).

2.2 The main window

The main window of *Grez* is displayed in Figure 2.1. It consists of the following parts; from top to bottom these are:

- In the *main menu* there are various commands for opening and generating graph transformation systems, configuring external programs and changing user preferences.
- In the *rule selection box* the user can select the rule of the currently opened graph transformation system which is displayed.
- In the *rule display* the currently selected rule is displayed. One can change the layout of the rule by dragging nodes and (control points of) edges. When multiple nodes and edges are selected, they can be moved simultaneously by dragging the selection box or rotated by dragging the quarter circle in the top left corner of the selection box. Currently it is not possible to modify the structure of the rule.
- In the *action panel* the user can select the action she wants to perform (prove, compare or trace) and give the algorithm or algorithms which are to be used in this action. In the right of the action panel is the *Execute* but-

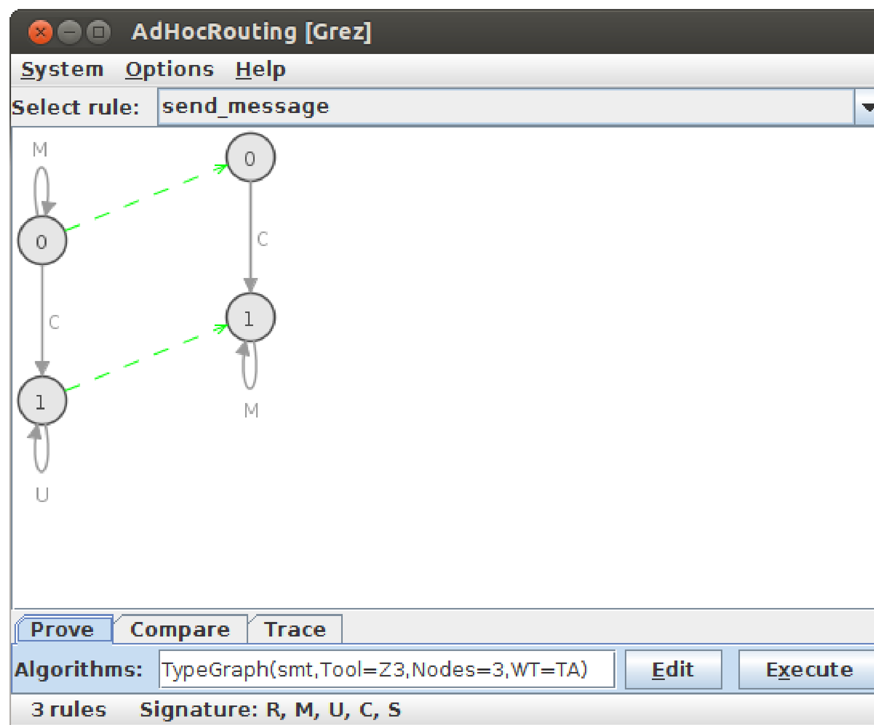


Figure 2.1: Main window of *Grez*.

ton which executes the currently selected action.

- At the bottom is the *status bar* which displays information about the currently open graph transformation system, in particular the number of rules and the edge labels occurring in it.

2.3 The main menu

Grez's main menu contains the following menus and commands:

- *System* – Commands for obtaining graph transformation systems.
  - *Open GTS* – Open a system from a file; see § 2.4.1.
  - *Generate random GTS* – Generates a random graph transformation system with user-specified parameters; see § 2.4.2.
  - *Exit* – Quit *Grez*
- *Options* – Command for configuring *Grez*.
  - *Termination tools* – Configure external termination tools used by *Grez*; see § 2.7.2.
  - *SMT solvers* – Configure SMT solvers used by *Grez*; see § 2.7.1.
  - *GUI options* – Change some preferences for the graphical user interface; see § 2.8.
  - *Change number of worker threads* – Change the number of worker threads

that *Grez* uses when proving or comparing algorithms (the default is the number of separate CPU cores in the computer).

- *Help* – Obtain information about the program; see § 2.9
  - *Visit website* – Open *Grez*'s website in a browser window.
  - *About* – Display copyright, license and version information of *Grez*.

2.4 Opening graph transformation systems

There are two ways to open a graph transformation system: reading a graph transformation system from a file or generating a random one.

2.4.1 Reading a system from a file

A graph transformation system can be read from a file in two ways, on the command-line and through the graphical user interface.

- *On the command line.* Specify the file containing the GTS on the command-line. If this file contains more than one GTS, the name of the GTS must be specified using the `--system=`*name* command-line option. See Chapter 3 for more information.

- *Through the graphical user interface.* Select *Open GTS* from the *System* menu. A file dialog appears. Select the file containing the graph transformation system you want to open. When selecting a file, in the right of the dialog a list of graph transformation systems in this file appears. Select the one you want to open and click *Open*.

2.4.2 Generating random systems

Grez provides the option of automatically generating GTSs. To generate a random graph transformation system, select the *Generate random GTS* menu item from the *System* menu. A dialog appears in which the parameters of graph transformation system generation can be specified.

The parameters can be divided in four parts: general parameters, parameters for the left-hand side, parameters for the right-hand side and parameters for the interface of the rule.

General parameters.

- *Number of rules* – This is a single number which specifies the number of rules in the randomly generated GTS.

- *Signature* – The labels that can occur in the GTS and their arities. This is a comma-separated list of entries of the form $\mathit{label} : \mathit{arity}$, where label is the string displayed on the edge and arity is the arity of the label, that is the number of nodes edges labeled with this label are incident to. Alternatively, press the *Edit* button to edit the signature through a dialog.

Left-hand and right-hand side parameters.

- *Number of nodes.* Distribution for the number of nodes in a graph (see below).

- *Density.* Distribution for the density of the graph. The density is a floating point number ≥ 0 which determines the number of edges in the graph depending on the number of nodes. In particular, if n is the number of nodes and d the density, the number of edges in the graph will be $d \cdot n^2$. In typical cases, the density will lie between 0 and 1.

- *Connectedness.* This parameter control to which extend the graph is connected. It has three possible values:

  - *Arbitrary.* All graphs can be generated, regardless of whether they are connected or not.
  - *No isolated nodes.* Non-connected graphs can be generated, but each node will always be incident to at least one edge.
  - *Connected.* Only connected graphs will be generated.

Interface parameters.

- *Number of nodes.* Distribution for the number of nodes in the interface (see below).
- *Discrete interfaces.* When checked, only interfaces without edges will be generated. Otherwise, the interface will be the maximal interface such that each interface edge of the left-hand side has a corresponding edge in the right-hand side.

Specifying distributions. Distributions can be given through a dialog by clicking the *Edit* button, or entered manually by giving a string representation of the distribution. The followings distributions are available for all options where a distribution is required:

- *Non-random.* A single number which is always chosen. The syntax is simply this number, for example 3 or `0.25`.
- *Uniform.* All numbers within a specified range have an equal probability. The string representation of such a distribution is `uniform(`*min*`,`*max*`)`.
- *Normal.* Normal distribution with a given mean and standard deviation. The string representation of such a distribution is `normal(`*mean*`,`*sd*`)`, where *sd* is the standard deviation.

2.5 Actions

Grez has three main actions to perform on graph transformation systems: it can try to automatically *prove* that a given graph transformation system terminates or not using a number of algorithms (§ 2.5.1), it can *compare* the result of a number of algorithms (§ 2.5.2), and it can produce a *trace* for a single algorithm (§ 2.5.3).

2.5.1 Proving termination

To prove termination of a graph transformation system, choose the *Prove* tab from the action panel, give the algorithms which are to be used for proving termination, and press the *Execute* button.

When proving termination, *Grez* runs a number of user-selected algorithms concurrently. As soon as one algorithm returns a proof of termination or non-termination of the graph transformation system under consideration, all algorithms are stopped and the found proof is reported to the user. If an algorithm finds a relative termination proof, all algorithms are stopped and the procedure is repeated on the smaller graph transformation system from this relative termination proof.

Algorithms are run concurrently, but it is not the case that all algorithms are started at the same time: only as many algorithms are run concurrently as CPU cores. The order in which the algorithms are executed depends on their order in the list.

There are three possible outcomes of this action: a proof that the currently opened graph transformation system terminates, a proof that the currently opened graph transformation system does not terminate, or, if none of the algorithms could find a termination or non-termination proof, no proof at all.

2.5.2 Comparing algorithms

To compare the results of a number of algorithms on the currently opened graph transformation system, select the *Compare* tab from the action panel, give a list of algorithm to compare and click the *Execute* button.

When comparing algorithms, all algorithms specified by the user are run until they finish, either by giving a (relative or non-relative) termination proof, a non-termination proof or report that they cannot find a proof. The results of all algorithms are reported to the user. If some algorithm return a relative termination proof the relative termination proof will be reported and the procedure will not be repeated on the smaller graph transformation system.

2.5.3 Generating traces

Some algorithms can generate a trace to make visible in what order certain operations are tried out. To generate a trace of an algorithm, select the *Trace* tab of the action panel, enter the algorithm to generate a trace of, and click the *Execute* button. The selected algorithm is executed, the steps of the algorithm are stored, and after the algorithm finishes its trace and its result are reported to the user.

Note that generating a trace is not supported by all algorithms.

2.6 Specifying algorithms

There are two ways to specify the algorithms which are going to be run: the string representation of the algorithm or algorithm list can be edited directly, or it can be edited through a dialog. If you want to edit the string directly, see § 3.3 for the syntax. If you want to use the dialog press the *Edit* button to the right of the text box. If the currently selected action is *Prove* or *Compare*, the *Algorithm list editor* appears to specify the list of algorithms; if the *Trace* action is selected, the *Algorithm editor* appears to specify the algorithm.

2.6.1 The algorithm list editor

The algorithm list editor displays the list of (representations of) algorithm on the left and a number of command buttons on the right. Available command buttons are:

- *Add* – Opens the *Algorithm editor* and adds the algorithm specified there to the algorithm list.
- *Remove* – Removes the currently selected algorithm from the list.
- *Clear* – Removes all algorithms from the list.
- *Edit* – Opens the *Algorithm editor* to edit the currently selected algorithm.
- *Move up* – Moves the currently selected algorithm one position up in the list.
- *Move down* – Moves the currently selected algorithm one position down in the list.
- *Default* – Replaces the list by the default list of algorithms.
- *Complete* – Adds to the algorithm list all algorithms of the default list, for which there is not already an algorithm of the same type (but possibly with different parameters) in the list.

