Raven – A tool suite for computing and manipulating graph automata

April 25, 2014

Contents

1 Introduction

1.1 History

The implementation of RAVEN[1] started in 2008 with a tool which implements the graph automaton accepting all graphs containing a given subgraph [2]. The application of this tool was to check whether the language (accepted by the implemented graph automaton) is an invariant for a given graph transformation rule by computing the Myhill-Nerode quasi-order and checking whether the left-hand side of the transformation rule is related to the right-hand side (w. r. t. the Myhill-Nerode quasi-order).

This tool had two main drawbacks which led to the development of RAVEN. On the one hand the graph automaton had been implemented in an explicit fashion. The number of states grows exponentially in the size of the maximum permitted interface of the graph automaton, which has a direct impact on the size of the explicit representation. As shown by practical examples, it was not possible to compute graph automata which exceed a rather small maximum interface size. On the other hand one needs to build the deterministic graph automaton to be able to compute the Myhill-Nerode quasi-order. But since the computation of the deterministic graph automaton (by the powerset construction) is not feasible due to the exponential blow-up, the tool used the simulation quasi-order as an over-approximation. This quasi-order had also been represented explicitly which led to the problem that the simulation quasi-order had been computable only for graph automata up to a maximum interface size of 4, even for rather simple subgraphs [4].

1.2 Raven today

To overcome these issues the development of RAVEN has been started. The key to an efficient new tool has been the usage of BDDs as data structures to represent graph automata. The usage of a symbolical instead of an explicit representation tries to prevent the exponential state space blow-up. In addition algorithms for language inclusion checks, universality checks, emptiness checks, memberships checks and algorithms to compute atomic cospan decompositions for a given cospan (which includes graphs seen as cospans with empty inner and outer interfaces) have been implemented.

[1] Available at `http://www.ti.inf.uni-due.de/research/tools/raven/`

2 People

The following people (in alphabetic order) are or were involved either in the theoretical development or in the implementation of the tool:

- Christoph Blume (Universität Duisburg-Essen, Germany)
- H. J. Sander Bruggink (Universität Duisburg-Essen, Germany)
- Dominik Engelke (Universität Duisburg-Essen, Germany)
- Martin Friedrich (Universität Duisburg-Essen, Germany)
- Weixiang Guan (Universität Duisburg-Essen, Germany)
- Barbara König (Universität Duisburg-Essen, Germany)
- Dennis Nolte (Universität Duisburg-Essen, Germany)
- Laura Steinert (Universität Duisburg-Essen, Germany)

The current maintainer of the tool and the documentation is Christoph Blume. The web site of Raven is at `http://www.ti.inf.uni-due.de/research/tools/raven/`. Please address questions to `christoph.blume@uni-due.de`.

3 System and Software Requirements

3.1 The System Architecture

This section describes the system architecture of RAVEN which is depicted in Figure 3.1.

The architecture of RAVEN can roughly be divided into six parts:

- the input components (depicted on the left) which are again split up into two groups: the *user interface* and the *file readers*,
- the *repository* (depicted in the center) which is one of the core components of RAVEN, that is a database for all current objects,
- the *decomposer* unit (depicted on the bottom) which is used to transform graphs and cospans to equivalent atomic cospan decompositions,
- the *goal* components (depicted in the middle around the repository) which are also core components providing different techniques to perform universality, language inclusion, invariant, emptiness and membership checks,
- the *algorithms* unit (depicted on the top) which is used by the different goal components
- the output components (depicted on the right) which are also split up into two groups: the *user interface* and the *file writers*,

In the following the components are described in more detail.

The system starts by reading in the user's input. Depending on which goals the user wants to achieve different data structures must be provided. The data structures which can be handled by RAVEN are:

- *Graphs* can either be directly created by the user via the user interface or by loading a file in GXL format[1] (see Appendix 5),
- *Signatures* contain strings representing the letters $connect_*$, *fuse*, *perm*, *res*, *trans*, *vertex* (also called *atomic cospans*, see Table 4.1 on page 17) which are used to define the input alphabet of the different graph automata. Signatures can also be created in the user interface or by loading a file in a special line-oriented format,
- *Cospans* can be produced through the user interface or by loading a file in GXL format (see Appendix 5),

[1] GXL is a XML-based standard exchange language for different kinds of graphs widely spread in the graph transformation community [12, 9, 8], see also `http://www.gupro.de/GXL/`

- *(Atomic) Cospan Decomposition* can either be generated by the user calling the decomposer unit on a graph or a cospan, which will be explained in detail below, or by loading a file in a special line-oriented format,

- *Graph Automata* can be created by choosing an automaton type (out of a list of predefined automata) and defining the automaton's properties, such as the permitted inner, outer and maximum interface and some further specific properties, or by loading a graph automaton from a file (see Appendix 5 for further information).

After the data structures have been read, every object is stored under a certain name in the repository. Later on, the user can use the objects in the repository by handing over the names of the desired objects to the several goals.

Some of the goals, namely the membership and the invariant checking goal, expect the user to input an atomic cospan decomposition. Hence, Raven provides the opportunity to automatically decompose a graph or a cospan into such an atomic cospan decomposition. This is done in several steps depending on the object one wants to decompose:

- In case of graphs, the first step is to compute a tree decomposition of the graph. This is done by two algorithms. The first one is a heuristic which computes a list of the nodes of the graph depending on different criteria [5] (see also Table 4.3). The second algorithm generates the actual tree decomposition depending on the node list. The heuristic to find the node list can either be chosen by the user or the default heuristic, called GreedyDegree in the literature, is chosen.

  In the second step, the tree decomposition (which is essentially a tree) is traversed to obtain a linearization of the decomposition. In order to get a path decomposition, one needs to perform further operations, since it could be the case that the bags (of the linearization) containing a given node (of the graph) do not form a path, which would be a violation of the definition of a path decomposition. Therefore, it is checked for every graph node and for every path from one bag to another bag which both contain the given node, whether the node is also contained in all bags on the path. If this is not the case, the node is added to all bags on the path which do not contain the node. This yields a valid path decomposition of the graph.

  In the last step, the path decomposition is transformed into an atomic cospan decomposition. This is done in a "bag-wise" manner, i. e. the algorithm processes the path decomposition bag by bag and transforms the contents of each bag into an equivalent sequence of atomic cospans.

- In case of cospans, five different steps are required to obtain an atomic cospan decomposition. The first step is to add a *vertex*-operation for each node in the middle graph of the given cospan. Note that there are additional nodes which are accessible from the inner interface (of the given cospan). In the second step all edges of the middle graph are added by permuting the incident nodes to the last position of the outer interface with the help of *shift*- and *trans*-operations and a subsequent *connect*$_*$-operation, where $*$ is just a placeholder for the corresponding label. Next, in step three, the nodes added in step one are fused with the already existing nodes according to the inner interface (of the given cospan). At this point the middle graph obtained by the atomic cospan decomposition matches the middle graph of the given cospan. Hence, in step four, we can permute the outer interface (of the atomic cospan decomposition) by adding *shift*- and *trans*-operations repeatedly such that the nodes which are not accessible by the outer interface of the given cospan allocate the last positions of the outer interface of the atomic cospan decomposition. Then, the fifth and last step is to remove these last nodes from the outer interface by adding enough *res*-operations. This yields the desired atomic cospan decomposition.

An object in the repository can be used as input to the different goal components:

- The *universality* method checks whether the given graph automaton is universal, i. e. whether the accepted language of the $\langle i, j\rangle$-graph automaton contains all cospans of the form $c\colon D_i \leftrightarrow D_j$. In case that the given graph automaton is *not* universal, a counter-example is returned.
- The *language inclusion* goal expects two $\langle i, j\rangle$-graph automata and checks whether the language of the first graph automaton is contained in the language of the second graph automaton. In case the language inclusion does not hold, each algorithm computes a counter-example and returns it to the user.
- The *invariant checking* goal checks whether the language of a given $\langle 0, j\rangle$-graph automaton is an invariant for a graph transformation rule $\rho = \langle \ell, r\rangle$ given as two cospans of the form $\ell, r\colon \emptyset \leftrightarrow D_i$. The check is then based on the language inclusion goal for the graph automata $\mathcal{A}[\ell]$ and $\mathcal{A}[r]$. Again, if the language of the given graph automaton is not an invariant for the given transformation rule, a counter-example is computed and presented to the user.
- The *emptiness checking* goal expects a graph automaton and checks whether the language of the automaton is empty.
- The *membership checking* goal runs a given $\langle i, j\rangle$-graph automaton on a given cospan of the form $c\colon D_i \leftrightarrow D_j$ and checks whether the cospan is accepted by the graph automaton.

