LoLA

A Low Level Petri Net Analyser

http://service-technology.org/lola

Version 1.15-unreleased, 29 April 2010

Karsten Wolf

About this document:

This manual is for LoLA – A Low Level Petri Net Analyser, last updated 29 April 2010.

Table of Contents

1 About LoLA 1

1.1 Selected Case Studies 1
1.1.1 Verification of a GALS wrapper 1
1.1.2 Validation of a Petri Net Semantics for WS-BPEL 2
1.1.3 Verification of WS-BPEL choreographies 2
1.1.4 Garavel's challenge in the Petri Net Mailing List 2
1.1.5 Exploration of biochemical networks 2
1.2 Integration 3

2 Net File Format 4

2.1 Basics 4
2.1.1 Identifiers 4
2.1.2 Numbers 5
2.2 File Format for Place/Transition Nets 5
2.2.1 Places 5
2.2.2 Initial Marking 5
2.2.3 Transitions and Arcs 6
2.3 File Format for High Level Nets 6
2.3.1 Sorts 6
2.3.2 Operations 8
2.3.3 Expressions and Left Values 10
2.3.4 Terms and Multiterms 12
2.3.5 Places 12
2.3.6 Initial Marking 13
2.3.7 Transitions and Arcs 13

3 Supported Properties 15

3.1 Properties of the Whole Net 15
3.1.1 Checking Reversibility 15
3.1.2 Checking Deadlock freedom 15
3.1.3 Checking Existence of Home Markings 16
3.1.4 Checking Boundedness 16
3.1.5 Checking Liveness 16
3.1.6 Checking Quasi-Liveness 16
3.1.7 Checking Nothing 17
3.1.8 Not Checking 17
3.2 Properties of a Marking 17
3.2.1 Specification 17
3.2.2 Checking Reachability 18
3.2.3 Checking Coverability 18
3.2.4 Checking Home Status 18
3.3 Properties of a Place 19
3.3.1 Specification 19
3.3.2 Checking Boundedness 19
3.3.3 Checking Death 20
3.3.4 Checking Liveness 20
3.4 Properties of a Transition 20

3.4.1 Specification ... 20
3.4.2 Checking Death ... 20
3.4.3 Checking Liveness ... 21
3.5 Properties of a State Predicate ... 21
3.5.1 Specification ... 21
3.5.2 Checking Reachability ... 22
3.5.3 Checking Liveness ... 22
3.5.4 Checking Fairness ... 23
3.5.5 Checking Stabilization ... 23
3.5.6 Checking Eventual Occurrence ... 23
3.6 Properties of a CTL Formula ... 24
3.6.1 Specification ... 24
3.6.2 Model Checking ... 25

4 Reduction Techniques ... 26
4.1 Symmetries ... 26
4.2 Stubborn Sets ... 27
4.3 Sweep-Line Method ... 28
4.4 Cycle Coverage ... 29
4.5 Coverability Graph ... 29
4.6 Attracted Execution ... 30
4.7 Invariant Based Compression ... 30

5 Output ... 32
5.1 Return Value ... 32
5.2 Witness Path ... 32
5.3 Witness State ... 32
5.4 Computed Portion of the State Space ... 33
5.5 Place/Transition Net ... 33
5.6 Net Automorphisms ... 34
5.7 The Generated Progress Measure ... 34
5.8 Status information ... 35
5.9 State Limit ... 35

6 Download ... 36

7 First Steps ... 37
7.1 Setup and Installation ... 37
7.2 Contents of the Distribution ... 37

8 Additional Utilities ... 38
8.1 Drawing Reachability Graphs: graph2dot ... 38

9 Version History ... 41

Index ... 45

1 About LoLA

Abstract

LoLA (a Low Level Petri Net Analyzer) has been implemented for the validation of reduction techniques for place/transition net state spaces. Its particular strengths include

- A large number of available state space reduction techniques many of which may be applied jointly;
- A high degree of automation for various state space reduction techniques
- Availability of dedicated variations of state space reduction techniques for several frequently used properties
- efficient implementation exploiting the particular nature of Petri net models
- simple textual interaction for easy integration into other tools

LoLA has been tested on several UNIX platforms (FreeBSD, Solaris), Linux, as well as under Windows using the CYGWIN environment.

1.1 Selected Case Studies

The following list gives a short summary of some case studies involving the use of LoLA.

- Verification of a GALS (globally asynchronous locally synchronous system) wrapper
- Validation of a Petri Net Semantics for WS-BPEL (Web Service Business Process Execution Language)
- Verification of WS-BPEL choreographies
- Garavel's challenge in the Petri Net Mailing List
- Exploration of biochemical networks

1.1.1 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 analysed 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 analysed 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 eight hazard situations in the model. Six of them were ruled out by the engineers due to timing constraints which were not modeled. The remaining two hazards were confirmed as real problems. The circuit was redesigned and another verification confirmed the absence of hazard situations.

More information:

- Milos Krstic, Eckhard Grass, and Christian Stahl. **Request-Driven GALS Technique for Wireless Communication System**. In *Proceedings of the 11th International Symposium on Advanced Research in Asynchronous Circuits and Systems (ASYNC 2005)*, New York, NY, USA, pages 76-85, March 2005. IEEE Computer Society.

1.1.2 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 into 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. [DOI]

1.1.3 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 1000 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. [DOI]

1.1.4 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 (binary decision diagram) based) verification or the use of the covering step graph technique, the full state space consisted of almost 10^{22} states.

More information:

- The original posting
- The summary of responses

1.1.5 Exploration of 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.

1.2 Integration

LoLA has been integrated into various other tools.