2.6.2 The algorithm editor

In the algorithm editor we can specify the algorithm type and for each type we can give a number of parameters. The following algorithm types are available:

- *Cycle finder.* Tries to prove non-termination of the selected graph transformation system by trying to construct a cycle. This algorithm is described in § 5.4.1. The parameters are:

- *Initial graph size* – number of nodes in the initial graph;
- *Length limit* – length of the transformation sequences which are generated.

- *Node counter.* Tries to prove termination by showing that the number of nodes decreases in each rule application. This algorithm is described in § 5.4.2.

- *Edge counter.* Tries to prove termination by showing that the number of edges with a label from a certain set decreases in each rule application. This algorithm is described in § 5.4.3.

- *Type graph finder.* Tries to prove termination by the weighted type graph method described in § 5.4.5. The parameters are:
  - *Number of nodes in type graph* – maximal number of nodes in the type graph;
  - *Maximum weight* – maximal weight (only available if the *Use SMT* option is turned off; if on, there is no limit on the maximum weight);
  - *Possible semirings* – which kind of evaluation can be used;
  - *Use SMT* – whether to use an external SMT solver; click the *Configure* button to specify which SMT solver to use.
  - *Use non-relative termination* – whether to use a relative or non-relative termination. If non-relative termination is used and the SMT solver fails to find a proof, the relative termination is used afterwards.

- *Match bound finder.* Tries to prove termination by using the match bound method described in § 5.4.6. The *Length limit* parameter specifies after how many iterations the algorithm will be aborted, while the other parameters specify which type graph simplifications will be used.

- *Petri-net approximator.* Tries to prove termination by approximating the graph transformation system by a Petri-net. This algorithm is described in § 5.4.4. This algorithm requires an external SMT solver. Click the *Configure* button to specify which SMT solver to use.

- *External tool.* Sends the graph transformation to an external tool, possibly after performing some translation.

2.7 Configuring external tools

Two types of external tools can be used with *Grez*: external SMT solvers and external termination tools. The external SMT solvers are used by some algorithms to solve constraints, while external termination tools serve as a kind of oracle: *Grez* instructs them to prove termination of a graph transformation system, after translating it to a format suitable for the tool, and trusts that the result is correct.

External SMT solvers and termination tools need to be configured before *Grez* can use them. In particular, *Grez* needs to know where their executable files are, what command-line arguments the tools require, how and in what format they expect their input, and how to interpret their output.

2.7.1 Configuring SMT solvers

To configure the external SMT solvers used by *Grez*, select the *SMT solvers* item from the *Options* menu. A dialog appears with which you can add, remove and edit the configurations for the external SMT solvers. For each SMT solver the following information is required:

- *Name* – The name of the SMT tool. This name is used to refer to the tool in user interactions.
- *Executable* – Full path of the executable file of the tool. Click the *Browse* button to select the executable in a file dialog box.
- *Command-line arguments* – The command-line arguments to be passed to the tool when called from *Grez*. If the command-line arguments contain the literal string '%FILE%' or '$FILE', this string is replaced by the input file name. If it is required to pass an input file name to the tool, but none of the two string is present, then *Grez* adds the input file name to the end of the command-line.
- *Input method* – Whether the SMT solver expects its input to be passed through the standard input stream, or via a temporary file on disk.

Grez expects SMT solvers to write their output in SMT-LIB 2 format to the standard output stream.

In principle *Grez* support any SMT solver which can read files in SMT-LIB 2 format and produces results and models in the same format. Some SMT solvers which claim to be compatible with SMT-LIB 2, however, are not or not completely. *Grez* was tested and found to work with *CVC4* [1] and *Z3* [2]. Other solvers might also work; it is up to the user to try out.

2.7.2 Configuring termination tools

To configure external termination tools used by *Grez*, select the *Termination tools* item from the *Options* menu. A dialog appears with which you can add, remove and edit the configurations for the external termination tools. For each termination tool the following information is required:

- *Name* – The name of the tool. This name is used to refer to the tool in user interactions.
- *Executable* – Full path of the executable file of the tool. Click the *Browse* button to select the executable in a file dialog box.
- *Command-line arguments* – The command-line arguments to be passed to the tool when called from *Grez*. If the command-line arguments contain the literal string '%FILE%' or '$FILE', this string is replaced by the input file name. If it is required to pass an input file name to the tool, but none of the two string is present, then *Grez* adds the input file name to the end of the command-line.
- *Input format* – The file format in which *Grez* will translate the graph transformation system prior to passing it to the tool. The possible values are:
  - *Simple graph format* – Pass the graph transformation system in simple graph format to the external tool.
  - *String rewrite system* – Builds a cycle rewrite system which is equivalent to the graph transformation system in terms of termination, and then calls the tool on that.
- *Input method* – Whether the termination tool expects its input to be passed through the standard input stream, or via a temporary file on disk.
- *Terminating regex* – If the tool's output matches this regular expression, *Grez* will interpret this to mean: *the system is terminating*.
- *Non-terminating regex* – If the tool's output matches this regular expression, *Grez* will interpret this to mean: *the system is not terminating*.

2.8 Changing user preferences

By choosing *GUI options* from the *Options* menu the user can change the way *Grez* displays graphs and rules, and specify when the log window is closed after executing an action.

The following options can be changed by the user:

- *Layouter* – Select the layouter which produces the initial layout (that is, determines the initial position of the nodes and edges) of the graphs and rules. Possible options are:
  - *Leveled* – Locate nodes with minimum number of incoming edges at the top, nodes directly reachable from those nodes at the level below, etc. In general produces good results for graphs containing only binary edges.
  - *Spring embedder* – Use a spring embedder to layout graphs and rules. In general produces satisfiable results.
- *Rule style* – Select how rules are displayed. Possible options are:
  - *Separated* – The left-hand side and right-hand side are displayed separately, and the correspondence morphism is displayed with dashed lines.
  - *Unified* – The left-hand side and right-hand side are displayed in a merged graph; colors indicate which nodes and edges are removed and created.
- *Close log when finished* – During proving, comparing or tracing a log is displayed which reports to the user what the algorithms are currently doing. This option controls when this log is automatically closed. Possible values are:
  - *Never* – Never automatically close the log. The user must click the *Close* button manually to close the log.
  - *When successful* – The log is automatically closed when one of the algorithms returned a result, but not when no termination or non-termination proof could be found.
  - *Always* – The log is always automatically closed after all algorithms finished. This causes *Grez* to finish silently when no proof can be found.

2.9 Getting help

Grez does not provide an online help system. Its documentation consists of this user manual and the web site. Under the *Help* menu you can find two menu items: *Visit website* will open *Grez*'s website in your standard browser, and *About* displays version and license information of *Grez*.

Three

The Command-Line Interface

Almost all functionality of *Grez* can also be accessed from the command line. This is mainly useful if you want to start *Grez* from a shell script or another application, for example to perform batch processing. In this chapter we describe the command-line interface of *Grez*.

One thing you cannot do with the command-line interface is configuring external tools, such as other termination provers and SMT solvers. For this you need the graphical user interface (see § 2.7). After an external tool has been configured, it is available through the command-line interface, however.

3.1 Using *Grez* from the command-line

From the directory where the file `grez.jar` is located, *Grez* is started from the command-line with following command:

```
java -jar grez.jar --cli <args>
```

When a suitable shell script is created in the path (see § 1.5), one can also use (in any directory) the following:

```
grez --cli <args>
```

In both cases the `<args>` consist of a number of command-line options and, optionally, a single input file. An option starts with `-`, `--` or `/`, and any other command-line argument is interpreted as an file name. The `--cli` option makes sure that *Grez*'s graphical user interface is not started. The command-line options and input file may occur in any order. The available command-line options are listed in the next section.

3.2 Command-line options

Options start with `-`, `--` or `/`. These three are equivalent; for example, `-help`, `--help` and `/help` are all recognized as a command-line option and have the same effect. The available command-line options are:

3.2.1 General options

`--help, -h` – Display usage information (basically the information of this section) and abort.

`--version, -v` – Display version information and abort.

`--cli, -c` – Use the command-line interface, that is, do *not* start the graphical user interface.

If none of the above options are present, the graphical user interface is started.

3.2.2 Action type and system selection

`--action=`*act* – Select the action to perform, possibilities for *act* are: `prove` (try to prove termination or non-termination of a graph transformation system) `compare` (compare results of different algorithms) and `trace` (produce a trace of a supported algorithm).

`--algos=`*list* – Select the algorithm(s) to try. *list* is a comma-separated list of algorithms (see § 3.3).

`--threads=`*num* – Specify the number of worker threads that *Grez* uses when executing the *Prove* or *Compare* actions. The default value is the number of separate CPU cores in the machine. The *Trace* action uses a single thread regardless of the value of this parameter.

`--system=`*system* – If the input file contains more than one graph transformation system, select the one with the given name.

`--random` – Generate a random graph transformation system.

`--beta` – Enable algorithms that are based on unpublished results and/or currently in beta stage. These algorithms are disabled by default. Note that these algorithms are *not documented*.

3.2.3 Reporting and proof generation

`--progress=`*style* – Set the progress report style, possibilities for *style* are: `silent` (no progress report), `nice` (the default), `nice-unicode` and `standard`.

`--rtype=`*ext* – Set the file type of the generated report (proof/disproof), possibilities for *ext* are: `silent` (do not write report), `text` (text only; default) and `pdf` (PDF). For the last option a LaTeX distribution is required.

`--rfile=`*file* – Set the file where the generated report is written; default is standard out if `rtype=text` and `grez-output.pdf` if `rtype=pdf`.

`--silent` – Shorthand for `--rftype=silent --progress=silent`.

3.2.4 Random system generation

For more information on the meaning of these options, and the syntax for specifying distributions, see § 2.4.2.

`--left-size=`*dist* – Set the distribution for the number of nodes in the left-hand side.

`--right-size=`*dist* – Set the distribution for the number of nodes in the right-hand side.

`--if-size=`*dist* – Set the number of nodes in the interface.

