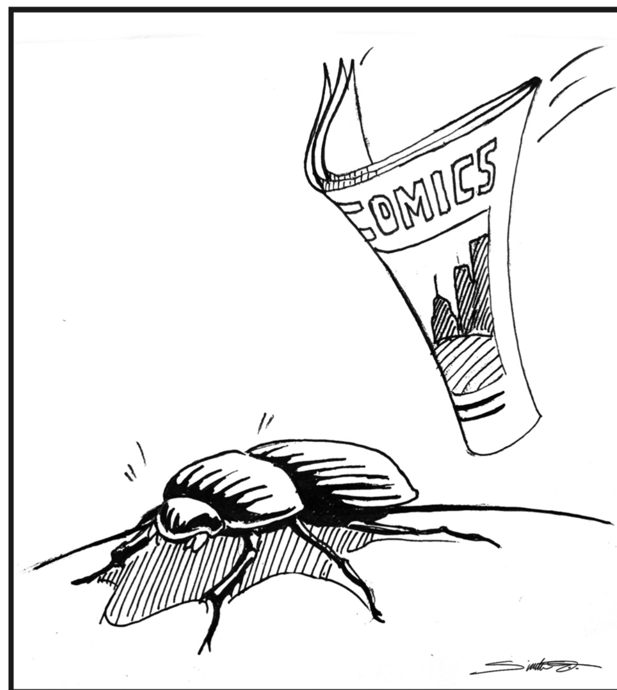
The COMICS Tool
Version 1.0

Computing Minimal Counterexamples for Discrete-Time Markov Chains

Manual

Nils Jansen, Erika Ábrahám



Contents

1 Introduction **5**
1.1 Foundations 6

2 Getting Started **7**
2.1 Running the Binaries 7
2.1.1 Linux 7
2.1.2 Microsoft Windows 7
2.2 Compiling the Source Code 7
2.2.1 Compiling for Linux 8
2.2.2 Compiling for MAC OS 9
2.2.3 Running the self-compiled binaries 9

3 Input Format **11**
3.1 The .dtmc-Format 11
3.2 The .tra-Format 11
3.3 The .lab-Format 12
3.4 The .conf-Format 12
3.5 The .xml-Format 14

4 Usage **15**
4.1 Command-line Mode 15
4.1.1 Basic Options 15
4.1.2 Heuristics 17
4.1.3 Benchmarks 18
4.1.4 Output 18
4.2 Interactive Mode 18
4.2.1 Introducing the GUI 18
4.2.2 Structure 19
4.2.3 Creating a DTMC 21
4.2.4 Model Checking 23
4.2.5 Counterexample Generation 23

1 Introduction

COMICS is a stand-alone tool which performs model checking and the generation of counterexamples for discrete-time Markov Chains (DTMCs). For an input DTMC COMICS computes an abstract system that carries the model checking information and uses this result to compute a critical subsystem, which induces a counterexample. This abstract subsystem can be refined and concretized hierarchically. The tool comes with a command line version as well as a graphical user interface which allows the user to interactively influence the refinement process of the counterexample. For more details on the approaches implemented in this tool, we refer to [1, 4, 5].

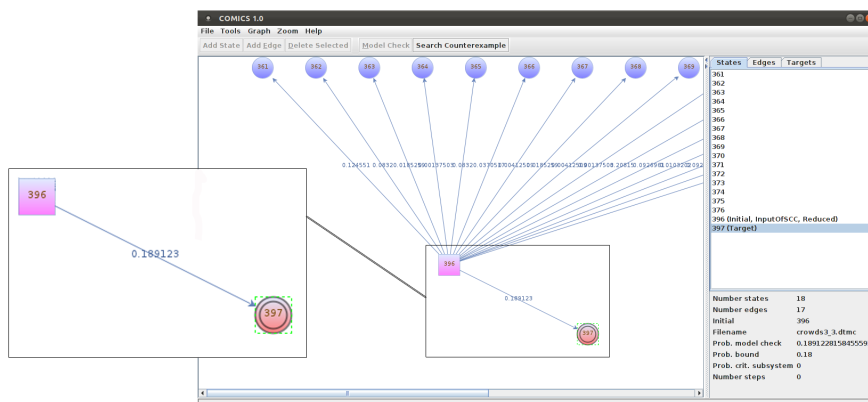


Figure 1.1: Screenshot of `COMICS`'s GUI with an instance of the *crowds protocol* benchmark

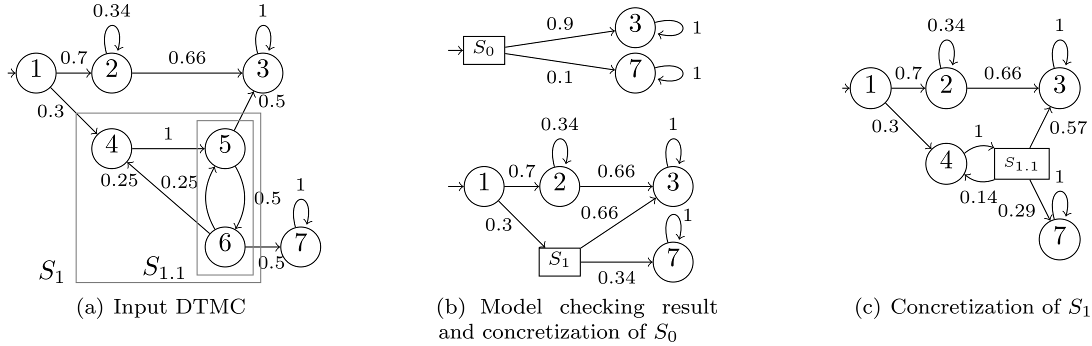
In this manual we firstly give a very short introduction to the basic algorithms which are used in `COMICS`. In Section 2 we describe how to get the binaries and how to compile the sources for Linux, Microsoft Windows as well as MAC OS X. Section 3 lists the different input formats the tool accepts as well as the corresponding syntax and usage. In Section 4 we explain, how `COMICS` is used both for the GUI and the command line version together with a running example which will describe all features of our tool.

We will continuously update this manual, please consider to visit our homepage again for a new version.

http://www-i2.informatik.rwth-aachen.de/i2/comics/

In case of any questions or remarks, please feel free to contact

Nils Jansen
Theory of Hybrid Systems
RWTH Aachen
52056 Aachen
Germany
nils.jansen@cs.rwth-aachen.de



(a) Input DTMC

(b) Model checking result and concretization of S_0

(c) Concretization of S_1

Figure 1.2: Example SCC-based model checking

1.1 Foundations

In this section we briefly explain the algorithms implemented in `COMICS` (see [4] and [5] for more details). We use the standard definitions for DTMCs and *probabilistic computation tree logic* PCTL. For an detailed introduction to these topics as well as related work we refer to [3].

We consider model checking and counterexample generation for DTMCs and time-unbounded PCTL properties, which can be reduced to a DTMC M with one *initial state* s_{I}, a set of *target states* T, and an upper probability bound $\lambda \in [0, 1]$ on reaching T from s_{I}. Originally, a *counterexample* for a DTMC M and a *reachability property* is defined as set of finite paths of M leading from s_{I} to T with a cumulated probability mass greater than λ.