If the language inclusion goal has been chosen, one of the algorithms solve the language inclusion problem. Please note that in case of the simulation-based antichain algorithm, the simulation pre-order for the given automata is computed as a pre-processing step.

If the user has chosen the universality goal, there exist algorithms which are essentially the same as for the language inclusion goal (see [13, 1, 6] for further information about these universality algorithms).

If the invariant checking goal has been selected by the user, the input is transformed as described above and afterwards the chosen language inclusion algorithms is called.

The other two goals are directly implemented on top of the underlying data structures, which are given as input, i. e. both the emptiness and the membership

Library	License	URL
ANTLR	ANTLR 4 License	`http://www.antlr.org/`
BuDDy	Public Domain	`http://buddy.sourceforge.net/`
GraphViz	Eclipse Public License 1.0	`http://www.graphviz.org/`
JAnsi	Apache License 2.0	`http://jansi.fusesource.org/`
JavaBDD	GNU LGPL 2.0	`http://javabdd.sourceforge.net/`
JDOM	Apache-style License	`http://www.jdom.org/`
JGoodies	BSD 2-Clause License	`http://www.jdom.org/`
JGraphX	BSD 3-clause License	`http://www.jgraph.com/jgraph.html`
LibTW	Public Domain	`http://www.treewidth.com/`

Table 3.1: Libraries on which Raven depends

check goal call methods which are implemented in the graph automaton data structure. Therefore, no further algorithms are needed here.

At last, the system can either visualize the data structures contained in the repository using the different viewers or write the data structures in the different file formats which are described in Chapter 5.

3.2 The System Requirements

The tool suite Raven has been implemented in the Java programming language[2] and offers both a command-line and a graphical user interface. Furthermore, Raven depends on a number of libraries/programs which are listed in the following:

- JavaBDD, a Java library for manipulating Bdds which offers an interface to the well-known BuDDy library [11]
- BuDDy, a highly efficient Bdd library written in C [10]
- LibTW, a Java library for computing tree decompositions of graphs [7]
- ANTLR, a parser generator library written in Java,
- JDOM, a library for reading, manipulating and writing Xml documents written in Java
- JGraphX, a Java library for visualizing graphs
- JAnsi, a library written in Java to use ANSI escape sequences to format console outputs
- Graphviz, a program for automatically layouting graphs

All the libraries and programs given above are free software (see Table 3.1 for further information). All the libraries are shipped with Raven. Since Raven as well as many of the libraries above are written in Java and since BuDDy as well as Graphviz are available for many platforms, Raven can be used on Linux, MacOS and Windows.

[2]Raven depends on a Java Runtime Environment 1.7 or higher.

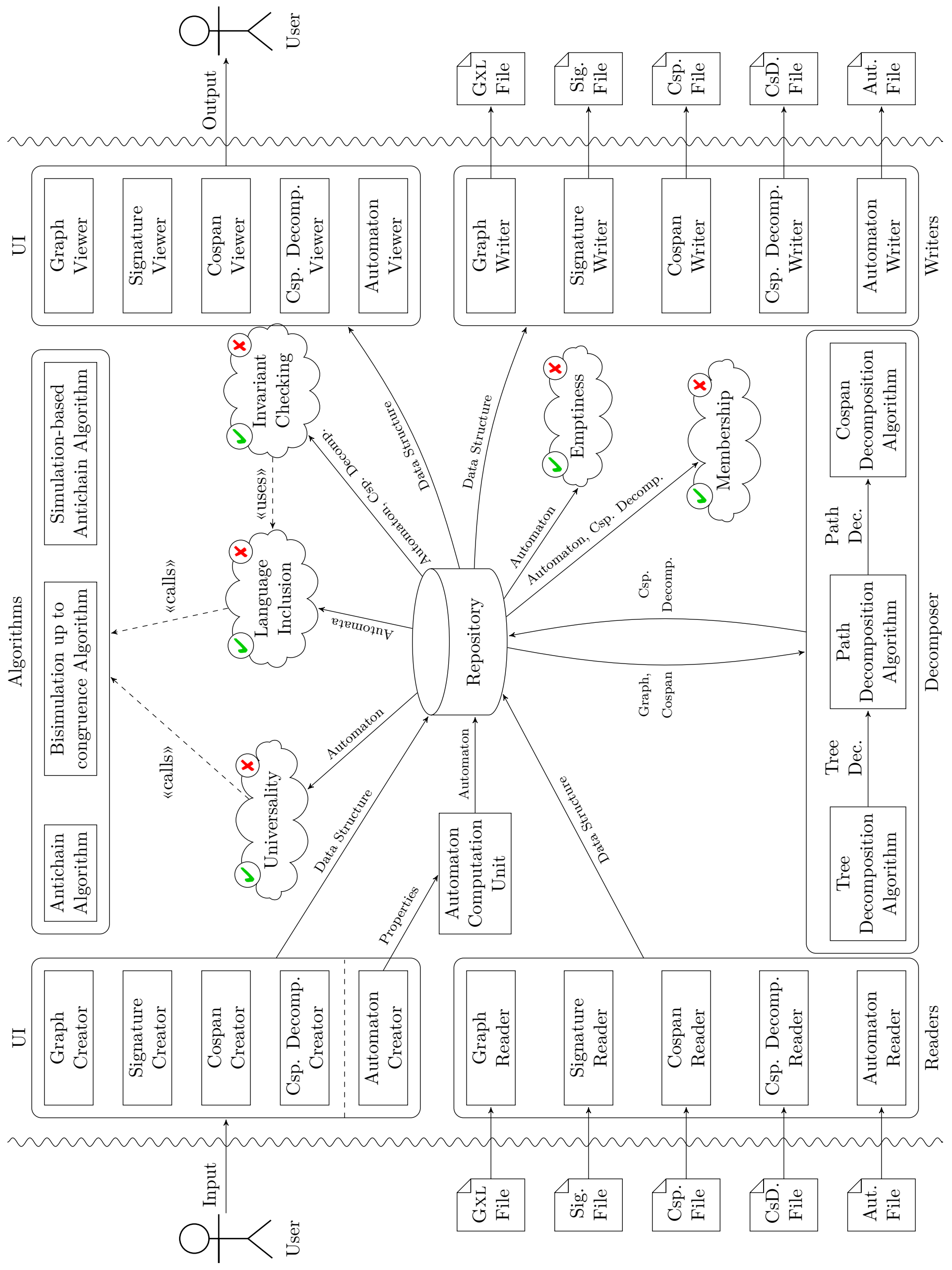


Figure 3.1: System Architecture of Raven

4 Installation, Usage and Functionality

4.1 Installation

After installing a Java Runtime Environment (JRE) Version 1.7 (or newer) and installing the Graphviz graph visualization software[1], no further installation is necessary to run Raven. Please make sure, that Raven is allowed to create new files and directories (in the directory where you installed Raven).

4.2 Usage

Raven offers both a graphical user interface (GUI) and a command-line interface (CLI). The recommended way of using Raven is with the GUI. To start the program with GUI the `raven.jar` can be directly executed or called from a shell by:

```
java -jar raven.jar
```

If you want to start Raven with the command-line interface, you can call

```
java -jar raven.jar -c [scriptFile]
```

from a shell, where `scriptFile` is an optional parameter which indicates a Raven script file contained in the `script`-directory. Please note, that the command-line is *not* interactive if you pass the `scriptFile`-parameter to Raven, i. e. Raven will be terminated after the execution of the given Raven script file.