- The Petri Net Kernel
- The Model Checking Kit
- CPN-AMI

2 Net File Format

In LoLA, the Petri net to be analysed needs to be provided in textual representation. LoLA supports place-transition nets and high-level nets in the shape of an interpreted algebraic Petri net. A place-transition net description starts with the keyword '`NET`', followed by the specification of

- places
- the initial marking
- transitions and arcs

A high level net description starts with the keyword '`SPECIFICATION`', followed by the definition of

- sorts (data domains)
- operations

The defined symbols can be used in the actual net description which is subsequently provided using the keyword '`NET`', followed by the specification of

- places
- the initial marking
- transitions and arcs

2.1 Basics

2.1.1 Identifiers

The rules for building identifiers in LoLA are quite liberal. This way, it should be easy to translate various file formats into LoLA format. Basically, every string of printable characters that does not contain any of the following characters is an identifier: `,`, `;`, `:`, `(`, `)`, `{`, `}`

Exceptions from this rule are numbers and the following reserved strings:

`RECORD`, `END`, `SORT`, `FUNCTION`, `SAFE`, `DO`, `ARRAY`, `STRONG`, `WEAK`, `FAIR`, `ENUMERATE`, `CONSTANT`, `BOOLEAN`, `OF`, `BEGIN`, `WHILE`, `IF`, `THEN`, `ELSE`, `SWITCH`, `CASE`, `NEXTSTEP`, `REPEAT`, `FOR`, `TO`, `ALL`, `EXIT`, `EXISTS`, `RETURN`, `TRUE`, `FALSE`, `MOD`, `VAR`, `GUARD`, `STATE`, `PATH`, `GENERATOR`, `ANALYSE`, `PLACE`, `TRANSITION`, `MARKING`, `CONSUME`, `PRODUCE`, `FORMULA`, `EXPATH`, `ALLPATH`, `ALWAYS`, `UNTIL`, `EVENTUALLY`, `AND`, `OR`, `NOT`, `<->`, `<>`, `->`, `=`, `[`, `]`, `.`, `+`, `-`, `*`, `/`, `:`, `;`, `|`, `(`, `)`, `,`, `>`, `<`, `#`, `>=`, `<=`

Examples:

```
p1
|||=
helloworld
[]....8[[[
```

Identifiers must be separated from other parts of the net file specification using white space characters. White space characters include the blank, newline, and tab characters as well as comments. A comment in LoLA is any text between a pair of curling brackets '`{`', '`}`' on the same line.

Example:

```
{ A { comment }
```

Future versions of LoLA may have additional reserved words. New reserved words shall, however, always consist of capital letters only.

2.1.2 Numbers

Every sequence of digits, separated by white space (blank, tab, newline, or comment, i.e. text enclosed in curling brackets '`{`', '`}`'), is a number. LoLA does not have a concept of signed numbers.

Example:

```
0
11
0001117
```

2.2 File Format for Place/Transition Nets

2.2.1 Places

In LoLA, every place is represented as a unique name. The name can be an identifier according to the general rules for building identifiers or a number. The set of places is specified by the keyword '`PLACE`', followed by a list of sections. Each section starts with a capacity specification and is followed by a comma-separated list of the places names, finished by a '`;`'. The capacity statement may by empty or consist of the keyword '`SAFE`', optionally followed by a number. Using '`SAFE`' without number is equivalent to the specification '`SAFE 1`'. If the file '`userconfig.H`' contains the directive '`#CAPACITY k`', an empty capacity specification is equivalent to the specification '`SAFE k`'. Otherwise, an empty specification represents an unbounded capacity (which is internally approximated by a capacity of 2^{32}). A capacity statement specifies the maximum number of tokens expected on the places which are specified subsequently. LoLA uses the capacity statement only for a compact representation of markings. The firing rule is not effected by the capacity specification. Optionally, the validity of the capacity specifications can be checked during state space generation. For this purpose, the directive '`CHECKCAPACITY`' must be active in the file '`userconfig.H`'.

Example:

```
PLACE SAFE p1, 17, helloworld, p[]....8[[[;
p2 , p3 ; SAFE 7 : p4 , p5;
```

specifies 8 places. Places '`p1`', '`17`', '`hellworld`', '`p[]....8[[[`' are expected to never contain more than 1 token. Places '`p2`', '`p3`' have either unknown bound (without '`#CAPACITY`' in '`userconfig.H`'), or the bound specified in '`userconfig.H`'. Places '`p4`' and '`p5`' are expected to contain at most 7 tokens.

2.2.2 Initial Marking

The initial marking of the net is specified in a separate section starting with the keyword '`MARKING`' and finished by a '`;`'. In between, there is a comma-separated list. Each list item consists of a place name, a '`:`', and a number. The number specifies the number of tokens initially being on the mentioned place. Places which are not mentioned get 0 tokens initially. For places mentioned more than once, the specified token counts are summed up.

Example:

```
MARKING p1 : 13, p2 : 3 , p1 : 4 ;
```

assigns 17 tokens to place '`p1`', 3 tokens to place '`p2`', and 0 tokens to any other place in the net.

2.2.3 Transitions and Arcs

In LoLA, there is, for each transition, a distinguished section for defining that transition and all connecting arcs. The section starts with the keyword '`TRANSITION`' followed by the name of the transition. This name may be built general rules for building identifiers or a number.

Then, optionally, a fairness assumption '`WEAK FAIR`' or '`STRONG FAIR`' may be specified. The assumptions are effective only for the verification of a few properties. A transition is treated *weakly unfair* in an infinite transition sequence iff it is, from some point in the sequence, permanently enabled but never fired. It is treated *strongly unfair* iff it is infinitely often enabled but only finitely often fired.

After that, the list of incoming arcs is specified. This part starts with the keyword '`CONSUME`' and ends with a '`;`'. Between these symbols, there is a comma-separated list of arc specifications. Each arc specification consists of a place name, a '`:`', and a number. It represents an arc from the mentioned place to the currently specified transition. The number represents the multiplicity of the arc.

Finally, the list of outgoing arcs is specified. This part starts with the keyword '`PRODUCE`' and ends with a '`;`' Between these symbols, there is a comma-separated list of arc specifications. Each arc specification consists of a place name, a '`:`', and a number. It represents an arc from the currently specified transition to the mentioned place. The number represents the multiplicity of the arc.