In [1] we proposed a model checking approach for DTMCs based on *hierarchical abstraction*. Each SCC of the underlying graph of the input DTMC is abstracted by a state whose outgoing transitions lead to states outside the SCC and carry the whole probability mass of reaching those states when once entering the SCC. This abstraction is done recursively in a bottom-up manner: before abstracting an SCC we first apply abstraction to the sub-SCCs nested in it. The final result is an abstract DTMC whose only transitions lead from the initial state of the input DTMC to absorbing states and carry the corresponding reachability probabilities. Fig. 1.2(a) depicts a DTMC and its nested SCC structure: SCC S_1 contains SCC $S_{1.1}$. The upper graph of Fig. 1.2(b) depicts the result of the model checking: The probability to reach the target state 3 from the initial state 0 is 0.9. This abstract DTMC can be also concretized hierarchically. The lower graph of Fig. 1.2(b) shows the concretization of the abstract state S_0: The outgoing edges of S_1 carry the probability mass of all paths leading from the input state 4 of S_1 to the output states 3 and 7. Fig. 1.2(c) shows a further concretization of S_1 while SCC $S_{1.1}$ is still abstracted. Concretizing also $S_{1.1}$ would result in the DTMC of Figure 3.1.

Based on this approach, we presented a method to compute and represent counterexamples as *critical subsystems* in [4]. These subsystems consist of subsets of the original DTMC's states and transitions such that the probability of reaching target states from the initial state within the subsystem still exceeds the probability bound λ. The method first computes an abstract critical subsystem for the abstract DTMC resulting from model checking. Inside this abstract DTMC one or more abstract states are selected and concretized, and a critical subsystem is determined for the concretized system. This process may be repeated until the system is fully concretized. We suggested two methods for the computation of critical subsystems: The *global search* (`GS`) looks for for most probable paths through the whole system until the involved states and transitions form a critical subsystem. The *local search* (`LS`) builds critical subsystems incrementally by extending subsystems with most probable path fragments.

Figure 1.1 shows one abstracted instance of the *crowds protocol* benchmark [6], where the probability of reaching the unique target state is displayed in the information panel on the right as well as on the edge leading from the initial state to the target state. The initial state is abstract and can therefore be expanded.

2 Getting Started

2.1 Running the Binaries

So far, we offer only binaries for Windows and Linux. If you are running a MAC OS system, please use the sources.

2.1.1 Linux

If you are using one of the pre-compiled binaries and you are running a 64 bit Debian system, you can start COMICS by changing into directory comics-1.0 and typing

```
./comics.sh
```

If this does not work, make sure that the script file is marked as executable:

```
chmod +x comics.sh
```

If you are running a 32 bit Debian system, you have to type

```
./comics_32.sh
```

By invoking COMICS without parameters or by

```
./comics.sh --help
```

the help output is displayed, which is depicted in Example 4.1. If you want to start the graphical user interface, you have to run

```
./xcomics.sh
```

For details on the usage, see Section 4.

2.1.2 Microsoft Windows

Using Microsoft Windows 7, you can start the GUI by executing

```
xcomics.bat
```

If you want to use the command-line tool, start

```
comics.exe
```

from the Windows console.

2.2 Compiling the Source Code

We have tested the compilation process for Linux, Windows and MAC OS. As the compilation for Windows requires many third party packages, we recommend to use the pre-compiled binaries for Windows. However, a description will follow!
In case of any problems with building the tool, please contact nils.jansen@cs.rwth-aachen.de, we will most probably be able to help you!

2.2.1 Compiling for Linux

Compiling with bash scripts

In order to build COMICS you have to use the GNU build system (Autotools). The following non-standard packages have to be installed:

- A Java Development Kit (JDK)[1]
- GNU Automatic Configure Script Builder (autoconf)
- GNU Generic Library Support Script (libtool)
- GNU C++ Compiler (g++)

The JAVA GUI has to be compiled together with the JNI shared libraries. You have to change into the scripts folder and execute

```
./build_comics.sh path_to_java
```

which is the easiest way to build COMICS. If you add as a parameter your JAVA_HOME, which is the path to your JDK installation, or your JAVA_HOME is already set system-wide, the necessary JNI libraries are built by this script. Please make sure that you have a complete Java Development Kid installed! If your JDK is, e. g., the *OpenJDK Development Kid*, possible ways to build COMICS complete and properly are:

(a) `./build_comics.sh /usr/lib/jvm/java-6-openjdk`

(b) `export JAVA_HOME=/usr/lib/jvm/java-6-openjdk`
`./build_comics.sh`

(c) `./build_comics.sh` (and JAVA_HOME is already set)

An indication of problems with building the JNI-libraries is the GUI throwing an JAVA exception when calling *Model Checking* or *Search Counterexample*. We have tested the compilation process both on 64bit and 32bit Debian systems using

- OpenJDK Development Kid 6 (openjdk-6-jdk)
- GNU Autoconf 2.65
- GNU Libtool 2.2.6b
- g++ 4.4.3

You can also compile the command-line tool without using our predefined bash-scripts using inside the cmdl_src directory.

```
autoreconf --install
```

```
./configure
```

```
make
```

[1]Please make sure that a complete JDK is installed, and not only a JRE (Java Runtime Environment).

2.2.2 Compiling for MAC OS

The build process is similar to Linux, please make sure to have a complete JDK installed. If your version of GNU Autoconf doesn't match, please change the version number specified inside the file

```
cmdl_src/configure.ac
```

in the first line

```
AC_PREREQ([2.64])
```

to your version number. Afterwards, calling from the `scripts` folder

```
./build_comics_mac.sh path_to_jdk
```

should successfully build the tool. Note, that for many MAC systems the JAVA_HOME (path to jdk) should be */usr/libexec/java_home*. Don't use any quotation marks for the paths!

2.2.3 Running the self-compiled binaries

If your sources are successfully built, you can start `COMICS` by changing into directory `scripts` and typing[2]

```
./comics.sh
```

By invoking `COMICS` without parameters or by

```
./comics.sh --help
```

the help output is displayed, which is depicted in Example 4.1. If you want to start the GUI, you have to run

```
./xcomics.sh
```

For details on the usage, see Section 4.

[2]As the sources are compiled on your own system, it doesn't matter, if you have a 32 bit or a 64 bit system.

3 Input Format

The input format of `COMICS` may either describe the explicit representation of a DTMC or an abstract graph which can be concretized. The `.dtmc`-files describe a DTMC with target states and one unique initial state. `.dtmc`-files are the original input for `COMICS`. `.tra`- and `.lab`-files are the DTMC-input for MRMC and can be imported[1] into `COMICS`. Abstract graphs are represented by `.xml`-files.
In this chapter we will introduce all formats in detail and give some examples. Consider the following DTMC with input state 1 and target states 3 and 8.

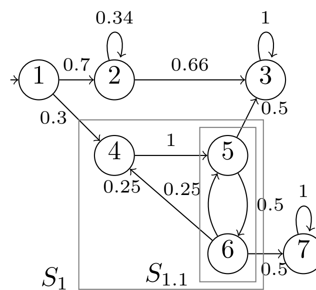


Figure 3.1: Example DTMC

3.1 The .dtmc-Format

The `.dtmc`-files contain the *number of states and transitions* of an DTMC. An *unique initial state* and an *arbitrary number of target states* are defined, and the *transition probability matrix* is given. The abstract syntax is defined as follows:

```
STATES <number of states>
TRANSITIONS <number of transitions>
INITIAL <initial state>
(TARGET <target state>\n)* //Arbitrarily many target states
(<start state> <end state> <probability>\n)* //Transitions
```

Syntax of .dtmc-Files

States are presented as positive integers starting with 1. Transitions are given by two integers identifying the start state and the end state of the transition and its probability which has to be out of $[0, 1]$. The transitions have to be ordered in ascending order w. r. t. to firstly the start states and secondly the end states. Consider Example 3.2 which represents the example DTMC depicted in Figure 3.1.

