

LoLA

A Low Level Petri Net Analyser
<http://service-technology.org/lola>
Version 2.0, 21 May 2017

Karsten Wolf and Niels Lohmann

Table of Contents

1	About LoLA	1
1.1	Introduction	1
1.1.1	Objective	1
1.1.2	Input	1
1.1.3	Property	1
1.1.4	State Space Exploration	1
1.1.5	Structural Exploration	2
1.1.6	Memory	2
1.1.7	User interaction	2
1.1.8	Output	2
1.1.9	More information	2
1.2	Success stories	3
1.2.1	Verification of a GALS wrapper	3
1.2.2	Validation of a Petri net semantics for WS-BPEL	3
1.2.3	Verification of WS-BPEL choreographies	3
1.2.4	Garavel's Challenge in the Petri Net mailing list	4
1.2.5	Exploring biochemical networks	4
1.2.6	Soundness of business process models	4
1.2.7	Semantic web service composition	5
1.2.8	LoLA as a "sparring partner"	5
1.2.9	The Model Checking Contest at the PETRI NETS conferences	5
2	Bootstrapping LoLA	6
2.1	Setting up LoLA	6
2.1.1	Download	6
2.1.2	Setup and installation	6
2.1.3	Using OS X	7
2.2	Troubleshooting	7
2.3	First steps	7
3	Petri Net File Format	9
3.1	Informal description	9
3.2	EBNF grammar	9
3.3	Regular expressions for terminal symbols	10
3.4	Compatibility	10
4	Modes of operation	11
4.1	Check nothing ('--check=none')	11
4.2	Compute the state space ('--check=full')	11
4.3	Verify a property in temporal logic ('--check=modelchecking', default)	11

5	Properties	13
5.1	CTL* formulae	13
5.2	Bound properties	15
5.3	Syntax for properties	15
5.3.1	Constituents of a CTL* formulae	15
5.3.2	EBNF grammar for properties	16
5.3.3	Regular expressions for terminal symbols	16
5.4	Compatibility	16
5.5	Implicitly supported properties	17
5.5.1	Reachability	17
5.5.2	Invariance	17
5.5.3	Deadlocks	18
5.5.4	k -boundedness of a place	18
5.5.5	k -boundedness of the whole net	18
5.5.6	Boundedness of a place	19
5.5.7	Boundedness of the whole net	19
5.5.8	Dead transition	19
5.5.9	Quasi-liveness	19
5.5.10	Liveness of a transition	20
5.5.11	Liveness of the net	20
5.5.12	Liveness of a state predicate	20
5.5.13	Reversibility	20
5.5.14	Causal precedence of a transition	21
5.5.15	Relaxed Soundness of a workflow net	21
5.5.16	Soundness of a workflow net	21
5.6	Unsupported properties	21
5.7	General recommendations	21
6	Search strategies	23
6.1	Available values	23
6.1.1	Depth first search ('--search=depth', default)	23
6.1.2	Sweep-line method ('--search=sweep')	23
6.1.3	Coverability graph ('--search=cover')	24
6.2	Setting resource limits	24
6.3	Best practice	24
7	Reduction techniques	26
7.1	Partial order reduction: the stubborn set method	26
7.1.1	The method	26
7.1.2	Unique features in LoLA	26
7.1.3	Options	26
7.1.4	Best practice	27
7.2	The symmetry method	27
7.2.1	The method	27
7.2.2	Unique features in LoLA	27
7.2.3	Options	27
7.2.4	Best Practice	28
7.3	Cycle reduction	28
7.3.1	The method	28
7.3.2	Unique features in LoLA	28
7.3.3	Options	28

8	Nonstandard Verification Techniques	29
8.1	Memoryless search (<code>--findpath=ORDER</code>)	29
8.2	The state equation (<code>--stateequation=ORDER</code>)	29
8.3	The siphon/trap property (<code>--siphontrap=ORDER</code>)	30
9	Storage concepts	32
9.1	Encoding	32
9.1.1	Compression by capacities	32
9.1.2	Variable length encoding	32
9.1.3	Compression by place invariants	33
9.1.4	Meaning of the values	33
9.2	Store	34
9.2.1	Prefix tree (<code>--store=prefix</code> , default)	34
9.2.2	Bloom filter (<code>--store=bloom</code>)	34
9.2.3	STL store (<code>--store=stl</code>)	35
9.2.4	Comparison store (<code>--store=comp</code>)	35
9.3	Useful combinations of encoder and store	35
10	Output formats	37
10.1	Markings	37
10.2	Paths	37
10.3	JavaScript Object Notations (JSON)	38
10.3.1	Example	41
11	Error messages	43
11.1	Syntax errors [#01]	43
11.2	Command line errors [#02]	43
11.3	File input/output errors [#03]	44
11.4	Thread error [#04]	44
12	Integration guide	45
13	Utilities	46
13.1	Remote reporting (<code>listener</code>)	46
13.2	Remote termination (<code>killer</code>)	46
13.3	Translating PNML to LoLA	47
13.4	Translating XML property files to LoLA	47
14	Examples	48
14.1	Reader/writer mutual exclusion (<code>data</code>)	48
14.1.1	Overview	48
14.1.2	Write mutual exclusion	48
14.1.3	The complete state space as benchmark	51
14.1.4	Read/write mutual exclusion	51
14.2	Simple mutex algorithm (<code>mutex</code>)	52
14.2.1	Overview	52
14.2.2	Mutual exclusion (safety property)	52
14.2.3	Mutual exclusion (liveness property)	53
14.3	High-end multiprocessor architecture (<code>vasy</code>)	54
14.3.1	Overview	54
14.3.2	Quasi-liveness	54

15	Difference to Previous Releases	56
15.1	From LoLA 1 to LoLA 2	56
15.1.1	General workflow	56
15.1.2	Net input	56
15.1.3	Property specification	56
15.1.4	Supported properties	56
15.1.5	Atomic propositions	56
15.1.6	Reduction Techniques	56
15.1.7	Computed information	56
15.1.8	Progress messages	57
15.1.9	Multicore Support	57
15.1.10	Internal Architecture	57
15.1.11	Code Quality	57
15.2	From LoLA 2.0 to LoLA 2.1	57
15.2.1	List of formulas	57
15.2.2	Boolean combinations	57
15.2.3	Portfolio approach	57
15.2.4	Partial order reduction for LTL and CTL model checking	57
15.2.5	New stubborn set approach for reachability and deadlock	57
15.2.6	Bound properties	58
15.2.7	Efficiency improvements	58
15.2.8	Scalability issues	58
15.2.9	JSON output	58
15.2.10	AG/EF properties	58
15.3	Did you know?	58
	Index	60

1 About LoLA

1.1 Introduction

1.1.1 Objective

With LoLA, you can investigate properties of discrete systems that enjoy a high degree of concurrency, i.e. systems that have many causally unrelated events. Typical application domains include business process models, asynchronous circuits, biochemical reaction chains, protocols but exclude sequential software, real-time systems, and synchronous circuits.

1.1.2 Input

LoLA expects its input to be given as a Petri net, one of the most suitable formalisms for concurrent systems. LoLA uses the class of place/transition nets that has a very simple and clear semantics. A Petri net can be passed to LoLA in a clearly arranged ASCII based language. LoLA does not provide any graphical user interface for painting a Petri net. However, several graphical Petri net modeling tools are able to export files that can be read by LoLA or translated into the LoLA format (see [\[File formats\]](#), page [\[File formats\]](#)).

1.1.3 Property

Properties analyzed by LoLA include typical properties of concurrent systems such as deadlock freedom and properties specified in a temporal logic. Most properties have binary values, and LoLA shall return true, false, or unknown (if an incomplete technique is used, or a verification run has not finished). The only exception is the computation of maximum number of tokens. Here, the result is a natural number.

A property can be passed as command line parameter or as file. Given a Petri net and a property, LoLA explores the state space (reachability graph) of the Petri net to evaluate the property. For some properties, LoLA is also able to explore structural criteria. As soon as the value is determined, LoLA quits state space exploration. LoLA is not a tool for randomly playing a manually controlled token game, nor for using the Petri net as a control engine for any kind of environment, nor for visualising the state space.

1.1.4 State Space Exploration

The state space exploration methods implemented in LoLA belong to the class of *explicit* techniques, as opposed to symbolic (e.g. BDD based or SAT based) exploration methods. That is, states are explored one by one and state explosion is tackled by exploring only a part of the state space. This part is by construction equivalent to the full state space with respect to the investigated property. The collection of available state space reduction techniques is unique in several regards:

- With the partial order reduction, the symmetry method, the sweep-line method, the coverability graph construction, and other techniques, LoLA provides a larger collection of explicit-state reduction techniques than any other explicit state exploration tool.
- In LoLA, many of these techniques can be applied jointly thus getting better state space reduction.
- For the partial order reduction, LoLA uses unique methods that provide more reduction power than the popularly used LTL preserving or CTL preserving methods
- The symmetry method used in LoLA is based on the graph automorphisms of the Petri net. This method is able to deal with arbitrarily complex patterns of symmetry, as opposed to the scalar set approach used in several other tools.

- LoLA takes care that, for the given property, application of a reduction technique is sound. It automatically chooses a version of the respective method that preserves the property under investigation. If there is no such version, the reduction technique is not applied.
- LoLA can apply the sweep-line method without requiring the user to provide a progress measure – it computes a suitable progress measure on its own.
- LoLA employs results from the unique Petri net structure theory for further state space reduction or efficiency benefits. In particular, it employs information about place invariants, transition invariants, and conflict clusters of the net.

1.1.5 Structural Exploration

For some properties, LoLA provides purely structural verification methods stemming from Petri net theory. These methods can be applied instead of, prior to, or parallel to state space exploration. These methods include:

- The siphon/trap-property for deadlock check
- The Petri net state equation for reachability queries

1.1.6 Memory

LoLA uses the main memory of your machine for handling the state space. It supports several data structures for representing the states. Some of these structures are inherently subject to loss of information, that is they do not guarantee exhaustive search. If LoLA is run on a multicore machine, it can distribute some of its computations over a given number of cores.

1.1.7 User interaction

Once started, LoLA does not require any user interaction. It is purely working in batch mode. The only action that a user can take on a running LoLA is to abort it. There is no way to interactively query a once computed state space. If you want to explore several properties, you want to run LoLA multiple times. The reason for that design is that reduction techniques depend on the given property, so there is hardly one state space that is able to answer more than one question. Unlike some other verification tools, LoLA does not produce source code that needs subsequent compilation.

1.1.8 Output

Output by LoLA is organized such that it is easy to use LoLA as a backend engine in other tools, or in scripts. If applicable, LoLA supports its results by witness states, or witness paths. LoLA can also provide results gathered in preprocessing, it reports run time and size of the explored part of the state space, and it emits progress messages. Most messages can be suppressed, be deferred to files, or deferred to a remote machine. For further processing, LoLA can collect all produced information as a JSON (Javascript Object Notation) object.

1.1.9 More information

- Karsten Schmidt. **LoLA: A Low Level Analyser**. In *Mogens Nielsen and Dan Simpson, editors, Application and Theory of Petri Nets 2000: 21st International Conference, ICATPN 2000, Aarhus, Denmark, June 2000*. Proceedings, volume 1825 of Lecture Notes in Computer Science, pages 465-474, June 2000. Springer-Verlag.
- Karsten Wolf. **Generating Petri Net State Spaces**. In *Jetty Kleijn and Alex Yakovlev, editors, 28th International Conference on Applications and Theory of Petri Nets and Other Models of Concurrency, ICATPN 2007, Siedlce, Poland, June 25-29, 2007*. Proceedings, volume 4546 of Lecture Notes in Computer Science, pages 29-42, June 2007. Springer-Verlag. Invited lecture

- Karsten Wolf. **The Petri net twist in explicit model checking.** *Software and System Modeling* 14(2): pages 711-717 (2015)
- Karsten Wolf. **Running LoLA 2.0 in a Model Checking Competition.** *Transactions on Petri Nets and Other Models of Concurrency* 11: pages 274-285 (2016)

1.2 Success stories

1.2.1 Verification of a GALS wrapper

A GALS circuit is a complex integrated circuit where several components operate locally synchronously but exchange information asynchronously. GALS technology promises lower energy consumption and higher clock frequency.

In a joint project, researchers at Humboldt-Universität zu Berlin and the Semiconductor Research Institute in Frankfurt (Oder) analyzed a GALS circuit that implements a device for coding/decoding signals of wireless LAN connections according to the 802.11 protocol. They were particularly concerned with parts of the circuit they called wrapper. A wrapper is attached to each synchronous component of a GALS circuit. It is responsible for managing the asynchronously incoming data, pausing the local clock in case of no pending data, and shipping the outgoing signals to the respective next component. They modeled a wrapper as a place-transition net and analyzed the occurrence of hazard situations. A hazard is a situation where, according to two incoming signals within a very short time interval, output signals may assume undefined values. In the model, a hazard situation corresponds to a particular reachable state predicate. LoLA was used with stubborn sets and the sweep-line method as reduction techniques. Analysis revealed 8 hazard situations in the model. 6 of them were ruled out by the engineers due to timing constraints which were not modeled. The remaining 2 hazards were confirmed as real problems. The circuit was redesigned and another verification confirmed the absence of hazard situations.

More information:

- Christian Stahl, Wolfgang Reisig, and Milos Krstic. **Hazard Detection in a GALS Wrapper: A Case Study.** In Jörg Desel and Y. Watanabe, editors, *Proceedings of the Fifth International Conference on Application of Concurrency to System Design (ACSD'05)*, St. Malo, France, pages 234-243, June 2005. IEEE Computer Society.

1.2.2 Validation of a Petri net semantics for WS-BPEL

The language WS-BPEL has been proposed by an industrial consortium for the specification of Web services. Researchers at Humboldt-Universität zu Berlin proposed a formal semantics for WS-BPEL on the basis of high-level Petri nets (with a straightforward place-transition net abstraction that ignores data dependencies). Due to tricky concepts in the language, the translation of WS-BPEL to Petri nets required a validation. The validation was carried out through an automated translation of WS-BPEL into Petri nets and a subsequent analysis of the resulting Petri nets using LoLA. LoLA was used with stubborn sets and the sweep-line method as most frequently used reduction techniques.

More information:

- Sebastian Hinz, Karsten Schmidt, and Christian Stahl. **Transforming BPEL to Petri Nets.** In Wil M. P. van der Aalst, B. Benatallah, F. Casati, and F. Curbera, editors, *Proceedings of the Third International Conference on Business Process Management (BPM 2005)*, volume 3649 of Lecture Notes in Computer Science, Nancy, France, pages 220-235, September 2005.

1.2.3 Verification of WS-BPEL choreographies

The language WS-BPEL has been proposed by an industrial consortium for the specification of Web services. Researchers at Humboldt-Universität zu Berlin developed a tool for translating

WS-BPEL processes and choreographies into place-transition nets. LoLA has been used for checking several properties on the choreographies. They used stubborn sets and the symmetry method. The latter method turned out to be useful in those cases where choreographies involved a large number of instances of one and the same process. This way, choreographies with more than 1,000 service instances could be verified.

More information:

- Niels Lohmann, Oliver Kopp, Frank Leymann, and Wolfgang Reisig. **Analyzing BPEL4Chor: Verification and Participant Synthesis.** In Marlon Dumas and Reiko Heckel, editors, *Web Services and Formal Methods, Forth International Workshop, WS-FM 2007*, Brisbane, Australia, September 28-29, 2007, Proceedings, volume 4937 of Lecture Notes in Computer Science, pages 46-60, 2008. Springer-Verlag.

1.2.4 Garavel's Challenge in the Petri Net mailing list

In 2003, H. Garavel posted a place/transition net to the Petri net mailing list. It consisted of 485 places and 776 transitions. He was interested in quasi-liveness, i.e. the absence of any transition that is dead in the initial marking. According to the posting, the example stems from the translation of a LOTOS specification into Petri nets. There were four responses reporting successful verification. One of them involved LoLA. With LoLA, we checked each transition separately for non-death. We succeeded for all but two transitions. For the remaining transitions, goal-oriented execution confirmed non-death. According to the other responses which involved either symbolic (BDD based) verification or the use of the covering step graph technique, the full state space consisted of almost 10^{22} states.