Example:

```
TRANSITION t1 WEAK FAIR
CONSUME p1 : 2 , p2 : 4 ;
PRODUCE p1 : 2 , p3 : 2 ;
```

is a transition which is to be treated weakly fair for some properties. It tests '`p1`' for the presence of 2 tokens, removes 4 tokens from '`p2`', and puts 2 tokens onto p3.

```
TRANSITION t2
CONSUME p1 : 1 ;
PRODUCE ;
```

may be treated unfair. It removes a token from '`p1`' and does not put tokens anywhere.

2.3 File Format for High Level Nets

2.3.1 Sorts

Sorts represents domains for tokens on places. A sort name can be buit according to the general rules for building identifiers. The set of sorts is specified by the keyword '`SORTS`', followed by a list of sort definitions. Each definition consists of a sort name, a '`:`', and a sort description which ends with a '`;`'. The following descriptions are available:

- the description '`BOOLEAN`' with values '`TRUE`' and '`FALSE`';

  Example:

  ```
  s1 = BOOLEAN ;
  ```

- an interval of natural numbers, specified as a comma-separated pair of numbers, enclosed in brackets '`[`' and '`]`'. The description represents all values which are natural numbers greater or equal to the left number, and less or equal to the right number.

Example:

```
s3 = [ 3 , 7 ] ;
```

- the keyword '`ENUMERATE`', followed by a white space separated list of identifiers and the keyword '`END`'. Each identifier represents a distinguished value.

Example:

```
s4 = ENUMERATE
       blue white red
     END ;
```

- the keyword '`ARRAY`' followed by a scalar sort description, the keyword '`OF`', and another arbitrary sort description. The description represents arrays (vectors) where the first sort description represents the set of indices while the second sort description represents the values of components. Any of the sort descriptions mentioned in the first four items of this list are scalar while the remaining two are not.

Example:

```
s5 = ARRAY s2 OF [ 1 , 3 ];
```

- The keyword '`RECORD`' followed by a list of component definitions, finished by the keyword '`END`'. A component definition consists of an identifier, a '`:`', and a sort description. A record description represents cross products of values where each component represents one dimension of the cross product.

Example:

```
s6 = RECORD
       r1 : BOOLEAN;
       r2 : ARRAY [ 1 , 3 ] OF s2
     END ;
```

- any previously specified sort name; represents the description of the mentioned sort.

Example:

```
s2 = s1 ;
```

LoLA considers a canonical ordering on each set of values that can be described by a sort description. The ordering is defined as follows:

- for '`BOOLEAN`': '`FALSE`' < '`TRUE`';
- for intervals: the usual ordering on the natural numbers;
- for enumerations: ascending according to appearance in the description; for arrays: for the smallest index where both values differ, the corresponding component determines the order;
- for records: the values of the first differing components (in order of definition) determine the order.
- According to these rules, there are a unique least element, a unique largest element, and a canonical order of enumeration of all values of a sort description.

Every value of a sort description has a text representation.

- The text representations of the boolean constants are '`TRUE`' and '`FALSE`';

Example:
`TRUE`

- The text representation of a number is the decimal ASCII representation of that number;

  Example:
  `42`

- The text representation of an enumerated value is the ASCII representation of that value;

  Example:
  `red`

- The text representation of a value of an array is a '`|`'-separated list of the component values, in ascending order of their index, enclosed in brackets;

  Example:
  `[1|5|7|3]`

- The text representation of a value of a record is a '`|`'-separated list of the values of the record components (in the order of definition), enclosed in '`<`' and '`>`'.

  Example:
  `<1|TRUE|[1|2|5]|red>`

The text representation is used in the translation from a high-level net to a low-level net.

For sorts, several compatibility rules apply:

- Every sort is compatible to itself and renamings.
- All integer sorts are compatible to each other
- Two '`ARRAY`' types are compatible if their component sorts are compatible and their index sorts represent the same number of values
- Two record types are compatible if they have the same number of components, and the components have pairwise compatible sorts (in the order of specification of the components).

2.3.2 Operations

Operations represent mappings between sorts. The specification of an operation consists of an operation symbol which can later on be used in terms, a typing which controls the construction of terms, and a meaning which is basically a side-effect free program.

The specification of an operation starts with the keyword '`FUNCTION`'. It follows, enclosed in parenthesis, the specification of argument typing and, separated by a '`:`', the specification of a return type. The specification of argument type may be empty or a '`;`'-separated list. Each entry in the list is formed by a comma-separated list of identifiers, followed by a '`:`' and a sort description. The return type is a sort description. Each identifier represents an argument of the specified function. The order of arguments corresponds to the order of appearance of the respective identifiers. The identifiers for the arguments are used in the description of the meaning of the operation.

The specification of the meaning of an operation consists of a declaration part and a statement which is enclosed in the pair '`BEGIN`' and '`END`' of keywords. The declaration part consists of the keyword '`VAR`' and a '`;`'-separated list of declarations. Each declaration consists of a '`,`'-separated list of identifiers, a '`:`', and a sort description. Each declared variable represents a

value which is, upon each execution of the subsequent statement, initialized with the least value of its sort.

The meaning of an operation defines a mapping from the cross-product of domains which are represented by the argument sorts, to the of the data domain represented by the return type. A statement can have any of the following shapes where '*S1*' and '*S2*' are substatements, '*X*' is a declared variable or represents an argument, '*E*', '*E1*', '*E1*', ..., are expressions, and '*L*' is a left value.

`EXIT`

finish execution and return the multiset of values collected so far

`RETURN` *E*

evaluate expression '*E*' and add the resulting value to the collection of values to be returned; continue execution!