3.2 The .tra-Format

The `.tra`-format is the input format of MRMC and defined similar to the `.dtmc`-format. The only difference is that it contains no information about target states and input state. The `.dtmc`-format

[1] So far only by using the graphical user interface.

```
STATES 7
TRANSITIONS 12
INITIAL 1
TARGET 3
1 2 0.7
1 4 0.25
2 2 0.34
2 3 0.34
3 3 1.0
4 5 1.0
5 3 0.5
5 6 0.5
6 4 0.25
6 5 0.25
6 7 0.5
7 7 1.0
```

Figure 3.2: Example `.dtmc`-file

was designed to have the same syntax in order to guarantee the compatibility of both tools and to easily use the PRISM export facilities. Note, that - so far - `.tra`-files are only supported when using the graphical user interface of `COMICS`.

```
STATES <number of states>
TRANSITIONS <number of transitions>
(<start state> <end state> <probability>\n)* //Transitions
```

Syntax of .tra-Files

3.3 The .lab-Format

`.lab`-files contain information about the list of atomic propositions and their assignment to the states of a DTMC. It consists firstly of the declaration of a list of atomic propositions and secondly of states and a list of propositions they are labeled with. This list has to be subset of the declared list of propositions. Note, that - so far - `.lab`-files are only supported when using the graphical user interface of `COMICS`.

```
#DECLARATION
<proposition_list>
#END
(<state proposition_list>)*
```

Syntax of .lab-Files

The `.tra` and `.lab` files depicted in Figure 3.3 and Figure 3.4 together define again the DTMC of Figure 3.1 while state 8 is additionally labeled with `ERRORLABEL`. Note, that one state can have several labels.

3.4 The .conf-Format

The `.conf`-files contains all information needed for a counterexample search. Although all parameters can also be set as command-line parameters or inside the graphical user interface, this

```
STATES 7
TRANSITIONS 12
1 2 0.7
1 4 0.25
2 2 0.34
2 3 0.34
3 3 1.0
4 5 1.0
5 3 0.5
5 6 0.5
6 4 0.25
6 5 0.25
6 7 0.5
7 7 1.0
```

Figure 3.3: Example `.tra`-file

```
#DECLARATION
INITIAL TARGET ERRORLABEL
#END
1 INITIAL
3 TARGET
7 ERRORLABEL
```

Figure 3.4: Example `.lab`-file

simplifies structuring the tasks to perform.

```
TASK counterexample | modelchecking
PROBABILITY_BOUND double
DTMC_FILE *.tra | *.xml | *.dtmc
REPRESENTATION subsystem | pathset
SEARCH_ALGORITHM global |local
ABSTRACTION concrete | abstract
```

Syntax of .conf-Files

The main task may be the search for a *counterexample* or just getting the *model checking* result. A *probability bound* can be defined which shall not exceed the model checking probability of the input system, otherwise the counterexample search will not be carried out. The input files are as mentioned above. The search algorithm can be specified: *global* search or *local* search.

Finally, the user can choose between performing the hierarchical counterexample search on an `abstract` system or just search for a critical subsystem directly on the concrete graph. An example can be found in Figure 3.5: a search for counterexample against a probability bound of 0.2 will be performed on an input file named `example_manual.dtmc`. The search algorithm will be the global search, and the task will start on an abstract system. This file as well as the corresponding `.dtmc` file can be found in any installation of `COMICS` inside the folder `examples/example_manual/`. The `.dtmc` file corresponds to the system depicted in Figure 3.1.

```
TASK CounterExample
PROBABILITY_BOUND 0.2
DTMC_FILE example_manual.dtmc
SEARCH_ALGORITHM global
ABSTRACTION abstract
```

Figure 3.5: Example `.conf`-file

3.5 The .xml-Format

The `.xml`-format stores the whole abstract graph which results from SCC-based model checking [1]. The root node is named `dtmc`. It has children `scc`, `target`, and `prob`. `scc` has as attributes a unique `id` and a identifying node `node0`. Its children are `inp` (input states), `out` (output states), `vtx` (remaining states), and `edge` (graph edges). Moreover, again `scc` can be child of `scc`. `edge` has a flag `abs` wether it is abstract or not. `target` identifies the target states of the DTMC and `prob` stores the model checking probability.
Note, that if the probability is defined, no model checking is performed for `.xml` files, as the probability of reaching target states in the whole system is already saved inside the document. For the command-line version, the probability actually **has** to be defined inside the `.xml` file, otherwise no search for a counterexample will be performed. An additional feature of the `.xml`-format is the storage of state positions on the graph panel, i. e., , if a graph is stored in `.xml`-format, the positions of all states will be as before if the file is loaded later.
A deeper understanding of this format is not crucial for using `COMICS`, as the `.xml`-files should not be edited. They are only used to save the abstract graph structure, changes are done at the risk of having an inconsistent system.

4 Usage

In this chapter, we describe the usage of `COMICS` both for the command-line tool (C++) and the GUI (JAVA with JNI shared libraries).

4.1 Command-line Mode

If you are using one of the pre-compiled binaries and you are running a 64 bit Debian system, you can start `COMICS` by changing into directory `comics-1.0` and typing

```
./comics.sh
```

If this does not work, make sure that the binary file is marked as executable:

```
chmod +x comics.sh
```

If you are running a 32 bit Debian system, you have to type

```
./comics.sh_32.sh
```

By invoking `COMICS` without parameters or by

```
./comics.sh --help
```

the help output is displayed, which is depicted in Example 4.1. Of course, you can also start the binary files directly instead of the bash scripts.
Many of the basic parameters can already be predefined by loading a `.conf`-file. If parameters of the `.conf`-file and the command-line are inconsistent, those given in the command-line are chosen.
We will shortly explain all of the possible parameters. Comics can be started with an input `.conf`-file by

```
./comics <inputFile.conf>
```

In case, a `.dtmc`-file was specified inside the `.conf`-file, the corresponding tasks are performed directly. If an `.xml`-file was specified, model checking will not be performed as the result is already saved inside the `.xml`-file. Thus, it makes no sense to invoke model checking as only task when loading an `.xml`-file.
To have a quick start, just run

```
./comics.sh examples/example_manual/example_manual.conf --concrete
```

in order to try `COMICS` on the system depicted in Figure 3.1, using the parameters specified as in the `.conf` file of Figure 3.5. The pre-defined search on an abstract system is overwritten by forcing the `--concrete` parameter.

4.1.1 Basic Options

- `--outputdtmc <filename>`

  The resulting critical subsystem is saved in the current directory as `<filename.dtmc>`. If nothing is specified, a file “result.dtmc” is saved.

```
COMICS - Computing Minimal Subsystems
Copyright (c) RWTH Aachen University 2012
Authors:  Nils Jansen, Erika Abraham, Jens Katelaan, Maik Scheffler,
Matthias Volk, Andreas Vorpahl

This is COMICS 1.0.  Possible ways to call are:
./comics filename [options]
NOTICE: filepaths have to be relative to the executable!
with options:

 --outputdtmc <filename>   (Save result as <outputFilename.dtmc>)
 --outputxml <filename>    (Save result as <outputFilename.xml>)
 --saveIterations          (Save results of all iterations)
 --dtmc                    (DTMC file only, no config file)
 --only_model_checking     (Do not search for a counterexample)
 --no_model_checking       (Skip model checking (enforces concrete search))
 --abstract | --concrete   (Abstract or concrete search algorithm)
 --global | --local        (Global or local path search)
 --subsystem | --pathset   (Subsystem- or path-based counter example)
 --iterationcount <count>  (Do <count> many iterations)
 --stepsize <size>         (One iteration has <size> concretization steps)
 --probBound <bound>       (Probability Bound for counterexample search)

Heuristics for the choice of abstract state to concretize:
 --choose_by_probability   (By highest outgoing probability)
 --choose_by_degree        (By lowest input/output degree)
 --choose_by_membership    (By highest relative membership in subsystem)

Benchmarking:
 --search-benchmark          (Try global, local, concrete, abstract)
 --concretization-benchmark  (Try concretizing 1, sqrt, all SCCs in each step)
 --heuristics-benchmark      (Try different concretization heuristics)
 --complete-benchmark        (Combine all of the above benchmarks - slow!)
Default behavior:  --abstract --global --closure
```

Figure 4.1: Example help-output

- `--outputxml <filename>`

  The resulting critical subsystem is saved as abstract graph possibly containing information about abstracted SCCs and the model checking result for the subsystem. The filename is `<filename.xml>`

- `--saveIterations`

  Every time a critical subsystem is computed for an abstract graph, the corresponding `.dtmc`-file is stored in the current directory before further concretization is performed and a new computation is invoked.