The Petri net is part of the LoLA distribution and can be found in the `examples/vasy` folder. The example files are explained in Chapter 14 [Examples], page 48.

More information:

- The original posting: <http://www.informatik.uni-hamburg.de/cgi-bin/TGI/pnml/getpost?id=2003/07/2709>
- The summary of responses: <http://www.informatik.uni-hamburg.de/cgi-bin/TGI/pnml/getpost?id=2003/09/2736>

1.2.5 Exploring biochemical networks

A biochemical network reflects substances and known reactions for their mutual transformation. Researchers at SRI use LoLA in the exploration of Petri net models of such networks. They use the capability of LoLA to produce witness paths which are interpreted as reaction sequences.

More information:

- Carolyn Talcott, David L. Dill. **The pathway logic assistant.** *Third International Workshop on Computational Methods in Systems Biology, 2005.*

1.2.6 Soundness of business process models

Soundness is a fundamental correctness criterion in the area of business process models. There exists several domain-specific approaches to verify soundness, for instance graph-theoretic approaches exploiting block structured models over algebraic techniques based on invariants. An experiment conducted with industrial business process models from IBM customers showed that LoLA could verify the models in a matter of milliseconds and hence offered the same performance than domain-specific techniques.

More information:

- Dirk Fahland, Cédric Favre, Jana Koehler, Niels Lohmann, Hagen Völzer, and Karsten Wolf. **Analysis on Demand: Instantaneous Soundness Checking of Industrial Business Process Models.** *Data Knowl. Eng.*, 70(5):448-466, 2011.

- The Petri net files are available for download at <http://service-technology.org/soundness>

1.2.7 Semantic web service composition

Given a formally specified goal, a web service composition algorithm aims at the automated selection and composition of services from a given repository that is able to reach the goal. The problem can be translated into the reachability problem for a Petri nets that is generated from the goal and the ontology based descriptions of the services in the repository. Using LoLA for solving the problem turned out to be an interesting alternative to the use of standard AI procedures.

More information:

- Artur Niewiadomski and Karsten Wolf. **LoLA as Abstract Planning Engine of PlanICS.** *Proc. iWorkshop PNSE*, :349-350, 2015.

1.2.8 LoLA as a “sparring partner”

Some researchers have compared the performance of their domain-specific solution of selected problems to the performance of LoLA on a Petri net translation of that problem:

- in the context of verifying parameterized Boolean programs
- in the context of task partitioning of multiprocessor embedded systems.

In all cases, experimental results suggest excellent performance of LoLA: not significantly worse than the promoted domain-specific tool and better than other general purpose tools mentioned.

More information:

- A. Kaiser, D. Kroening, T. Wahl. **Dynamic cutoff detection in parameterized concurrent programs.** *CAV 2010*.
- D. Das, P.P. Chakrabarti, R. Kumar. **Functional verification of task partitioning for multiprocessor embedded systems.** *ACM Transactions on Design Automation of Electronic Systems (TODAES) Volume 12 Issue 4, September 2007 Article No. 44*

1.2.9 The Model Checking Contest at the PETRI NETS conferences

The contests have been established in 2011. In all issues since then, LoLA participated in various disciplines and proved competitiveness with other Petri net verification tools.

More information:

- <http://mcc.lip6.fr>

2 Bootstrapping LoLA

2.1 Setting up LoLA

2.1.1 Download

The most recent version of LoLA can be downloaded at <http://service-technology.org/files/lola>. As of May 2017, the most recent version is **2.0**.

The use of LoLA is free under the GNU Affero General Public License (AGPL) which is part of the distribution (see file COPYING). An online version is available at <http://www.gnu.org/licenses/agpl-3.0.html>.

2.1.2 Setup and installation

To unpack the downloaded tarball `lola-2.0.tar.gz`, go to your download directory and execute

```
$ gunzip lola-2.0.tar.gz
$ tar xf lola-2.0.tar
```

This creates a directory `lola-2.0` which contains the LoLA distribution. You then need to configure LoLA by executing

```
$ cd lola-2.0
$ ./configure
```

The configuration should finish with a message like

```
=====
Successfully configured LoLA 2.0.
-> compile LoLA with 'make'.
=====
```

Then, execute

```
$ make
```

to compile LoLA. You may ignore potential compiler warnings. If everything is OK, you should see a message like

```
=====
Successfully compiled LoLA 2.0.
-> check out LoLA's help with 'src/lola --help'
-> install LoLA to /usr/local/bin with 'make install'
=====
```

This indicates that the LoLA binary `lola` has been successfully built in the `src` directory.

To install LoLA, you may optionally execute

```
$ sudo make install
```

to copy all required files to `/usr/local/bin`. An installed LoLA has the advantage that LoLA can be called from anywhere and that you will not need the `lola-2.0` directory any more. In the following, we assume that LoLA has been installed and can be called by simply executing `'lola'`.

You can change the installation directory by calling `'./configure --prefix=DIRECTORY'`. Then, the required files will be installed to `DIRECTORY/bin`.

2.1.3 Using OS X

With Apple OS X, you can easily install LoLA using the Homebrew package manager (<http://brew.sh>) which provides a ‘brew’ command. You can then install LoLA with

```
$ brew cask homebrew/science
$ brew install lola
```

You can also install the latest development version from LoLA’s source code repository by executing

```
$ brew install lola --HEAD
```

Any required tools are installed automatically.

2.2 Troubleshooting

The only prerequisite of LoLA is a working C++ compiler such as GCC or Clang. Successful compilations have been reported from machines running GNU/Linux, Apple OS X, and Microsoft Windows (running cygwin). We are aware of problems on Solaris and FreeBSD machines, but have too little input to fix these issues at the moment. Feedback would be greatly appreciated!

If an error occurs, you should see a warning like:

```
configure: error: C++ compiler cannot create executables
See ‘config.log’ for more details.
```

In that case, please review the file `config.log`. If you cannot resolve the problem, please send the file to lola@service-technology.org.

If the compilation succeeds but you encounter any other problems, you can help us by executing

```
$ ./configure --enable-debug
$ make check
```

which runs a large test suite.¹ Please then explain your problem with LoLA in an email to lola@service-technology.org and attach the file `tests/testsuite.log`.

2.3 First steps

As first step, you may want to get to know LoLA and execute

```
$ lola --help
```

to display information about the command line parameters of LoLA. This gives you a brief overview of the most important parameters of LoLA. With

```
$ lola --detailed-help
```

a more detailed help is printed.

In most of the cases, LoLA requires at least one input file (usually a Petri net) and a parameter which property to check. To an example, change to the ‘`examples/mutex`’ directory of the LoLA distribution and execute

¹ Note you need GNU Autoconf to generate the testsuite and GNU Bash to run it.

```
$ lola mutex.lola --formula="EF DEADLOCK"
```

This will make LoLA check for deadlocks ('--formula="EF DEADLOCK"') in the Petri net given as file `mutex.lola`. Beside a lot of output, you shall observe the message

```
lola: RESULT
lola:  result: no
lola:  produced by: state space
lola:  The net does not have deadlocks.
lola:  6 markings, 10 edges
```

indicating that this net is deadlock free. In the remainder of this manual, all other parameters are explained in detail.

3 Petri Net File Format

LoLA uses a proprietary file format. It aims at being both simple to write or generate and easy to parse while trying to be as little verbose as possible. Consequently, we decided against the standard PNML file format. In our distribution, you shall find utilities that help you translating PNML files into LoLA file, though, see [Utilities], page 46.

In the following, we provide the EBNF grammar for the file format. Thereby, we use the following conventions:

- Terminal symbols such as keywords are given in quotes, e.g. ‘PLACE’. Thereby, ‘IDENTIFIER’ and ‘NUMBER’ denote a placeholder for identifiers and numerical constants, respectively, for which we provide a regular expression in [Regular expressions for terminal symbols], page 10.
- Any non-capitalized word denotes a nonterminal symbol which is eventually defined using ‘:=’.
- Alternatives are separated with ‘|’. Symbols followed by ‘?’ may be skipped. Symbols followed by ‘+’ may be repeated an arbitrary number of times. Symbols followed ‘*’ may be repeated an arbitrary number of times or skipped. Parentheses are used to group symbols with respect to skipping and repetition.

3.1 Informal description

LoLA’s Petri net file format consists of three parts:

1. The places are defined as a comma-separated list of place names, beginning with keyword ‘PLACE’. For a (sub-)list of places, an optional token bound can be defined using keyword ‘SAFE’ followed by a number (1 if no number is provided). This bound is not enforced by LoLA, but rather used to allow for bit-perfect data structures. If no bound is given, $2^{32} - 1$ is used as maximal bound. If you are analysing Petri nets where the number of tokens exceeds the bound, the results of LoLA may become incorrect.
2. An initial marking is a comma-separated list of assignments of place names with token numbers, beginning with keyword ‘MARKING’. If no number is provided (i.e., only a place name is listed), an initial marking of one token is assumed. If places are mentioned several times, the corresponding number of tokens is added. That is, MARKINGp1 , p1 : 1; results in an initial marking with two tokens on place p1.
3. Transitions are specified one by one, each beginning with keyword ‘TRANSITION’, followed by a name and an optional fairness assumption (‘STRONG FAIR’ or ‘WEAK FAIR’). Then, the transition’s preset (‘CONSUME’) and postset (‘PRODUCE’) is given in the same fashion of a marking in the ‘MARKING’ section (i.e., an arc weight of 1 is assumed if not given explicitly).

A lot of example files are provided in the `examples` folder of the LoLA distribution.

3.2 EBNF grammar

```

net ::= 'PLACE' place_lists 'MARKING' marking_list? ';' transition+

place_lists ::= ( capacity? place_list ';' )+

capacity ::= 'SAFE' 'NUMBER'? ':'

place_list ::= nodeident ( ',' nodeident )*

nodeident ::= 'IDENTIFIER' | 'NUMBER'

marking_list ::= marking ( ',' marking )*

marking ::= nodeident ( ':' 'NUMBER' )?

```

```

transition ::= 'TRANSITION' nodeident fairness?
              'CONSUME' arc_list? ';' 'PRODUCE' arc_list? ';'

fairness ::= ( 'STRONG' | 'WEAK' ) 'FAIR'

arc_list ::= arc ( ',' arc )*

arc ::= nodeident ( ':' 'NUMBER' )?

```

3.3 Regular expressions for terminal symbols

NUMBER

Any nonempty sequence of digits.

`"-"?[0-9]+`

IDENTIFIER

Any nonempty sequence of characters, excluding `'`, `,`, `;`, `:`, `(`, `)`, `{`, `}`, or whitespace (spaces, tabs, or newlines).

`[^,;:()\t\n\r\{\}]`

3.4 Compatibility

Low level Petri net files from earlier versions of LoLA (before version 2.0) can be read without any adjustments. High level Petri net files need to be translated into a low-level version first. This can be done using the `-n` option available in LoLA 1.x.

4 Modes of operation

You select the mode of operation using the ‘`--check=MODE`’ command line option where ‘`MODE`’ can be one of the values ‘`none`’, ‘`full`’, or ‘`modelchecking`’. If no value is given, ‘`modelchecking`’ is used as default.

4.1 Check nothing (‘`--check=none`’)

LoLA only performs preprocessing but does not do any actual state space exploration. This option is useful for checking whether the input is syntactically correct, or for getting information from preprocessing.

```
$ lola --check=none phils10.lola
lola: NET
lola:   reading net from phils10.lola
...
lola: TASK
lola:   checking nothing (--check=none)
...
```

4.2 Compute the state space (‘`--check=full`’)

LoLA explores the reachable states without evaluating any particular property. This option is useful for experiencing the size of a state space. It applies the selected state space reduction techniques.

```
$ lola --check=full phils10.lola
lola: NET
lola:   reading net from phils10.lola
...
lola: TASK
lola:   creating the whole state space (--check=full)
...
lola: RESULT
lola:   result: no
lola:   State space generated.
lola:   272 markings, 370 edges
...
```

Note that some reduction techniques are switched on by default and need to be switched off for getting the brute force state space.

```
$ lola --check=full phils10.lola --stubborn=off
...
lola: RESULT
lola:   result: no
lola:   State space generated.
lola:   59048 markings, 393650 edges
...
```

4.3 Verify a property in temporal logic (‘`--check=modelchecking`’, default)

LoLA evaluates a property (see below). The particular property is passed to LoLA by the command line option ‘`--formula=FORMULA`’. The value ‘`FORMULA`’ is either a string that directly describes a formula, or file name that contains such a string. For linear time properties, the

formula can be replaced by a Büchi automaton. In this case, LoLA searches for a path that is accepted by the given automaton. The automaton is passed to LoLA by the command line option ‘--buechi=STRING’ where ‘STRING’ is a file name containing the description of a Büchi automaton.

```
$ lola --formula="EF ea.2 > 0" phils10.lola
lola: NET
lola:   reading net from phils10.lola
lola:   finished parsing
lola:   closed net file phils10.lola
lola:   90/65536 symbol table entries, 0 collisions
lola:   preprocessing...
lola:   finding significant places
lola:   50 places, 40 transitions, 30 significant places
lola:   computing forward-conflicting sets
lola:   computing back-conflicting sets
lola:   60 transition conflict sets
lola: TASK
lola:   E (F ((1 <= ea.2)))
lola:   E (F ((1 <= ea.2)))
lola:   checking reachability
lola:   Planning: workflow for reachability check: search (--findpath=off)
lola:   processed formula: E (F ((1 <= ea.2)))
lola:   processed formula length: 19
lola:   1 rewrites
lola: STORE
lola:   using a bit-perfect encoder (--encoder=bit)
lola:   using 4 bytes per marking, with 2 unused bits
lola:   using a prefix tree store (--store=prefix)
lola: SEARCH (state space)
lola:   state space: using reachability graph (--search=depth)
lola:   state space: using reachability preserving stubborn set method with insertion algo-
rithm (--stubborn=tarjan)
lola: RUNNING
lola: RESULT
lola:   result: yes
lola:   produced by: state space
lola:   The predicate is reachable.
lola:   3 markings, 2 edges
lola: memory consumption: 3224 KB
lola: time consumption: 0 seconds
```

5 Properties

There are two kinds of properties: properties specified in the temporal logic CTL*, and bound properties. For a CTL* property, LoLA computes its Boolean value. For a bound property, LoLA computes its numerical value. You can call LoLA with a ':'-separated list of properties where CTL* and bound properties may be arbitrarily mixed. LoLA processes such a list sequentially. For each property, a separate child process is spawned that produces its own state space. Only if a child process terminates, the next process is launched. This way,

- There is no competition for computing resources among the child processes;
- The efforts for resetting data structures between two state space explorations are deferred to the operating system.

The given command line options refer to all listed properties. LoLA reports status and result separately for every property. After termination of all state space explorations, LoLA produces a summary of the individual results. If LoLA is aborted, it reports the results obtained so far.

Example:

```
> lola phils10.lola --formula="EF DEADLOCK : AGEF INITIAL : MAX(ea.1 + ea.2)"
...
lola: =====
lola: ...considering subproblem: E (F (DEADLOCK))
lola: =====
lola: SUBTASK
lola:   checking reachability of deadlocks
...
lola: SUBRESULT
lola:   result: yes
lola:   produced by: state space
lola:   The net has deadlock(s).
...
lola: =====
lola: ...considering subproblem: A (G (E (F (((fo.10 <= 1) AND (1 <= fo.10) AN... (shortened)
lola: =====
...
lola: SUBRESULT
lola:   result: no
lola:   produced by: CTL model checker
lola:   The net does not satisfy the given formula.
...
lola: =====
lola: ...considering subproblem: MAX(ea.1 + ea.2)
lola: =====
...
lola: SUBRESULT
lola:   result: 1
lola:   produced by: state space
lola:   The maximum value of the given expression is 1
...
lola: RESULT
lola:
SUMMARY:  yes no 1
...
```

5.1 CTL* formulae

Syntactically, LoLA can read (but not necessarily verify) any CTL* formula. After reading the formula, LoLA first rewrites the formula according to known tautologies of temporal logic. This way, the verification problem finally approached by LoLA may be simpler than the one originally given.