`--left-dens=`*dist* – Set the distribution for the density of the left-hand side.

`--right-dens=`*dist* – Set the distribution for the density of the right-hand side.

`--dens=`*dist* – Set the distribution for the density of the left-hand side and right-hand side at the same time.

`--left-conn=`*conn* – Set the connectedness of the left-hand side, possibilities are `all`, `connected` and `noiso`.

`--right-conn=`*conn* – Set the connectedness of the right-hand side, possibilities are `all`, `connected` and `noiso`.

`--conn=`*conn* – Set connectedness of the left-hand side and right-hand side at the same time.

`--discrete=`*bool* – Set whether or not the interface is discrete (default value: true).

`--rules=`*num* – Set the number of rules to generate.

`--labels=`*list* – Set the signature of the generated graph transformation system. The value *list* is a comma-separated list of entries of the form *label* : *arity* .

3.3 Specifying algorithms

The general syntax for specifying algorithms is the following:

Algorithm (*Parameters*)

where *Parameters* is a comma-separated list of *param*=*value* pairs. If an algorithm has no parameters the parentheses are optional.

The *value* can be any string not containing a comma. If the parameter requires an integer value, *value* must be an non-negative integer (a sequence of digits), and if the parameter is a boolean parameter, *value* must be either '`true`' or '`false`'; if the value is to be set to true, '`=true`' can be omitted.

The possible algorithms are list below. Here, an arbitrary string is denoted by *s* , a (non-negative) integer by *n* and a boolean value by *b* .

`CycleFinder` – Tries to prove non-termination by constructing a cycle. This algorithm is described in § 5.4.1.

`EdgeCounter` – Tries to prove termination by showing that the number of edges with a label from a certain set decreases in each rule application. This algorithm is described in § 5.4.3.

`External(tool=`*s* `)` – Calls an external termination tool, the name of which is given by the `tool` parameter. External tools can only be configure using the graphical user interface, see § 2.7.2.

`MatchBound(size=`*n*`,limit=`*n*`,mini=`*b*`,in=`*b*`, out=`*b* `)` – Tries to prove termination by using the match bound method described in § 5.4.6. The `size` parameter specifies the size of the initial graph, the `limit` parameter specifies after how many iterations the algorithm is aborted. The parameters `mini`, `in` and `out` turn on or off *conservative minimization*, *incoming factorization* and *outgoing factorization*, respectively.

`NodeCounter` – Tries to prove termination by showing that the number of nodes decreases in each transformation step. This algorithm is described in § 5.4.2.

`PetriNet(tool=`*s* `)` – Tries to prove termination by approximating the graph transformation system by a Petri-net. This algorithm is described in § 5.4.4. This algorithm requires an external SMT solver, the name of which is specified by the `tool` parameter.

`TypeGraph(smt=`*b*`,tool=`*s*`,nodes=`*n*`,max=`*n*`, wt=`*s*`, nrt=`*b* `)` – Tries to prove termination by the weighted type graph method described in § 5.4.5. The parameters are: `smt` – whether or not to use an external SMT solver; `tool` – SMT solver to use (only if SMT is used); `nodes` – maximum number of nodes in generated type graph; `max` – maximum weight of generated type graph (only if SMT is not used); `wt` – evaluation type; if it contains '`T`' tropical evaluation may be used, if it contains '`A`' arctic evaluation can be used, if it contains '`N`' arithmetic evaluation can be used; `nrt` – wether or not to use non-relative termination. This option is only available if an SMT-solver is used

Four

Specifying Graph Transformation Systems: The Simple Graph Format

Currently, *Grez* does not provide a graphical user interface for constructing graph transformation systems. It can generate random graph transformation systems, or read existing ones from a file stored on disk. If the user wants to find a termination proof of a custom graph transformation system, he must use an external program to create a file which describes the wanted graph transformation system.

The file format that *Grez* natively supports, called the *simple graph format* (SGF), is a text-based format that describes various objects which play a role in graph transformation, such as graphs (and hypergraphs), morphisms and rules. The format was developed to have a single parser and input format that can be used in different graph transformation tools.

Unlike other text-based formats, such as XML-derived formats, which are mainly designed to be easily parsed by a computer program, SGF is designed to be easily written, read and maintained in source form by a human being – the word *simple* in the name reflects that it is supposedly simple to write, not that it describes simple graphs. One of the results of the emphasis on easy writing rather than easy parsing is that there is sometimes more than one way to define the same object.

In this chapter we describe the fragment of SGF which describes objects that *Grez* understands, namely graphs, rules, morphisms and graph transformation systems.

4.1 A first tour

An SGF file consists of a list of object definitions. Each of the objects defined in an SGF file can either be nameless or be named by an identifier. If it is named by an identifier, object definitions occurring later in the file may refer to the object by using its name.

The types of object that are mainly important for *Grez* are graph transformation rules (*rules* for short) and graph transformation systems, which are sets of rules. In this section the most common way to define such objects is presented. For a complete reference on how these objects can be defined, the reader is referred to § 4.2.

4.1.1 Defining rules

Rules are defined in the following way:

```
<name> = rule {
  <left-hand side>
  =>
```

```
  <right-hand side>
};
```

The '<name> =' part is optional if you want to define a rule with a default name. The left-hand side and right-hand side are blocks (delimited with '{' and '}') defining the nodes and edges of the respective graph (the node and edge definitions are separated by semicolons). An edge can be defined in the following ways:

- `e:A(n1,...,nk)` denotes a (hyper)edge with name `e` and label `A` which is incident to the nodes `n1`,...,`nk`. Giving a name to an edge is optional; if it is absent the edge will not be in the correspondence morphism of the rule.
- `n1 --e:A-> n2` denotes a binary edge with source node `n1` and target node `n2`. Again, the name of the edge is optional. The definition `n1 --e:A-> n2` is equivalent to the definition `e:A(n1,n2)`.
- `n2 <-e:A-- n1` denotes a binary edge with source node `n1` and target node `n2`. Again, the name of the edge is optional. The definition `n2 <-e:A-- n1` is equivalent to the definitions `n1 --e:A-> n2` and `e:A(n1,n2)`.

The latter two variants can be "chained". For example

```
n1 --e:A-> n2 --B-> n2 <-A-- n3
```

defines three edges: one `A`-labelled edge from `n1` to `n2`, one `B`-labelled edge from `n2` to `n2` (a loop) and one `A`-labelled edge from `n3` to `n2` (a loop). Only the first of these edges is given a name, `e`.

All nodes occurring in edge definitions are also implicitly added to the graph. If more nodes are to be added to the graph, we can do so by the '`node`' keyword:

```
node n4
```

adds a node named `n4` to the graph. In practice, the '`node`' keyword is only required to add nodes to the graph which are not incident to any edge (that is, *isolated* nodes).

The correspondence morphism of the rule is implicitly defined to be the partial morphism which maps nodes and edges of the left-hand side to the node or edge of the right-hand side with the same name, if it exists.

Example 4.1.1. Consider the following SGF code:

```
MyRule = rule {
  { n1 --A-> n2 --A-> n3 }
  =>
  { B(n1,n4,n3) }
};
```

defines a rule which removes pairs of consecutive `A`-labelled edges and replaces them by a single hyperedge labelled `B`. The node `n2` of the left-hand side is removed, and the node `n4` of the right-hand side is created when this rule is applied.

4.1.2 Defining graph transformation systems

A graph transformation system is defined as follows:

```
<name> = gts {
  rules = [
    <list of rules>
  ];
};
```

The list of rules is a comma-separated list or rules. Usually, the list entries are names of rules defined earlier, but it is also possible to define rules in-place using the `rule{...}` syntax of the previous section.

Example 4.1.2. A graph transformation system which only contains the rule of Example 4.1.1 is defined by the following SGF code:

```
MyGTS = gts {
  rules = [ MyRule ];
};
```

4.2 Grammar

In this section we specify the grammar of the fragment of SGF which describes graph transformation systems and the objects on which they depend.

In the grammar fragments we use the following conventions. Terminal symbols are given in typewriter font and enclosed in double quotes (for example `"graph"` and `"{"`). Non-terminal symbol are displayed in a slanted font. We use a kind of extended Backus-Naur form: the right-hand sides of the rules are basically regular expression consisting of optional parts (?), choice (|) and repetition ($*$).

4.2.1 Tokens

SGF tokens are delimited by whitespace (space, tab, carriage return and line feed) characters. Otherwise, whitespace is ignored (this means, in particular, that newlines are ignored and do not carry any meaning).

The following tokens are keywords in SGF:

`copy`	`ifgraph` ⋆	`rule`
`from`	`morphism`	`system` ⋆
`graph`	`node`	`to`
`gts`	`not` ⋆	`trace` ⋆

The keywords marked with a star are used to describe objects which are not used in the context of termination proving; they are mentioned here for the sake of completeness but can otherwise be ignored when specifying graph transformation systems for use with *Grez*. They are not further mentioned in this chapter.

An identifier in SGF is any non-empty sequence of letters (`a`–`z` and `A`–`Z`), digits (`0`–`9`) and the special symbols `@`, `_` and `.` which is not a keyword. Identifiers and keywords in SGF are case-sensitive, that is `myRule` is not the same token as `myrule` or `MYRULE`. Identifiers function both as names (of defined objects, nodes and edges) and as labels (of edges):

name → *identifier*
label → *identifier*

C/C++-style comments can be used throughout SGF code: on encountering `//` everything until the end of the line is ignored, and additionally every symbol between `/*` and `*/` is ignored.

4.2.2 Object definitions

An SGF file consists of a list of object definitions:

sgf → *object*$*$

Each object definition defines an object of a certain type. For *Grez*, only graphs (`graph`), morphisms (`morphism`), graph transformation rules (`rule`) and graph transformation systems (`gts`) are relevant.

Optionally, each defined object is given a name by preceding the object definition with `*name* =`, where the name is an identifier. By giving an object a name it is possible to refer to the object later.

Each object definition ends with a semicolon.

object → (*name* `"="`) ?
(*graph*
| *morphism*
| *rule*
| *gts*
| *copy*) `";"`

To make a copy of a previously defined object, we can use the copy keyword:

copy → `"copy"` `"("` *name* `")"`

A declaration of the form `copy(`*name*`)` makes a copy of the object with the given name. This is especially useful if you wish to store the same object under two names[1].