- `--dtmc`

  A `.dtmc`-file is given as input instead of the standard `.conf`-file.

- `--only_model_checking`

  Only the model checking result is returned, no search for a counterexample is performed.

- `--no_model_checking`

  No model checking is performed, only possible for the direct search on a concrete system. Note, that the tool might not terminate, if the given probability bound is not exceeded by the input system.

- `--abstract`

  The search for a counterexample is performed on the abstract graph involving concretization.

- `--concrete`

  The search for a counterexample is performed directly on the concrete graph.

- `--global`

  The global search approach is used to find a counterexample, which may be represented as a critical subsystem or a set of paths of the input system.

- `--local`

  The local search approach is used to find a critical subsystem.

- `--subsystem`

  The resulting counterexample is represented as critical subsystem.

- `--pathsum`

  The resulting counterexample is represented as set of paths.

- `--iterationcount <count>`

  The number of search iterations is fixed by `<count>`. Afterwards, the program terminates with a possibly still abstract counterexample. If this parameter is not set, the program will iterate with a concrete counterexample not containing any abstract SCCs. For large benchmarks, this concrete system may still be very large.

- `--stepsize <count>`

  The number of concretization steps per iteration is fixed by `<size>`.

- `--probBound`

  The probability bound for the counterexample search. If a `.conf`-file is loaded the value specified there will be overwritten.

4.1.2 Heuristics

We have implemented a large number of heuristics for the automatic choice of the next abstract SCC to concretize. The three heuristics which performed best are offered for selection.

- `--choose_by_probability`

  The abstract SCC which has the *highest* outgoing probability *inside* the critical subsystem is chosen.

- `--choose_by_degree`

  The abstract SCC with the *lowest* input/output-degree is chosen.

- `--choose_by_membership`

  The abstract SCC with the *highest* number of ingoing and outgoing transitions is chosen. To this we refer as the *highest-membership* inside the critical subsystem.

In our tests, the highest-membership heuristic performed best, while the choice by input/output degree delivered mostly worse results than using no heuristics.

4.1.3 Benchmarks

In order to do extensive tests with our tool, the user can choose between a number of benchmarking options, were several tasks are performed.

- `--search-benchmark`

  A test for the global search vs. the local search is performed both on abstract and on concrete systems.

- `--concretization-benchmark`

  A hierarchical search on an abstract system is performed, where
  1. At each step only 1 abstract SCC is concretized.
  2. For n visible abstract SCCs in the system, at each step $\sqrt{n}$ SCCs are concretized. (SQRT-heuristics).
  3. At each step all abstract SCCs are concretized.

  In our tests, the SQRT-heuristic performed best.

- `--heuristics-benchmark`

  The three heuristics for automatic choice of next abstract SCCs are tested against each other, see Section 4.1.2.

4.1.4 Output

As mentioned before, the resulting critical subsystem is stored in the current directy either as `.dtmc`- or as `.xml`-file. Additionally, the intermediate results may be saved. In any case, a short summary of the search process is saved as `counter_example_summary.txt`.
An example of such a summary is depicted in Figure 4.2. The task was as given in the `.conf` file of Figure 3.5 which is available in the `COMICS` installation. The command was

```
./comics.sh examples/example_manual/example_manual.conf --concrete
```

4.2 Interactive Mode

4.2.1 Introducing the GUI

The GUI was implemented using the `JGraph` library [2] and aims at making the process of finding a counterexample user-interactive and thereby increasing the usability of probabilistic counterexamples.

```
---------------------------------
----------- STATISTICS ----------
---------------------------------
Task:  Counterexample for P(t)<0.2
Counter example size:  4 states, 3 transitions
Model Checking result of original system:  0.9
Closure probability:  0.7
Time of counter example computation:  0 secs [Without Pre-/Post processing]
Search params:  CONCRETE, GLOBAL, CLOSURE-based
Base node selection for concretization:  Select All Visible
Additional selection criterion:  None
Number of concretizations per step:  1
During the 1 refinement steps the following number of shortest paths/closures
were computed:
-----------------------------------------------
step | #paths | #closures | #conc.  scc
-----------------------------------------------
1    | 1      | 1          | 1
-----------------------------------------------
Total #paths:  1, total #closures:  1, total #concretized SCCs:  1
```

Figure 4.2: Example: Summary-file of Counterexample Generation

A user can load instances of DTMC benchmarks by using the `.dtmc`, `.tra` or `.xml` formats. Additionally, he can model his own examples by the in-built graph-editor. SCC-based model checking [1] can be performed by calling the C++-library via JNI. Afterwards, critical subsystems for certain probability bounds can be computed automatically [4] or user-guided in the sense that the states can be conretized for further inspection. Note, that the GUI is not suited to load rather large problem instances. We strongly suggest to apply some steps of the counterexample search by using the command-line mode and afterwards to load the small, abstract graph into the GUI. If the graph is loaded but still too big or unordered for debugging purposes, a smooth zoom and several layouting algorithms are offered.
If a user opens `.xml`-file representing an *abstract graph*, the actual model checking result of the system is included. Therefore, the graph can not be edited but only concretized or abstracted. The counterexample generation can be called at any point. If a *concrete graph* is loaded, model checking has to be applied beforehand in order to guarantee completeness in the sense that the probability actually *can* be exceeded.

4.2.2 Structure

Consider at first Figure 4.3: On top is the *menubar*, right below is the *toolbar* located. On the left side is the *graph-panel*, where DTMCs are displayed, on the right side is the *information-panel*, where a user can see lists of all states, edges and target states. The DTMC of Fig. 1.2(a) is displayed in the graph-panel, while all relevant information is also displayed at the information panel. This small example is accessible on the homepage[1] as well as in the examples folder in the downloaded version of `COMICS`.