L `=` *E*

replace the value of '*L*' with the result of evaluating expression '*E*'

S1 `;` *S2*

execute first statement '*S1*' and then statement '*S2*'

`WHILE E DO S1 END`

perform a loop that consists of evaluating expression '*E*' (of sort '`BOOLEAN`') first, and then executing '*S1*'. Leave the loop as soon as '*E*' evaluates to '`FALSE`'

`REPEAT` *S1* `UNTIL` *E* `END`

perform a loop that consists of executing '*S1*' first and then evaluating expression '*E*' (of sort '`BOOLEAN`'). Leave the loop as soon as '*E*' evaluates to '`TRUE`'

`FOR` *X* `:=` *E1* `TO` *E2* `DO` *S1* `END`

execute '*S1*' once for each value of scalar expression '*X*' between the value of expression '*E1*' and the value of expression '*E2*'

`FOR ALL` *X* `DO` *S1* `END`

perform '*S1*' for each value in the domain of the sort of '*X*', in the canonical order of that domain

`IF` *E* `THEN` *S1* `END`

execute '*S1*' if evaluation of '*E*' yields '`TRUE`'

`IF` *E* `THEN` *S1* `ELSE` *S2* `END`

execute '*S1*' if evaluation of '*E*' yields '`TRUE`', otherwise execute '*S2*'

`SWITCH` *E* `CASE` *E1* `:` *S1* `CASE` *E2* `:` *S2* `...` `ELSE` *S* `END`

Evaluate expression '*E*' and execute the first statement '*Si*' where '*Ei*' has the same value as '*E*'. If no expresion matches, execute '*S*'. The part '`ELSE` *S*' is optional. If it is absent, nothing is executed in a situation where no case expression matches.

All used variables must be declared as arguments or in the declarations section. There are no variables or side-effects. Wherever variables, expressions, or left values occur, attached sorts must be compatible. When integer values (or arrays, records having integer components) are assigned, they are aligned to the target sort. That is, we add or subtract the size of the target interval iteratively until the resulting value fits in the target domain.

Example:

```
FUNCTION allelements() : s
{ returns all elements of domain s }
VAR
  x : s ;
  noprime : ARRAY [ 2 , 10000 ] OF BOOLEAN { initially all entries FALSE }
BEGIN
  FOR ALL x DO
     RETURN x
  END
END
```

Example:

```
FUNCTION allprimesuntil(n : [ 2 , 10000 ] ) : [2 , 10000]
{ returns all prime numbers until n which is expected to be less or equal to 10000 }
VAR
  x , y : [ 2 , 10000 ] ;
  noprime : ARRAY [ 2 , 10000 ] OF BOOLEAN { initially all entries FALSE }
BEGIN
  FOR x = 2 TO n DO
    IF NOT noprime[ x ] THEN
      RETURN x
      y = x
      WHILE x * y <= n DO
        y = y * x ;
        noprime [ y ] = TRUE
      END
    END
  END
END
```

Example:

```
FUNCTION iscontained(x : s ; a : ARRAY i OF s ) : BOOLEAN
{ returns TRUE iff x occurs in a }
VAR
  y : i ;
BEGIN
  FOR ALL y DO
    IF a [ y ] = x THEN
      RETURN TRUE
      EXIT
    END
  END
  RETURN FALSE
END
```

2.3.3 Expressions and Left Values

Expressions can be used in the descriptions of the meaning of operations, in transition guards, and in the specification of state predicates and CTL formulas. An expression represents a single value that may depend on arguments of the operation and values of variables.

Expressions can be built according as follows ('E1' and 'E2' are subexpressions):

- Left values: A variable is an expression. The current value of the variable forms the value of the expression. Its sort is the sort of the variable (specified in the variable declaration. If 'L' is a left value of an array sort, and 'E' an expression of the corresponding index type, then 'L [E]' is an expression, too. Its sort is the component sort of the array sort, its value is the i-th component of the value of 'L' if 'E' evaluates to the ith element of its sort. If 'L' is a left value of a record type, and bla one of its components, then 'L . bla' is a left value. Its value is the corresponding component of the value of 'L', its sort is the sort specified for component bla.
- Numbers, symbols of an enumeration type, the keywords 'TRUE' and 'FALSE' are expressions. There value corresponds to the depicted item. The type is integral, an enumeration type (the one that mentions the item), or 'BOOLEAN', resp.
- Logical connectives: If 'E1' and 'E2' are expressions of type 'BOOLEAN', so are 'E1 <-> E2', 'E1 -> E2', 'E1 AND E2', 'E1 OR E2', and 'NOT E1'. The value is the logical "if and only if", "implies", conjunction, disjunction, or negation (resp.) of the values of the subexpressions.
- Comparisons: If 'E1' and 'E2' are expressions, then 'E1 < E2', 'E1 > E2', 'E1 <= E2', 'E1 >= E2', 'E1 = E2', 'E1 <> E2', and 'E1 # E2' are expressions of type 'BOOLEAN'. If the sorts of 'E1' and 'E2' are incompatible, all comparisons except '<>' and '#' evaluate to 'FALSE'. Otherwise, the comparisons represent the usual relations less than, greater than, less or equal, greater or equal, equal, inequal, and an alternative representation of inequal. For scalar sorts, the comparisons are evaluated according to the canonical order of values. For Arrays, and Records, the comparison return 'TRUE' iff it returns true for all pairwise comparisons of the components.
- Arithmetic operations: If 'E1' and 'E2' are expressions of an integral sort, or arrays or records thereof, then 'E1 + E2', 'E1 - E2', 'E1 * E2', 'E1 / E2', 'E1 MOD E2', and '- E1' are expressions of the same sort as well. Values correspond to the addition, subtraction, multiplication, division, remainder, and sign change operations, resp. For arrays or records, the operation is performed component-wise.
- Parenthesis: If 'E' is an expression, so is '(E)'. This way, operation precedence can be controlled.
- Aggregation: If 'E1',..., 'Ek' are expressions with compatible sort, then '[E1 | E2 ... | Ek]' is an expression of sort 'ARRAY [1 , k] OF' that sort. Its components get value according to the values of 'E1',..., 'Ek'.
- Function call: If 'bla' is an operation with k arguments and return sort 's', 'E1',..., 'Ek' expressions compatible to the corresponding argument sorts of bla, then 'bla(E1,...,Ek)' is an expression of sort 's'. Its value is the value returned by executing the meaning of 'bla', with argument values set to the values of 'E1',..., 'Ek'. If 'bla' returns 0 or more than 1 value, LoLA terminates with a run-time error.