In a second step, LoLA categorizes the resulting formula. For each category, dedicated verification routines are available. LoLA uses the following categories (**phi** represents a formula without temporal operators):

- CTL – A formula in the CTL fragment of CTL*
- LTL – A formula in the LTL fragment of CTL*
- Reachability – A formula of shape EF **phi**
- Invariance – A formula of shape AG **phi**
- Deadlock – A formula of shape EF DEADLOCK
- NoDeadlock – A formula of shape AG NOT DEADLOCK
- Initial – A formula of shape **phi**
- Liveness – A formula of shape AG EF **phi**
- Possible Liveness – A formula of shape EF AG EF **phi**
- Possible invariance – A formula of shape EF AG **phi**
- Always possible invariance – A formula of shape AG EF AG **phi**

If a formula happens to be a Boolean combination of subformulas, each subformula is categorized on its own. Furthermore, LoLA explores each such subformula separately (in sequence) using the technique that fits to the category of the respective subformula.

```
$ lola --formula="EF ea.2 > 0 AND FG hl.1 = 0" phils10.lola
...
lola: TASK
lola:   (E (F ((1 <= ea.2))) AND F (G ((hl.1 <= 0))))
lola:   checking a Boolean combination of subproblems
lola: RUNNING
lola:   =====
lola:   ...considering subproblem: E (F ((1 <= ea.2)))
lola:   =====
lola:   SUBTASK
lola:     checking reachability
...
lola: SUBRESULT
lola:   result: yes
lola:   produced by: state space
lola:   The predicate is reachable.
lola:   3 markings, 2 edges
lola:   =====
lola:   =====
lola:   ...considering subproblem: F (G ((hl.1 <= 0)))
lola:   =====
lola:   SUBTASK
lola:     checking stabilization
...
lola: SUBRESULT
lola:   result: no
lola:   produced by: LTL model checker
lola:   The net does not satisfy the given formula (language of the product automaton is nonempty).
lola:   12 markings, 13 edges
lola:   =====
lola: RESULT
lola:   result: no
lola:   The Boolean predicate is false.
...
```

If a formula fits into more than one category, the more specific category is chosen. That is, EF **phi** is categorised as reachability although it is as well a valid CTL formula. If LTL and CTL are the remaining candidates, LTL is preferred. That is, input AF **phi** will be verified as the LTL

formula $F \phi$. The latter priority can be overridden by the user with the command line option `--preference=ctl`. The choice between LTL and CTL has a semantic consequence when the system contains deadlocks. In a deadlock state, LTL assumes that the system stutters in this state, that is, the deadlock state is assumed to be repeated forever. CTL evaluates the existential path quantifier to false and the universal path quantifier to true, interpreting a deadlock as a state that does not have any outgoing path. This means that, in a deadlock state, "AX FALSE" evaluates to true in CTL and to false in LTL.

The categorisation of LoLA actually has a few more classes:

- Fairness – A formula of shape $GF \phi$
- Stabilisation – A formula of shape $FG \phi$
- Eventual occurrence – A formula of shape $F \phi$

For these categories, no specific algorithm has been implemented yet, so the problems are all deferred to the general LTL model checker.

If a given formula does not fall into any of the listed categories, LoLA cannot verify it and shall exit without producing a result.

5.2 Bound properties

You can use LoLA for computing the maximum value that an expression can get. The expression is a formal sum of place names. Its evaluation in a marking yields an integer value. The result is the maximum value of the expression w.r.t. all reachable markings.

5.3 Syntax for properties

5.3.1 Constituents of a CTL* formulae

A formula is built from the following elements:

- an atomic proposition consisting of an integer comparison ('=', '!=', '>', '>=', '<', '<=') of terms built over place names, integers, ω markings (' ω '), and addition ('+') and subtraction ('-') thereof,
- the Boolean constants 'TRUE' and 'FALSE',
- the keyword 'FIREABLE' followed by a transition name which is internally unfolded to an atomic proposition that is true iff the given transition is activated,
- the keyword 'DEADLOCK' for an atomic proposition that is true in a state that activates no transition,
- the keyword 'INITIAL' which is internally unfolded to an atomic proposition that is true only in the initial marking,
- the Boolean operators for negation ('NOT'), conjunction ('AND'), disjunction ('OR'), exclusive disjunction ('XOR'), implication ('->'), and equivalence ('<->'),
- the temporal operators for eventual occurrence ('EVENTUALLY' or 'F'), global occurrence ('GLOBALLY' or 'G'), next state ('NEXTSTATE' or 'X'), until ('UNTIL' or 'U'), and release ('RELEASE' or 'R'),
- the shortcuts for reachability ('REACHABLE' = 'EF'), invariance ('INVARIANT' = 'AG'), and impossibility ('IMPOSSIBLE' = 'AG NOT'),
- the universal ('ALLPATH' or 'A') and existential ('EXPATH' or 'E') path quantifiers,
- and parentheses to override associativities.

5.3.2 EBNF grammar for properties

```

list ::= 'FORMULA'? property (':' property)* ';'

property ::= bound | statepredicate

bound ::= 'MAX' '(' term ')

statepredicate ::= '(' statepredicate ')'
                | atomic_proposition
                | 'NOT' statepredicate
                | statepredicate boolean_operator statepredicate
                | path_quantifier statepredicate
                | unary_temporal_operator statepredicate
                | '(' statepredicate binary_temporal_operator statepredicate ')'

boolean_operator ::= 'AND' | 'OR' | 'XOR' | '->' | '<->'

unary_temporal_operator ::= 'ALWAYS' | 'EVENTUALLY' | 'NEXTSTATE'
                          | 'REACHABLE' | 'INVARIANT' | 'IMPOSSIBLE'

binary_temporal_operator ::= 'UNTIL' | 'RELEASE'

path_quantifier ::= 'ALLPATH' | 'EXPATH'

atomic_proposition ::= term term_comparison_operator term
                   | 'TRUE'
                   | 'FALSE'
                   | 'FIREABLE' '(' 'IDENTIFIER' ')
                   | 'INITIAL'
                   | 'DEADLOCK'

term_comparison_operator ::= '=' | '!=' | '>' | '>=' | '<' | '<='

term ::= '(' term ')'
      | 'IDENTIFIER'
      | 'NUMBER'
      | term '+' term
      | term '-' term
      | 'NUMBER' '*' term
      | 'oo'

```

5.3.3 Regular expressions for terminal symbols

NUMBER

Any nonempty sequence of digits, optionally preceded by a minus sign.
 "-"?[0-9]+

IDENTIFIER

Any nonempty sequence of characters, excluding ',', ';', ':', '(', ')', '{', '}', or whitespace (spaces, tabs, or newlines).
 [^,;:()\t\n\r\{\}\s]+

5.4 Compatibility

Formulae from earlier versions of LoLA (version 1.x) are in principle compatible to this grammar. However, depending on the mode of LoLA 1.x, a temporal operator needs to be added:

mode in LoLA 1.x	formula in LoLA 1.x	formula in LoLA 2.0	notes
BOUNDEDNET	—	—	1

¹ Not directly supported, see [boundedness], page 19.

BOUNDEDPLACE	ANALYSE PLACE p	$AG\ p < \infty$	2
DEADLOCK	—	EF DEADLOCK	
DEADTRANSITION	ANALYSE TRANSITION t	$AG\ NOT\ FIREABLE(t)$	
EVENTUALLYPROP	FORMULA ϕ	$F\ \phi$	
FAIRPROP	FORMULA ϕ	$GF\ \phi$	
FINDPATH	FORMULA ϕ	ϕ	3
FULL	—	—	4
HOME	—	not supported	
LIVEPROP	FORMULA ϕ	$AGEF\ \phi$	
MODELCHECKING	FORMULA ϕ	ϕ	
NONE	—	—	5
REACHABILITY	ANALYSE MARKING m	$EF\ \phi$	6
REVERSIBILITY	—	$AGEF\ INITIAL$	
STABLEPROP	FORMULA ϕ	$FG\ \phi$	
STATEPREDICATE	FORMULA ϕ	$EF\ \phi$	
STATESPACE	—	not supported	

5.5 Implicitly supported properties

Throughout this section, let a *state predicate* be a temporal logic formula that does not contain temporal operators. It describes properties that can be satisfied or violated by individual states, regardless of subsequent reachable states.

5.5.1 Reachability

If you want to check reachability of a state that satisfies a given state predicate P , you should apply the command line option ‘`--formula="EF P"`’. LoLA will recognize this formula as a reachability query and apply specialized techniques for its evaluation.

```
$ lola --formula="EF (ea.2 > 0 AND ea.3 > 0)" phils10.lola
lola: NET
lola:   reading net from phils10.lola
...
lola: TASK
lola:   E (F (((1 <= ea.2) AND (1 <= ea.3))))
lola:   checking reachability
...
lola: RESULT
lola:   result: no
lola:   produced by: state space
lola:   The predicate is unreachable.
...
```

5.5.2 Invariance

If you want to check whether a state predicate P is satisfied by *all* reachable states, you should apply the command line option ‘`--formula="AG P"`’. LoLA will transform this formula into a reachability query for $\neg P$ and apply specialized techniques for its evaluation.

² Together with option ‘`--search=cover`’.

³ Together with option ‘`--search=findpath`’, see [Memoryless search], page 29.

⁴ Together with option ‘`--check=full`’, see [Compute the state space], page 11.

⁵ Together with option ‘`--check=none`’, see [Check nothing], page 11.

⁶ The state predicate ϕ thereby needs to express marking m , e.g. ‘ $p1 = 1\ AND\ p2 = 0$ ’ for marking ‘ $p1:1, p2:0$ ’.

```

$ lola --formula="AG (ea.2 = 0 OR ea.3 = 0)" phils10.lola
...
lola: TASK
lola:   A (G (((ea.2 <= 0)) OR ((ea.3 <= 0))))
lola:   checking invariance
...
lola: RESULT
lola:   result: yes
lola:   produced by: state space
lola:   The predicate is invariant.
...

```

5.5.3 Deadlocks

LoLA checks for the existence of deadlock markings. A *deadlock marking* is a marking where no transition is enabled. LoLA aborts state space exploration as soon as it detects the first such state. If other options permit, the found state, and a sequence of transitions leading from the initial marking of the found state can be reported.

```

$ lola --formula="EF DEADLOCK" phils10.lola
...
lola: TASK
lola:   E (F (DEADLOCK))
lola:   checking reachability of deadlocks
...
lola: RESULT
lola:   result: yes
lola:   produced by: state space
lola:   The net has deadlock(s).
...

```

5.5.4 k -boundedness of a place

If you want to check whether a certain place p is k -bounded, for a given bound k , you should apply the command line option ‘--formula="AG $p \leq k$ "’. LoLA will transform this formula into a reachability query for $p > k$ and apply specialized techniques for its evaluation.

```

$ lola --formula="AG hl.3 <= 1" phils10.lola
...
lola: TASK
lola:   A (G ((hl.3 <= 1)))
lola:   checking invariance
...
lola: RESULT
lola:   result: yes
lola:   produced by: state space
lola:   The predicate is invariant.
...

```

5.5.5 k -boundedness of the whole net

For checking whether the whole net is k -bounded, you can repeat the k -boundedness check for each place individually. State spaces that preserve k -boundedness of a single place are orders of magnitude smaller than a state space that preserves that property for all places at once. Consequently, chances are that the proposed approach replaces a single state space that does not fit into memory by many state spaces that do fit into memory. Since it is quite easy to run LoLA in shell scripts, the repeated application of LoLA is quite well manageable.

5.5.6 Boundedness of a place

If you want to check whether a certain place p is bounded (i.e., there exists a k such that p is k -bounded), you should apply the command line options ‘`--encoder=full`’ and ‘`--formula="AG $p < \infty$ "`’. Thereby, ‘ ∞ ’ stands for ω , representing a boundary for arbitrary tokens on the place. With the ‘`--search=cover`’ parameter, the *coverability graph* is built which is guaranteed to be finite even for unbounded nets. Finally, ‘`--encoder=full`’ chooses a special marking encoder which is required in combination with the coverability graph, see [Encoding], page 32.

```
$ lola --search=cover --encoder=full --formula="AG hl.3 < oo" phils10.lola
...
lola: TASK
lola:   A (G ((hl.3 < oo)))
lola:   checking invariance
...
lola: RESULT
lola:   result: yes
lola:   produced by: state space
lola:   The predicate is invariant.
...
```

5.5.7 Boundedness of the whole net

For checking whether the whole net is bounded, you can repeat the boundedness check for each place individually. Coverability graphs that preserve boundedness of a single place are orders of magnitude smaller than a coverability graph that preserves that property for all places at once. Consequently, chances are that the proposed approach replaces a single coverability graph that does not fit into memory by many coverability graphs that do fit into memory. Since it is quite easy to run LoLA in shell scripts, the repeated application of LoLA is quite well manageable.

5.5.8 Dead transition

A transition is *dead* in a given marking m if it is not enabled in any marking reachable from m . If you want to check whether a certain transition t is dead in the initial marking, you should apply the command line option ‘`--formula="AG NOT FIREABLE(t)"`’. LoLA will transform this formula into a reachability query for a state predicate that expresses the negated enabling condition of t and apply specialized techniques for its evaluation.

```
$ lola --search=cover --encoder=full --formula="AG NOT FIREABLE(t1.[y=3])" phils10.lola
...
lola: TASK
lola:   A (G (NOT FIREABLE(t1.[y=3])))
lola:   checking invariance
...
lola: RESULT
lola:   result: no
lola:   produced by: state space
lola:   The predicate is not invariant.
lola:   1 markings, 0 edges
...
```

5.5.9 Quasi-liveness

A net is *quasi-live* if it does not have any dead transition. You can check quasi-liveness by applying the dead transition check for each transition individually. State spaces that preserve single dead transitions are orders of magnitude smaller than state spaces that preserve quasi-liveness. Consequently, chances are that the proposed approach replaces a single state space

that does not fit into memory by many state spaces that do fit into memory. Since it is quite easy to run LoLA in shell scripts, the repeated application of LoLA is quite well manageable.

5.5.10 Liveness of a transition

A transition is *live* if it not dead in any reachable marking. If you want to check whether a certain transition t is live, you should apply the command line option ‘`--formula="AGEF FIREABLE(t)"`’.

```
$ lola --formula="AGEF FIREABLE(t1.[y=3])" phils10.lola
...
lola: TASK
lola:   A (G (E (F (FIREABLE(t1.[y=3])))))
lola:   checking liveness
...
lola: RESULT
lola:   result: yes
lola:   produced by: tscs_search
lola:   The predicate is live.
...
```

5.5.11 Liveness of the net

A net is *live* if all its transitions are live. You can check liveness by applying the live transition check for each transition individually. State spaces that preserve single live transitions are orders of magnitude smaller than state spaces that preserve liveness of the whole net. Consequently, chances are that the proposed approach replaces a single state space that does not fit into memory by many state spaces that do fit into memory. Since it is quite easy to run LoLA in shell scripts, the repeated application of LoLA is quite well manageable.

5.5.12 Liveness of a state predicate

A state predicate is *live* if it is reachable from all reachable markings. If you want to check whether a certain predicate P is live, you should apply the command line option ‘`--formula="AGEF P "`’. In the current release, LoLA evaluates this query by a CTL model checking algorithm. However, upcoming releases will transform this formula into a specialized query, with dedicated state space reduction techniques.

```
$ lola --formula="AGEF hl.1 > 0" phils10.lola
...
lola: TASK
lola:   A (G (E (F ((1 <= hl.1)))))
lola:   A (G (E (F ((1 <= hl.1)))))
lola:   checking liveness
...
lola: RESULT
lola:   result: yes
lola:   produced by: tscs_search
lola:   The predicate is live.
...
```

5.5.13 Reversibility

A net is *reversible* if the initial marking is reachable from every reachable state. If you want to check reversibility for a net, you should apply the command line option ‘`--formula="AGEF INITIAL"`’. In the current release, LoLA evaluates this query by a CTL model checking algorithm. However, upcoming releases will transform this formula into a specialized query, with dedicated state space reduction techniques.

```

$ lola --formula="AGEF INITIAL" phils10.lola
...
lola: TASK
lola:   A (G (E (F (((fo.10 <= 1) AND (1 <= fo.10) AND (ea.1 <= 0) AND ...
lola:   checking liveness
...
lola: RESULT
lola:   result: no
lola:   produced by: tscc_search
lola:   The predicate is not live.
...