4.3 Functionality

In this section we give a short introduction to the main features of Raven. For this we first start Raven with GUI, which brings up the main window of Raven (see Figure 4.1). This window consists of five main components:

① The *goal panel* on which the user can choose the different goals to use,

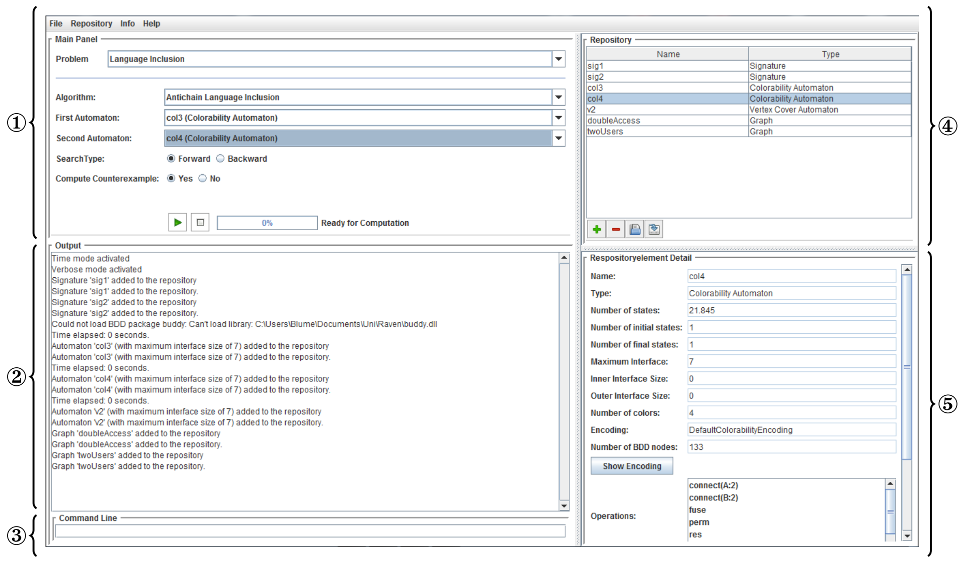
② The *output panel* which shows the user text-based status information about Raven,

③ The *command-line input field* which provides the user with the opportunity to directly use console commands in the graphical use interface,

④ The *repository panel* which presents a list of all data objects (signatures, cospans, . . .) which are currently contained in the repository,

⑤ The *data information panel* which gives detailed information about the currently selected data object.

[1]Graphviz is available at `http://www.graphviz.org/`



Main Panel

Problem	Language Inclusion
Algorithm:	Antichain Language Inclusion
First Automaton:	col3 (Colorability Automaton)
Second Automaton:	col4 (Colorability Automaton)
SearchType:	[✓ Forward] Backward
Compute Counterexample:	[✓ Yes] No

Ready for Computation

Output

```
Time mode activated
Verbose mode activated
Signature 'sig1' added to the repository
Signature 'sig1' added to the repository.
Signature 'sig2' added to the repository
Signature 'sig2' added to the repository.
Could not load BDD package buddy: Can't load library: C:\Users\Blume\Documents\Uni\Raven\buddy.dll
Time elapsed: 0 seconds.
Automaton 'col3' (with maximum interface size of 7) added to the repository
Automaton 'col3' (with maximum interface size of 7) added to the repository.
Time elapsed: 0 seconds.
Automaton 'col4' (with maximum interface size of 7) added to the repository
Automaton 'col4' (with maximum interface size of 7) added to the repository.
Time elapsed: 0 seconds.
Automaton 'v2' (with maximum interface size of 7) added to the repository
Automaton 'v2' (with maximum interface size of 7) added to the repository.
Graph 'doubleAccess' added to the repository
Graph 'doubleAccess' added to the repository.
Graph 'twoUsers' added to the repository
Graph 'twoUsers' added to the repository.
```

Command Line

Repository

Name	Type
sig1	Signature
sig2	Signature
col3	Colorability Automaton
col4	Colorability Automaton
v2	Vertex Cover Automaton
doubleAccess	Graph
twoUsers	Graph

Respositoryelement Detail

Name:	col4
Type:	Colorability Automaton
Number of states:	21.845
Number of initial states:	1
Number of final states:	1
Maximum Interface:	7
Inner Interface Size:	0
Outer Interface Size:	0
Number of colors:	4
Encoding:	DefaultColorabilityEncoding
Number of BDD nodes:	133
Operations:	connect(A:2) connect(B:2) fuse perm res

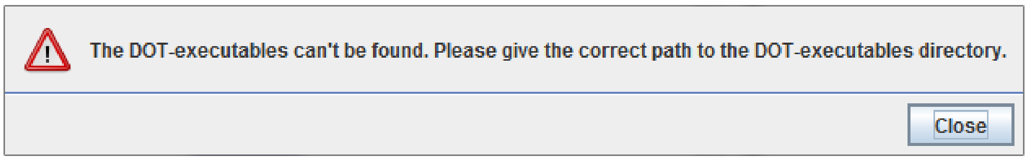
Figure 4.1: Raven GUI

4.3.1 Preparations

The first time we start Raven, there may pop up an information dialog as depicted in Figure 4.2. If this is the case, we have to configure the correct path to the Dot executable. To do so, first we choose `Info -> Variables and properties`, which brings up the *Variables and Properties* dialog (see Figure 4.3). We can now add new variables by pressing the button (✚) labelled with a green plus sign or remove non-default variables by pressing the button (▬) labelled with a red minus sign. There exists a default `dot` variable which needs to be configured by selecting the key and pressing the button (✍) labelled with the edit symbol. We have to enter the full path of the Dot executable which is needed to layout the graphs used by Raven. By closing the dialog all changes will be saved automatically. Now we can start with the creation of data structures, which we will analyse with some of the implemented algorithms.

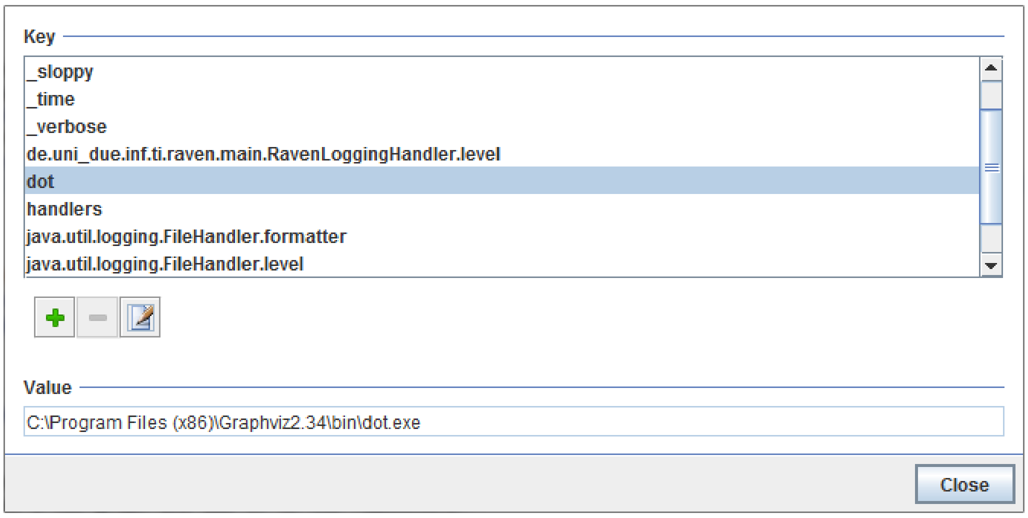
4.3.2 Creation of data structures

We start by creating some data structures. First, we create a new graph and add it to the repository. This can be done by choosing either `Repository -> Create graph` from the main menu or the item `Create graph` which appears when pressing the button (✚) labelled with a green plus sign located on the repository panel (④, Figure 4.1). The *visual graph creator* component appears, which is depicted in Figure 4.4. The use is rather intuitive. In the text field on



The DOT-executables can't be found. Please give the correct path to the DOT-executables directory.