Examples:

```
[ TRUE | TRUE | FALSE | FALSE ] <-> [ TRUE | FALSE | TRUE | FALSE ]
```

evaluates to

```
[ TRUE | FALSE | FALSE | TRUE ]
```

and

```
[ 1 | 2 ] * [ 2 | 3 ]
```

evaluates to

```
[ 2 | 6 ]
```

2.3.4 Terms and Multiterms

Terms represents combinations of defined operations, that is, mappings between domains which are specified as sorts.

A Term is a variable (which must be declared in the context of the term occurrence, or an operation symbol with a comma-separated list of (sub-)terms, enclosed in parenthesis. The number of subterms must fit to the number of specified arguments for the operation.

If a term is a variable, its sort that has been attached to the variable in its declaration. If a term is an operation, its sort is the specified return domain of the operation. Each subterm must have a sort that is compatible with the sort of the corresponding argument in the top-level operation.

Given a value for each variable that occurs in a term, a term can be evaluated to a multiset of values from the domain that is represented by its sort. If subterms evaluate to multisets, the top-level term is evaluated for each combination of values of the subterms and the results are summed up (using multiset addition).

Assume that '`all()`' evaluates to the multiset '`[1,2,3]`', '`x`' has value 2, '`even(1)`' = '`even(3)`' = '`FALSE`', and '`even(2)`' = '`TRUE`'. Let '`cross(x,y)`' evaluate to a '`RECORD`' with component a taking value '`x`' and component '`b`' taking value '`y`'. '`even(all())`' evaluates to '`[2 FALSE, TRUE ]`', '`cross( all(), all())`' evaluates to '`[<1|1>,<1|2>,<1|3>,<2,1>,<2,2>,<2,3>,<3,1>,<3,2>,<3,3>]`', and '`even(x)`' evaluates to '`[TRUE]`'.

A multiterm can be a term, a term followed by a '`:`' and a number k, or two multiterms with enclosed '`+`'. A multiterm evaluates to a multiset that corresponds to the evaluation of the denoted term, the multiset where each element occurs k time as often as in the specified term, or the multiset where the occurrence of each element corresponds to the sum of its occurrences in the two involved multiterms, resp.

Example:

`even(twotimesthree()):3 + even(one()) + even(two()):5`

evaluates to the multiset that assigns 7 to '`FALSE`' and 5 to '`TRUE`', under the assumption that '`even(twotimesthree())`' evaluates to '`TRUE`' occurring twice, '`even(one())`' evaluates to a single occurrence of '`FALSE`', and '`even(two())`' evaluates to a single occurrence of '`TRUE`'.

2.3.5 Places

In LoLA, every place is represented as a unique name. The name can be an identifier according to the general rules for building identifiers or a number. The set of places is specified by the keyword '`PLACE`', followed by a list of sections. Each section starts with a capacity specification and is followed by a comma-separated list, finished by a '`;`'. Each entra in the list consists of a place name, a '`:`', and a sort name. The sort name specifies the data domain for tokens on the mentioned place. The capacity statement may by empty or consist of the keyword '`SAFE`', optionally followed by a number. Using '`SAFE`' without number is equivalent to the specification '`SAFE 1`'. If the file '`userconfig.H`' contains the directive '`#CAPACITY` *k*', an empty capacity specification is equivalent to the specification '`SAFE` *k*'. Otherwise, an empty specification represents an unbounded capacity (which is internally approximated by a capacity of 2^{32}). A capacity statement specifies the maximum number of tokens of a particular value expected on the places which are speficied subsequently. LoLA uses the capacity statement only for a compact representation of markings. The firing rule is not effected by the capacity specification. Optionally, the validity of the capacity specifications can be checked during state space generation. For this purpose, the directive '`CHECKCAPACITY`' must be active in the file '`userconfig.H`'. The specification of HL-net places may be arbitrarily mixed with the specification of place/transition net places.

This option must, however, be used with care as LoLA translates every HL net place into a set of place/transition net places. Every resulting place has a name that consists of the name of the HL Net place, a '.', and a text representation of a value. LoLA does not avoid resulting name clashes with specified place/transition net place names.

Example:

```
PLACE SAFE p1 : phil, 17 : phil, helloworld, p[]....8[[[ : bla;
p2 : bla , p3 : bla; SAFE 7 : p4 : bla , p5;
```

specifies 8 high level places. Places '`p1`', '`17`' contain tokens of sort (domain) '`phil`', places '`helloworld`' and '`p5`' are in fact low level places, i.e. they contain black tokens. The remaining places contain tokens of sort (domain) '`bla`'. Places '`p1`', '`17`', '`hellworld`', '`p[]....8[[[`' are expected to never contain more than 1 token per value. Places '`p2`', '`p3`' have either unknown bound (without '`#CAPACITY`' in '`userconfig.H`'), or the bound specified in '`userconfig.H`'. Places '`p4`' and '`p5`' are expected to contain at most 7 tokens.

If sort '`bla`' is defined as '`bla = [ 1 , 3 ] ;`' then LoLA will internally consider places '`p2.1`', '`p2.2`', '`p2.3`', '`p3.1`' and so on.

2.3.6 Initial Marking

The initial marking of the net is specified in a separate section starting with the keyword '`MARKING`' and finished by a '`;`'. In between, there is a comma-separated list. Each list item consists either of a low level place name, a '`:`', and a number, or a high level place name, a '`:`', and a multiterm of the sort which is specified for the corresponding place. The number specifies the number of tokens initially being on the mentioned low level place. The multiterm represents the number of tokens of each value on a high level place. Places which are not mentioned get 0 tokens initially. For places mentioned more than once, the specified token counts are summed up.

Example:

```
MARKING p1 : allprimes(), p2 : 3 , p1 : succ(allprimes()) , p1.7 : 3;
```

assigns as many tokens to place p1 as the sum of results of evaluating '`allprimes()`' and '`succ(allprimes())`', with 3 additional tokens on the instance '`p1.7`', and 3 tokens to place '`p2`'. All other places have no tokens in the initial marking.