```

5.5.14 Causal precedence of a transition

Transition t *causally precedes* a state predicate P if every path from the initial marking to a marking satisfying P contains an occurrence of t . If you want to check causal precedence, you can either directly represent it as a terribly complicated CTL formula, or you can add a new place q with a single initial token to your net and make it a pre-place of t . Then check reachability of $P \text{ AND } (q = 1)$. t causally precedes P if and only if that reachability check returns ‘no’. The latter approach has the advantage that LoLA can apply its reachability checks and is not forced into general model checking algorithms.

5.5.15 Relaxed Soundness of a workflow net

A *workflow net* has a distinguished source place and a distinguished sink place. Initially, only the source place is marked. It is desired that finally only the sink place is marked.

A workflow net is *relaxed sound* if, for each transition t , there is a path from the initial to the final marking that contains t . In LoLA, relaxed soundness can be checked by verifying, for each transition t individually, causal precedence of t with respect to a state predicate that describes the final marking: $p_1 = 0 \text{ AND } \dots \text{ AND } p_n = 0 \text{ AND sink} = 1$.

5.5.16 Soundness of a workflow net

A workflow net is *sound* if the final marking is reachable from all reachable markings, and the net has no dead transitions. This amounts to liveness of a state predicate that describes the final marking, e.g. $p_1 = 0 \text{ AND } \dots \text{ AND } p_n = 0 \text{ AND sink} = 1$, and checking for dead transitions, as described above. Again, we recommend to split the soundness check into many individual runs of LoLA.

5.6 Unsupported properties

Home states. A marking is a *home state* if it is reachable from every reachable marking. For a given marking, this can be checked as a liveness of a state predicate where the predicate describes the candidate marking. However, the question whether the net has home markings cannot be answered with this release.

5.7 General recommendations

Divide and conquer. If you can separate a global query into many local queries (as for liveness of a net versus liveness of all transitions), you should do so. The global property will always have a much higher probability to run out of memory than the worst local verification problem.

Stay simple. If you can express your problem as a reachability or deadlock problem, you should do so. Consider the possibility that this reduction can be achieved through a modification of the net itself, as demonstrated above for causal precedence. The particular strength of LoLA is on reachability checking. Here, technology is most advanced, and the number of available methods is larger than for any other class of properties.

Approximate. If your verification problem turns out to be too hard for LoLA, try verifying another property that implies the desired property or its negation.

Abstract. You can also try to abstract your input system. Did you model details that are irrelevant for the property under investigation? Can you coarsen sequences of activities into a single activity?

Do not break symmetry. If your model consists of replicated blocks, make sure that you do not break its symmetry, for instance by sequential initialisation of the components. Make sure that every detail of your model preserves the symmetry.

Keep concurrency. Try to avoid side conditions in your model. A place is a side condition for a transition if there are arcs in both directions between these nodes. If you have plenty of side conditions, you probably think of a Petri nets in terms of an ordinary program. Let your transitions pass the permission that is represented by your side condition through the net. This way, you add concurrency and the reduction techniques in LoLA work better.

6 Search strategies

LoLA offers several search strategies for exploring the state space. Several features depend on this choice: run time, whether or not you can get a witness path, its length, and whether or not state space exploration is exhaustive.

You select the search strategy using the command line option `--search=SEARCH` where `SEARCH` is one of the values `depth`, `sweep`, `findpath`, or `cover`. If no value is given, `depth` is used as default search strategy.

6.1 Available values

6.1.1 Depth first search (`--search=depth`, default)

LoLA explores the state space in depth first order. This is the fastest among the exhaustive strategies. Witness paths can be produced but are not necessarily the shortest possible witnesses.

6.1.2 Sweep-line method (`--search=sweep`)

The sweep-line method depends on a progress measure assigned to states and explores states in ascending progress order. States with progress value smaller than the current explored ones are removed from memory. Thus, the sweep-line method requires less space than depth-first or breadth-first search. If states are reached that have potentially been removed before, they are stored permanently, and their successors are explored (in subsequent “rounds”). Hence, the state space is explored exhaustively.

LoLA computes the required progress measure automatically. Self-defined measures are not supported. The sweep-line method is only available if the given property can be reduced to a simple reachability problem. Otherwise, LoLA applies depth-first search anyway. Using the sweep-line method, no witness path can be produced. Since some markings may be visited after prior removal, the numbers of visited markings and fired transitions is generally larger than with depth-first search (but peak memory usage can be smaller).

- Karsten Wolf. **Automated Generation of a Progress Measure for the Sweep-Line Method.** *STTT*, 8(3):195-203, June 2006.

```

$ lola --formula="EF DEADLOCK" --search=sweep phils10.lola
lola: NET
lola:   reading net from phils10.lola
...
lola:   preprocessing...
...
lola:   calculating the progress measure
lola: TASK
lola:   E (F (DEADLOCK))
lola:   checking reachability of deadlocks
lola:   Planning: workflow for deadlock check: search (--findpath=off,--siphontrap=off)
lola: SEARCH (state space)
lola:   state space: using sweep-line method (--search=sweep)
lola:   state space: using deadlock preserving stubborn set method with insertion algorithm(--stubborn=t
lola: RUNNING
lola:   transition progress range [-3,1], transients in [-3,1]
lola:   using 200 bytes per marking, including 0 wasted bytes
lola:   using 4 bytes per marking, with 2 unused bits
lola:   82 persistent markings, 27 transient markings (max)
lola: RESULT
lola:   result: yes
lola:   produced by: state space
lola:   The net has deadlock(s).
lola:   109 markings, 410 edges
...

```

6.1.3 Coverability graph ('--search=cover')

LoLA computes the coverability graph instead of a reachability graph. The coverability graph stores limits of sequences of markings rather than individual markings and is always finite (while the state space of a Petri net may be infinite. Verification results obtained from a coverability graph may be imprecise, that is, you may get *unknown* as a verification result. If LoLA returns *yes* or *no*, validity of that result is asserted by coverability graph theory, though. LoLA explores a coverability graph in breadth first order (which has superior performance here). Witness states derived from coverability graphs may assign omega (printed as 'oo' to some places meaning that arbitrarily large numbers of tokens may be put on them. If omega appears in a witness state, the corresponding witness path contains subsequences that are marked as *repeating*. Executing such parts increasingly often, you can replay a sequence of markings that produces increasing number of tokens on the places marking with omega.

6.2 Setting resource limits

A state space may easily grow larger than the available physical memory. Once starting to use swap space, LoLA will get incredibly slow and you should abort it. If you do not want to observe and kill LoLA manually, you can set two limits that cause LoLA to quit before having explored the whole state space. With '--timelimit=SECONDS', you can force LoLA to quit after 'SECONDS' seconds of work. With '--markinglimit=MARKINGS' you can force LoLA to quit after having explored 'MARKINGS' markings. Both values are integer. The default is that LoLA runs forever and produces arbitrarily many markings.

6.3 Best practice

Normally, depth first search is the option of your choice. If depth first search runs out of memory, and your property can be transformed into a simple reachability query, you may try the sweep-line method next. Memoryless search is useful if all other methods fail, or can be run in parallel on a second machine. If you need to manually inspect a witness path and depth-first search produces a prohibitively long one, you may run the same query in breadth-first mode for trying to get a shorter path.

You should be aware that you have other options for reducing the memory consumption of state space exploration. You may try other state space reduction techniques, and you can switch to a different storage model (see the appropriate chapters of this manual).

7 Reduction techniques

LoLA is the tool that offers the broadest variety of state space reduction techniques in the realm of explicit state exploration. Each individual technique has unique features. In addition, the opportunities to combine different reduction techniques are unprecedented. Most reduction techniques unfold their full power when applied to simple properties, especially to reachability and deadlock checking.

7.1 Partial order reduction: the stubborn set method

7.1.1 The method

The stubborn set method is one of the independently developed methods that are collectively called partial order reduction. At each marking, a subset of transitions is computed (called a stubborn set) and only enabled transitions from the stubborn set are explored. Stubborn set calculation follows rules that assure that desired properties are preserved in the reduced state space. We believe that partial order reduction, including the stubborn set method, is the most powerful state space reduction technique and has a decisive impact on the alleviation of state explosion. On the other hand, its application does not cause any remarkable disadvantages. Even the run-time penalty for computing stubborn sets in every marking tends to be moderate. Moreover, we experienced that some search strategies (`--search=findpath` and `--search=sweep`) as well as some other state space reduction techniques (e.g. `--cycle`) unfold their power only if applied in combination with the partial order reduction. For these reasons, the stubborn set method is *always* applied if actual properties are explored (option `--check=modelchecking`) while it is switched off in full state space generation (`--check=full`) and irrelevant for `--check=none`.

7.1.2 Unique features in LoLA

In the current release of LoLA, stubborn sets are applied to deadlock checking and for checking properties that can be transformed into reachability problems. For deadlock checking, we use the established method by Valmari, 1989. For reachability, we use a unique dedicated technique. Unlike methods that preserve whole temporal logics, our approach does not require presence of so-called invisible transitions (transitions that do not alter elementary propositions of the given property). That is, we can get substantial reduction where a general LTL or CTL model checker would not.

- Karsten Schmidt. **Stubborn Sets for Standard Properties.** *In Applications and Theory of Petri Nets 1999: 20th International Conference, ICATPN'99, Williamsburg, Virginia, USA*, June 1999. Proceedings, volume 1639 of Lecture Notes in Computer Science, pages 46-65, June 1999. Springer-Verlag.
- Lars Michael Kristensen, Karsten Schmidt, and Antti Valmari. **Question-Guided Stubborn Set Methods for State Properties.** *Formal Methods in System Design*, 29(3):215-251, November 2006.

7.1.3 Options

Generally, there are several stubborn sets that meet the requirements for property preservation in a given marking. If the enabled transition in one stubborn set is a subset of another, the smaller one is always to be preferred as it will result in better reduction. If it contains less enabled transitions but they are not included in the other stubborn set, either one may cause better reduction. Two styles of computing stubborn sets are available. The first one (activated with `--stubborn=tarjan`) investigates a dependency graph between transitions. It has linear time complexity and it does not necessarily result in inclusion-minimal stubborn sets. This is the default if no `--stubborn` option is present at the command line. The second option

(activated with ‘`--stubborn=deletion`’) applies a technique where iteratively transitions are removed from the set of all transition until it is not possible to remove any further transition without violating the requirements for property preservation. This technique has quadratic run time complexity but results in inclusion-minimal stubborn sets.

7.1.4 Best practice

If you expect that a deadlock or a witness state is present in your state space, you should prefer ‘`--stubborn=tarjan`’. It explores more states per time unit, tends to attract exploration quickly to a satisfying state in reachability checks, and tends to produce witness paths of reasonable length. If you expect that no deadlock or no specified state is reachable, you should prefer ‘`--stubborn=deletion`’. In this case, ultimately the whole state space is explored and the slower exploration rate is more than compensated by the significantly smaller number of states. If you have two machines available, it makes sense to run LoLA on both. Here, the deletion version should be run on the machine with more available memory: If the ‘`tarjan`’ version does not find a state quickly, the ‘`deletion`’ version is more likely to do the job.

7.2 The symmetry method

7.2.1 The method

Symmetric structure implies symmetric behavior. Thus, symmetry detected on the net structure can be used to fold the state space: if two markings are identical up to symmetry, only one of them needs to be explored.

7.2.2 Unique features in LoLA

LoLA investigates symmetries by inspecting the graph automorphisms present in the Petri net itself and the investigated formula (if applicable). The result is a generating set. Through graph automorphisms, any shape of symmetrical structure in a Petri net can be detected (while many other tools are restricted to distinguished symmetry patterns such as “scalar sets”). During state space exploration, each encountered marking is transformed into a lexicographically smaller but symmetric one. Symmetry is detected through identity between the transformed images of the original marking. LoLA does not dare to transform a marking into *the* lexicographically smallest symmetric one as there are no polynomial solutions known for that task which needs to be executed at each encountered marking. Thus, the reduced state space is not necessarily the smallest one that could be theoretically obtained by the symmetry method. On the other hand, the applied method guarantees a moderate run-time penalty during state space exploration. Unfortunately, the actual investigation of graph automorphisms during pre-processing may be time-consuming, so the method can be switched off.

- Karsten Schmidt. **How to Calculate Symmetries of Petri Nets.** *Acta Inf.*, 36(7):545-590, 2000.
- Karsten Schmidt. **Integrating Low Level Symmetries into Reachability Analysis.** In *Susanne Graf and Michael I. Schwartzbach, editors, Tools and Algorithms for the Construction and Analysis of Systems: 6th International Conference, TACAS 2000, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2000, Berlin, Germany, March/April 2000. Proceedings, volume 1785 of Lecture Notes in Computer Science, pages 315-330, March 2000. Springer-Verlag.*

7.2.3 Options

With ‘`--symmtimelimit=SECONDS`’, you can set a limit for the time LoLA spends on symmetry calculation. If the time limit is exceeded, LoLA will leave symmetry calculation and use the symmetries encountered so far for state space reduction. This does not jeopardize correctness,

but may lead to a significantly larger state space. Alternatively, pressing **CTRL+c** during symmetry calculation has the same effect. LoLA will not necessarily leave symmetry calculation promptly, as it needs to leave in a well-defined state.

7.2.4 Best Practice

Computing symmetries may take time but is usually worth it. In rare cases, the generating set can exceed available memory. Then, setting a time limit may be useful. If you know that your Petri net has no symmetry, you do not want to use the symmetry method since LoLA may need some time to find out. If you believe that your system is symmetric but LoLA does not find symmetries, you may want to check your model or its translation to the LoLA format. It is quite easy to break symmetry, for instance through sequential initialization of otherwise concurrent processes.

7.3 Cycle reduction

7.3.1 The method

When this technique is switched on, LoLA investigates linear dependencies between the transition firing vectors and obtains information about transition sequences that form cycles in the state space. This is done during pre-processing. During actual state space exploration, only a subset of encountered markings is actually stored while other markings, if re-visited, are explored again. The pre-processed information is used for taking care that the markings that are not stored do not form cycles. Thus, termination is guaranteed.

7.3.2 Unique features in LoLA

The method as such is unique. It is an example for the benefits that Petri net structure theory provides in state space verification.

- Karsten Schmidt. **Using Petri Net Invariants in State Space Construction.** In *Hubert Garavel and John Hatcliff, editors, Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2003), 9th International Conference, Part of ETAPS 2003, Warsaw, Poland*, volume 2619 of Lecture Notes in Computer Science, pages 473-488, April 2003. Springer-Verlag.

7.3.3 Options

The method is activated by the ‘`--cycle`’ command line option. It is used only if the explored property is deadlock checking or a simple reachability query. The method may cause a prohibitive increase of run time. For alleviating that problem, a heuristic parameter has been introduced that can be set by ‘`--cycleheuristic=K`’. Small values produce less reduction but better run-time while large values cause better reduction but longer run-time. Correctness of the is independent of this value.

8 Nonstandard Verification Techniques

8.1 Memoryless search ('--findpath=ORDER')

Depending on `ORDER`, this technique is applied in parallel to ('--findpath=par'), before ('--findpath=seq'), or instead of ('--findpath=alone') state space exploration. By default, it is not applied at all ('--findpath=off').

LoLA explores the state space without recording visited states at all. It stops if, by chance, a state witnessing the given property is encountered. Witness paths produced by this method may contain cycles and are not necessarily the shortest possible witness paths. The search is not exhaustive.

If a certain depth is exceeded, search resets to the initial state and explores another path (transitions are randomly selected). The depth at which LoLA resets to the initial state can be controlled using the '--depthlimit=DEPTH' command line option where 'DEPTH' is any integer greater than 1. The default depth limit is one million.