The Menubar

The following menus are offered:

- File

[1] http://www-i2.informatik.rwth-aachen.de/i2/comics/

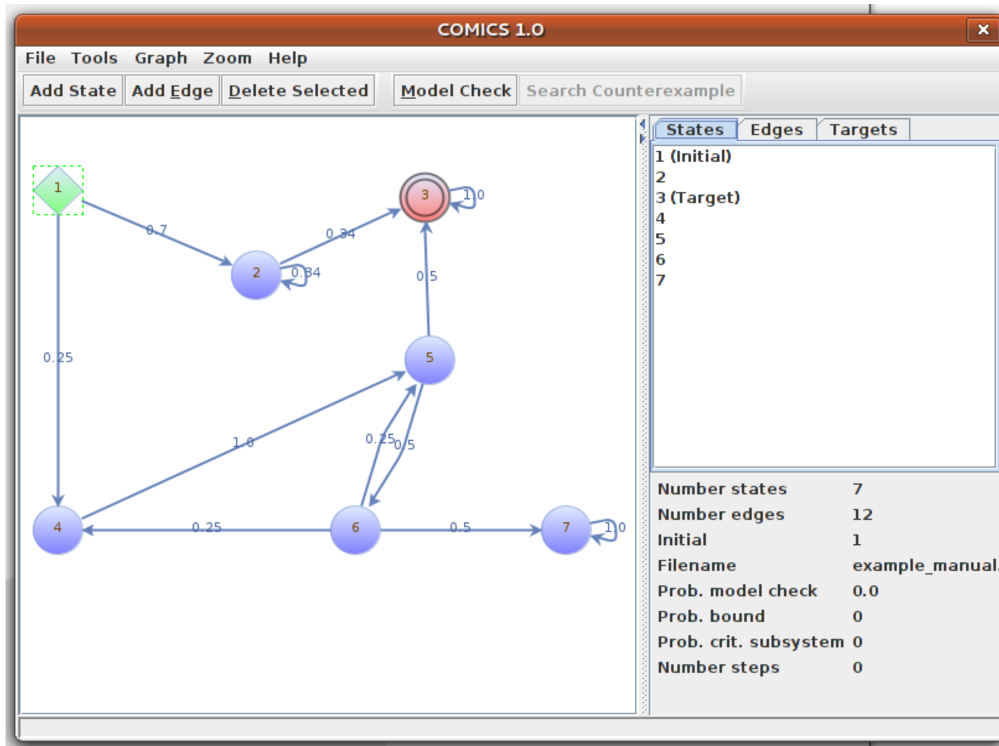


Figure 4.3: COMICS - GUI

- *Reset*: Resets the current DTMC
- *Open file*: Opens `.dtmc`, `.conf`, `.tra` or `.xml`
- *Import labels*: Opens `.lab` and gives the choice to declare target states and initial state by labels
- *Save*: Saves `.dtmc`
- *Save as XML*: Saves `.xml`[2]
- *Export to MRMC*: Saves `.tra`
- *Eport labels*: Writes information about target states and input states into `.lab`
- *Exit*: Exits the GUI

• Tools
  - *Search states*: Searches for states
  - *Center initial state*: Initial state is highlighted and centered
  - *Transform to 1 target*: Introduces new unique target state
  - *Hide isolated states*: States without ingoing or outgoing edges are hidden
  - *Check consistency*: Checks if all states have discrete probability distributions

• Graph
  - *Choose color*: The color of all state-types can be chosen
  - *Layout*: A number of different layout algorithms is offered

• Zoom

[2] Note, that if no model checking result is specified, the corresponding tag will be left empty inside the `.xml` file.

- Help

The Toolbar

The toolbar includes all necessary operations for creating and modifying graphs as well as buttons for model checking and counterexample generation. Note, that in some cases not all functions are available, in particular: First, the graph is not modifiable when an `.xml`-file is loaded, as the total model checking result of the system has to stay invariant. Secondly, before model checking is applied, no counterexample search is possible.

4.2.3 Creating a DTMC

The first operation is to add a state. This is simply done by activating the *Add State* button and clicking on the graph-panel, see Figure 4.4. Edges can be added, if the *Add Edge* Button is activated either by subsequently selecting the first and second state of the edge or by dragging a line from one state to another (Figure 4.5). A dialog pops up which asks for the probability of the edge. Note, that the entered number has to be a floating point number out of [0, 1].

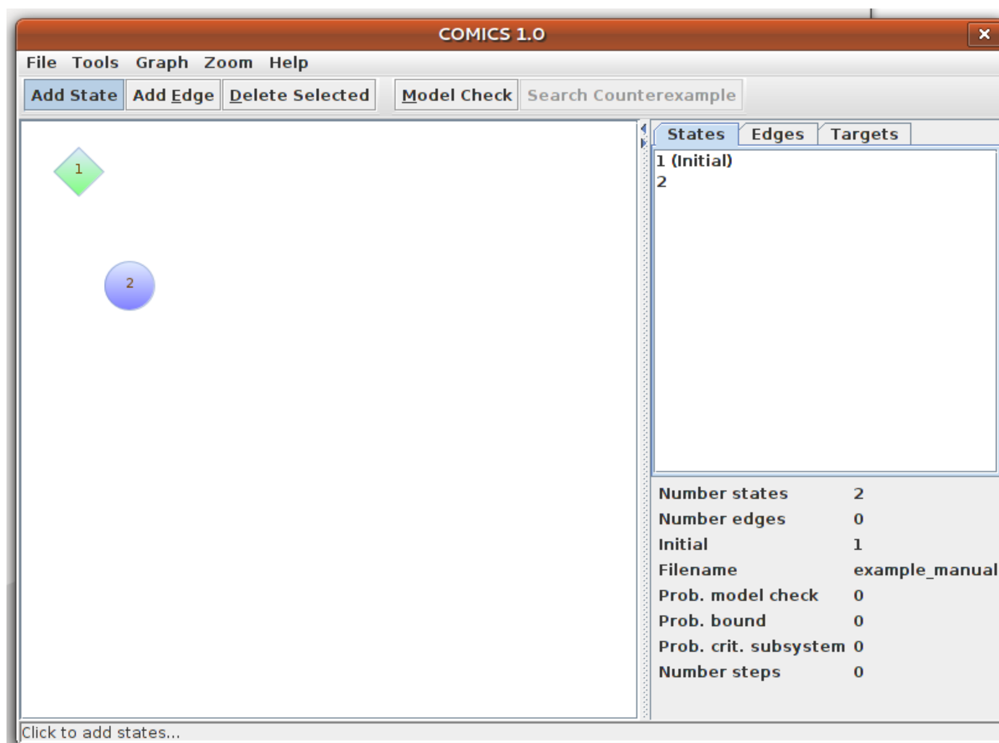


Figure 4.4: GUI - Add State

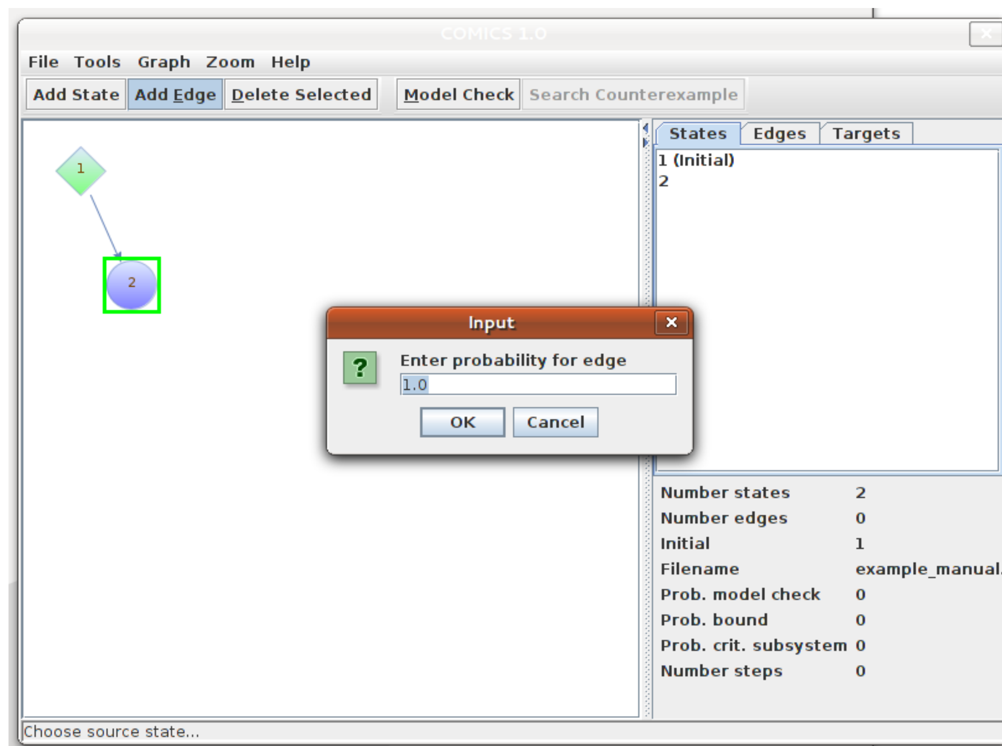


Figure 4.5: GUI - Add Edge

The context menu for states offers to select or de-select a state as target states (Figure 4.6) and to define the unique inital state.