Figure 4.2: Raven Dot Executable not found


Key

```
_sloppy
_time
_verbose
de.uni_due.inf.ti.raven.main.RavenLoggingHandler.level
dot
handlers
java.util.logging.FileHandler.formatter
java.util.logging.FileHandler.level
```

Value

```
C:\Program Files (x86)\Graphviz2.34\bin\dot.exe
```

Figure 4.3: RAVEN Variables and Properties

the top side of the window, the user has to define a (unique) name for the new graph which is used to address the graph after it has been added to the repository. By pressing the `Add node` button the user can add new nodes to the graph. A click on the `Add edge` button brings up a new dialog window on which the user can specify the (unique) name, the label and incident nodes of a new edge. The graph is then added to the repository by pressing the `Accept` button.

Next, we want to add a graph automaton. But before we can do this, we need to add a signature to define the automaton's input alphabet. Again, we either choose `Repository -> Create signature` from the main menu or the item `Create signature` which appears when pressing the button (✚) labelled with a green plus sign located on the repository panel. The *visual signature creator* appears, which is shown in Figure 4.5. First of all, the user has to define a (unique) name for the new signature. Subsequently the user can add the letters which should be contained in the signature by pressing the button (✚) labelled with a green plus sign. The available letters are given in the following table:

Letter	**Description**
$connect_*$	Adds a new edge labelled with $*$
fuse	Fuses the nodes accessible by the last two interface nodes
perm	Shifts the complete interface by one node
res	Restricts the last node from the interface
trans	Transposes the first two interface nodes
vertex	Adds a new node to the middle graph and to the outer interface

Table 4.1: Available letters for the input alphabet

Note that $*$ is a placeholder for an arbitrary label which can be determined by the user. If the user wants to remove a letter, this can be done by selecting the respective letter and pressing the button (━) labelled by a red minus sign. Once all operations have been added to the signature, the user can press the `Accept` button and add it to the repository.

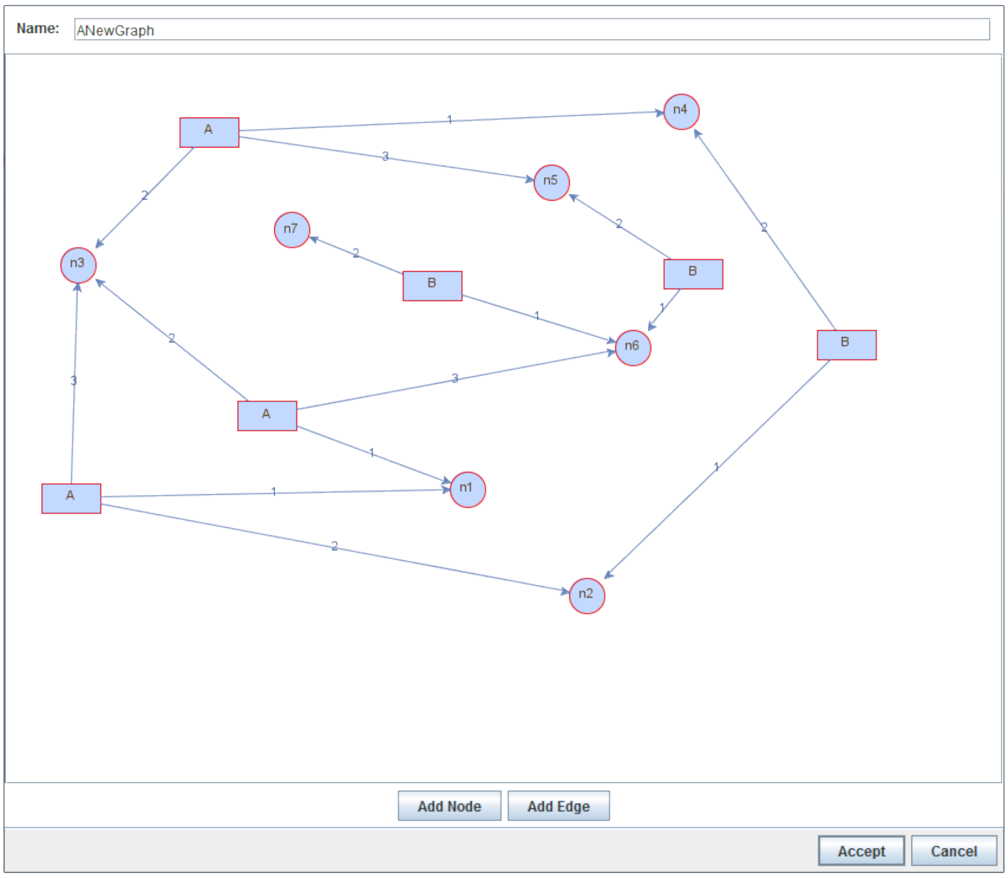


Figure 4.4: RAVEN Visual Graph Creator

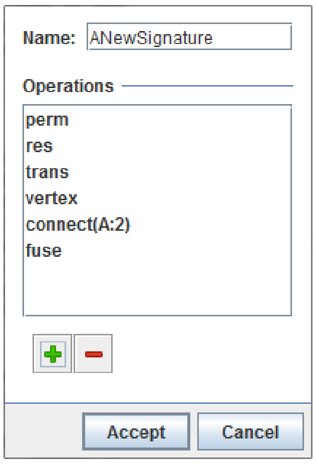
Now, we can create a new graph automaton. As before, the *visual automaton creator* dialog can be invoked either by the main menu, choosing `Repository -> Create graph automaton`, or by pressing the button (✚) labelled with a green plus sign located on the repository panel and selecting the `Create automaton` item. Either way the visual automaton creator, which is depicted in Figure 4.6, is shown to the user. Analogously to the dialogs explained above, first of all the user has to define a (unique) name for the new graph automaton. Next, the user can choose the type of the graph automaton, which specifies the language accepted by the automaton. The list consists of the following types:

Type	**Description**
Colorability	Graph automata which accept a cospan if and only if the middle graph G of the cospan can be colored by at most k colors, i.e. there exists a k-coloring such that no adjacent nodes have the same color
Dominating Set	Graph automata which accept a cospan if and only if the middle graph G of the cospan contains a dominating set of size at most k, i.e. a set D of nodes of G with size at most k such that each node of G is either in D or adjacent to a node in D

Edge/Vertex Counting	Graph automata which count the number of specific edges or nodes of the middle graph of the input cospan modulo a fixed divisor,
Link	Graph automata which check that the middle graph of the input cospan consists of an edge which is incident to at least one node of the inner interface and to at least one node of the outer interface of the input cospan
Maximum/Minimum Edge/Vertex	Graph automata accepting only cospans whose middle graph have a particular maximum (minimum) number of edges or nodes,
No Isolated Nodes	Graph automata which accept only those cospans whose middle graphs do not contain any isolated node
Path	Graph automata which check that there exists an $\langle S, T\rangle$-path in the middle graph of the input cospan, where S is a subset of the inner interface and T is a subset of the outer interface, i. e. there exists a path starting at some node contained in S to some other node contained in T
Product	Graph automata whose accepted language is the intersection of the languages of two given graph automata,
Subgraph	Graph automata which accept all cospans whose middle graphs contain a specific graph as a subgraph
Union	Graph automata whose accepted language is the union of the languages of two given graph automata,
Vertex cover	Graph automata accepting a cospan if and only if the middle graph G of the cospan has a vertex cover of size at most k, i. e. there exists a set C of nodes of G with size at most k such that each edge is incident to at least one node of C

Furthermore, the user has to specify the signature used by the new graph automaton as well as the inner, outer and maximum interface size of the cospans accepted by the automaton. However, it is recommended to choose the maximum interface size as low as possible, since the size (and therefore the time of computation) of a graph automaton depends exponentially on the maximum interface size. At last, the user has to define some type-specific properties such as the number of colors in case of the colorability type or the maximum size of the dominating set in case of the dominating set type, et cetera. If all required properties have been given, the user can start the computation of the new graph automaton by clicking the `Accept` button. When the computation is finished, the graph automaton is added to the repository.