With another command line option, '--retrylimit=RETRIES', you can force LoLA to terminate after 'RETRIES' resets to the initial marking. If 'RETRIES' is set to 0, LoLA goes on forever, exploring an unlimited number of paths each having length 'DEPTH' (unless the desired state is found). If search is successful, LoLA terminates with the respective result. Parallel or subsequent search is cancelled. If the number of retries is reached without having found a state, subsequent or parallel search continue exploring the property. In the case, that the method is applied alone ('--findpath=alone'), the value "unknown" is returned as result. If a parallel search finishes, memoryless search is cancelled.

This method is available only if the given property can be transformed into a simple reachability query. Otherwise, LoLA performs depth-first search anyway.

- Karsten Schmidt. **LoLA wird Pfadfinder**. In *6. Workshop Algorithmen und Werkzeuge für Petrinetze (AWPN'99)*, Frankfurt, Germany, October 11.-12., 1999, volume 26 of CEUR Workshop Proceedings, pages 48-53, October 1999. CEUR-WS.org.

```
$ lola --findpath=alone --formula="EF FIREABLE(t520)" vasy.lola
...
lola: TASK
lola:   E (F (FIREABLE(t520)))
lola:   checking reachability
lola:   Planning: workflow for reachability check: findpath (--findpath=alone)
lola: SEARCH (findpath)
lola:   findpath: using reachability preserving stubborn set method with insertion algorithm (--stubborn)
lola: RUNNING
lola:   findpath: starting randomized, memory-less exploration (--findpath)
lola:   findpath: searching for paths with maximal depth 1000000 (--depthlimit)
lola:   findpath: no retry limit given (--retrylimit)
lola:   findpath: transitions are chosen hash-driven
lola: RESULT
lola:   result: yes
lola:   produced by: findpath
lola:   The predicate is reachable.
...
```

8.2 The state equation ('--stateequation=ORDER')

Depending on `ORDER`, this technique is applied in parallel to ('--stateequation=par'), before ('--stateequation=seq'), or instead of ('--stateequation=alone') state space exploration. By default, it is not applied at all ('--stateequation=off').

LoLA invokes the tool Sara that verifies reachability based on the Petri net state equation. The state equation is a linear system of equations that establishes a necessary condition for reachability. If the state equation has no solution, Sara returns false (and so does LoLA). If the state equation has a solution (which happens to be the transition count vector of a firing sequence), Sara checks whether an executable firing sequence fitting to the given count vector exists. If so, it returns true. Otherwise it checks for reasonable modifications to the state equation that yield alternative solutions. If no reasonable modifications exist, it returns false.

If Sara returns, LoLA terminates with the respective result. Parallel or subsequent search is cancelled. If parallel search finishes, Sara is cancelled.

This method is available only if the given property can be transformed into a simple reachability query not containing any DEADLOCK atomic proposition. Otherwise, LoLA performs depth-first search anyway.

- Harro Wimmel and Karsten Wolf. **Applying CEGAR to the Petri Net State Equation.** *Logical Methods in Computer Science* 8(3) (2012)

```
$ lola --stateequation=par --formula="EF hr.1 > 1" phils10.lola
...
lola: TASK
lola:   E (F ((2 <= hr.1)))
lola:   checking reachability
lola:   Planning: workflow for reachability check: stateequation||search (--findpath=off)
...
lola: RUNNING
lola:   state equation: Generated DNF with 1 literals and 1 conjunctive subformulas
lola:   state equation: write sara problem file to stateEquationProblem.sara
lola:   state equation: calling and running sara
sara: try reading problem file stateEquationProblem.sara.
sara: place or transition ordering is non-deterministic

lola:   state equation: solution impossible
lola: RESULT
lola:   result: no
lola:   produced by: state equation
lola:   The predicate is unreachable.
...
```

8.3 The siphon/trap property ('--siphontrap=ORDER')

Depending on ORDER, this technique is applied in parallel to ('--siphontrap=par'), before ('--siphontrap=seq'), or instead of ('--siphontrap=alone') state space exploration. By default, it is not applied at all ('--siphontrap=off').

LoLA checks whether there exists a siphon in the given Petri net that does not contain a sufficiently marked trap. If such a siphon does not exist, and the net is homogenous (all arc multiplicities are the same), then the net is known to be deadlock-free. The check is executed by translating the problem into a Boolean formula and running an integrated SAT checker (MINISAT) on it. You can pass command line argument to MINISAT by choosing '--minisatargs=STRING'. For valid arguments, refer to the Minisat documentation.

If Minisat established deadlock freedom, LoLA terminates with the respective result. Parallel or subsequent search is cancelled. Otherwise, parallel or subsequent search continues. In the case, that the method is applied alone ('--stateequation=alone'), the value "unknown" is returned as result. If normal search yields a result, Minisat is cancelled.

If Minisat finds a siphon without an initially marked trap, such a siphon and its maximal included trap can be shown by LoLA using the parameter '--siphonwitness[=FILE]'.

Sometimes, the formula to be passed to Minisat gets so large that Minisat cannot handle it. For such situations, there is an option ‘`--siphondepth=INT`’ in LoLA. If this option is given, LoLA produces a shorter formula. For this formula, the Minisat check may result in one of the following values:

- All siphons contain an initially marked trap;
- There is a siphon without initially marked trap contained;
- The formula is too short to get any of the above answers.

In the third case, you may try a larger number for ‘`--siphondepth=INT`’. The larger the value gets, the larger the generated formula becomes, and the more likely it is to get one of the first two answers in the above list.

In any case, LoLA is fully aware of what is preserved by the formula and will never produce a wrong result.

This method is available only if the given property can be transformed into a deadlock (EF DEADLOCK) or AG NOT DEADLOCK query. Otherwise, LoLA performs depth-first search anyway.

- Olivia Oanea, Harro Wimmel, and Karsten Wolf. **New Algorithms for Deciding the Siphon-Trap Property.** *Petri Nets* 2010: 267-286

```
$ lola --siphontrap=seq --formula="EF DEADLOCK" phils10.lola
...
lola: TASK
lola:   E (F (DEADLOCK))
lola:   checking reachability of deadlocks
lola:   Planning: workflow for deadlock check: siphon;search (--findpath=off,--siphontrap=seq)
...
lola: SEARCH (state space)
lola:   state space: using reachability graph (--search=depth)
lola:   state space: using deadlock preserving stubborn set method with insertion algorithm(--stubborn=t
lola: RUNNING
lola:   STP: formula with 4672 variables and 13424 clauses shipped to Minisat
lola:   stp: The siphon/trap property does not hold
lola: RESULT
lola:   result: yes
lola:   produced by: state space
lola:   The net has deadlock(s).
lola:   13 markings, 13 edges
...
```

9 Storage concepts

For managing visited markings, LoLA performs two tasks. First, it transforms a marking into a sequence of bits. Second, it handles a data structure where such sequences can be searched (for finding out whether that marking had been seen before) and inserted (if they have not been seen). In LoLA, the two tasks are separated. We call the first task *encoding* and the second one *storing*. For both task, LoLA offers several solutions that can be selected independently.

9.1 Encoding

With encoding, you can control the length of a bit vector that is stored later on as a representation of a marking in the main memory. If your focus is on getting to the memory limits, you want to choose strong compression with some runtime penalty. If your focus is on runtime while you believe to have sufficient memory resources, you may choose a weaker compression.

You select the encoding with the ‘`--encoder=ENCODER`’ option on the command line. ‘*ENCODER*’ can be any of the following values: ‘`bit`’, ‘`copy`’, ‘`simplecompressed`’, or ‘`fullcopy`’. The default value is ‘`--encoder=bit`’. LoLA uses three sources for compression: *capacities*, *variable length encoding*, and *place invariants*.

9.1.1 Compression by capacities

When specifying a Petri net, you can use the `SAFE i` statement in the place declaration list. Using that statement, you promise that no reachable marking will ever have more than *i* tokens on the concerned places. LoLA will compute the number of bits that are necessary to represent all values between 0 and *i* and compress a marking into a bit sequence where every place gets exactly the computed number of bits.

The coding is dense, i.e. the number is *not* rounded up to full bytes for all individual places. Only at the end of the complete bit vector, a few bits may be wasted. If a place is declared without a ‘`SAFE i`’ statement, 32 bits are reserved for that place. If compression by capacity is not used, the marking of each place is represented by 32 bits.

Compression by capacity is useful if your model has a significant number of places where small capacities are known by construction. Many translations from other formalisms into Petri nets result in 1-safe Petri nets. If that fact is disclosed to LoLA using the ‘`SAFE 1`’ statement in the net input file, only one bit per place is stored in every reachable marking.

```
$ cat phils10.lola
PLACE SAFE 1:
ea.1, ea.2, ea.3, ea.4, ea.5, ea.6, ea.7, ea.8, ea.9, hl.1, fo.1, hl.2, fo.2,
...

$ lola --encoder=bit phils10.lola --check=none
...
lola: using a bit-perfect encoder (--encoder)
lola: using 4 bytes per marking, with 2 unused bits
...

$ lola --encoder=copy phils10.lola --check=none
...
lola: using a copy encoder (--encoder)
lola: using 120 bytes per marking, including 0 wasted bytes
...
```

9.1.2 Variable length encoding

This encoding scheme supports nets where no bounds for the places are known in advance. It uses the assumption that even in unbounded nets or nets with large bounds, small values occur

more frequently than large values. Consequently, short bit sequences are related to small values and larger sequences to large values. The compression is not as strong as in the case of tight known bounds but better than shipping uncompressed bit sequences. Since the length of a bit sequence varies, LoLA does not report any bit vector length.

9.1.3 Compression by place invariants

A *place invariant* is a mapping that assigns a *weight* to each place such that all reachable markings get the same weighted token sum. Invariants correspond to solutions x of the linear System of equations $Cx = 0$ where C is the incidence matrix of the Petri nets. Having such a *place invariant*, the number of tokens on one place p with nonzero weight is fully determined by the number of tokens on the remaining places and the constant overall weight (which can be calculated for the initial marking). In other words, if two markings do not differ on any place except p , they are equal on p as well. We call p an *insignificant* place. Since a Petri net may have several place invariants, more than one place may be insignificant.

During preprocessing, LoLA analyzes the linear dependencies in the incidence matrix and comes up with a factoring of the place set into significant and insignificant places. When compression by place invariants is active, LoLA only ships significant places to the resulting bit vector. Typically, compression by place invariants reduces the length of the resulting bit vector to 30–70 percent of the original size. Since experience tells that preprocessing does not require much run time, its use is strongly recommended.

For efficiency reasons, LoLA only determines *whether* a place is insignificant but not how. That is, LoLA does not fully explore place invariants and does not store them permanently. For this reason, bit vectors in the store, although uniquely characterizing a marking, cannot be used to restore a full marking for later use. If such functionality is ever useful (in future releases), switching off compression by capacities may make sense. During preprocessing, LoLA reports the number of significant places. This way, the user can easily experience the compression ration through place invariants.

- Karsten Schmidt. **Using Petri Net Invariants in State Space Construction.** In *Hubert Garavel and John Hatcliff, editors, Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2003), 9th International Conference, Part of ETAPS 2003, Warsaw, Poland*, volume 2619 of Lecture Notes in Computer Science, pages 473-488, April 2003. Springer-Verlag.

```
$ lola phils10.lola --check=none
...
lola: finding significant places
lola: 50 places, 40 transitions, 30 significant places
...
```

9.1.4 Meaning of the values

The values for ‘*ENCODER*’ have the following meaning:

- ‘--encoder=bit’ (default value). Use compression by capacity and compression by place invariants. This is the best option for nets with known tight bounds for many places and exhaustive search.
- ‘--encoder=copy’ Use compression by place invariants only. This may be an option if memory is not at stake, or the Bloom store is used (see [Store], page 34).
- ‘--encoder=simplecompressed’ Use compression by place invariants and variable length encoding. This may be useful if no tight bounds for places are known.
- ‘--encoder=fullcopy’ Do not apply any compression. This value is strongly deprecated. We use it for comparing the effect of the other encodings.

9.2 Store

During state space exploration, LoLA maintains a store. A store contains information on whether or not a marking (i.e. an encoded bit sequence) had been visited before. LoLA supports several data structures for organizing its store. The decision for a particular store includes the decision on whether or not state space exploration is exhaustive, regardless of the applied search strategy.

You can select a particular data structure using the ‘`--store=STORE`’ option on the command line. Possible values for ‘`STORE`’ include ‘`prefix`’ (the default value), ‘`bloom`’, ‘`comp`’, and ‘`stl`’.

9.2.1 Prefix tree (‘`--store=prefix`’, default)

This data structure merges common prefixes of stored bit sequences. If a new bit sequence arrives from the encoder, LoLA identifies the largest prefix that this sequence has in common with any other stored sequence and stores only those bits that do not belong to that prefix. Systematic traversal of the sequence fragments is realized by some constant-size management overhead. Prefix trees are the strongest storage concept in LoLA for exhaustive state space exploration.

9.2.2 Bloom filter (‘`--store=bloom`’)

This data structure does not store states at all. It rather records hash values obtained from the encoded bit sequences. If, for a new bit sequence, the hash value has not been recorded before, it is treated as new (which is always sound). If the hash value has not been recorded, the sequence is treated as already visited (which may be wrong due to hash collision). Consequently, the Bloom filter store is inherently incomplete since, in case of a hash conflict, only one of the conflicting markings is explored.

Risk of hash conflicts can be reduced by operating on multiple hash tables, each using a hash function that is stochastically independent of the others. You can control the number of hash functions to be used by the ‘`--hashfunctions=INT`’ command line options. If that option is absent and Bloom filtering is used, LoLA operates on two hash functions.

During state space exploration, the probability of a false positive is printed:

```
$ lola phils1000.lola --check=full --store=bloom
...
lola:      95278 markings,      427819 edges,      19056 markings/sec,      0 secs
lola: 2147483648 hash table size      false positive probability: 0.0000000079
lola:      188148 markings,      849006 edges,      18574 markings/sec,      5 secs
lola: 2147483648 hash table size      false positive probability: 0.0000000307
...
```

Furthermore, the statistically optimal number of hash functions to minimize this probability is printed once the exploration is completed:

```
$ lola phils10.lola --check=full --store=bloom
...
lola: Bloom filter: probability of false positive is 0.0000000030
lola: Bloom filter: optimal number of hash functions is 15.2
...

$ lola phils10.lola --check=full --store=bloom --hashfunctions=15
...
lola: Bloom filter: probability of false positive is 0.0000000000
lola: Bloom filter: optimal number of hash functions is 15.2
...
```

Note that using the Bloom filter yields inconclusive results in case no witness state was found:


```

$ lola phils10.lola --formula="EF (ea.1 = 1 AND ea.2 = 1)" --store=prefix
...
lola: result: no
lola: The net does not satisfy the given formula.
lola: 28098 markings, 44878 edges

$ lola phils10.lola --formula="EF (ea.1 = 1 AND ea.2 = 1)" --store=bloom
...
lola: result: unknown
lola: The net may or may not satisfy the given formula.
lola: 28098 markings, 44878 edges

```

9.2.3 STL store('--store=stl')

Use the set data type of the C++ Standard Template Library (STL) for storing bit sequences. This value is strongly deprecated. We use it for teaching our students the principle functionality of a store, and for enjoying the superior performance of prefix trees and Bloom filters compared to this one.

9.2.4 Comparison store('--store=comp')

This is a debugging option only. It manages two other stores concurrently (e.g. a trusted implementation and a new data structure) and checks whether they agree on search queries.

```

$ lola phils14.lola --store=prefix --check=full
...
lola: using a prefix store (--store)
...
lola: 4782968 markings, 44641030 edges
...

$ lola phils14.lola --store=bloom --check=full
...
lola: using a specialized store (--store)
...
lola: 4782928 markings, 44640646 edges
...

$ lola phils14.lola --store=bloom --hashfunctions=1 --check=full
...
lola: 4763253 markings, 44454472 edges
...