4.2.3 Graphs

Graphs are defined with the keyword '`graph`', followed by a description of the nodes and edges of the graph.

graph → `"graph"` `"{"` *graph_desc* `"}"`

The description of the nodes and edges of the graph is block delimited with curly braces containing a semicolon-separated list of node and edge definitions:

graph_desc → *node_edge* (`";"` *node_edge*?) ∗
node_edge → *node* | *edge* | *binedge*

Defining nodes. Nodes can be defined by the '`node`' keyword followed by the name of the node. Nodes have no label (see Chapter 5).

node → `"node"` *name*

A definition of the form '`node` *name*' makes sure that the graph contains a node with the given name; if the graph already contains a node with that name, a second node with the same name will *not* be added.

Defining edges. Edges can be defined in two ways: in a functional style, which allows edges of arbitrary arity to be defined – both binary edges and hyperedges – and in an "arrow" style, which only allows binary edges to be defined but which is much more convenient in most circumstances. In both cases edges have a label and, optionally, a name, in the grammar called an edge annotation:

edge_annot → (*name* `":"`)? *label*

In the functional style, edges are defined by definitions of the form '$A(n_1,\ldots,n_k)$', where A is the annotation (label and optionally the name) of the edge and $n_1,\ldots,n_k$ are the names of the nodes incident to the edge.

edge → *edge_annot* `"("` *inc_nodes*? `")"`
inc_nodes → *name* (`","` *name*)∗

In the arrow style binary edges are defined by entries of the form '*src* `--`*A*`->` *tgt*' and '*tgt* `<-`*A*`--` *src*', which denote binary edges from node *src* to node *tgt*. These entries can be chained; for example: '`x --A-> y <-B-- z`'.

binedge → *name* (*direction name*)∗
direction → `"--"` *edge_annot* `"->"` | `"<-"` *edge_annot* `"--"`

For both the functional style and the arrow style it holds that if one of the node names does not yet occur in the graph, a node of that name is added to the graph.

Names. Within a single graph, each name can occur only once. That is, for each name there can be at most on node or edge with that name. If a node name occurs in a graph definition multiple times, then it refers to the same node in all cases. In particular, the edge definition '`x --A-> x`' defines an `A`-labeled loop on the node named `x`. On the other hand, each edge can only be defined once. Defining two edges with the same name will result in an error, even if they have the same label in both cases.

4.2.4 Morphisms

Morphisms are defined by the keyword '`morphism`' as follows:

[1] Note, however, that an actual *copy* is made of the object; thus the object will be represented in memory twice.

```
morphism → "morphism"
               ("from" name)?
               ("to" name)?
               "{" mappings? "}"
mappings → mapping ( ";" mapping )*
mapping  → name "=>" name
```

Every morphism has a domain graph (given after the ‘from’ keyword) and a codomain graph (given after the ‘to’ keyword). In some cases (for example when defining a rule) the domain and codomain can be determined from the context; in these cases explicitly specifying the domain and the codomain is optional. In all other cases the domain and codomain must be specified.

The body of a morphism definition consists of a semicolon-separated list of individual mappings which map nodes of the domain graph to nodes of the codomain and edges of the domain to edges of the codomain. When mapping an edge it is not necessary to explicitly map the incident nodes of that edge also; this is done automatically.

If the specified mapping does not represent a (partial) morphism – in particular this happens when a node or an edge of the domain is mapped to two different nodes or edges of the codomain – an error is reported.

4.2.5 Transformation rules

A transformation rule[2] consists of a left-hand side, a right-hand side and a correspondence morphism which associates a subset of the nodes and edges of the left-hand side with nodes and edges of the right-hand side. There are two ways in which rules can be specified: firstly, the three components of a rule can be given explicitly, and secondly there is a short-hand form where the correspondence morphism is automatically constructed from the names of the nodes and edges of the left-hand side and the right-hand side.

[2] In addition to these three components, SGF allows the definition of negative application conditions. Since they are ignored by *Grez*, they are not described here.

```
rule → exp_rule | short_rule
```

Explicit form. In the explicit form, a rule is defined by the following grammar:

```
exp_rule  → "rule" "{"
              ( "left" "=" graph_ref
              | "right" "=" graph_ref
              | "morphism" "="
                  morph_ref
graph_ref → graph | name
morph_ref → morphism | name
```

The three components of a rule are explicitly specified, either by referring by name to a previously defined graph or morphism, or by defining the components in-place (the `graph` and `morphism` non-terminals are defined in § 4.2.3 and § 4.2.4, respectively). The left-hand side, right-hand side and morphism can be given in any order, and from their definition until the end of the rule declaration scope, `left`, `right` and `morphism` function as names for the respective objects.

Short-hand form. In the short-hand form, a rule is defined by the following grammar:

```
short_rule → "rule" "{"
                "{" graph_desc "}"
                "=>"
                "{" graph_desc "}"
             "}"
```

The definition of graphs is the same as in § 4.2.3. The correspondence morphism is automatically generated from the graph definitions by mapping each node and edge of the left-hand side to the node or edge of the right-hand side with the same name, if it exists.

4.2.6 Transformation systems

A transformation system is a set of rules. Transformation systems are defined by the following grammar:

```
gts      → "gts" "{"
           "rules" "=" "["
             rules?
           "]" ";" "}"
rules    → rule_ref ("," rule_ref)∗
rule_ref → rule | name
```

The list of rules is a comma-separated list in which each entry is an in-place rule definition (the non-terminal `rule` is defined in § 4.2.5) or the name of a previously defined rule.

4.3 Examples

4.3.1 Two systems in one file

It is possible to define two or more graph transformation systems in a single SGF file. This can be useful if the two systems share certain graphs or rules. For example:

```
A = graph { 1 --A-> 2 };
B = graph { 1 --B-> 2 };

AtoB = rule {
  left = A;
  right = B;
  morphism = morphism {
    1 => 1;
    2 => 2;
  }
}

BtoA = rule {
  left = B;
  right = A;
  morphism = morphism {
    1 => 1;
    2 => 2;
  }
}

SysAtoB = gts {
  rules = [ AtoB ];
};

SysBtoA = gts {
  rules = [ BtoA ];
};

SysAeqB = gts {
  rules = [ AtoB, BtoA ];
};
```

If the user tries to open an SGF file which contains two graph transformation systems, she must specify the name of the system she wants to open, either by using the `--system=<name>` command-line option or selecting it in the open dialog of the graphical user interface.

4.3.2 In-place definition

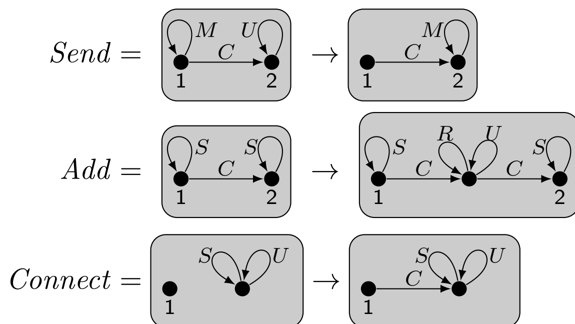
In almost all cases where you can use the name of a previously defined object, you can also define that object in-place. (The exception to this rule is in the definition of a morphism: after the `from` and `to` keywords you can only use the name of a previously defined graph.) For example:

```
MakeTriangle = gts {
  rules = [
    rule {
      { T(1,2,3) }
      =>
      { 1 --A-> 2 --A-> 3 --A-> 1 }
    }
  ];
}
```

If you define an object in place, you cannot give it a name and thus you cannot refer to it later. Defining an object in-place can be advantageous if you need an object only once and do not want to introduce a new name.

4.3.3 Ad-hoc routing

The following example is adapted from [5]. The example describes a simple ad-hoc routing protocol in a dynamically changing network. A message (M) traverses a network of servers (S), routers (R) and directed connections (C) between them. The message can only be sent to unvisited (U) nodes. In addition, rules which modify the network's layout are active. The graph transformation system which models the protocol consists of the following rules:



The numbers below the nodes indicate what nodes are mapped to each other by the correspondence morphism.

This graph transformation system is represented by the following SGF code:

```
Send = rule {
  { 1 --M-> 1 --C-> 2 --U-> 2 }
  =>
  { 1 --C-> 2 --M-> 2 }
};

Add = rule {
  { 1 --S-> 1 --C-> 2 --S-> 2 }
  =>
  { 1 --S-> 1 --C-> n
      --C-> 2 --S-> 2;
    n --R-> n --U-> n; }
};

Connect = rule {
  { node 1;
    r --S-> r --U-> r; }
  =>
  { 1 --C-> n;
    n --S-> n --U-> n; }
};

AdHocRouting = gts {
  rules = [Send,Add,Connect];
};
```

Five

Theoretical Foundations

In this chapter we present some of the theoretical foundations of the termination proving techniques used in *Grez*. We have choses to keep the text on a high level, referring to the literature for the details.

5.1 Graph transformation

Grez uses so-called *hypergraphs*. A hypergraph differs from a normal graph in that edges can be incident to any number of nodes. Formally, a hypergraph G consists of a finite set of nodes V_G (sometimes also called vertices in the literature) and a finite set of edges E_G disjoint from V_G. Each edge $e \in E_G$ has a label $lab_G(e)$ and a sequence of incident nodes $att_G(e) \subseteq V_G^*$. Note that in this approach nodes do not have labels; they solely serve as a means to connect edges. In the following a hypergraph will be simply called a *graph*.

We will sometimes confuse the graph G with the set of its components: $G = V_G \cup E_G$. In this chapter we will additionally assume – without loss of generality – that, for two graphs G and H, V_G and E_H are disjoint (in other words, we assume global, disjoint universes of nodes and edges).

The graph labels come from a fixed set Λ, called the *signature*. In *Grez*, the signature consists of all identifiers, that is, strings consisting of letters, digits and some special characters (see § 4.2.1) can function as labels. The subset of labels that occur in a graph G is called the *signature of* G.

If G is a graph, e is an edge of G and v a node of G, then we will say that e and v are *incident* if $v \in att_G(e)$. Furthermore, two edges e_1 and e_2 are *adjacent* if there exists a node v which is incident to both e_1 and e_2. Similarly, two nodes v_1 and v_2 are called *adjacent* if there exists an edge which is incident to both v_1 and v_2.