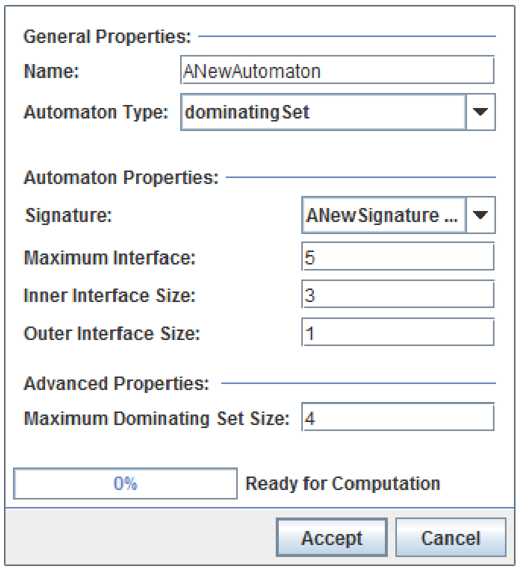
The next data structure we want to create is a new cospan. For this purpose the user has two possibilities in Raven. The first is to directly create a new cospan by creating the middle graph of the cospan and defining which nodes are


Name: ANewSignature

Operations

perm
res
trans
vertex
connect(A:2)
fuse

Figure 4.5: RAVEN Visual Signature Creator


General Properties:

Name: ANewAutomaton

Automaton Type: dominatingSet

Automaton Properties:

Signature: ANewSignature ...

Maximum Interface: 5

Inner Interface Size: 3

Outer Interface Size: 1

Advanced Properties:

Maximum Dominating Set Size: 4

0% Ready for Computation

Figure 4.6: RAVEN Visual Automaton Creator

in the inner and which ones are in the outer interface. The second is to give the cospan in terms of an atomic cospan decomposition. The advantage of the latter option is that the user has more influence on the cospan decomposition, since no decomposition must be computed. The disadvantage is that the user has to know the decomposition of the desired cospan. In order to create a new cospan the user has to choose either `Repository -> Create cospan` from the main menu or the button (✚) labelled with a green plus sign located on the repository panel and selecting the `Create cospan` item. The *visual cospan creator* dialog appears which is shown in Figure 4.7. The use of this dialog is similar to the visual graph creator. Therefore we omit the explanation of the visual cospan creator as far as possible. The only difference is that the user can add nodes of the middle graph to the inner and outer interface respectively. This can be done by selecting the certain node and clicking on the buttons labelled with the different arrows. The color of the nodes change with respect to the following criteria:

Color	**Criterion**
Gray	*The node is accessible by no interface*
Orange	*The node is accessible only by the inner interface*
Green	*The node is accessible only by the outer interface*
White	*The node is accessible by both, the inner and the outer interface*

To create a new cospan decomposition the user has to choose the `Create cospan decomposition` item instead of the `Create cospan` item. This brings up the *signature selection* dialog, which can be seen in Figure 4.8. The user can define the inner interface size and subsequently selects a signature which defines the available atomic cospans for the cospan decomposition. If the user has not created a signature, a default signature can be used. After accepting the chosen settings, the *visual cospan decomposition creator* dialog shows up, which can be seen in Figure 4.9. First of all, the user has to define the (unique) name

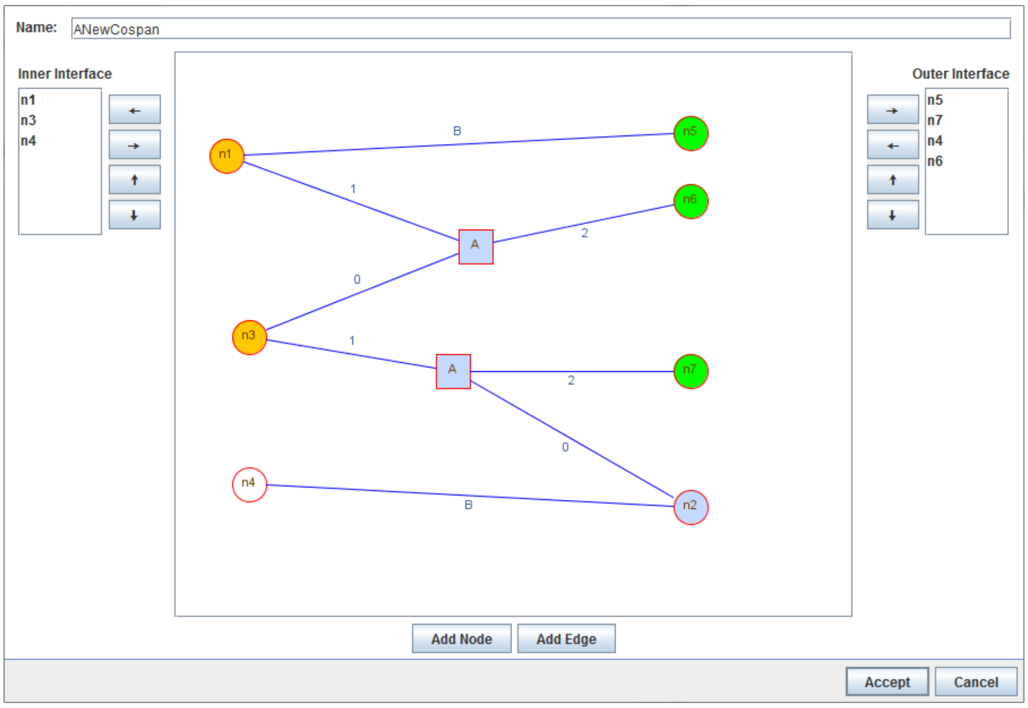
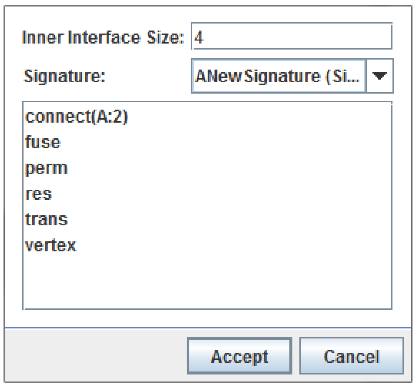


Figure 4.7: Raven Visual Cospan Creator

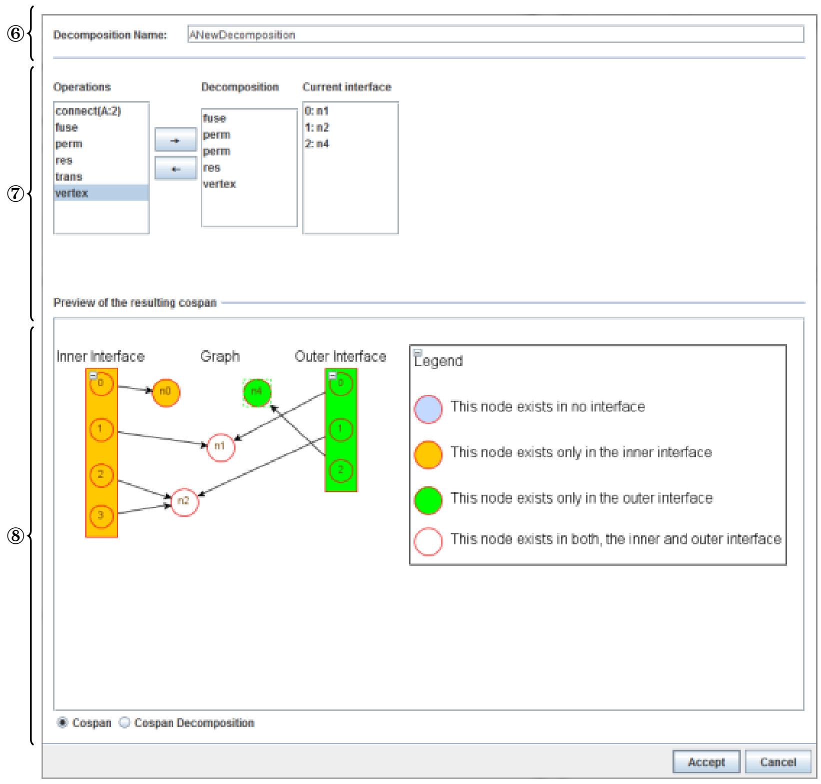
of the cospan decomposition (⑥, Figure 4.9). Afterwards we can compose the new atomic cospan decomposition out of the list of available atomic cospans by selecting the desired atomic cospan and pressing the arrow buttons. The list of available atomic cospans is depicted on the upper left of the dialog window (⑦, Figure 4.9). Every time a new atomic cospan is added to the decomposition, the cospan list, depicted in the middle, and the cospan view, depicted in the bottom part (⑧, Figure 4.9) of the dialog, are updated. Furthermore, the user is provided with the current (outer) interface of the cospan by the list on the right. The user can switch between a cospan view and a decomposition view by selecting the corresponding radio button under the view. Once, the atomic cospan decomposition is complete, the user can click the `Accept` button such that the decomposition is added to the repository.