```

9.3 Useful combinations of encoder and store

If you want to explore the state space exhaustively, you should use the prefix tree store in combination with a strongly compressing encoder. In case of bounds for places that are recorded in the net input file, the bit encoder is the best option. This particular combination does not require any command line specification as this is the default setting. In case of unknown bounds for places, you may want to try the simplecompressed encoder.

If you are satisfied with a potentially incomplete exploration (be it that you use LoLA for debugging only, be it that exhaustive search ran out of memory), you want to use the Bloom filter store. Encoding is not as important as for exhaustive search since markings are reduced to hash values anyway. However, shorter bit sequences require less complex computations for determining the hash value of a marking. For this purpose, you should use an encoder that at least uses the compression by place invariants since this compression comes at no cost (except pre-processing). In other words, any encoder other than the fullcopy encoder is fine in combination with the Bloom filter store.

If you select ‘`--search=findpath`’ as your search strategy ([Memoryless search], page 29), you do not need to worry about encoders nor stores at all since that search strategy does not keep any information about visited markings.

10 Output formats

LoLA will output runtime information on standard error. This output is purely informational and may change in future versions. If you want to process information, we advice using the JSON output, see [JSON], page 38.

You can silence all output using the ‘`--quiet`’ parameter.

```
$ lola --formula="EF DEADLOCK" --quiet phils10.lola
```

10.1 Markings

Depending on the property and the outcome of the analysis, LoLA can provide a marking that demonstrates why the property is (not) satisfied). The marking is printed with the ‘`--state=FILE`’ parameter. If no filename ‘*FILE*’ is given, the state is printed on standard output.

The output lists the names of all marked places, followed by a colon (‘:’) and the number of tokens on the place or ‘`oo`’ in case the place is unbounded. Note place names are not necessarily sorted alphabetically and unmarked places are not printed.

```
h1.1 : 1
h1.2 : 1
h1.3 : 1
h1.4 : 1
h1.5 : 1
h1.6 : 1
h1.7 : 1
h1.8 : 1
h1.9 : 1
h1.10 : 1
```

The marking can also be printed in JSON format, see [JSON], page 38.

10.2 Paths

Depending on the property and the outcome of the analysis, LoLA can provide a path from the initial marking to a marking that demonstrates why the property is (not) satisfied) (see [Markings], page 37). The path is printed with the ‘`--path=FILE`’ parameter. If no filename ‘*FILE*’ is given, the path is printed on standard output.

The output lists the transitions. In case of CTL, LTL, or boundedness properties, the path may contain cycles. The begin and end of the cycles is labeled with ‘`===begin of cycle===`’ and ‘`===end of cycle===`’, respectively.

```
t1.[y=1]
t1.[y=2]
t1.[y=3]
t1.[y=4]
t1.[y=5]
t1.[y=6]
t1.[y=7]
t1.[y=8]
t1.[y=9]
t1.[y=10]
```

The path can also be printed in JSON format, see [JSON], page 38.

Furthermore, the path can be printed in different shapes that can be chosen with the ‘--pathshape=SHAPE’ parameter. With ‘--pathshape=fullrun’, a distributed run is generated. With ‘--pathshape=run’, this distributed run is cropped to only those conditions that contribute to reaching the target marking. With ‘--pathshape=event’, an event structure is printed. The result is printed in Graphviz format (see <http://www.graphviz.org>).

10.3 JavaScript Object Notations (JSON)

With the command line parameter ‘--json’, LoLA outputs a JSON representation of its output to the standard output or a given filename (e.g., ‘--json=output.json’). The result is a JSON object whose entries are as follows:

`analysis` [object]

This object collects information on the performed analysis.

`analysis.type` [string]

The type of the analysis (the parameter given with ‘--check’).

`analysis.stats` [object]

This object collects information on the constructed state space.

`analysis.stats.edges` [number]

The number of fired transitions.

`analysis.stats.states` [number]

The calculated markings. Note that in case of some search strategies, not all of these markings are actually stored.

`analysis.result` [boolean]

The result of the analysis. In case no result was found (e.g. due to reaching of a resource limit), this entry is missing. In case the result is unknown (e.g. using a Bloom store where no witness was found), the result is ‘null’.

`analysis.formula` [object]

This object collects information on the formula in case the analysis type is ‘modelchecking’.

`analysis.formula.atomic_propositons` [number]

The number of atomic propositions occurring in the formula.

`analysis.formula.parsed` [string]

The formula as interpreted by the parser before processing. Note this string may be different from the original input, because the parser may add brackets and spaces.

`analysis.formula.parsed_size` [number]

The length of the formula before processing. This value is mostly valuable in case of very long formulae to make a prediction about the estimated duration of processing.

`analysis.formula.place_mentioned` [number]

The total number of places mentioned in the formula.

`analysis.formula.place_unique` [number]

The unique number of places mentioned in the formula. As most reduction techniques especially perform well when only a part of the net is affected, this number gives a good hint in how “local” the property is.

`analysis.formula.processed` [string]

The formula as interpreted by the parser after processing. Processing tries to remove redundancies and unfolds more complex properties (e.g. ‘FIREABLE’) or operators

(e.g. ‘<->’) to simpler ones. Note that the outmost temporal operator may be removed in case reachability or invariance formulae are checked – in this case, the formula only contains the state predicate to check. See [\[Supported Properties\]](#), page [\[undefined\]](#) for more information.

`analysis.formula.processed_size` [number]

The length of the formula after processing.

`analysis.formula.rule_applications` [number]

The number of rewrite rules applied in preprocessing.

`analysis.formula.type` [string]

The type of the formula. LoLA tries to find the most special type to choose the most efficient algorithm. Therefore, formulae like ‘EF *P*’ are not checked using CTL routines, but rather using more efficient reachability algorithms.

`call` [object]

This object collects information on the call and caller of LoLA.

`call.architecture` [number]

The datapath width of the architecture in bits. This number is useful to check whether the compiler version of LoLA can make use of more than 4 gigabytes of memory (in this case, 64 bit are required).

`call.assertions` [boolean]

Whether assertions are checked at runtime. This setting should only be used to debug LoLA. Make sure you configured LoLA with ‘./configure --enable-optimizations’. See Chapter 2 [Bootstrapping LoLA], page 6 for more information.

`call.build_system` [string]

An identifier of the build system used to compile LoLA, including the kernel name, kernel version, architecture, and vendor. Examples are ‘x86_64-apple-darwin13.2.0’ (OS X), ‘x86_64-unknown-linux-gnu’ (Linux), or ‘x86_64-unknown-cygwin’ (Windows running Cygwin).

`call.optimizations` [boolean]

Whether optimizations were used at compile time. This setting should only be switched off to debug LoLA. Make sure you configured LoLA with ‘./configure --enable-optimizations’. See Chapter 2 [Bootstrapping LoLA], page 6 for more information.

`call.package_version` [string]

The version number of LoLA. As of May 2017, the most recent version is **2.0**.

`call.parameters` [array]

An array containing the used command line parameters.

`call.signal` [string]

In case LoLA was aborted (e.g. CTRL+c), this string contains the signal that was used internally (e.g. ‘Interrupt: 2’).

`call.error` [string]

In case of an error, a textual description thereof, for instance ‘No such file or directory’.

`call.svn_version` [string]

If LoLA was compiled from the original source code repository, this entry contains the revision number of the compiled source code. Otherwise, the value is ‘Unversioned directory’.

`call.hostname` [string]

The hostname of the caller.

`files` [object]

This object collects information on opened files. For each file opened by LoLA, a subobject is created that uses the purpose of the file (e.g., `'net'`, `'formula'`) as key [string] and collects the filename (`'filename'` [string]) and the size in kilobytes (`'size'` [number]).

`limits` [object]

This object contains the set limits of the execution.

`limits.markings` [number]

The maximal number of markings to be constructed (set with the `'--markinglimit'` parameter). The value is `'null'` in case no limit is provided.

`limits.time` [number]

The maximal runtime of LoLA in seconds (set with the `'--timelimit'` parameter). The value is `'null'` in case no limit is provided.

`limits.symmetrytime` [number]

The maximal time LoLA may spend calculating symmetry in seconds (set with the `'--symmtimelimit'` parameter). The value is `'null'` in case no limit is provided. Note that LoLA does not immediately terminate once the limit is reached, but rather tries to calculate all possible symmetries from the already calculated generators.

`net` [object]

An object collecting information on the input net.

`net.conflict_sets` [number]

The number of conflict sets. A conflict set contains those transitions that are in conflict. These sets are explicitly stored to speed up the firing of transitions.

`net.filename` [string]

The filename of the input net.

`net.places` [number]

The number of place of the input net.

`net.places_significant` [number]

The number of significant places of the input net, see [Compression by place invariants], page 33.

`net.transitions` [number]

The number of transitions of the input net.

`path` [array]

An array of transition names expressing a witness/counterexample path for the given property (e.g., a path from the initial marking to a deadlock state). In case of CTL, LTL, or boundedness properties, the path may contain cycles. In this case, the path array contains sub-array expressing these cycles.

Note: To include this entry, use the `'--jsoninclude=path'` parameter.

`state` [object]

An object expressing a witness/counterexample marking for the given property. The marking is given as mapping from place names to integers.

Note: To include this entry, use the `'--jsoninclude=state'` parameter.

`store` [object]

An object collecting information on the used marking store, see Chapter 9 [Storage concepts], page 32.

`store.bucketing` [number]

The number of buckets if bucketing is used; can be set with ‘`--bucketing=BUCKETS`’.

`store.encoder` [string]

The used state encoder, see [Encoding], page 32.

`store.threads` [number]

The number of threads used; can be set with ‘`--threads=THREADS`’.

`store.search` [string]

The used search strategy, see Chapter 6 [Search strategies], page 23.

`store.type` [string]

The used store, see [Store], page 34.

`symmetries` [object]

An object collecting information on the calculated symmetries.

`symmetries.generators` [number]

The number of generators calculated.

`symmetries.dead_branches` [number]

The number of dead branches visited during calculation of generators.

`symmetries.represented` [number]

The number of symmetries represented by the calculated generators.

10.3.1 Example

```

{
  "analysis": {
    "formula": {
      "atomic_propositions": 2,
      "parsed": "AG (ea.1 != 1)",
      "parsed_size": 14,
      "places_mentioned": 2,
      "places_mentioned_unique": 1,
      "processed": "(ea.1 > 0 AND ea.1 <= 1)",
      "processed_size": 24,
      "rewrites": 7,
      "type": "invariance"
    },
    "result": false,
    "stats": {
      "edges": 2,
      "states": 3
    },
    "type": "modelchecking"
  },
  "call": {
    "architecture": 64,
    "assertions": false,
    "build_system": "x86_64-apple-darwin14.0.0",
    "error": null,
    "hostname": "stewie-2.local",
    "optimizations": true,
    "package_version": "2.0",
    "parameters": [
      "--formula=AG ea.1 != 1",
      "--json=../output13.tmp",
      "--jsoninclude=path",
      "phils10.lola"
    ],
    "signal": null,
    "svn_version": "9643:9646M"
  },
  "files": {
    "net": {
      "filename": "phils10.lola",
      "size": 3
    }
  },
  "limits": {
    "markings": null,
    "time": null
  },
  "net": {
    "conflict_sets": 60,
    "filename": "phils10.lola",
    "places": 50,
    "places_significant": 30,
    "transitions": 40
  },
  "path": [
    "t1.[y=1]",
    "tr.[y=1]"
  ],
  "store": {
    "bucketing": 16,
    "encoder": "bit-perfect",
    "search": "depth_first_search",
    "threads": 1,
    "type": "prefix"
  }
}

```


11 Error messages

In case LoLA encounters a problem, an error message is displayed together with an error code (#01–#04), and LoLA exits with exit code 1.

A complete example looks like this:

```
$ lola --check=none net.lola
lola: reading net from net.lola
lola: place 'p5' does not exist
lola: net.lola:7:9 - error near 'p5'

6 CONSUME p1, p2;
7 PRODUCE p5;
  ^^

lola: syntax error -- aborting [#01]
lola: see manual for a documentation of this error
```

11.1 Syntax errors [#01]

syntax error, unexpected x, expecting y

This error occurs if the input does not match the grammars described in [\[File formats\]](#), page [\[undefined\]](#). Sometimes, an excerpt of the input file is displayed to help locating the source of the error. However, note that diagnosing syntax errors is not perfect, so the reported location may not be the root cause of the syntax error.

11.2 Command line errors [#02]

These errors indicate that the given command line parameters are wrong, for instance that a required argument is missing or that a combination of arguments is incompatible.

invalid command-line parameter(s)

The command-line parameters do not match the requirements. The message is usually combined with an indication which parameter is wrong, for instance ‘lola: option ‘--formula’ requires an argument’. Check the help output (see ‘lola --help’ or ‘lola --detailed-help’) for more information.

too many files given - expecting at most one

LoLA can only read at most one net file – if no file is given, LoLA reads from standard input. The error message occurs if you called LoLA with more than one net file. Remember formula files need to be passed using ‘--formula=myformulafile’.

--check=modelchecking given without --formula or --buechi

In the ‘modelchecking’ mode, either a formula or a Büchi automaton must be passed using the ‘--formula’ or ‘--buechi’ parameter.

specified store does not fit the given task

this encoder can not decode states

this store cannot return states

Not all combinations of tasks, encoders, and stores are supported. Please refer to Chapter 9 [\[Storage concepts\]](#), page 32 or [\[undefined\]](#) [\[Supported Properties\]](#), page [\[undefined\]](#) for more information.

11.3 File input/output errors [#03]

could not close *purpose* file *filename*

could not open *purpose* file *filename*

When LoLA encounters a problem opening files for reading or writing, a respective error message is shown. Usually, the message is accompanied with additional information such as ‘last error message: No such file or directory’.

11.4 Thread error [#04]

thread could not be created

mutexes could not be created

mutex conditions could not be created

named semaphores could not be created

named semaphore could not be closed and/or unlinked

LoLA uses POSIX threads to realize multi-threaded execution. As the resources are limited, errors may occur that are related to the pthread API.

12 Integration guide

LoLA follows the UNIX principle of “everything is a file”. Thereby, integrating LoLA into other tools boils down to choosing the needed command line parameters, providing the input (net, formula) as files, and reading the generated output(s). As LoLA allows to read from standard input and write to standard output, it can further be integrated without generating files at all. With the structured JSON output (see [JSON], page 38), it is furthermore very easy to extract special portions of the output, for instance using a JSON processor like `jq` (see <http://stedolan.github.io/jq/>).

```
$ lola phils10.lola --formula="EF DEADLOCK" --quiet --json | jq ".analysis.result,
.analysis.stats.states"
true
29
```

As another example, consider the following (very simplistic) Python script that passes a net (given as string) to LoLA via standard input and reads the JSON output from standard output into a dictionary.

```
#!/usr/bin/env python

from subprocess import Popen, PIPE
import json

net = """
PLACE p1, p2;
MARKING p1;
TRANSITION t
CONSUME p1;
PRODUCE p2;
"""

lola = Popen(["lola", "--formula=\"EF DEADLOCK\"", "--quiet", "--json"], stdin=PIPE, stdout=PIPE)
output = lola.communicate(input=net)

result = json.loads(output[0])

net_has_deadlock = result['analysis']['result']
```

13 Utilities

LoLA follows the UNIX principle of having one tool for one purpose. As such, several helper functions have been moved from the main tool into smaller utility. These utilities are intended to simplify scripting LoLA and to run it on remote locations.

The utilities are compiled together with LoLA and are located in the `utils` directory. Note the utilities are not installed with `'make install'`.

13.1 Remote reporting (listener)

LoLA can send all reporting information to a remote destination via UDP. With the remote reporting utility, `listener`, these reports can be received.

The `listener` tool runs until aborted with `CTRL+c` and prints all received reports to the standard output. The input port is `'5555'` by default and can be changed in the source code.