A *morphism* f from a graph G to a graph H is a function which maps nodes of G to nodes of H and edges of G to edges of H, such that, for each edge $e \in E_G$, $lab_G(e) = lab_H(f(e))$ and $f(att_G(e)) = att_H(f(e))$.[1]

Grez employs the so-called *double pushout approach* to graph transformation [6]. However, for performance and convenience reasons, graph transformation rules are represented similar to *single pushout* rules. A graph transformation rule consists of two graphs, a left-hand side L and a right-hand side R, and a partial, injective morphism from L to R. The morphism f will be called the *correspondence morphism*. Without loss of generality it will always be assumed that the correspondence morphism f is of the following

[1] In this case, f is extended to sequences of nodes as follows: $f(v_1 \cdots v_n) = f(v_1) \cdots f(v_n)$.

form:

$$f(x) = \begin{cases} x & \text{if } x \in L \cap R \\ \text{undefined} & \text{otherwise.} \end{cases}$$

Using this assumption, the equivalent rule in the normal double pushout style is obtained by taking $L \cap R$ as the interface and the two injections as the rule span.

A graph transformation step can be algorithmically described as follows. Let a rule $\rho = L \to R$ with correspondence morphism f be given. Suppose there exists an injective morphism m, the *match*, from L into some graph G. The rule can be applied to m if for every edge $e \in E_G$ not in the image of m and node $v \in att_G(e)$ it holds, that if v is in the image of m, f is defined on the pre-image of v under m. In this case we say that the *dangling edge condition* is satisfied. If the rule is applicable, all images of elements in L for which f is undefined are removed from G. This gives us the "context" graph C. Furthermore the elements of R that do not have a preimage in L are added and connected with the remaining elements, as specified f. This results in the graph H, and we write $G \Rightarrow_\rho H$. The dangling edge condition ensures that nodes can only be deleted if all incident edges are deleted.

Finally, a *graph transformation system* is a finite set of graph transformation rules. The rewrite relation induced by the graph transformation system is the union of the rewrite relations induced by its rules.

5.2 Termination

There are two kinds of termination: *uniform* and *non-uniform termination*. A graph transformation system is uniformly terminating if it does not allow an infinite transformation sequence $G_0 \Rightarrow G_1 \Rightarrow \cdots$, where G_0 is an arbitrary graph. A graph transformation system is non-uniformly terminating with respect to a set of graphs $\mathcal{L}$ if it does not allow an infinite transformation sequence $G_0 \Rightarrow G_1 \Rightarrow \cdots$ with $G_0 \in \mathcal{L}$. *Grez* only proves uniform termination (although in future versions also non-uniform termination might be supported).

A lot of research has been done on techniques for proving (uniform) termination of rewrite systems, in particular of term and string rewrite systems. An overview of such techniques can be found in [8, Chap. 6]. In general, termination of a term or string rewrite system is proven by finding a suitable well-founded ordering $>$ on the terms or strings. In particular, it is convenient if the ordering has the following property: if $l > r$ for some rule $\rho = l \to r$, and $s \Rightarrow_\rho t$, then $s > t$. Now, if we want to show that a term or string rewrite system is terminating, it suffices to check that $l > r$ for each rule of the system.

In graph transformation the rules do only consist of a left-hand side and a right-hand side, but also of the correspondence morphism. So we do not want to compare the left-hand side and the right-hand side in isolation, but also want to consider the correspondence morphism to get stronger termination arguments. We will call a rule ρ *decreasing* (resp. *non-increasing*) if it holds that $G \Rightarrow_\rho H$ implies $G > H$ (resp. $G \geq H$), where $>$ is a well-founded ordering on graphs which depends on the specific technique and $\geq$ is the reflexive closure of $>$. Now it holds that a graph transformation system is terminating if all its rules are decreasing.

A useful notion is that of *relative termination*. Let $\mathcal{R}$ be a set of graph transformation rules (that is, $\mathcal{R}$ is a graph transformation system). If $\mathcal{R}$ can be partitioned into two sets $\mathcal{R}^>$ and $\mathcal{R}^= = \mathcal{R} \backslash \mathcal{R}^>$ such that all rules of $\mathcal{R}^>$ are decreasing and all rules of $\mathcal{R}^=$ are non-increasing, then $\mathcal{R}$ is terminating if and only if $\mathcal{R}^=$ is terminating. Often, proving that $\mathcal{R}^=$ is terminating is considerably easier than proving that $\mathcal{R}$ is terminating. Thus, iterative termination proofs are produced. Many termination techniques employed by *Grez* support

relative termination.

5.3 Satisfiability modulo theories[2]

Some algorithms used by *Grez* solve constraints by translating them into a *satisfiability modulo theories* (SMT) formula and then calling an external SMT solver. Here we briefly describe SMT.

The SMT problem is a decision problem for logical formulas with respect to combinations of background theories expressed in classical first-order logic with equality. Examples of theories typically used in computer science are the theory of real numbers, the theory of integers, and the theories of various data structures such as lists, arrays, bit vectors and so on. SMT can be thought of as a form of the constraint satisfaction problem and thus a certain formalized approach to constraint programming.

Formally speaking, an SMT instance is a formula in first-order logic, where some function and predicate symbols have additional interpretations, and SMT is the problem of determining whether such a formula is satisfiable. In other words, imagine an instance of the Boolean satisfiability problem (SAT) in which some of the binary variables are replaced by predicates over a suitable set of non-binary variables. A predicate is basically a binary-valued function of non-binary variables. Example predicates include linear inequalities (for example $3x + 2y - z \geq 4$) or equalities involving uninterpreted terms and function symbols (for example $f(f(u, v), v) = f(u, v)$ where f is some unspecified function of two unspecified arguments.) These predicates are classified according to the theory they belong to. For instance, linear inequalities over real variables are evaluated using the rules of the theory of linear real arithmetic, whereas predicates involving uninterpreted terms and function symbols are evaluated using the rules of the theory of uninterpreted functions with equality (sometimes referred to as the empty theory). Other theories include the theories of arrays and list structures (useful for modeling and verifying software programs), and the theory of bit vectors (useful in modeling and verifying hardware designs). Most SMT solvers support only quantifier free fragments of their logics.

Grez employs only the theories of linear integer arithmetic (beta algorithms also non-linear integer arithmetic) and uninterpreted functions, using quantifier-free logic.

[2]This section is adapted from Wikipedia [10].

5.4 Algorithms for proving termination and non-termination

5.4.1 Finding cycles

Grez implements a simple cycle finder which exhaustively generates all transition sequences up to a given length which start with an arbitrary initial graph with a given size. If a cycle exists among these finitely many transition sequences this constitutes a proof of non-termination.

5.4.2 Node counting

The node counting technique is a simple termination technique which supports relative termination. Let $\#(G)$ denote the number of nodes of a graph G. This induces a well-founded order on graph by counting the number of nodes.

A rule $\rho = L \to R$ is *node-decreasing* if $\#(L) > \#(R)$ and it is *non-node-increasing* if $\#(L) \geq \#(R)$. Clearly, when $G \Rightarrow_\rho H$, $\#(G) > \#(H)$ if ρ is node-decreasing and $\#(G) \geq \#(H)$ if ρ is non-node-increasing. This leads to the following termination argument:

Theorem 5.4.1. *If $\mathcal{R}$ and $\mathcal{S}$ are graph transformation systems such that all rules of $\mathcal{R}$ are node-decreasing and all rules of $\mathcal{S}$ are non-node-*

increasing, then $\mathcal{R} \cup \mathcal{S}$ is terminating if and only if $\mathcal{S}$ is.

The *node counter* algorithm used by *Grez* uses this termination argument. It just checks if the condition applies and returns a termination proof if it does.

5.4.3 Edge counting

The edge counting technique is a simple termination technique which supports relative termination. Let $S \subseteq \Lambda$ be a set of labels; then, for a graph G, $\#_S(G)$ denotes the number of edges in G which are labelled with a label from S.

A rule $\rho = L \to R$ is *edge-decreasing* with respect to S if $\#_S(L) > \#_S(R)$ and it is *non-edge-increasing* with respect to S if $\#_S(L) \geq \#_S(R)$. Clearly, when $G \Rightarrow_\rho H$, $\#_S(G) > \#_S(H)$ if ρ is edge-decreasing with respect to S and $\#_S(G) \geq \#_S(H)$ if ρ is non-edge-increasing with respect to S. This leads to the following termination argument:

Theorem 5.4.2. *Let $S \subseteq \Lambda$ be a set of labels. If $\mathcal{R}$ and $\mathcal{S}$ are graph transformation systems such that all rules of $\mathcal{R}$ are edge-decreasing with respect to S and all rules of $\mathcal{S}$ are non-edge-increasing with respect to S, then $\mathcal{R} \cup \mathcal{S}$ is terminating if and only if $\mathcal{S}$ is.*

The *edge counter* algorithm used by *Grez* uses this termination argument. It tries all subsets of the graph transformation system's signature and returns a termination proof if the above condition holds for one of those subsets.

5.4.4 Petri-net approximation

In [9] a termination technique was presented which over-approximates the transition sequences of a graph transformation system by a Petri-net. If the Petri-net is terminating, which is the case if a certain system of linear inequalities does not have any trivial solutions, then the graph transformation system is terminating as well. In fact, the paper [9] extends this result to graph transformation systems with negative application conditions (of a certain kind), but since *Grez* does not support negative application conditions at this point, this extension was not implemented.

The Petri-net that approximates the graph transformation system has a place for each label in the signature and a transition for each graph transformation rule. The pre- and post-conditions of the Petri-net are as follows: for each edge labelled with a label A in the left-hand side of a rule ρ the place corresponding to A is added as a pre-condition to the transition corresponding to ρ, and, for each edge labelled with a label B in the right-hand side of a rule ρ the place corresponding to B is added as a post-condition to the transition corresponding to ρ. Note that the Petri-net only models the number of edges with a certain label; it does not model the structure of the graph.

Clearly, each transformation sequence of the graph transformation system induces a firing sequence of the Petri-net of the same length. Thus, termination of the Petri-net implies termination of the graph transformation system – but not vice versa.