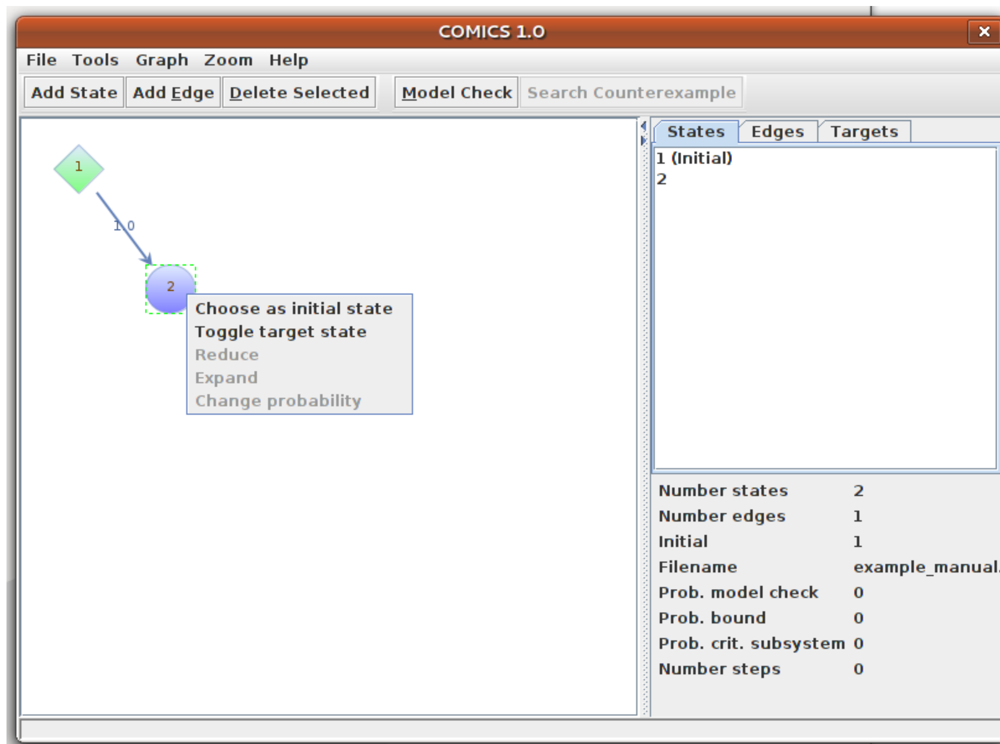


Figure 4.6: GUI - Context menu for states

4.2.4 Model Checking

If the initial state and at least one target state are specified, you can apply SCC-based model checking by just pressing the button *Model Checking*. The corresponding JNI library is called and the resulting abstract graph is displayed. Figure 4.7 shows the *abstract graph* resulting from applying model checking to the graph of Figure 4.3. The probability of reaching target states from the initial state is displayed in the *information-panel*. In case the input DTMC had states with outgoing probabilities less than 1, the user is asked, if corresponding self-loops should be added. The graph resulting from SCC-based model checking may be refinable. You can open the context menu of *abstract SCCs*, which are magenta colored rectangular states, by right-clicking them and - if available - choose the option *expand*. This is one *concretization step*. The result of expanding state 1 is depicted in Figure 4.8 and corresponds the example shown in Figure 1.2(b).

4.2.5 Counterexample Generation

You can now apply the search for a counterexample, this is done by pressing the button *Search Counterexample*. Note, that this button is only available, if the model checking result is defined. If you press the button, a dialogue pops up where the first parameters for the counterexample search can be specified (See Figure 4.9). The choices are:

- Perform a search on the *Abstract* system including concretization steps or a search directly on the *Concrete system*.
- Use the *Global* search algorithm or the *Local* search algorithm.
- Represent the counterexample as critical *Subsystem* or as a set of paths (*Pathset*).

If a `.conf` file was loaded, all options are selected as defined in the file. Note, that the local search implies the representation as critical subsystem.