Machine running listener

```
$ listener
lola: listening on port 5555
lola: 127.0.0.1 21:51:01: pid = 45541
lola: 127.0.0.1 21:51:01: reading net from phils10.lola
lola: 127.0.0.1 21:51:01: finished parsing
lola: 127.0.0.1 21:51:01: closed net file phils10.lola
lola: 127.0.0.1 21:51:01: 90/65536 symbol table entries, 0 collisions
lola: 127.0.0.1 21:51:01: preprocessing net
lola: 127.0.0.1 21:51:01: computing forward-conflicting sets
lola: 127.0.0.1 21:51:01: computing back-conflicting sets
lola: 127.0.0.1 21:51:01: 60 transition conflict sets
lola: 127.0.0.1 21:51:01: finding significant places
lola: 127.0.0.1 21:51:01: 50 places, 40 transitions, 30 significant places
lola: 127.0.0.1 21:51:01: read: AG (EF (ea.1 = 1))
lola: 127.0.0.1 21:51:01: formula length: 18
lola: 127.0.0.1 21:51:01: checking liveness
lola: 127.0.0.1 21:51:01: processing formula
lola: 127.0.0.1 21:51:01: processed formula: !(E(TRUE U !(E(TRUE U (ea.1 <= 1 AND ea.1 > 0))))))
lola: 127.0.0.1 21:51:01: processed formula length: 50
lola: 127.0.0.1 21:51:01: 4 rewrites
lola: 127.0.0.1 21:51:01: formula mentions 1 of 50 places; total mentions: 2
lola: 127.0.0.1 21:51:01: using a bit-perfect encoder (--encoder)
lola: 127.0.0.1 21:51:01: using 120 bytes per marking, with 0 unused bits
lola: 127.0.0.1 21:51:01: using a prefix store (--store)
lola: 127.0.0.1 21:51:01: checking a formula (--check=modelchecking)
lola: 127.0.0.1 21:51:01: finished preprocessing
lola: 127.0.0.1 21:51:01: CTL formula contains 2 significant temporal operators and needs
9 bytes of payload
lola: 127.0.0.1 21:51:01: result: no
lola: 127.0.0.1 21:51:01: The net does not satisfy the given formula.
lola: 127.0.0.1 21:51:01: 3113 markings, 13384 edges
lola: 127.0.0.1 21:51:01: killed reporter thread
lola: 127.0.0.1 21:51:01: done
```

Machine running LoLA

```
$ lola phils10.lola --formula="AGEF ea.1 = 1" --reporter=socket
```

13.2 Remote termination (killer)

LoLA can be remotely terminated by sending special UDP packages to a running instance. With the remote termination utility, `killer`, such packages can be sent.

The **killer** tool sends a termination package to a predefined address (default: 'localhost') and port (default: '5556'). The package further has a payload (default: 'goodbye') that needs to match the running LoLA. All default values can be changed in the source code.

Machine running LoLA

```
$ lola garavel.lola --formula="EF FIREABLE(t553)" --remoteTermination
lola: enabling remote termination (--remoteTermination)
lola: setting up listener socket at port 5556 - secret is goodbye
lola: reading net from garavel.lola
...
lola: finished preprocessing
lola:      205181 markings,      235592 edges,      41036 markings/sec,      0 secs
lola:      403256 markings,      467135 edges,      39615 markings/sec,      5 secs
lola: received KILL packet (goodbye) from 127.0.0.1 - shutting down
lola: caught signal User defined signal 1: 30 - aborting LoLA
lola: killed listener thread
```

Machine running killer

```
$ lola: sending KILL packet (goodbye) to localhost:5556
```

13.3 Translating PNML to LoLA

Given a place/transition Petri net described in the Petri Net Markup Language (PNML), you may try the utility **pnml2lola** for translating it into the LoLA net file format. This tool is very simple and has been developed for the particular PNML files occurring in the yearly model checking contests (MCC). Most certainly, our tool is NOT robust w.r.t. all the bells and whistles generally permitted in XML documents.

The tool reads from stdin and writes to stdout:

```
$ pnml2lola < given.pnml > produced.lola
```

It cannot process PNML documents that model a high-level Petri net.

13.4 Translating XML property files to LoLA

Given an XML description of a property for place/transition nets as used in the 2016 and 2017 editions of the MCC, a corresponding property file in LoLA can be generated. The tool is called **formula2lola**. It reads from stdin and writes to stdout:

```
$ formula2lola < given.xml > produced.lolatask
```

It is NOT robust w.r.t. XML bells and whistles but seems to work fine for the property files automatically generated in recent editions of the MCC. It cannot process formulas designed for high level nets.

14 Examples

In the following, we describe some examples from the `example` folder:

folder	description
<code>data</code>	reader/writer mutual exclusion
<code>mutex</code>	simple mutex algorithm
<code>vasy</code>	high-end multiprocessor architecture (Vasy example)

14.1 Reader/writer mutual exclusion (data)

14.1.1 Overview

The **Reader/writer mutual exclusion** example models a system with readers and writers. Reading can be conducted concurrently whereas writing has to be done exclusively. This is modeled by a number of semaphores (one for each reader) that need to be collected by a writer prior to writing.

A graphical version is depicted in `data.pdf`. The model was originally modeled as high-level Petri net (`data.hllola`), so versions with different numbers of readers and writers can be quickly generated. The folder `data` contains a version of 10, 20, 50, 100 readers/writers, respectively. Furthermore, the folder contains two formulae for each version: `write-mutex-i.formula` expressing that at most one writer can write at a time and `rw-mutex-i.formula`, expressing that at if data is read, no data is written.

14.1.2 Write mutual exclusion

Let us first verify the mutual exclusion of writers:

```

$ lola data-10x10.lola -f write-mutex-10.formula
lola: reading net from data-10x10.lola
lola: finished parsing
lola: closed net file data-10x10.lola
lola: 90/65536 symbol table entries, 0 collisions
lola: preprocessing net
lola: computing forward-conflicting sets
lola: computing back-conflicting sets
lola: 31 transition conflict sets
lola: finding significant places
lola: 50 places, 40 transitions, 20 significant places
lola: reading formula from write-mutex-10.formula
lola: read: AG (wri.1 + wri.2 + wri.3 + wri.4 + wri.5 + wri.6 + wri.7 + wri.8
+ wri.9 + wri.10 <= 1)
lola: formula length: 88
lola: checking invariance
lola: processing formula
lola: processed formula: wri.1 + wri.2 + wri.3 + wri.4 + wri.5 + wri.6 + wri.7
+ wri.8 + wri.9 + wri.10 > 1
lola: processed formula length: 82
lola: 65 rewrites
lola: processed formula with 1 atomic propositions
lola: formula mentions 10 of 50 places; total mentions: 10
lola: closed formula file write-mutex-10.formula
lola: using a bit-perfect encoder (--encoder)
lola: using 4 bytes per marking, with 12 unused bits
lola: using a prefix store (--store)
lola: checking a formula (--check=modelchecking)
lola: finished preprocessing
lola: result: yes
lola: The net satisfies the given formula.
lola: 21 markings, 40 edges

```

As this is the first example, we shall explain line by line:

```

lola: reading net from data-10x10.lola
lola: finished parsing
lola: closed net file data-10x10.lola
lola: 90/65536 symbol table entries, 0 collisions

```

These lines describe the parsing process: LoLA read the file `data-10x10.lola` and stored it in its symbol table.

```

lola: preprocessing net
lola: computing forward-conflicting sets
lola: computing back-conflicting sets
lola: 31 transition conflict sets
lola: finding significant places
lola: 50 places, 40 transitions, 20 significant places

```

The preprocessing begins: To efficiently fire transitions, LoLA computes so-called conflicting sets – this may take a while for large nets. Next, the place invariant are used to find the significant places (see [Compression by place invariants], page 33). This example has 50 places, but only 20 significant places. Consequently, only the markings of these 20 places are stored yielding a 40 percent reduction.

```

lola: reading formula from write-mutex-10.formula
lola: read: AG (wri.1 + wri.2 + wri.3 + wri.4 + wri.5 + wri.6 + wri.7 + wri.8
+ wri.9 + wri.10 <= 1)
lola: formula length: 88

```

Next, the formula is read from file `write-mutex-10.formula`. This formula expresses the mutual exclusion: the sum of the markings on the places expressing

reading processes (`'wri.1' - 'wri.10'`) must be at most one in all reachable markings (`'AG'`).

```
lola: checking invariance
lola: processing formula
lola: processed formula: wri.1 + wri.2 + wri.3 + wri.4 + wri.5 + wri.6 + wri.7
+ wri.8 + wri.9 + wri.10 > 1
lola: processed formula length: 82
```

LoLA detects that only one temporal operator (`'AG'`) occurs, making this an *invariance* property. As formulas of type `'AG phi'` can be transformed into `'NOT EF NOT phi'`, LoLA will check the reachability of a state in which more than one of the places `'wri.1' - 'wri.10'` is marked. The result of this check will then be negated.

```
lola: 65 rewrites
lola: processed formula with 1 atomic propositions
lola: formula mentions 10 of 50 places; total mentions: 10
lola: closed formula file write-mutex-10.formula
```

These are statistical outputs: to detect the formula type and to transform it into a simpler form, it was rewritten in 65 steps. The formula contains one atomic proposition (the sum of the places must be greater 1) and mentions 10 of 50 places. The latter is a metric how “local” the formula is in the sense that LoLA’s reductions are most efficient if only few places are mentioned in the formula.

```
lola: using a bit-perfect encoder (--encoder)
lola: using 4 bytes per marking, with 12 unused bits
lola: using a prefix store (--store)
lola: checking a formula (--check=modelchecking)
lola: finished preprocessing
```

After processing the formula, information about the encoder and store are printed (see Chapter 9 [Storage concepts], page 32 for more information). As we neither specified an encoder nor a store, the default values are used. A bit-perfect encoder expresses each marking with 4 bytes which are stored by a prefix store. This concludes the preprocessing and begins the checking of a formula (see [Temporal logic], page 11).

```
lola: result: yes
lola: The net satisfies the given formula.
```

LoLA could verify the formula: all reachable markings satisfy the mutual exclusion of the writers. Note LoLA originally checked `'NOT EF NOT phi'` and returned `'yes'`, because it could not find a reachable marking that violated `'NOT phi'`. That is, LoLA’s final output always answers whether the original input formula holds and LoLA takes care about any necessary negations or simplifications.

```
lola: 21 markings, 40 edges
```

Finally, LoLA reports the number of markings it generated (21) and how many transitions were fired (40).

Also the larger versions satisfy this mutex:


```

$ lola data-10x10.lola -f write-mutex-10.formula
...
lola: result: yes
lola: The net satisfies the given formula.
lola: 41 markings, 80 edges

$ lola data-20x20.lola -f write-mutex-20.formula
...
lola: result: yes
lola: The net satisfies the given formula.
lola: 41 markings, 80 edges

$ lola data-50x50.lola -f write-mutex-50.formula
...
lola: result: yes
lola: The net satisfies the given formula.
lola: 101 markings, 200 edges

$ lola data-100x100.lola -f write-mutex-100.formula
...
lola: result: yes
lola: The net satisfies the given formula.
lola: 201 markings, 400 edges

```

14.1.3 The complete state space as benchmark

The previous example showed we could prove mutual exclusion of the writer processes by checking no more than 201 markings. To get a feeling for the size of the complete state space, let us build it without reduction:

```

$ lola data-20x20.lola --check=full
...
lola: 1048238 markings, 6239951 edges, 209648 markings/sec, 0 secs
lola: 1048500 markings, 12677119 edges, 52 markings/sec, 5 secs
lola: 1048576 markings, 19175337 edges, 15 markings/sec, 10 secs
lola: result: no
lola: 1048596 markings, 20971560 edges

```

As we can see, the number of reachable markings explodes. However, we were able to prove mutual exclusion by only visiting a small fraction of these states. Note the result of ‘`--check=full`’ is always ‘no’.

14.1.4 Read/write mutual exclusion

The files `rw-mutex-i.formula` contain formulae expressing that if one process reads data, no other process may write data. This mutual exclusion is formalized as ‘`AG ((rea.1 + rea.2 + ... > 0) -> (wri.1 + wri.2 + ... = 0))`’. Again, formal sums of places allow to naturally and compactly express properties.

We now verify these formulae with the symmetry reduction enabled:

```

$ lola data-10x10.lola -f rw-mutex-10.formula --symmetry
...
lola: computing symmetries (--symmetry)
lola: computed 90 generators (21 in search tree, 69 by composition)
lola: representing 1.31682E+13 symmetries
lola: 0 dead branches visited in search tree
...
lola: result: yes
lola: The net satisfies the given formula.
lola: 3 markings, 22 edges

$ lola data-100x100.lola -f rw-mutex-100.formula --symmetry
...
lola: computing symmetries (--symmetry)
lola: computed 9900 generators (202 in search tree, 9698 by composition)
lola: representing 8.70978E+315 symmetries
lola: 0 dead branches visited in search tree
...
lola: result: yes
lola: The net satisfies the given formula.
lola: 3 markings, 202 edges

```

As we can see, the net and our property are highly symmetry, and LoLA calculates up to $8.70978 \cdot 10^{315}$ symmetries. The total number of states required to prove that the mutual exclusion holds is fixed to 3 and does not grow with additional processes.

14.2 Simple mutex algorithm (mutex)

14.2.1 Overview

This example net models a simple mutex algorithm in which the mutual exclusion between two agents with respect to a critical resource is implemented by a semaphore ‘key’.

A graphical version is depicted in `mutex.pdf`. The file `mutex.hllola` is a high-level version in which the number of agents can be set arbitrarily. The low-level net `mutex.lola` is a version with two agents.

Note the net’s transitions are annotated with fairness assumptions: Transitions ‘g2.0’ and ‘g2.1’ modeling the entry of the critical sections need to be treated strong fairly; that is, if an agent infinitely often requests the entry, it must be granted infinitely often. Transitions ‘g3.0’ and ‘g3.1’ modeling the exit of the critical sections need to be treated weak fairly; that is, agents must not stay in the critical section forever.

14.2.2 Mutual exclusion (safety property)

The file `mutex_safety.formula` models the mutual exclusion as safety property. It states ‘IMPOSSIBLE (critical.0 = 1 AND critical.1 = 2)’ which is equivalent to the CTL formula ‘AG NOT (critical.0 = 1 AND critical.1 = 2)’, which in turn is equivalent to the question whether a marking satisfying ‘(critical.0 = 1 AND critical.1 = 2)’ is reachable.

```

$ lola mutex.lola -f mutex_safety.formula
...
lola: reading formula from mutex_safety.formula
lola: read: AG (!((critical.0 = 1 AND critical.1 = 1)))
lola: formula length: 43
lola: checking invariance
lola: processing formula
lola: processed formula: (critical.0 <= 1 AND critical.0 > 0 AND
critical.1 <= 2 AND critical.1 > 0)
...
lola: The net satisfies the given formula.
lola: 7 markings, 13 edges

```

Again, LoLA communicates how the input formula was read and rewritten. Note that LoLA checks ‘critical.0 = 1’ by ‘(critical.0 ≤ 1 AND critical.0 > 0)’.

14.2.3 Mutual exclusion (liveness property)

File `mutex_liveness.formula` contains a liveness property ‘GF (critical.0 = 0 AND critical.1 = 0)’ expressing that neither agent stays in the critical section forever:

```
$ lola mutex.lola -f mutex_liveness.formula
...
lola: reading formula from mutex_liveness.formula
lola: read: G (F ((critical.0 = 0 AND critical.1 = 0)))
lola: formula length: 43
lola: checking fairness
lola: fairness not yet implemented, converting to LTL...
lola: processing formula
lola: processed formula: G (F ((critical.0 ≤ 0 AND (critical.0 > -1 AND
(critical.1 ≤ 0 AND critical.1 > -1))))
lola: processed formula length: 89
lola: 5 rewrites
lola: transforming LTL-Formula into a Büchi-Automaton
lola: the resulting Büchi automaton has 2 states
...
lola: result: yes
lola: The net satisfies the given formula.
lola: 8 markings, 42 edges
```

LoLA recognizes the formula as fairness, but the dedicated fairness check from LoLA 1.x has not yet been implemented in LoLA 2. Therefore, the property is checked with standard LTL routines. As such, a Büchi automaton is generated from the formula.