A non-termination Petri-net is called *weakly repetitive* in the Petri-net literature. To check whether a Petri-net is weakly repetitive, the following well-known result can be used:

Proposition 5.4.3. *Let A be the incidence matrix of a Petri-net M. Then M is weakly repetitive if there exists a vector z, with $z > 0$ and $z \neq 0$, such that $A^T \times z \geq 0$.*

Grez's Petri-net approximation algorithm uses an external SMT solver to check whether the inequality of the above proposition has any solutions (since it is a *linear* inequality, this is a decidable problem).

5.4.5 Type graph techniques

A *type graph* for a graph transformation system $\mathcal{R}$ is a graph T, such that for each reachable graph G there exists a morphism from G to T. (Since *Grez* only supports uniform termination, effectively this means that for *each* graph G there exists a morphism from G to T.) We additionally require that for each rule $\rho \in \mathcal{R}$, (where $\rho = L \to R$ and ρ has correspondence morphism f) and morphism $m\colon L \to T$, there exists a morphism $m'\colon R \to T$ such that $(m' \circ f)(x) = m(x)$ for all x for which f is defined.

A *weighted type graph* is a type graph where each node and edge has been given a weight $\in \mathbb{N}$.

Rules can be evaluated with respect to a weighted type graph in two ways: tropically and arcticly (see below). In each of these cases a graph transformation system is proved to be terminating by finding a weighted type graph which satisfies the properties required by the chosen evaluation type. *Grez*'s type graph algorithm supports two methods for find (tropically, arcticly or arithmetically evaluated) weighted type graphs: the first is a naive implementation which exhaustively enumerates all weighted type graphs up to a certain number of nodes and maximum weight; the second encodes the properties as an SMT formula and runs an external SMT solver to find a weighted type graph with k nodes satisfying these properties (where k is a user-specified parameter).

Tropical evaluation. This termination technique was introduced in [5]. The technique supports relative termination.

Given a weighted type graph T, the weight of a morphism $m\colon G \to T$, denoted $wt(m)$, is the sum of all the weights in its image. Then a rule $\rho = L \to R$, with correspondence morphism f, is called *tropically decreasing* with respect to T if for each morphism $m_L\colon L \to T$ there exists a morphism $m_R\colon R \to T$ which agrees on f such that $wt(m_L) > wt(m_R)$; it is called *tropically non-increasing* with respect to T if for each morphism $m_L\colon L \to T$ there exists a morphism $m_R\colon R \to T$ which agrees on f such that $wt(m_L) \geq wt(m_R)$.

The following termination argument was proved in [5]:

Theorem 5.4.4. *Let T be a weighted type graph. If $\mathcal{R}$ and $\mathcal{S}$ are graph transformation systems such that all rules of $\mathcal{R}$ are tropically decreasing with respect to T and all rules of $\mathcal{S}$ are tropically non-increasing with respect to T, then $\mathcal{R} \cup \mathcal{S}$ is terminating if and only if $\mathcal{S}$ is.*

Arctic evaluation This termination technique was introduced in [5]. It is very similar to the tropical evaluation, except that rules are evaluated "from left to right". As the tropical evaluation, this technique supports relative termination.

Given a weighted type graph T, the weight of a morphism $m\colon G \to T$ is defined in the same way as with the tropical evaluation. A rule $\rho = L \to R$, with correspondence morphism f, is called *arcticly decreasing* with respect to T if for each morphism $m_R\colon R \to T$ there exists a morphism $m_L\colon L \to T$ which agrees on f such that $wt(m_L) > wt(m_R)$; it is called *arcticly non-increasing* with respect to T if for each morphism $m_R\colon R \to T$ there exists a morphism $m_L\colon L \to T$ which agrees on f such that $wt(m_L) \geq wt(m_R)$.

The following termination argument was proved in [5]:

Theorem 5.4.5. *Let T be a weighted type graph. If $\mathcal{R}$ and $\mathcal{S}$ are graph transformation systems such that all rules of $\mathcal{R}$ are arcticly decreasing with respect to T and all rules of $\mathcal{S}$ are arcticly non-increasing with respect to T, then $\mathcal{R} \cup \mathcal{S}$ is terminating if and only if $\mathcal{S}$ is.*

Arithmetical evaluation. This termination technique was introduced in [4]. The technique supports relative termination.

Given a weighted type graph T, the weight of a graph G , denoted $wt(G)$, is the sum of the weights of all morphisms $m\colon G \to T$, denoted $wt(m)$ which is the product of all the weights in its image. Then a rule $\rho = L \leftarrow\!\varphi_L\!- I -\!\varphi_R\!\to R$, with morphism $f\colon I \to T$, is called *arithmetically non-increasing* with respect to T if for all f it holds that $\sum_{\substack{m_L\colon L\to T\\ m_L\circ\varphi_L=f}} wt(m_L) \geq \sum_{\substack{m_R\colon R\to T\\ m_R\circ\varphi_R=f}} wt(m_R)$; it is called *arithmetically decreasing* with respect to T if the sum of weights for m_L is $>$ than the sum of weights for m_R with respect to $f'\colon I \to T$ where f' maps all nodes onto the flower node of T.

The following termination argument was proved in [4]:

Theorem 5.4.6. *Let T be a weighted type graph. If $\mathcal{R}$ and $\mathcal{S}$ are graph transformation systems such that all rules of $\mathcal{R}$ are arithmetically decreasing with respect to T and all rules of $\mathcal{S}$ are arithmetically non-increasing with respect to T, then $\mathcal{R} \cup \mathcal{S}$ is terminating if and only if $\mathcal{S}$ is.*

5.4.6 Match bounds

The match bound technique implemented in *Grez* is a slightly improved version of the technique presented in [3]. The idea is to annotate each edge with a *match height*, informally the number of rule applications responsible for creating the edge in question. If the match height of the graph transformation system is bounded, then the graph transformation system is terminating.

Formally, annotating a graph transformation system is done by constructing a new graph transformation systems which uses $\Lambda \times \mathbb{N}$ as label set, where Λ is the label set of the original system. When an occurrence of a left-hand side is replaced by a right-hand side, the new edges are annotated with a creation height which is equal to the smallest creation height of the left-hand side plus one. Now, the termination argument is as follows: if there exists a type graph (with annotated edges) for the annotated graph transformation system, then the match bound is bounded (because the type graph is finite by definition) and thus the original graph transformation system is terminating. See [3] for more details and formal definitions and proofs.

Grez's *match bound* algorithm uses the following algorithm to try to find a type graph for the annotated graph transformation system:

1. Start with the flower graph, that is the graph which consists of a single node v and for each label and edge which is only connected (as many times as the arity prescribes) to v. All edges are annotated with the creation height 0.
2. Find an occurrence of an annotated left-hand side in the current type graph. If such an occurrence exists, *extend* the type graph with the right-hand side. The left-hand side is not removed from the type graph. Note that the annotations of all edges in the right-hand side is the minimum annotation of the left-hand side plus one.
3. Reduce the type graph by user-specified type graph reductions (see below).
4. Continue with step 2 as long as the type graph can be modified this way.

If the algorithm terminates, then a type graph for the annotated graph transformation system has been found and thus the original graph transformation system is terminating.

Grez implements the following type graph reductions:

- *Conservative minimization* – While the type graph has an endomorphism (a morphism from the type graph to itself) which is not an isomorphism, the type graph is replaced by the image of this endomorphism. This reduction is conservative in the following sense. Let T be a type graph and T' its

minimization. Then there exists a morphism $f\colon G \to T$ if and only if there exists a morphism $f'\colon G \to T'$.

- *Outgoing edge factorization* – If there are two edges with the same label, which are connected to the same node in port 0 (in binary edges, this is the source port), then the nodes connected to port 1 (the target port) of the two edges are merged. This is a non-conservative reduction. Let T be a type graph and T' its outgoing edge factorization. Then it is the case that the existence of a morphism $f\colon G \to T$ implies the existence of a morphism $f'\colon G \to T'$, *but not the other way around*. For some graph transformation systems this reduction is necessary for the algorithm to terminate, while in others it causes non-termination.

- *Incoming edge factorization* – The same as outgoing edge factorization, but now nodes are merged if they are the source nodes of two edges with the same label and the same target node.

Appendix A

License texts

A.1 BSD License

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
3. Neither the name of the copyright holder nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

This software is provided by the copyright holders and contributors "as is" and any express or implied warranties, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose are disclaimed. In no event shall the copyright holder or contributors be liable for any direct, indirect, incidental, special, exemplary, or consequential damages (including, but not limited to, procurement of substitute goods or services; loss of use, data, or profits; or business interruption) however caused and on any theory of liability, whether in contract, strict liability, or tort (including negligence or otherwise) arising in any way out of the use of this software, even if advised of the possibility of such damage.

A.2 Apache License, version 2.0

1. Definitions. "License" shall mean the terms and conditions for use, reproduction, and distribution as defined by Sections 1 through 9 of this document.

"Licensor" shall mean the copyright owner or entity authorized by the copyright owner that is granting the License.

"Legal Entity" shall mean the union of the acting entity and all other entities that control, are controlled by, or are under common control with that entity. For the purposes of this definition, "control" means (i) the power, direct or indirect, to cause the direction or management of such entity, whether by contract or otherwise, or (ii) ownership of fifty percent (50%) or more of the outstanding shares, or (iii) beneficial ownership of such entity.

"You" (or "Your") shall mean an individual or Legal Entity exercising permissions granted by this License.

"Source" form shall mean the preferred form for

making modifications, including but not limited to software source code, documentation source, and configuration files.

"Object" form shall mean any form resulting from mechanical transformation or translation of a Source form, including but not limited to compiled object code, generated documentation, and conversions to other media types.

"Work" shall mean the work of authorship, whether in Source or Object form, made available under the License, as indicated by a copyright notice that is included in or attached to the work.

"Derivative Works" shall mean any work, whether in Source or Object form, that is based on (or derived from) the Work and for which the editorial revisions, annotations, elaborations, or other modifications represent, as a whole, an original work of authorship. For the purpose of this License, Derivative Works shall not include works that remain separable from, or merely link (or bind by name) to the interfaces of, the Work and Derivative Works thereof.