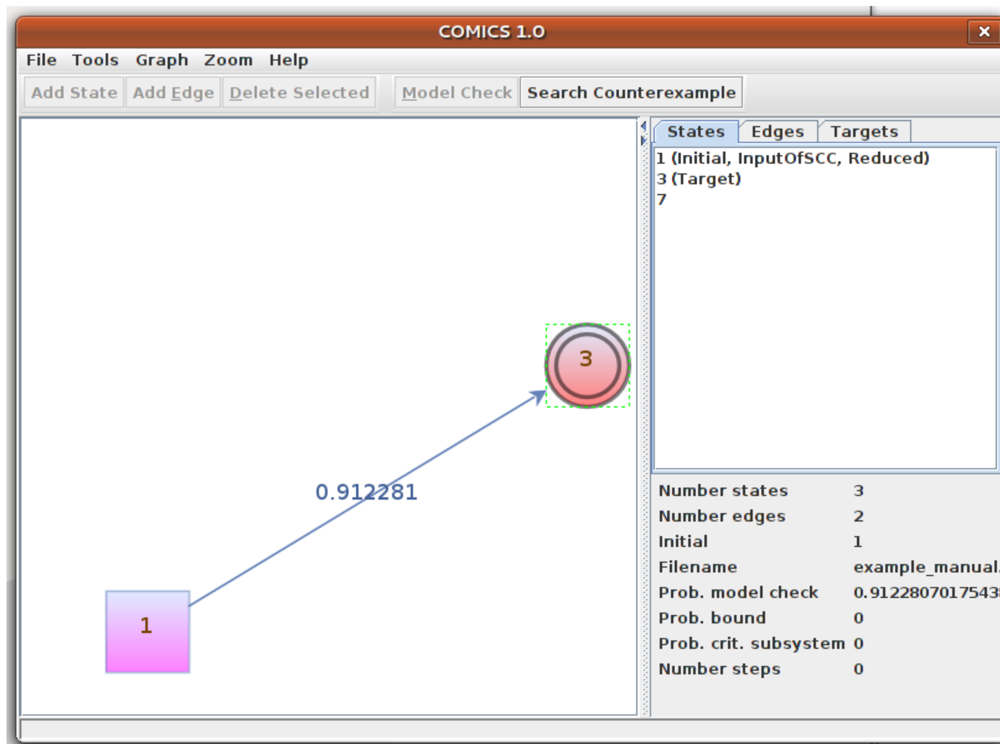


Figure 4.7: GUI - Model Checking Applied, Abstract State is refinable

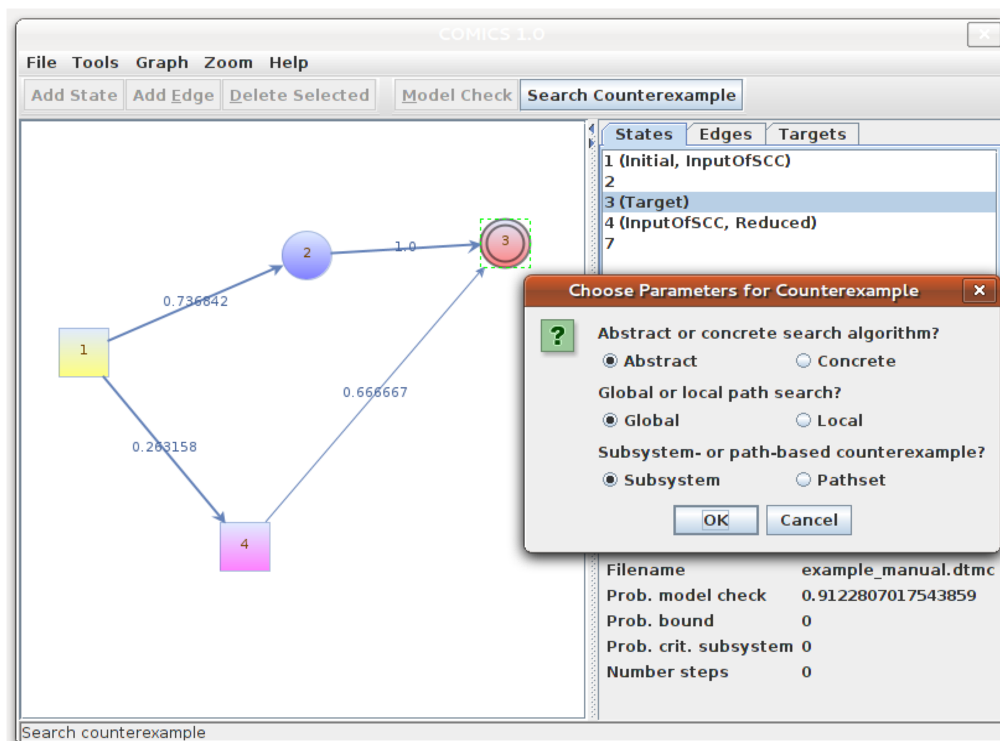


Figure 4.9: GUI - Specify Parameters for Counterexample Search

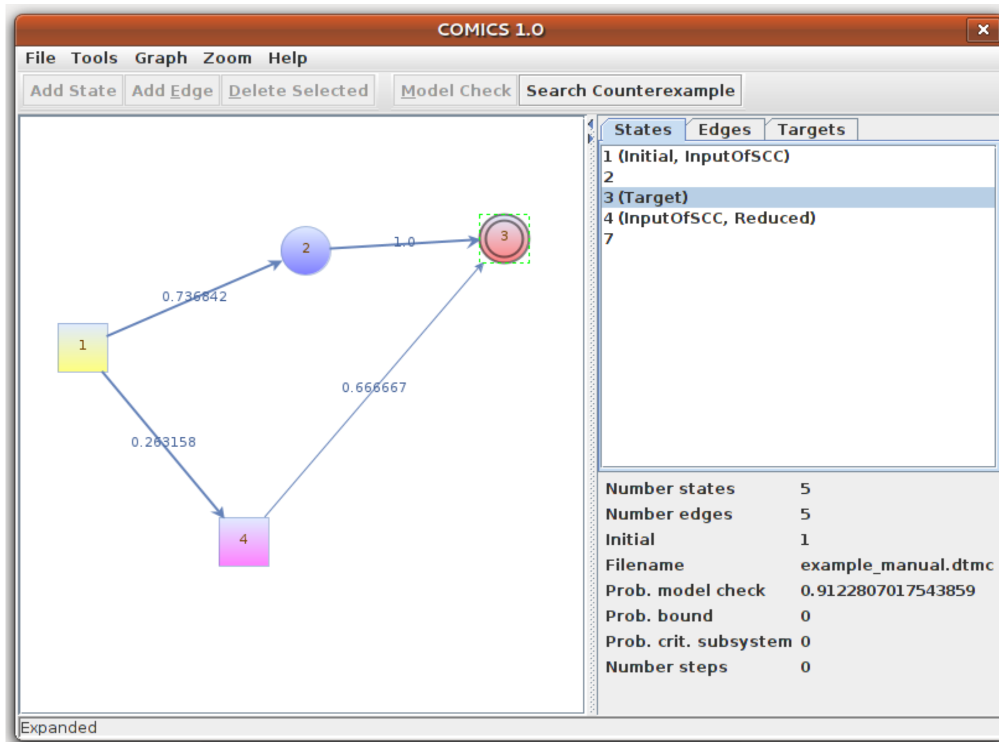


Figure 4.8: GUI - Abstract state 1 expanded

After the search parameters are set, you are asked to specify an probability bound which shall be reached or exceeded by the computed counterexample, see Figure 4.10. Offered is the actual model checking probability or, if available, the bound which is specified in the `.conf` file. The model checking probability is the highest bound that can be reached. Note, that in many cases this bound implies a hard computation.

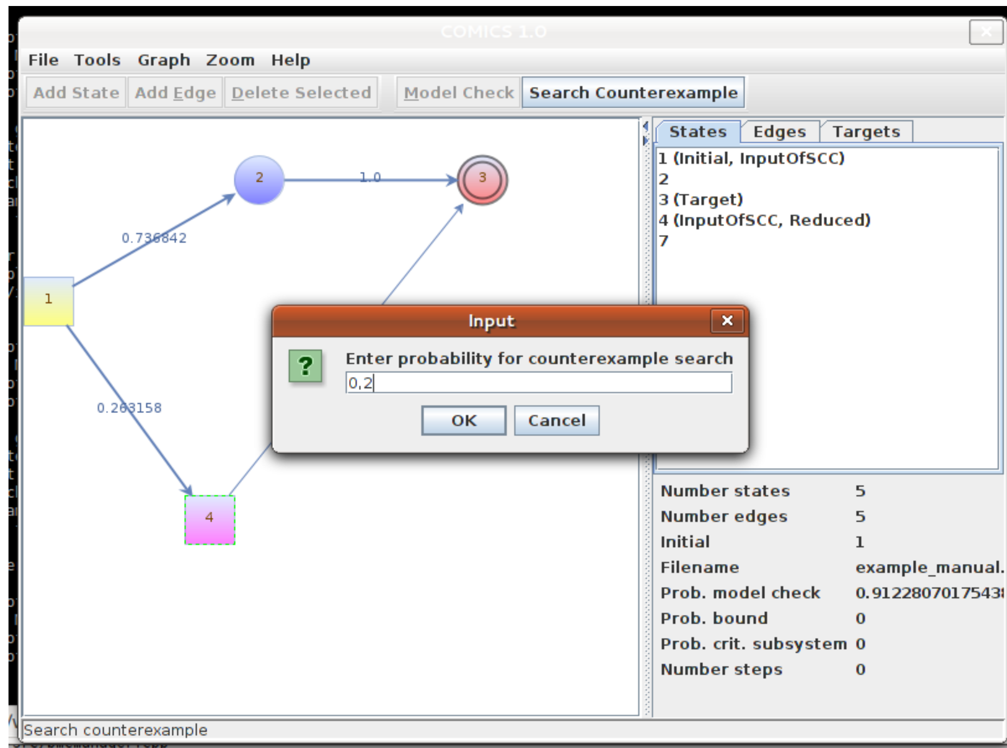


Figure 4.10: GUI - Specify Probability Bound for Counterexample

At last, several options are offered for the number of search iterations and the concretization of abstract SCCs (Figure 4.11):