Note this formula holds, because we ruled out behavior in which the critical section is never left with fairness assumptions. Consider file `mutex_unfair.lola` without these fairness assumptions:

```
$ lola mutex_unfair.lola -f mutex_liveness.formula
...
lola: result: no
lola: The net does not satisfy the given formula.
lola: 5 markings, 14 edges
```

We can display a counterexample that demonstrates why the liveness property is violated:

```
$ lola mutex_unfair.lola -f mutex_liveness.formula --path
...
lola: writing witness path to stdout
g1.1
g2.1
g0.0
===begin of cycle===
g0.0
===end of cycle===
lola: closed witness path file stdout
...
```

LoLA displays a sequence of transitions, that model a scenario in which one agent stays in the critical section forever: with ‘g1.1 g2.1’, agent 1 enters the critical section. Then, ‘g0.0’ is fired indefinitely, meaning agent 0 never leaves its idle state.

Note that cyclic counterexamples are usual in case of LTL formulae.

Finally, we can verify a *leads-to* property: ‘AGEF ((request.0 = 1) -> (EF critical.0 = 1))’: whenever an agent requests to enter the critical section, it will eventually enter it.

```
$ lola mutex.lola -f mutex_leadsto.formula
...
lola: result: yes
lola: The net satisfies the given formula.
lola: 8 markings, 35 edges
```

14.3 High-end multiprocessor architecture (vasy)

14.3.1 Overview

The model originates from an industrial case study, namely a model (8,500 lines of LOTOS and 3,000 lines of C) developed by Bull for its FAME high-end multiprocessor architecture. The source code of this model (in LOTOS and C) was automatically translated into an interpreted Petri net using the CÆSAR compiler of the CADP toolbox. The present benchmark was obtained by removing all data information (namely, data types, variables, conditions, actions, offers) from the interpreted Petri net in order to obtain a place/transition Petri net.

The model was used in the 2013 edition of the Model Checking Contest, and file `vasy.pdf` lists further information about the model.

14.3.2 Quasi-liveness

In the original specification, quasi-liveness was of interest. As LoLA does not directly support this property, we have to check, for each transition, whether a marking is reachable that enables that transition. This can be done with the CTL formula ‘EF FIREABLE(*t*)’:

```
$ lola vasy.lola -f "EF FIREABLE(t1)"
...
lola: read: EF (FIREABLE(t1))
lola: formula length: 17
lola: checking reachability
lola: processing formula
lola: processed formula: p.476 > 0
...
lola: result: yes
lola: The net satisfies the given formula.
lola: 4 markings, 3 edges
...
```

The ‘FIREABLE’ predicate is unfolded to a property that is true iff the transition is enabled. With a shell script, we can quickly check this property for each individual transition:

```
#!/bin/bash

for i in $(seq 1 775)
do
    lola vasy.lola -f "EF FIREABLE(t$i)"
done
```

This script exploits the fact that the net’s transitions are named ‘t1’ to ‘t775’.

In all but four cases, LoLA gives an answer within seconds. However, for the remaining transitions (‘t516’, ‘t517’, ‘t552’, and ‘t553’), LoLA takes a lot of time and may even run out of memory.

However, we can use a Bloom filter (see [Bloom filter], page 34) which stores a hash value of a few bits rather than the complete marking with several bytes. However, a Bloom filter may produce false positives; that is, LoLA may prematurely abort the state space exploration, because it erroneously assumes a marking has already been generated. Hence, if LoLA does not find a satisfying marking, we cannot be sure whether the property is indeed unsatisfiable.

```
$ lola vasy.lola -f "EF FIREABLE(t516)" --store=bloom
...
lola: using Bloom filter with 2147483648 bit (2048 MB)
lola: using Bloom filter with 2 hash functions (--hashfunctions)
lola: checking a formula (--check=modelchecking)
lola: finished preprocessing
lola:      404989 markings,      473106 edges,      80998 markings/sec,      0 secs
lola: 2147483648 hash table size      false positive probability: 0.0000001422
lola:      839627 markings,      979644 edges,      86928 markings/sec,      5 secs
lola: 2147483648 hash table size      false positive probability: 0.0000006110
lola:     1322988 markings,     1540832 edges,      96672 markings/sec,     10 secs
lola: 2147483648 hash table size      false positive probability: 0.0000015163
lola:     1782432 markings,     2075495 edges,      91889 markings/sec,     15 secs
lola: 2147483648 hash table size      false positive probability: 0.0000027511
lola:     2254965 markings,     2626085 edges,      94507 markings/sec,     20 secs
lola: 2147483648 hash table size      false positive probability: 0.0000044012
lola: Bloom filter: probability of false positive is 0.0000058831
lola: Bloom filter: optimal number of hash functions is 9.7
lola: result: yes
lola: The net satisfies the given formula.
lola: 2607528 markings, 3039534 edges
```

With two hash functions, LoLA uses 2 bits per marking and was able to prove the quasi-liveness of transition ‘t516’ after generating 2607528 markings.

15 Difference to Previous Releases

15.1 From LoLA 1 to LoLA 2

This chapter summarizes the differences since earlier versions of LoLA (1.x).

15.1.1 General workflow

In LoLA 1, you needed to specify the kind of property you wanted to verify (e.g. ‘REACHABILITY’ and the reduction techniques to be applied in a `userconfig.H` file and then compile LoLA. If you used LoLA for checking several properties, this led to a huge pile of executables and some confusion about the configuration they represented. In LoLA 2, you just have a single executable, and all features are controlled through the command line.

15.1.2 Net input

LoLA 2 does not support high level nets. As LoLA 1 is able to output a low level net when started on a high level net, a work-around is available, though. Future releases of LoLA 2 will provide support for high-level nets, but will use another language (closer to the C programming language).

15.1.3 Property specification

In LoLA 1, you had a long list of supported properties. In LoLA 2, you express most properties in the temporal logic CTL*. LoLA 2 rewrites your formula and checks whether it can use a specialized check, or whether to run a general CTL or LTL model checking algorithm. The rewriting process also eliminates subformulas that it can prove to be tautologies or contradictions.

15.1.4 Supported properties

Unlike LoLA 1, LoLA 2 has a complete LTL model checker. Home states are no longer supported. See [Property compatibility], page 16 for a complete comparison of the supported properties.

15.1.5 Atomic propositions

In LoLA 1, atomic propositions compared places (i.e. the number of tokens on them) to constant values, such as ‘ $p7 > 3$ ’. In LoLA 2, you can compare arbitrary formal sums of places, such as ‘ $3 * p3 + p4 \leq 6 * p5 + 7$ ’. In addition, atomic propositions ‘DEADLOCK’, ‘INITIAL’, and ‘FIREABLE(t)’, where t is the name of a transition, have been added. This way, formulas like ‘AG (DEADLOCK OR $p3 > 0$)’ or ‘AG EF INITIAL’ can be specified. For expression of boundedness, atomic propositions such as ‘ $p4 < \infty$ ’ can be used. The symbol ‘ ∞ ’ represents the value ω of coverability graph theory. See [Formula syntax], page 15 for a complete overview of the syntax.

15.1.6 Reduction Techniques

In LoLA 2, the symmetry method is available for arbitrary properties, even for properties given as a formula. This was not supported in LoLA 1. On the other hand, most options concerning the way symmetries are used are no longer available as we support only the integration we found most successful.

15.1.7 Computed information

Output of a complete reachability graph is no longer supported. For most other bits of information, LoLA 2 offers a structured output using the JSON format, see [JSON], page 38.

15.1.8 Progress messages

These messages have been reshaped. They are now emitted at given points in time (rather than given stages of computation). Messages can be suppressed, or deferred to a remote machine via socket communication, see [Remote reporting], page 46.

15.1.9 Multicore Support

Selected tasks in LoLA 2 can be spread over several cores.

15.1.10 Internal Architecture

Benefitting from our experience with LoLA 1, LoLA 2 has a more convincing internal structure. This way, it is easier to extend and maintain LoLA 2. Furthermore, the interplay between properties, stores, search engines, etc. is aligned with the modular structure and therefore less error-prone.

15.1.11 Code Quality

LoLA 2 is a complete re-implementation. Not a single line of code was copied from LoLA 1. For coding LoLA 2, we used a system that reveals test case coverage. In core modules, coverage is at or near one hundred percent. Most parts underwent code reviews. We have carefully hunted for memory leaks.

15.2 From LoLA 2.0 to LoLA 2.1

15.2.1 List of formulas

You can now specify a ':'-separated list of properties that is processed sequentially. With the `--localtimelimit` feature, you can distribute available runtime bounds to the individual queries.

15.2.2 Boolean combinations

If your property is a Boolean combination of subformulas, each subformula is evaluated separately (sequentially). If the value of the overall formula is determined, the remaining subformulas are not evaluated at all.

15.2.3 Portfolio approach

While LoLA 2.0 offered memoryless search as one of the available search strategies, you can apply this in parallel to a regular state space exploration in LoLA 2.1. Furthermore, you can evaluate the siphon/trap property (for deadlock search) or a method based on the Petri net state equation before or in parallel to a state space exploration. The method that returns true or false first, determines the overall run time.

15.2.4 Partial order reduction for LTL and CTL model checking

Partial order reduction (the stubborn set method) was available only for reachability and deadlock queries in LoLA 2.0. In LoLA 2.1, LTL and CTL preserving stubborn sets are available as well.

15.2.5 New stubborn set approach for reachability and deadlock

LoLA 2.0 offered two algorithms for computing stubborn sets. One was based on starting with a small set and adding transitions until all requirements are satisfied. The other was based on starting with all transitions and then deleting transitions as long as all requirements are satisfied. The first approach is quicker, the second leads to better reduction. LoLA 2.1 offers a combined approach where the starting point of the second approach is the result of the first one. With

a few additional implementation tricks, the new method offers a new tradeoff between runtime and reduction power.

15.2.6 Bound properties

Given a formal sum of places, LoLA 2.1 can determine the maximum value that this sum can have for any reachable marking. A dedicated stubborn set method and the symmetry method are available for this feature.

15.2.7 Efficiency improvements

The LTL model checker has been completely re-implemented. It is now able to accept counterexamples much earlier than the approach in LoLA 2.0.

Computation of canonical representatives in the symmetry method has been modified and runs much faster than in LoLA 2.0.

Stubborn set calculation now uses additional structural information (the conflict clusters of the Petri net) that is calculated during preprocessing. This way, calculation is often much faster, and smaller stubborn sets are returned.

15.2.8 Scalability issues

LoLA 2.0 had problems with translating large LTL formulas to Buchi automata, especially when the formula made excessive use of Boolean operators in atomic propositions. The translation in LoLA 2.1 scales much better and takes substantially less time. Furthermore, there LoLA 2.1 has additional capabilities for detecting duplicate formulas, tautologies, and contradictions in atomic formulas. This leads to smaller formulas, better run-time and less time for pre-processing.

In LoLA 2.0, the rewriting system for formulas became very slow and would even crash if the formula made excessive use of Boolean operators in atomic proposition. This problem has been fixed.

15.2.9 JSON output

The structure of the JSON object has been reshaped. You can optionally include a lot of additional information.

15.2.10 AG/EF properties

For properties $AG\ EF\ \phi$, $EF\ AG\ \phi$, $AG\ EF\ AG\ \phi$, and $EF\ AG\ EF\ \phi$ (where ϕ is a state predicate not containing temporal operators), LoLA 2.1 uses a dedicated verification technique that in essence is based on a quick detection of terminal strongly connected components in the state space. The corresponding stubborn set method is stronger than the general CTL preserving method. Note that any CTL property that piles up AG and EF operators in arbitrary alternation, reduces to the mentioned properties, to reachability, or to invariance (all with special support in LoLA 2.1).

15.3 Did you know?

- You can abort LoLA's execution any time with **CTRL+c**. In case LoLA is currently calculating symmetries, only that step is aborted and LoLA continues with incomplete symmetry information.
- Command line parameters can be abbreviated as long as the prefix is unambiguous. For instance, instead of '`--formula`', you can also write '`--form`' or even '`--f`'. The same holds for option names, so '`--check=full`' can be abbreviated by '`--check=f`' or '`-cf`'.
- For any command line option that expects a file name to write to, you can pass '-' to write to standard output.

- If no input file is given, LoLA reads from the standard input.
- When building the state space, LoLA outputs statistics about markings per seconds. When this number suddenly increases, this usually is a sign that the exploration nearly finished.

Index

!

!= integer comparison 15

+

+ integer addition 15

-

- integer subtraction 15
 --buechi parameter 11
 --check=full parameter 11
 --check=modelchecking parameter 11
 --check=none parameter 11
 --depthlimit parameter 29
 --detailed-help parameter 7
 --findpath parameter 29
 --formula parameter 11
 --help parameter 7
 --path parameter 37
 --pathshape parameter 37
 --quiet parameter 37
 --retrylimit parameter 29
 --search=cover parameter 24
 --search=depth parameter 23
 --search=sweep parameter 23
 --siphontrap parameter 30
 --state parameter 37
 --stateequation parameter 29
 -> Boolean implication operator 15

<

< integer comparison 15
 <-> Boolean equivalence operator 15
 <= integer comparison 15

=

= integer comparison 15

>

> integer comparison 15
 >= integer comparison 15

A

A universal path quantifier 15
 ALLPATH universal path quantifier 15
 AND Boolean conjunction operator 15

B

Bloom filter 34
 bound property 15
 boundedness property (net) 19
 boundedness property (place) 19

C

capacity 32
 causal precedence property 21
 command line error -- aborting [#02] message 43
 CONSUME keyword 9
 COPYING file 6
 coverability graph 19
 CTL 13
 cycle reduction 28

D

dead transition property 19
 DEADLOCK atomic proposition 15
 deadlock property 18
 download 6

E

E existential path quantifier 15
 EVENTUALLY eventual occurrence operator 15
 EXPATH existential path quantifier 15
 explicit state space exploration 1

F

F eventual occurrence operator 15
 FALSE Boolean constant 15
 file input/output error -- aborting [#03] message 44
 FIREABLE atomic proposition 15

G

G global occurrence operator 15
 GALS wrapper 3
 GLOBALLY global occurrence operator 15
 grammar, formulae 15
 grammar, Petri net 9
 graph automorphism 27

I

IMPOSSIBLE operator (shortcut for 'AG NOT') 15
 INITIAL atomic proposition 15
 input (general) 1
 invariance property 17
 INVARIANT operator (shortcut for 'AG') 15

K

k-boundedness property (net) 18
 k-boundedness property (place) 18
 killer remote termination utility 46

L

license 6
 listener remote reporting utility 46

liveness property (net)	20
liveness property (state predicate)	20
liveness property (transition)	20
LTL	13

M

MARKING keyword	9
marking output	37
memory (general)	2
model checking	11

N

NEXTSTATE next state operator	15
NOT Boolean negation operator	15

O

OR Boolean disjunction operator	15
output (general)	2

P

partial order reduction	26
path output	37
place invariant	33
PLACE keyword	9
place, (in)significant	33
prefix tree	34
PRODUCE keyword	9
property	11
property (general)	1

Q

quasi-liveness property	19
-------------------------------	----

R

R release operator	15
reachability property	17
REACHABLE operator (shortcut for 'EF')	15
relaxed soundness property (workflow net)	21
RELEASE release operator	15
reversibility property	20

S

SAFE keyword	9
scalar set	27
see manual for a documentation of this error	
message	43
simple property	13
siphon	30
siphon-trap-property	2
soundness property (workflow net)	21
state equation	2, 29
state predicate	17
state space exploration (general)	1
state space, compute	11
STRONG FAIR keyword	9
structural verification	2
stubborn set method	26
symbolic state space exploration	1
symmetry method	27
syntax error -- aborting [#01] message	43

T

temporal logic	11, 13
thread error -- aborting [#04] message	44
TRANSITION keyword	9
trap	30
TRUE Boolean constant	15

U

U until operator	15
UNTIL until operator	15
user interaction (general)	2

V

variable length encoding	32
--------------------------------	----

W

WEAK FAIR keyword	9
--------------------------------	---

X

X next state operator	15
XOR Boolean exclusive disjunction operator	15