2.3.7 Transitions and Arcs

In LoLA, there is, for each transition, a distinguished section for defining that transition and all connecting arcs. The section starts with the keyword '`TRANSITION`' followed by the name of the transition. This name may be built general rules for building identifiers or a number.

Then, optionally, a fairness assumption '`WEAK FAIR`' or '`STRONG FAIR`' may be specified. The assumptions are effective only for the verification of a few properties. A transition is treated *weakly unfair* in an infinite transition sequence iff it is, from some point in the sequence, permanently enabled but never fired. It is treated *strongly unfair* iff it is infinitely often enabled but only finitely often fired.

The next part of a transition specification is a variable declaration in exactly the same shape as in the definition of operations. Each assignment of values to these variables defines a firing mode of the transition. The set of firing modes can, optionally, be further restricted through a guard. A guard is specified as a Boolean valued expression, subsequent to the keyword '`GUARD`'. After that, the list of incoming arcs is specified. This part starts with the keyword '`CONSUME`' and ends with a '`;`' Between these symbols, there is a comma-separated list of arc specifications.

Each arc specification consists either of a low level place name, a ‘:’, and a number, or a high level place name, a ‘:’, and a multiterm of the same sort as the mentioned place. It represents an arc from the mentioned place to the currently specified transition. The number represents the multiplicity of the arc. The multiterm may contain the variables which have been specified local to this transition. The multiterm represents an arc expression which maps a firing mode of the transition to a multiset of values to be consumed from the mentioned place.

Finally, the list of outgoing arcs is specified. This part starts with the keyword ‘PRODUCE’ and ends with a ‘;’ Between these symbols, there is a comma-separated list of arc specifications. Each arc specification consists either of a low level place name, a ‘:’, and a number, or a high level place name, a ‘:’, and a multiterm of the same sort as the mentioned place. It represents an arc from the currently specified transition to the mentioned place. The number represents the multiplicity of the arc. The multiterm may contain the variables which have been specified local to this transition. The multiterm represents an arc expression which maps a firing mode of the transition to a multiset of values to be produced on the mentioned place.

Example:

```
TRANSITION t1 WEAK FAIR
VAR x,y: bla; z : phil;
GUARD x < y
CONSUME p1 : allprimes():2 + succ(x):15, p2 : 4 , p3 : z;
PRODUCE p1 : second(x) , p3 : y;
```

is a transition which is to be treated weakly fair for some properties. Fired in mode (x=2, y=3, z=hegel), it removes as many tokens from p1 as specified by allprimes():2 + succ(x), with succ(x) evaluated for x=2. It removes 4 tokens from low level place p2, and one token (of value hegel) from place p3. It produces the tokens as specified by the multiterm second(x), evaluated for x=2, on p1, and a single token of value 3 to place p3.

Internally, each high level transition is replaced by an equivalent set of low level transitions, one for each firing mode that satisfies the guard. The name of a ow level transition consists of the name of the corresponding high-level transition, a ‘.’, and a description of the firing mode which is enclosed in brackets ‘[’ and ‘]’. The firing mode is described as a ‘|’-separated list where an entry consists of a variable name, a ‘=’, and a textual representation of a value, according to the rules explained elsewhere.

Example:

The low level transition of the firing mode used in the previous example could have the name ‘`t1.[y=3|x=2|z=hegel]`’. There are no fixed rules for the order in which the variables appear.

It is typical for high level nets that many firing modes of a high-level transition correspond to dead low level transitions. This way, the internal representation of a high level net can easily cause a memory overflow. For such a case, we recommend to rule out as many as possible dead firing modes through the use of (otherwise unneeded) transition guards. For a variable assignment that violates the guard, we do not generate a low level transition.

A high level net description may contain high level transition definitions as well as low level transition definitions. In such a case, the user is responsible for avoiding name clashes between specified low level transitions and generated low level instances of high-level transitions.

3 Supported Properties

Using LoLA, you can verify various properties, including properties of

- the whole net
- a marking
- a places
- a transitions
- a state predicate
- a CTL-formula

3.1 Properties of the Whole Net

3.1.1 Checking Reversibility

A net is *reversible* iff the initial marking is reachable from every reachable marking.

Edit file '`userconfig.H`' and select the option '`#REVERSIBILITY`' (by uncommenting the appropriate line and commenting all other lines in the properties section). Call '`make`' for generating an executable file. Call that file with your net specification.

Example:

```
lola ph.net
```

The following reduction techniques may be applied:

- Stubborn sets (automatically set)
- Invariant based state compression (always recommended)

3.1.2 Checking Deadlock freedom

A *deadlock* is a marking (reachable from the initial marking) that does not enable any transition.

Edit file '`userconfig.H`' and select the option '`#DEADLOCK`' (by uncommenting the appropriate line and commenting all other lines in the properties section). Call '`make`' for generating an executable file. Call that file with your net specification.

Example:

```
lola ph.net
```

In case of a reachable deadlock, LoLA may produce a witness path and a witness state.

The following reduction techniques may be applied:

- Stubborn sets (always recommended)
- Symmetries
- The sweep-line method
- Cycle coverage
- Attracted execution
- Invariant based state compression (always recommended)

Both depth first search and breadth first search are available. Breadth first search is only recommended if you are interested in a shortest witness path to a deadlock. Choose the search strategy by selecting (commenting or uncommenting) the appropriate lines in the file '`userconfig.H`'.

3.1.3 Checking Existence of Home Markings

A *home marking* is a marking that is reachable from every reachable marking.

Edit file '`userconfig.H`' and select the option '`#HOME`' (by uncommenting the appropriate line and commenting all other lines in the properties section). Call '`make`' for generating an executable file. Call that file with your net specification.

Example:

```
lola ph.net
```

In case of an existing, LoLA may produce a witness state. A witness path is not directly available but may be generated through checking reachability of the witness marking.