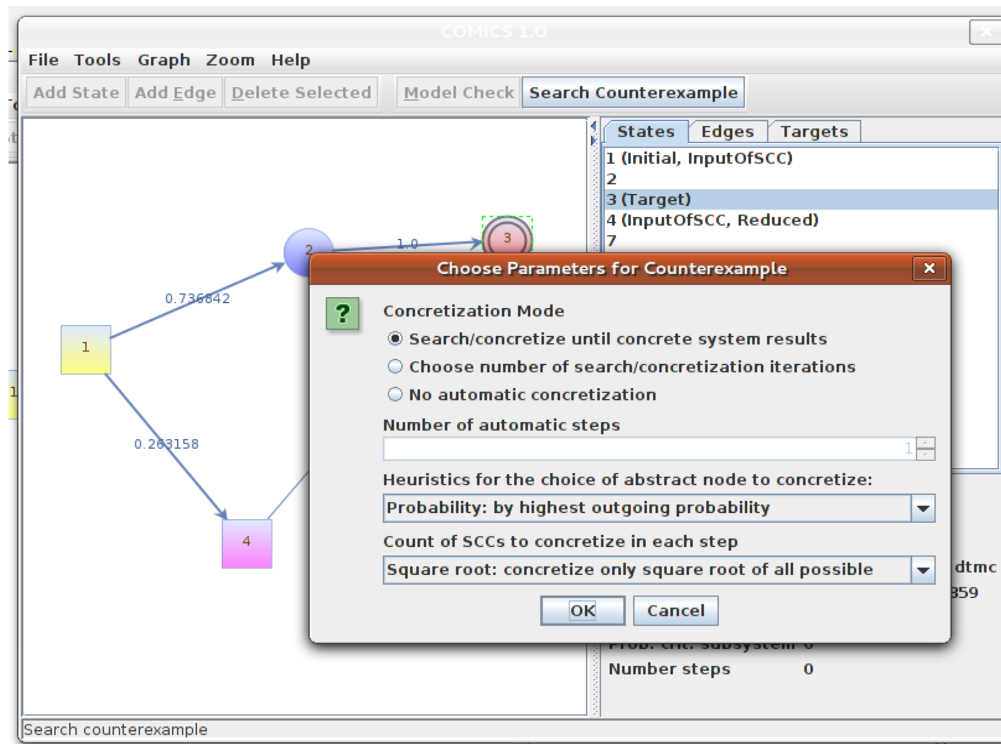


Figure 4.11: GUI - Specify Parameters for Concretization

Search/concretize until concrete system returns All concretizations and search steps are done automatically until a concrete system without any abstract SCCs is returned.

Choose number of search/concretization iterations The number of iterations can be specified by the user as *Number of automatic steps*. This may be very practical for large systems which cannot be displayed by the GUI, because after a certain number of iterations the current - still abstract - critical subsystem may be reasonably smaller.

No automatic concretization Concretization is only done by the user, who has the possibility to *expand* arbitrary states of interest (see Figures 4.7). One search iteration is done, and the critical subsystem is displayed. Afterwards, the user can start a new search. If no states were concretized before, an error message is given.

Heuristics for the abstract SCC to concretize As described in Section 4.1.2, we offer three heuristics for the choice of the next abstract SCC which will be automatically concretized. The three choices are:

Probability: by highest outgoing probability The abstract SCC which has the *highest* outgoing probability *inside* the critical subsystem is chosen.

Degree: by lowest input/output degree The abstract SCC with the *lowest* input/output-degree is chosen.

Membership: by highest relative membership in subsystem The abstract SCC with the *highest* number of ingoing and outgoing transitions is chosen. To this we refer as the *highest-membership* inside the critical subsystem.

Count of SCCs to concretize in each step The user can choose, how many of the currently visible abstract SCCs are to be concretized in *one* concretization step.

Square root: concretize only square root of all possible If n abstract SCCs are visible in the current critical subsystem, in each concretization step $\sqrt{n}$ abstract SCCs are concretized.

One: concretize only one Exactly one abstract SCCs is concretized in each step.

All: concretize all All visible abstract SCCs are concretized.

Finally, the concrete critical subsystem resulting if all parameters are chosen as offered for the input DTMC from Figure 4.3 is depicted in Figure 4.12.

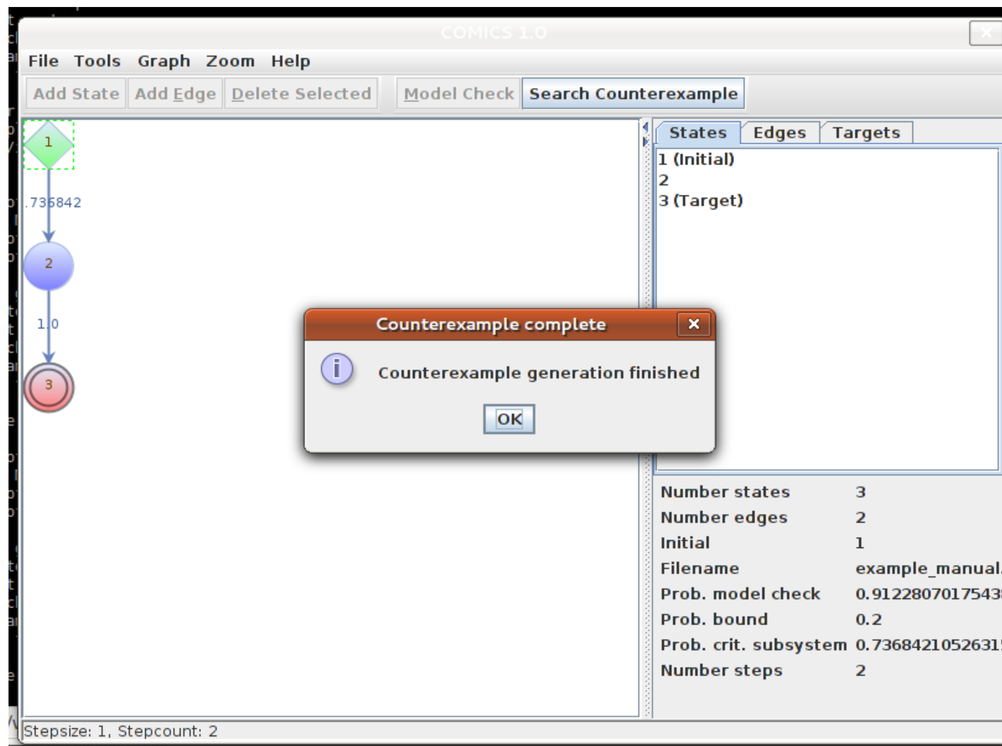


Figure 4.12: GUI - Resulting Critical Subsystem for example_manual.dtmc

Bibliography

[1] Erika Ábrahám, Nils Jansen, Ralf Wimmer, Joost-Pieter Katoen, and Bernd Becker. DTMC model checking by SCC reduction. In *Proc. of QEST*, pages 37–46. IEEE CS, 2010.

[2] Jay Bagga and Adrian Heinz. JGraph – A Java based system for drawing graphs and running graph algorithms. In *Proc. of Graph Drawing*, volume 2265 of *LNCS*, pages 459–460. Springer-Verlag, 2001.

[3] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking.* The MIT Press, 2008.

[4] Nils Jansen, Erika Ábrahám, Jens Katelaan, Ralf Wimmer, Joost-Pieter Katoen, and Bernd Becker. Hierarchical counterexamples for discrete-time Markov chains. In *Proc. of ATVA*, volume 6996 of *LNCS*, pages 443–452. Springer, 2011.

[5] Nils Jansen, Erika Ábrahám, Jens Katelaan, Ralf Wimmer, Joost-Pieter Katoen, and Bernd Becker. Hierarchical counterexamples for discrete-time Markov chains. Technical Report AIB-2011-11, RWTH Aachen University, 2011.

[6] Michael K. Reiter and Aviel D. Rubin. Crowds: Anonymity for web transactions. *ACM Trans. on Information and System Security*, 1(1):66–92, November 1998.