"Contribution" shall mean any work of authorship, including the original version of the Work and any modifications or additions to that Work or Derivative Works thereof, that is intentionally submitted to Licensor for inclusion in the Work by the copyright owner or by an individual or Legal Entity authorized to submit on behalf of the copyright owner. For the purposes of this definition, "submitted" means any form of electronic, verbal, or written communication sent to the Licensor or its representatives, including but not limited to communication on electronic mailing lists, source code control systems, and issue tracking systems that are managed by, or on behalf of, the Licensor for the purpose of discussing and improving the Work, but excluding communication that is conspicuously marked or otherwise designated in writing by the copyright owner as "Not a Contribution."

"Contributor" shall mean Licensor and any individual or Legal Entity on behalf of whom a Contribution has been received by Licensor and subsequently incorporated within the Work.

2. Grant of Copyright License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable copyright license to reproduce, prepare Derivative Works of, publicly display, publicly perform, sublicense, and distribute the Work and such Derivative Works in Source or Object form.

3. Grant of Patent License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable (except as stated in this section) patent license to make, have made, use, offer to sell, sell, import, and otherwise transfer the Work, where such license applies only to those patent claims licensable by such Contributor that are necessarily infringed by their Contribution(s) alone or by combination of their Contribution(s) with the Work to which such Contribution(s) was submitted. If You institute patent litigation against any entity (including a cross-claim or counterclaim in a lawsuit) alleging that the Work or a Contribution incorporated within the Work constitutes direct or contributory patent infringement, then any patent licenses granted to You under this License for that Work shall terminate as of the date such litigation is filed.

4. Redistribution. You may reproduce and distribute copies of the Work or Derivative Works thereof in any medium, with or without modifications, and in Source or Object form, provided that You meet the following conditions:

a. You must give any other recipients of the Work or Derivative Works a copy of this License; and

b. You must cause any modified files to carry prominent notices stating that You changed the files; and

c. You must retain, in the Source form of any Derivative Works that You distribute, all copyright, patent, trademark, and attribution notices from the Source form of the Work, excluding those notices that do not pertain to any part of the Derivative Works; and

d. If the Work includes a "NOTICE" text file as part of its distribution, then any Derivative Works that You distribute must include a readable copy of the attribution notices contained within such NOTICE file, excluding those notices that do not pertain to any part of the Derivative Works, in at least one of the following places: within a NOTICE text file distributed as part of the Derivative Works; within the Source form or documentation, if provided along with the Derivative Works; or, within a display generated by the Derivative Works, if and wherever such third-party notices normally appear. The contents of the NOTICE file are for informational purposes only and do not modify the License. You may add Your own attribution notices within Derivative Works that You distribute, alongside or as an addendum to the NOTICE text from the Work, provided that such additional attribution notices cannot be construed as modifying the License.

   You may add Your own copyright statement to Your modifications and may provide additional or different license terms and conditions for use, reproduction, or distribution of Your modifications, or for any such Derivative Works as a whole, provided Your use, reproduction, and distribution of the Work otherwise complies with the conditions stated in this License.

5. Submission of Contributions. Unless You explicitly state otherwise, any Contribution intentionally submitted for inclusion in the Work by You to the Licensor shall be under the terms and conditions of this License, without any additional terms or conditions. Notwithstanding the above, nothing herein shall supersede or modify the terms of any separate license agreement you may have executed with Licensor regarding such Contributions.

6. Trademarks. This License does not grant permission to use the trade names, trademarks, service marks, or product names of the Licensor, except as required for reasonable and customary use in describing the origin of the Work and reproducing the content of the NOTICE file.

7. Disclaimer of Warranty. Unless required by applicable law or agreed to in writing, Licensor provides the Work (and each Contributor provides its Contributions) on an "*as is*" *basis, without warranties or conditions of any kind*, either express or implied, including, without limitation, any warranties or conditions of *title*, *non-infringement*, *merchantability*, or *fitness for a particular purpose*. You are solely responsible for determining the appropriateness of using or redistributing the Work and assume any risks associated with Your exercise of permissions under this License.

8. Limitation of Liability. In no event and under no legal theory, whether in tort (including negligence), contract, or otherwise, unless required by applicable law (such as deliberate and grossly negligent acts) or agreed to in writing, shall any Contributor be liable to You for damages, including any direct, indirect, special, incidental, or consequential damages of any character arising as a result of this License or out of the use or inability to use the Work (including but not limited to damages for loss of goodwill, work stoppage, computer failure or malfunction, or any and all other commercial damages or losses), even if such Contributor has been advised of the possibility of such damages.

9. Accepting Warranty or Additional Liability. While redistributing the Work or Derivative Works thereof, You may choose to offer, and charge a fee for, acceptance of support, warranty, indemnity, or other liability obligations and/or rights consistent with this License. However, in accepting such obligations, You may act only on Your own behalf and on Your sole responsibility, not on behalf of any other Contributor, and only if You agree to indemnify, defend, and hold each Contributor harmless for any liability incurred by, or claims asserted against, such Contributor by reason of your accepting any such warranty or additional liability.

A.3 Creative Commons Attribution 4.0 International Public License

By exercising the Licensed Rights (defined below), You accept and agree to be bound by the terms and conditions of this Creative Commons Attribution 4.0 International Public License ("Public License"). To the extent this Public License may be interpreted as a contract, You are granted the Licensed Rights in consideration of Your acceptance of these terms and conditions, and the Licensor grants You such rights in consideration of benefits the Licensor receives from making the Licensed Material available under these terms and conditions.

Section 1 – Definitions

a. **Adapted Material** means material subject to Copyright and Similar Rights that is derived from or based upon the Licensed Material and in which the Licensed Material is translated, altered, arranged, transformed, or otherwise modified in a manner requiring permission under the Copyright and Similar Rights held by the Licensor. For purposes of this Public License, where the Licensed Material is a musical work, performance, or sound recording, Adapted Material is always produced where the Licensed Material is synched in timed relation with a moving image.

b. **Adapter's License** means the license You apply to Your Copyright and Similar Rights in Your contributions to Adapted Material in accordance with the terms and conditions of this Public License.

c. **Copyright and Similar Rights** means copyright and/or similar rights closely related to copyright including, without limitation, performance, broadcast, sound recording, and Sui Generis Database Rights, without regard to how the rights are labeled or categorized. For purposes of this Public License, the rights specified in Section 2(b)(1)-(2) are not Copyright and Similar Rights.

d. **Effective Technological Measures** means those measures that, in the absence of proper authority, may not be circumvented under laws fulfilling obligations under Article 11 of the WIPO Copyright Treaty adopted on December 20, 1996, and/or similar international agreements.

e. **Exceptions and Limitations** means fair use, fair dealing, and/or any other exception or limitation to Copyright and Similar Rights that applies to Your use of the Licensed Material.

f. **Licensed Material** means the artistic or literary work, database, or other material to which the Licensor applied this Public License.

g. **Licensed Rights** means the rights granted to You subject to the terms and conditions of this Public License, which are limited to all Copyright and Similar Rights that apply to Your use of the Licensed Material and that the Licensor has authority to license.

h. **Licensor** means the individual(s) or entity(ies) granting rights under this Public License.

i. **Share** means to provide material to the public by any means or process that requires permission under the Licensed Rights, such as reproduction, public display, public performance, distribution, dissemination, communication, or importation, and to make material available to the public including in ways that members of the public may access the material from a place and at a time individually chosen by them.

j. **Sui Generis Database Rights** means rights other than copyright resulting from Directive 96/9/EC of the European Parliament and of the Council of 11 March 1996 on the legal protection of databases, as amended and/or succeeded, as well as other essentially equivalent rights anywhere in the world.

k. **You** means the individual or entity exercising the Licensed Rights under this Public License. **Your** has a corresponding meaning.

Section 2 – Scope

a. **License grant.**

1. Subject to the terms and conditions of this Public License, the Licensor hereby grants You a worldwide, royalty-free, non-sublicensable, non-exclusive, irrevocable license to exercise the Licensed Rights in the Licensed Material to:

   A. reproduce and Share the Licensed Material, in whole or in part; and

   B. produce, reproduce, and Share Adapted Material.

2. *Exceptions and Limitations.* For the avoidance of doubt, where Exceptions and Limitations apply to Your use, this Public License does not apply, and You do not need to comply with its terms and conditions.

3. *Term.* The term of this Public License is specified in Section 6(a).

4. *Media and formats; technical modifications allowed.* The Licensor authorizes You to exercise the Licensed Rights in all media and formats whether now known or hereafter created, and to make technical modifications necessary to do so. The Licensor waives and/or agrees not to assert any right or authority to forbid You from making technical modifications necessary to exercise the Licensed Rights, including technical modifications necessary to circumvent Effective Technological Measures. For purposes of this Public License, simply making modifications authorized by this Section 2(a)(4) never produces Adapted Material.

5. *Downstream recipients.*

   A. *Offer from the Licensor – Licensed Material.* Every recipient of the Licensed Material automatically receives an offer from the Licensor to exercise the Licensed Rights under the terms and conditions of this Public License.

   B. *No downstream restrictions.* You may not offer or impose any additional or different terms or conditions on, or apply any Effective Technological Measures to, the Licensed Material if doing so restricts exercise of the Licensed Rights by any recipient of the Licensed Material.

6. *No endorsement.* Nothing in this Public License constitutes or may be construed as permission to assert or imply that You are, or that Your use of the Licensed Material is, connected with, or sponsored, endorsed, or granted official status by, the Licensor or others designated to receive attribution as provided in Section 3(a)(1)(A)(i).

b. **Other rights.**

1. Moral rights, such as the right of integrity, are not licensed under this Public License, nor are publicity, privacy, and/or other similar personality rights; however, to the extent possible, the Licensor waives and/or agrees not to assert any such rights held by the Licensor to the limited extent necessary to allow You to exercise the Licensed Rights, but not otherwise.
2. Patent and trademark rights are not licensed under this Public License.
3. To the extent possible, the Licensor waives any right to collect royalties from You for the exercise of the Licensed Rights, whether directly or through a collecting society under any voluntary or waivable statutory or compulsory licensing scheme. In all other cases the Licensor expressly reserves any right to collect such royalties.