The following reduction techniques may be applied:

- Stubborn sets (automatically set)
- Invariant based state compression (always recommended)

3.1.4 Checking Boundedness

A net is *bounded* iff it has finitely many reachable markings.

Edit file '`userconfig.H`' and select the option '`#BOUNDEDNET`' (by uncommenting the appropriate line and commenting all other lines in the properties section). Call '`make`' for generating an executable file. Call that file with your net specification.

Example:

```
lola ph.net
```

In case of an unbounded net, LoLA may produce a witness path which demonstrates the reachability of infinitely many different markings.

The following reduction techniques may be applied:

- Coverability graph (automatically set)
- Stubborn sets (always recommended)
- Symmetries
- Invariant based state compression (always recommended)

Both depth first search and breadth first search are available. Breadth first search is only recommended if you are interested in a shortest generalized witness path. Choose the search strategy by selecting (commenting or uncommenting) the appropriate lines in the file '`userconfig.H`'.

3.1.5 Checking Liveness

A net is *live* iff every transition is.

This verification problem is not directly supported in LoLA. You can, however, easily transform the problem into checking liveness for each individual transition. This way, you generate $|T|$ state spaces instead of one. However, the individual state spaces tend to be significantly smaller than any known reduced state space for the liveness problem for nets.

3.1.6 Checking Quasi-Liveness

A net is *quasi-live* iff no transition is dead in the initial marking

This verification problem is not directly supported in LoLA. You can, however, easily transform the problem into checking death for each individual transition. This way, you generate $|T|$ state spaces instead of one. However, the individual state spaces tend to be significantly smaller than any known reduced state space for the quasi-liveness problem for nets.

3.1.7 Checking Nothing

LoLA has the opportunity of generating a state space without checking any property. This feature is useful for evaluating reduction techniques, or for obtaining (and post-processing) the full state space.

Edit file '`userconfig.H`' and select the option '`#FULL`' (by uncommenting the appropriate line and commenting all other lines in the properties section). Call '`make`' for generating an executable file. Call that file with your net specification.

Example:

```
lola ph.net
```

The following reduction techniques may be applied:

- Stubborn sets (in deadlock preserving version)
- Symmetries
- The sweep-line method
- Cycle coverage
- Invariant based state compression

Both depth first search and breadth first search are available. Breadth first search is only recommended if you are interested in a shortest witness path to a deadlock. Choose the search strategy by selecting (commenting or uncommenting) the appropriate lines in the file '`userconfig.H`'.

3.1.8 Not Checking

LoLA has the opportunity of being run without generating any state space. This feature is useful for getting access to generated pre-processing information such as the unfolded version of a high-level net, the automorphisms generated for the symmetry method, or the progress measure generated for the sweep-line method.

Edit file '`userconfig.H`' and select the option '`#NONE`' (by uncommenting the appropriate line and commenting all other lines in the properties section). Call '`make`' for generating an executable file. Call that file with your net specification.

Example:

```
lola ph.net
```

The following reduction techniques may be applied:

- Symmetries
- The sweep-line method

3.2 Properties of a Marking

3.2.1 Specification

The marking to be analysed can be specified in a separate file. The file name is passed to LoLA using the command line option '`-a`'. If no file name is provided, LoLA generates a file name by replacing the extension of the net input file with '`.task`'. If option '`-A`' is used instead of '`-a`', the specification is read from the standard input stream.

The specification starts with '`ANALYSE MARKING`' followed by a description in the same syntax as for the initial marking (low level version or high level version) of the net. There is, however, no final '`;`'. Instead of a specification in a separate file, the specification can be immediately

appended to the specification of the net. This approach is recommended if LoLA is integrated into another tool and communicates via standard input/output streams.

Example for a low level specification:
`ANALYSE MARKING p1: 3 , p2 : 1 ; hello : 24`

Example for a high level specification:
`ANALYSE MARKING p1 : all(), bla: L(all())`

3.2.2 Checking Reachability

A marking is *reachable* iff there is a transition sequence that transforms the initial marking into the analysed one.

Edit file '`userconfig.H`' and select the option '`#REACHABILITY`' (by uncommenting the appropriate line and commenting all other lines in the properties section). Call '`make`' for generating an executable file. Call that file with your net specification and the specified marking to be analysed.

Example:
`lola ph.net -a ph.state`

In case of a reachable marking, LoLA may produce a witness path.

The following reduction techniques may be applied:

- Stubborn sets (always recommended)
- Symmetries
- The sweep-line method
- Cycle coverage
- Attracted execution
- Invariant based state compression (always recommended)

Both depth first search and breadth first search are available. Breadth first search is only recommended if you are interested in a shortest witness path. Choose the search strategy by selecting (commenting or uncommenting) the appropriate lines in the file '`userconfig.H`'.

3.2.3 Checking Coverability

A marking is *coverable* iff a marking is reachable which is pointwise greater or equal than the analysed one. This verification problem is not directly supported in LoLA. You can, however, easily transform the problem into a reachability problem for a state predicate.

Example:
Coverability of marking
`p1:1, p2:17, p3 : 5`
corresponds to reachability of the predicate
`p1 >= 1 AND p2 >= 17 AND p3 >= 5`

3.2.4 Checking Home Status

A marking is a *home marking* iff it is reachable from every reachable marking.

This verification problem is not directly supported in LoLA. You can, however, easily transform the problem into a liveness problem for a state predicate.

Example:
Marking
`p1:1, p2:17, p3 : 5`
is a home marking iff the predicate
`p1 = 1 AND p2 = 17 AND p3 = 5 AND p4 = 0 AND ...`
is live.

3.3 Properties of a Place

3.3.1 Specification

The place to be analysed can be specified in a separate file. The file name is passed to LoLA using the command line option '`-a`'. If no file name is provided, LoLA generates a file name by replacing the extension of the net input file with '`.task`'. If option '`-A`' is used instead of '`-a`', the specification is read from the standard input stream.