Inner Interface Size: 4

Signature: ANewSignature (Si...

connect(A:2)
fuse
perm
res
trans
vertex

Figure 4.8: Raven Signature Selector



Decomposition Name: ANewDecomposition

Operations	Decomposition	Current interface
connect(A:2)	fuse	0: n1
fuse	perm	1: n2
perm	perm	2: n4
res	res	
trans	vertex	
vertex		

Preview of the resulting cospan

Legend

●	This node exists in no interface
●	This node exists only in the inner interface
●	This node exists only in the outer interface
○	This node exists in both, the inner and outer interface

[✓ Cospan] Cospan Decomposition

Figure 4.9: Raven Visual Cospan Decomposition Creator

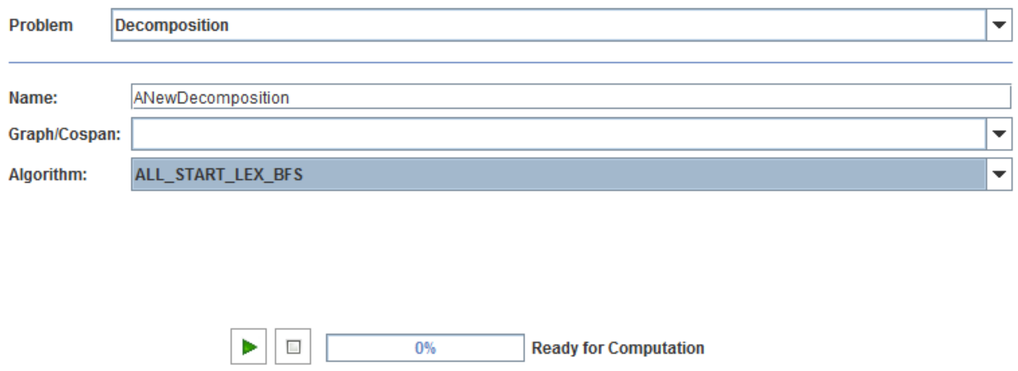
An alternative way to create a cospan decomposition is to auto-generate it, using the decomposition algorithms implemented in Raven. By selecting the `Decompose` goal from the goal panel (①, Figure 4.1), the decomposition panel shown in Figure 4.10 is presented. We have to type in a name for the new cospan decomposition we want to create. Then we have to choose one of the graph or cospan data structures, created in an earlier step. If we want to decompose a graph, we select one of the following algorithms to decompose it:

Type	**Description**
Greedy Degree	
Greedy Fill In	
All Start Lex BFS	
Lex BFS	

Table 4.3: Graph Decomposition Heuristics

Note that the given graph must not contain loops[2], since the given algorithms only work on loop-free graphs. To create a decomposition which may also describe a graph with loops and has a variable inner/outer interface size, the user has to select a cospan data object. In that case there only exists one selectable algorithm, which is an on-the-fly algorithm given as a default implementation. By pressing the button (▶) labelled with a green triangle, Raven decomposes the given data

[2]A *loop* in a hypergraph is an edge which is attached to some node more than once.


Problem: Decomposition

Name: ANewDecomposition

Graph/Cospan:

Algorithm: ALL_START_LEX_BFS

Figure 4.10: Raven Decomposition Goal Panel

structure and creates a new cospan decomposition object, which is saved in the repository.

After the user has created some data structures, the user can get further information about these objects by selecting them on the repository panel (④, Figure 4.1). The details about the selected object are then depicted on the data information panel (⑤, Figure 4.1). For example, if the user selects a graph automaton, the depicted details consist of properties which have been used during the creation and of further information such as the number of (all/the initial/the final) states of the automaton, the name of the BDD encoding or the (visual representation of the) BDDs used to encode the several transitions functions of the graph automaton. For the other data structures the user can also get additional information by selecting the certain object on the repository panel. In case the user selected a cospan, the corresponding graph is shown in the information panel. The user can save the graph in a scalable vector graphic file (SVG). In addition there exist information about the inner interface and the outer interface.

4.3.3 Algorithms of Raven

Now, we turn to the analysis of the graph automaton created beforehand. We assume that the user has created a graph with the help of the visual graph creator. The goal panel (①, Figure 4.1) consists of the following decision problems:

Decision Problem	**Description**
Language Inclusion	
Emptiness	
Membership	
Simulation	
Invariant Checking	
Decomposition	
Universality	

As an example we want to check whether the given graph is accepted by the graph automaton, i. e. we want to solve the membership problem. To do so, the user has to select the `Membership` goal from the goal panel (①, Figure 4.1) first. Then the desired graph automaton and the desired graph have to be selected from the respective lists which appear on the goal panel. To start the computation of the underlying membership algorithm we have to press the button (▶) labelled with a green triangle. In a similar way, the other goals can be chosen and started.

During the whole runtime of RAVEN the output panel (②, Figure 4.1) provides the user with additional information about the status of the program. Once the computation has finished, the result is displayed to the user. In case of the membership check only a small text dialog is displayed which is used to indicate whether the check has been successful or not. In case we have chosen the universality, language inclusion or invariant check, RAVEN also displays a counterexample if the result is negative and we have enabled the *Compute Counterexample* option on the goal panel (①, Figure 4.1). We will start with an example where we want to check whether the language of all 4-colorable graphs is included in the language of all 3-colorable graphs. First of all we create the respective automata for these problems as it was shown in the earlier steps. Now we have to select the `Language Inclusion` goal from the goal panel (①, Figure 4.1). We can choose one of the three algorithms *Antichain Language Inclusion* [13], *Simulation-based Antichain Language Inclusion* [1] or *Bisimulation up to congruence* [6] to compute whether the first automaton accepts a sub-language of the second automaton. If we select the antichain language inclusion algorithm, we have also the choice between a forward and a backward search variant. We select the *Compute Counterexample* option and start the computation by clicking on the button (▶) labelled with a green triangle. After the computation has been finished, we will see the computed counterexample shown in Figure 4.11. As expected the counterexample shows the graph of the 4-clique which is 4-colorable, but not 3-colorable. We can switch between the three different counterexample views *Graph*, *CospanDecomposition* and *Cospan* by selecting the corresponding radion buttons in the bottom of the dialog window.

The same kind of counterexample can be obtained if we check the universality of a 2-colorability automaton. The counterexample is the 3-clique shown in the cospan view in Figure 4.12 or shown in Figure 4.13 as a decomposition.