Section 3 – License Conditions Your exercise of the Licensed Rights is expressly made subject to the following conditions.

a. **Attribution.**

1. If You Share the Licensed Material (including in modified form), You must:
   A. retain the following if it is supplied by the Licensor with the Licensed Material:
      i. identification of the creator(s) of the Licensed Material and any others designated to receive attribution, in any reasonable manner requested by the Licensor (including by pseudonym if designated);
      ii. a copyright notice;
      iii. a notice that refers to this Public License;
      iv. a notice that refers to the disclaimer of warranties;
      v. a URI or hyperlink to the Licensed Material to the extent reasonably practicable;
   B. indicate if You modified the Licensed Material and retain an indication of any previous modifications; and
   C. indicate the Licensed Material is licensed under this Public License, and include the text of, or the URI or hyperlink to, this Public License.
2. You may satisfy the conditions in Section 3(a)(1) in any reasonable manner based on the medium, means, and context in which You Share the Licensed Material. For example, it may be reasonable to satisfy the conditions by providing a URI or hyperlink to a resource that includes the required information.
3. If requested by the Licensor, You must remove any of the information required by Section 3(a)(1)(A) to the extent reasonably practicable.
4. If You Share Adapted Material You produce, the Adapter's License You apply must not prevent recipients of the Adapted Material from complying with this Public License.

Section 4 – Sui Generis Database Rights Where the Licensed Rights include Sui Generis Database Rights that apply to Your use of the Licensed Material:

a. for the avoidance of doubt, Section 2(a)(1) grants You the right to extract, reuse, reproduce, and Share all or a substantial portion of the contents of the database;

b. if You include all or a substantial portion of the database contents in a database in which You have Sui Generis Database Rights, then the database in which You have Sui Generis Database Rights (but not its individual contents) is Adapted Material; and

c. You must comply with the conditions in Section 3(a) if You Share all or a substantial portion of the contents of the database.

For the avoidance of doubt, this Section 4 supplements and does not replace Your obligations under this Public License where the Licensed Rights include other Copyright and Similar Rights.

Section 5 – Disclaimer of Warranties and Limitation of Liability

a. **Unless otherwise separately undertaken by the Licensor, to the extent possible, the Licensor offers the Licensed Material as-is and as-available, and makes no representations or warranties of any kind concerning the Licensed Material, whether express, implied, statutory, or other. This includes, without limitation, warranties of title, merchantability, fitness for a particular purpose, non-infringement, absence of latent or other defects, accuracy, or the presence or absence of errors, whether or not known or discoverable. Where disclaimers of warranties are not allowed in full or in part, this disclaimer may not apply to You.**

b. **To the extent possible, in no event will the Licensor be liable to You on any legal theory (including, without limitation, negligence) or otherwise for any direct, special, indirect, incidental, consequential, punitive, exemplary, or other losses, costs, expenses, or damages arising out of this Public License or use of the Licensed Material, even if the Licensor has been advised of the possibility of such losses, costs, expenses, or damages. Where a limitation of liability is not allowed in full or in part, this limitation may not apply to You.**

c. The disclaimer of warranties and limitation of liability provided above shall be interpreted in a manner that, to the extent possible, most closely approximates an absolute disclaimer and waiver of all liability.

Section 6 – Term and Termination

a. This Public License applies for the term of the Copyright and Similar Rights licensed here. However, if You fail to comply with this Public License, then Your rights under this Public License terminate automatically.

b. Where Your right to use the Licensed Material has terminated under Section 6(a), it reinstates:

   1. automatically as of the date the violation is cured, provided it is cured within 30 days of Your discovery of the violation; or
   2. upon express reinstatement by the Licensor.

   For the avoidance of doubt, this Section 6(b) does not affect any right the Licensor may have to seek remedies for Your violations of this Public License.

c. For the avoidance of doubt, the Licensor may also offer the Licensed Material under separate terms or conditions or stop distributing the Licensed Material at any time; however, doing so will not terminate this Public License.

d. Sections 1, 5, 6, 7, and 8 survive termination of this Public License.

Section 7 – Other Terms and Conditions

a. The Licensor shall not be bound by any additional or different terms or conditions communicated by You unless expressly agreed.

b. Any arrangements, understandings, or agreements regarding the Licensed Material not stated herein are separate from and independent of the terms and conditions of this Public License.

Section 8 – Interpretation

a. For the avoidance of doubt, this Public License does not, and shall not be interpreted to, reduce, limit, restrict, or impose conditions on any use of the Licensed Material that could lawfully be made without permission under this Public License.

b. To the extent possible, if any provision of this Public License is deemed unenforceable, it shall be automatically reformed to the minimum extent necessary to make it enforceable. If the provision cannot be reformed, it shall be severed from this Public License without affecting the enforceability of the remaining terms and conditions.

c. No term or condition of this Public License will be waived and no failure to comply consented to unless expressly agreed to by the Licensor.

d. Nothing in this Public License constitutes or may be interpreted as a limitation upon, or waiver of, any privileges and immunities that apply to the Licensor or You, including from the legal processes of any jurisdiction or authority.

Bibliography

[1] Cvc4. http://cvc4.cs.nyu.edu/web/.

[2] Z3. http://z3.codeplex.com/.

[3] H.J. Sander Bruggink. Towards a systematic method for proving termination of graph transformation systems. In *Proceedings of GT-VC 2007*, 2007.

[4] H.J. Sander Bruggink, Barbara König, Dennis Nolte, and Hans Zantema. Proving termination of graph transformation systems using weighted type graphs over semirings. Submitted for ICGT 2015, 2015.

[5] H.J. Sander Bruggink, Barbara König, and Hans Zantema. Termination analysis of graph transformation systems. In *Proceedings of TCS 2014*, volume 8705 of *LNCS*. Springer, 2014.

[6] Andrea Corradini, Ugo Montanari, Francesca Rossi, Hartmut Ehrig, Reiko Heckel, and Michael Löwe. Algebraic approaches to graph transformation I: Basic concepts and double pushout approach. In Grzegorz Rozenberg, editor, *Handbook of Graph Grammars and Computing by Graph Transformation, vol. 1: Foundations*. World Scientific, 1997.

[7] Detlef Plump. Termination of graph rewriting is undecidable. *Fundementa Informaticae*, 33(2):201–209, 1998.

[8] Terese. *Term Rewriting Systems*, volume 55 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 2003.

[9] Dániel Varró, Szilvia Varró-Gyapay, Hartmut Ehrig, Ulrike Prange, and Gabriele Taentzer. Termination analysis of model transformations by petri nets. In *Proceedings of ICGT 2006*, volume 4178 of *LNCS*. Springer, 2006.

[10] Wikipedia. Satisfiability modulo theories. http://en.wikipedia.org/w/index.php?title=Satisfiability_Modulo_Theories&oldid=635242269, 2014.

Index

About menu item, 7, 12
action panel, 5
actions, 8–9
Add button, 9
adjacent, 25
Algorithm editor dialog, 9–10
Algorithm list editor dialog, 9
annotation, 20
arctic evaluation, 29

Change number of worker threads menu item, 7
Clear button, 9
command-line interface, 13–15
command-line options, 13, 14
  --action, 14
  --algos, 14
  --beta, 5, 14
  -c, 13
  --cli, 13
  --conn, 14
  --dens, 14
  --discrete, 14
  -h, 13
  --help, 13
  --if-size, 14
  --labels, 15
  --left-conn, 14
  --left-dens, 14
  --left-size, 14
  --progress, 14
  --random, 14
  --rfile, 14
  --right-conn, 14
  --right-dens, 14
  --right-size, 14
  --rtype, 14
  --rules, 14
  --silent, 14
  --system, 5, 7, 14
  --threads, 14
  -v, 13
  --version, 13
Compare action, 9
Complete button, 9
conservative minimization, 30
copy,
  texttttcopy19
Cycle finder algorithm, 27
cycle finder algorithm, 15
Cycle finder algorithms, 9
CycleFinder, 15

Default button, 9
double pushout approach, 25

edge counter algorithm, 10, 15, 28
edge counting, 28
edge factorization, 30
EdgeCounter, 15
Edit button, 9
Execute button, 6
Exit menu item, 6
External, 15
external tool algorithm, 10, 15

from, 20

Generate random GTS

dialog, 7–8
menu item, 6, 7
graph, 25
`graph`, 20
graph transformation rule, *see* rule
graph transformation system, 26
graphical user interface, 5–12
GTS, *see* graph transformation system
`gts`, 18, 21
GUI, *see* graphical user interface
GUI options menu item, 6

Help menu, 7, 12
hypergraph, *see* graph

incident, 25
incoming edge factorization, *see* edge factorization
installation, 2

main menu, 5, 6
main window, 5, 6
match bound algorithm, 10, 15, 30
match bounds, 29–30
`MatchBound`, 15
morphism, 25
`morphism`, 20
Move down button, 9
Move up button, 9

name of *Grez*, 1
`node`, 18, 20
node counter algorithm, 10, 15, 28
node counting, 27–28
`NodeCounter`, 15
non-uniform termination, *see* termination
Number of rules parameter, 7

Open GTS menu item, 6, 7
Options menu, 6
outgoing edge factorization, *see* edge factorization

Petri-net approximation, 28
Petri-net approximation algorithm, 10, 15, 28
`PetriNet`, 15
Prove action, 8

relative termination, *see* termination
Remove button, 9
rule, 25–26
`rule`, 17, 21
rule display, 5
rule selection box, 5

satisfiability modulo theories, 2, 27
SGF, *see* simple graph format
signature, 25
Signature parameter, 7
simple graph format, 17–23
comment, 19
defining edges, 18, 20
defining graph transformation systems, 18–19, 21–22
defining morphisms, 20–21
defining nodes, 18, 20
defining rules, 17–18, 21
identifier, 19
keyword, 19
SMT, *see* satisfiability modulo theories
SMT solver, 10–11
SMT solvers menu item, 6
status bar, 6
System menu, 6, 7

termination, 1, 26–27
non-uniform, 26
relative, 26
uniform, 26
termination tool, 10–11
Termination tools menu item, 6
`to`, 20
Trace action, 9
tropical evaluation, 29
type graph, 29
type graph algorithm, 10, 15, 29
`TypeGraph`, 15

uniform termination, *see* termination

Visit website menu item, 7, 12

weighted type graph, 29