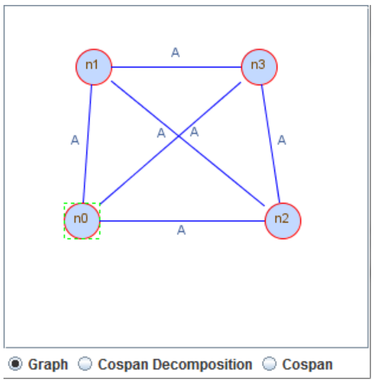


Figure 4.11: RAVEN Counterexample Language Inclusion

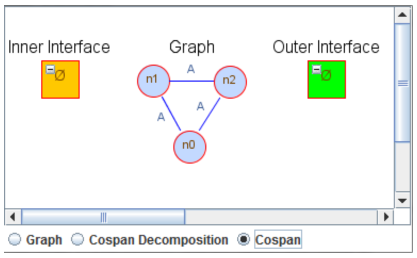


Figure 4.12: RAVEN Counterexample Universality

In case of an invariant check the counterexample consists of two graphs. The left graph represents the previous state and the right graph represents the state after applying the rule. As an example we want to check if the rule depicted in

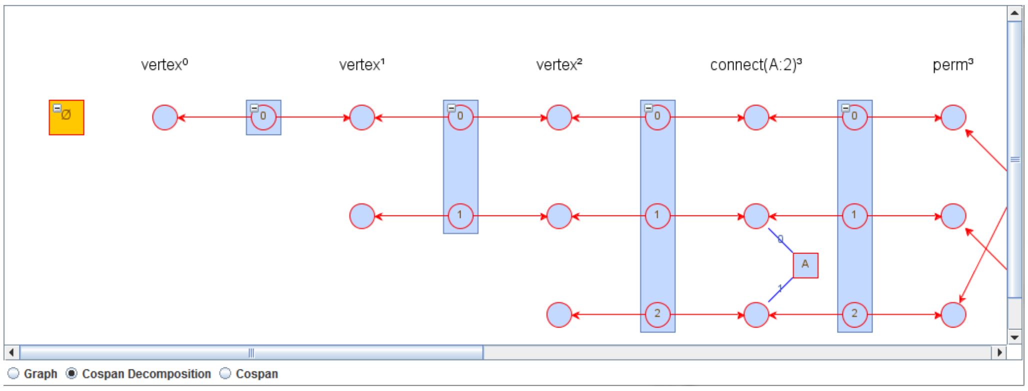


Figure 4.13: Raven Counterexample
Universality Decomposition

Figure 4.14 holds as an invariant for the 2-colorability automaton. To describe the rule we start creating the two cospans, which describe the left-hand side and the right-hand side of the rule. The cospans are depicted next to our rule in Figure 4.14. Now we can select the `Invariant Checking` goal from the goal panel (①, Figure 4.1). The invariant check is performed by a language inclusion check (for more information see [3]). The configurations are similar to the language inclusion goal panel.

Again we can choose one of the three algorithms *Antichain Invariant Checking*, *Simulation-based Invariant Checking* or *Bisimulation up to congruence* to compute whether the given rule is an invariant for the automaton or not. First, we have to choose an automaton which is used for the invariant check (in our case we select the 2-colorability automaton). Then we have to choose a cospan or cospan decomposition which describes the left-hand side of the transformation rule and a cospan or cospan decomposition which describes the right-hand side of the transformation rule. As already seen for the language inclusion check, we have the choice between a forward and a backward search variant, if we choose the antichain invariant checking algorithm. Again, we select the *Compute Counterexample* option and start the computation by clicking on the button (▶) labelled with a green triangle. Once the computation is finished, we will see the counterexample shown in Figure 4.15. The counterexample is a whitness that the language of all 2-colorable graphs is not an invariant for the given transformation rule. The graph containing two edges between to nodes where one of the edges

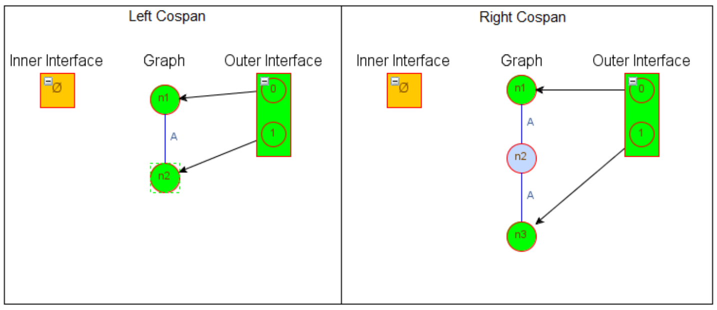


Figure 4.14: Raven Rule Example

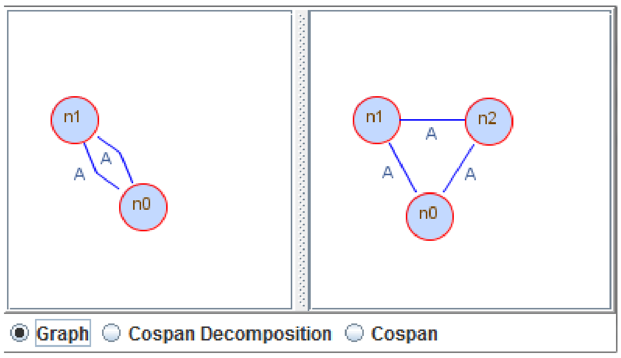


Figure 4.15: RAVEN Rule Counterexample

is rewritten into a new node which is adjacent to the former nodes. By this rewriting step we get the 3-clique which is not 2-colorable. As we can see the counterexample for rules consists of two graphs which describe two possible states of our graph automaton. The first graph is accepted by the given automaton, since it is 2-colorable. But by applying the transformation rule to the first graph we obtain a new graph which is not accepted by the automaton. Hence, enabling the counterexample option can help the user to understand the result of the analysis performed by RAVEN.

4.3.4 Saving/Loading with Raven

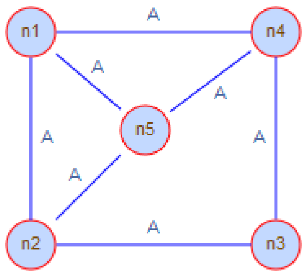
After the user has created some data structures, he can save and load his work. To avoid having a user create the same automaton, graph, cospan, signature or decomposition again after closing raven, he can save each data structure which he created in the repository. To do so the user has to choose `File -> Save`, or by pressing the button labelled with the save sign located on the repository panel and selecting one of the `Save` items. . . .

5 File Formats

5.1 The GXL Format for Graphs

In this section, we give an example of a graph represented in the GXL-format which is used to save graphs to `.gxl`-files. For a detailed description of the GXL-format see the website at `http://www.gupro.de/GXL/`.

For our example, we take the following graph:



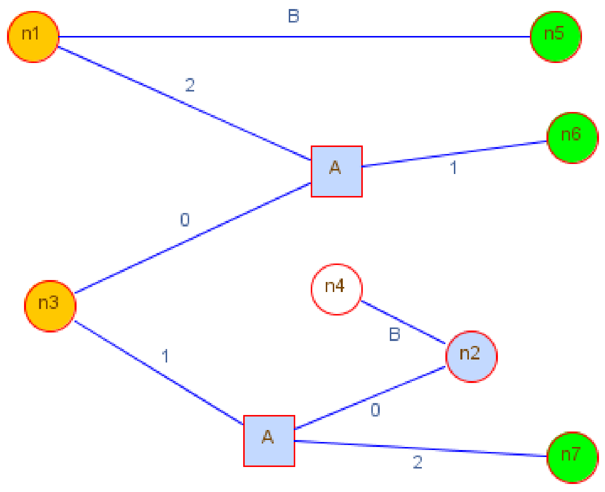
where the edge from node n_1 to node n_2 is named e_1, the edge from node n_1 to node n_3 is named e_2, and so on. The graph depicted above can then be represented by the following XML-file:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE gxl SYSTEM "http://www.gupro.de/GXL/gxl-1.0.dtd">

<gxl>
  <graph id="G" edgeids="true" edgemode="undirected"
         hypergraph="true">
    <node id="n1" />
    <node id="n2" />
    <node id="n3" />
    <node id="n4" />
    <node id="n5" />

    <rel id="e1">
      <attr name="label">
        <string>A</string>
      </attr>
      <relend target="n1" role="vertex" startorder="0" />
      <relend target="n2" role="vertex" startorder="1" />
    </rel>

    <rel id="e2">
      <attr name="label">
        <string>A</string>
      </attr>
      <relend target="n1" role="vertex" startorder="0" />
      <relend target="n4" role="vertex" startorder="1" />
    </rel>

    <rel id="e3">
      <attr name="label">
        <string>A</string>
      </attr>
      <relend target="n1" role="vertex" startorder="0" />
      <relend target="n5" role="vertex" startorder="1" />
    </rel>

    <rel id="e4">
      <attr name="label">
        <string>A</string>
      </attr>
      <relend target="n2" role="vertex" startorder="0" />
      <relend target="n3" role="vertex" startorder="1" />
    </rel>

    <rel id="e5">
      <attr name="label">
        <string>A</string>
      </attr>
      <relend target="n2" role="vertex" startorder="0" />
      <relend target="n5" role="vertex" startorder="1" />
    </rel>

    <rel id="e6">
      <attr name="label">
        <string>A</string>
      </attr>
      <relend target="n3" role="vertex" startorder="0" />
      <relend target="n4" role="vertex" startorder="1" />
    </rel>

    <rel id="e7">
      <attr name="label">
        <string>A</string>
      </attr>
      <relend target="n4" role="vertex" startorder="0" />
      <relend target="n5" role="vertex" startorder="1" />
    </rel>
  </graph>
</gxl>
```

5.2 The GXL Format for Cospans

In this section, we give an example of a cospan represented in the GXL-format which is used to save cospans to `.cos`-files. The file format is very similar to that used for graphs (see Section 5.1). The greatest difference is that we also need to save inner and outer interface of the cospan.

For our example, we take the following cospan:



where the B-edge incident to the nodes n_1 and n_5 is named e_1, the A-edge incident to the nodes n_1, n_3 and n_6 is named e_2, the A-edge incident to the nodes n_2, n_4 and n_7 is named e_3 and the B-edge incident to the nodes n_2 and n_4 is named e_4. The graph depicted above can then be represented by the following XML-file:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE gxl SYSTEM "http://www.gupro.de/GXL/gxl-1.0.dtd">

<gxl>
  <graph id="innerInterface">
    <attr name="n1">
      <string>"n1"</string>
    </attr>
    <attr name="n3">
      <string>"n3"</string>
    </attr>
    <attr name="n4">
      <string>"n4"</string>
    </attr>
  </graph>

  <graph id="middleGraph" edgeids="true" edgemode="undirected"
         hypergraph="true">
    <node id="n1" />
    <node id="n5" />
    <node id="n4" />
    <node id="n3" />
    <node id="n2" />
    <node id="n7" />
    <node id="n6" />

    <rel id="e3">
      <attr name="label">
        <string>A</string>
      </attr>
      <relend target="n2" role="vertex" startorder="0" />
      <relend target="n3" role="vertex" startorder="1" />
      <relend target="n7" role="vertex" startorder="2" />
    </rel>

    <rel id="e2">
      <attr name="label">
        <string>A</string>
      </attr>
      <relend target="n3" role="vertex" startorder="0" />
      <relend target="n6" role="vertex" startorder="1" />
      <relend target="n1" role="vertex" startorder="2" />
    </rel>

    <rel id="e4">
      <attr name="label">
        <string>B</string>
      </attr>
      <relend target="n2" role="vertex" startorder="0" />
      <relend target="n4" role="vertex" startorder="1" />
    </rel>

    <rel id="e1">
      <attr name="label">
        <string>B</string>
      </attr>
      <relend target="n1" role="vertex" startorder="0" />
      <relend target="n5" role="vertex" startorder="1" />
    </rel>
  </graph>

  <graph id="outerInterface">
    <attr name="n5">
      <string>"n5"</string>
    </attr>
    <attr name="n7">
      <string>"n7"</string>
    </attr>
    <attr name="n4">
      <string>"n4"</string>
    </attr>
    <attr name="n6">
      <string>"n6"</string>
    </attr>
  </graph>
</gxl>
```

5.3 The Automaton File Format

In this section, we briefly explain the `.aut`-file format which is used to save automata to files. Every `.aut`-file is essentially a ZIP-file consisting of different files depending on the automaton. Each `.aut`-file contains

- a file named `general.obj` which holds the serialized[1] representation of the corresponding automaton object,
- a file named `states.bdd`[2] holding information about the BDD representing the set of all states,
- a file named `initial.bdd` holding information about the BDD representing the set of initial states,
- a file named `final.bdd` holding information about the BDD representing the set of final states,
- a file named `nonFinal.bdd` holding information about the BDD representing the set of non-final states
- depending on the input alphabet of the corresponding automaton a `.bdd`-file for each atomic cospan of the input alphabet is contained: these `.bdd`-files hold information about the BDDs representing the transition relations for the particular atomic cospans.

5.4 The Signature File Format

In this section, we briefly explain the `.sig`-file format which is used to save signatures to files. Since a signature is just a set of atomic cospans (used as input alphabet for graph automata) the file format is rather simple. The format is text- and line-oriented, i. e. each line of the file consists of the name of exactly one atomic cospan.

5.5 The Cospan Decomposition File Format

In this section, we briefly explain the `.dec`-file format which is used to save cospan decompositions to files. Since a cospan decomposition is just a sequence of atomic cospans the file format is rather simple. The format is text- and line-oriented,

[1] See `http://docs.oracle.com/javase/7/docs/technotes/guides/serialization/` for further information about serialization in Java.
[2] For a description of the file format used in the `.bdd`-files see the manual of the BuDDy-package.

i. e. each line of the file consists of the name of exactly one atomic cospan. With one exception: the first line of the file indicates the size of the inner interface of the resulting cospan. The size of the outer interface of the resulting cospan (and all intermediate interface sizes) can then be computed by the sequence of atomic cospans.

Bibliography

[1] Abdulla, P.A., Chen, Y.F., Holík, L., Mayr, R., Vojnar, T.: When simulation meets antichains (on checking language inclusion of NFAs). In: Proc. of TACAS '10. pp. 158–174. LNCS 6015, Springer (2010)

[2] Blume, C.: Graphsprachen für die Spezifikation von Invarianten bei verteilten und dynamischen Systemen. Master's thesis, Universität Duisburg-Essen (2008), (in German)

[3] Blume, C., Bruggink, H.J.S., Engelke, D., König, B.: Efficient symbolic implementation of graph automata with applications to invariant checking. In: Ehrig, H., Engels, G., Kreowski, H.J., Rozenberg, G. (eds.) Graph Transformations. Lecture Notes in Computer Science, vol. 7562, pp. 264–278. Springer (2012)

[4] Blume, C., Bruggink, H.J.S., König, B.: Recognizable graph languages for checking invariants. In: Proceedings of GT-VMT 2010. ECEASST 29 (2010)

[5] Bodlaender, H.L., Koster, A.M.C.A.: Treewidth Computations I. Upper Bounds. Tech. Rep. UU-CS-2008-032, Department of Information and Computing Sciences, Utrecht University (September 2008)

[6] Bonchi, F., Pous, D.: Checking nfa equivalence with bisimulations up to congruence. In: Proceedings of the 40th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages. pp. 457–468. POPL' 13, ACM (2013)

[7] Thomas van Dijk, Jan-Pieter van den Heuvel, W.S.: Computing treewidth with LibTW (November 2006), `http://www.treewidth.com/docs/LibTW.pdf`, (online)

[8] Holt, R.C., Schürr, A., Sim, S.E., Winter, A.: GXL: A graph-based standard exchange format for reengineering. Science of Computer Programming 60(2), 149–170 (2006)

[9] Kullbach, B., Riediger, V., Winter, A.: An overview of the GXL graph exchange language. In: Diehl, S. (ed.) Software Visualization, Lecture Notes in Computer Science, vol. 2269, pp. 324–336. Springer (2002)

[10] Lind-Nielsen, J.: Buddy – a binary decision diagram package, `http://sourceforge.net/projects/buddy`, (online)

[11] Whaley, J.: Javabdd – java library for manipulating bdds, `http://javabdd.sourceforge.net/`, (online)

[12] Winter, A.: Exchanging graphs with gxl. In: Mutzel, P., Jünger, M., Leipert, S. (eds.) Graph Drawing, Lecture Notes in Computer Science, vol. 2265, pp. 485–500. Springer (2002)

[13] Wulf, M.D., Doyen, L., Henzinger, T.A., Raskin, J.F.: Antichains: A new algorithm for checking universality of finite automata. In: Proceedings of CAV 2006. pp. 17–30. LNCS 4144, Springer (2006)