The specification starts with '`ANALYSE PLACE`' followed by the name of the place. In the case of a high-level net, a place instance name as generated by LoLA must be used. For finding out the generated names, call LoLA with option '`-n`' (generates the low level version of the net). Instead of a specification in a separate file, the specification can be immediately appended to the specification of the net. This approach is recommended if LoLA is integrated into another tool and communicates via standard input/output streams.

Example for a low level specification:
`ANALYSE PLACE p1`

Example for a high level specification:
`ANALYSE PLACE hl.1`

3.3.2 Checking Boundedness

A place is *bounded* iff there is a fixed natural number that is greater than the number of tokens on that place, for every reachable marking.

Edit file '`userconfig.H`' and select the option '`#BOUNDEDPLACE`' (by uncommenting the appropriate line and commenting all other lines in the properties section). Call '`make`' for generating an executable file. Call that file with your net specification and the specified marking to be analysed.

Example:
`lola ph.net -a ph.place`

In case of an unbounded place, LoLA may produce a generalized witness path.

The following reduction techniques may be applied:

- Coverability graph (automatically set)
- Stubborn sets (always recommended)
- Symmetries

- Invariant based state compression (always recommended)

Both depth first search and breadth first search are available. Breadth first search is only recommended if you are interested in a shortest generalized witness path. Choose the search strategy by selecting (commenting or uncommenting) the appropriate lines in the file '`userconfig.H`'.

3.3.3 Checking Death

A place is *dead* iff it is unmarked in any reachable marking. This verification problem is not directly supported in LoLA. You can, however, easily transform the problem into a reachability problem for a state predicate.

Example:
Place '`p1`' is dead iff the predicate '`p1 > 0`' is not reachable.

3.3.4 Checking Liveness

A place is *live* iff, from every reachable marking, a marking can be reached that marks that place. This verification problem is not directly supported in LoLA. You can, however, easily transform the problem into a liveness problem for a state predicate.

Example:
Place '`p1`' is live iff the predicate '`p1 > 0`' is live.

3.4 Properties of a Transition

3.4.1 Specification

The transition to be analysed can be specified in a separate file. The file name is passed to LoLA using the command line option '`-a`'. If no file name is provided, LoLA generates a file name by replacing the extension of the net input file with '`.task`'. If option '`-A`' is used instead of '`-a`', the specification is read from the standard input stream.

The specification starts with '`ANALYSE TRANSITION`' followed by the name of the transition. In the case of a high-level net, a transition instance name as generated by LoLA must be used. For finding out the generated names, call LoLA with option '`-n`' (generates the low level version of the net). Instead of a specification in a separate file, the specification can be immediately appended to the specification of the net. This approach is recommended if LoLA is integrated into another tool and communicates via standard input/output streams.

Example for a low level specification:

```
ANALYSE TRANSITION t1
```

Example for a high level specification:

```
ANALYSE TRANSITION tr.[y=3]
```

3.4.2 Checking Death

A transition is *dead* iff it is disabled in every marking reachable from the initial marking.

Edit file '`userconfig.H`' and select the option '`#DEADTRANSITION`' (by uncommenting the appropriate line and commenting all other lines in the properties section). Call '`make`' for generating

an executable file. Call that file with your net specification and the specified marking to be analysed.

Example:
`lola ph.net -a ph.transition`

In case of an non-dead transition, LoLA may produce a witness path and a witness state.

The following reduction techniques may be applied:

- Coverability graph
- Stubborn sets (always recommended)
- Symmetries
- The sweep-line method
- Cycle Coverage
- Attracted execution
- Invariant based state compression (always recommended)

Both depth first search and breadth first search are available. Breadth first search is only recommended if you are interested in a shortest generalized witness path. Choose the search strategy by selecting (commenting or uncommenting) the appropriate lines in the file '`userconfig.H`'.

3.4.3 Checking Liveness

A transition is *live* iff, from every reachable marking, a marking is reachable that enables that transition. This verification problem is not directly supported in LoLA. You can, however, easily transform the problem into a liveness problem for a state predicate.

Example:
t1 with pre-places specified through
`CONSUME p1 : 2, p2 : 5;`
is live iff the predicate
`p1 >= 2 AND p2 >= 5`
is live.

3.5 Properties of a State Predicate

3.5.1 Specification

The predicate to be analysed can be specified in a separate file. The file name is passed to LoLA using the command line option '`-a`'. If no file name is provided, LoLA generates a file name by replacing the extension of the net input file with '`.task`'. If option '`-A`' is used instead of '`-a`', the specification is read from the standard input stream.

The specification starts with '`FORMULA`' followed by the description of the predicate. The syntax for a state predicate is the same as for a CTL-formula. As the only difference, temporal operators ('`NEXTSTEP`', '`ALWAYS`', '`EVENTUALLY`', '`UNTIL`') and path quantifiers ('`EXPATH`', '`ALLPATH`') cannot be used in a state predicate.

Instead of a specification in a separate file, the specification can be immediately appended to the specification of the net. This approach is recommended if LoLA is integrated into another tool and communicates via standard input/output streams.

Example for a low level specification:

```
FORMULA (p1 > 2 AND p3 = 4) OR p6 < 5
```

Example for a high level specification:

```
FORMULA ALL x : phil : [ x = 3 ] OR hasright . ( x ) > 0
```

3.5.2 Checking Reachability

A state predicate is reachable if there is a marking reachable from the initial marking where the given predicate is satisfied.

Edit file '`userconfig.H`' and select the option '`#STATEPREDICATE`' (by uncommenting the appropriate line and commenting all other lines in the properties section). Call '`make`' for generating an executable file. Call that file with your net specification and the specified marking to be analysed.

Example:

```
lola ph.net -a ph.statepredicate
```

In case of a reachable state predicate, LoLA may produce a witness path and a witness state.

The following reduction techniques may be applied:

- Stubborn sets (always recommended)
- The sweep-line method
- Cycle Coverage
- Attracted execution
- Invariant based state compression (always recommended)

Both depth first search and breadth first search are available. Breadth first search is only recommended if you are interested in a shortest witness path. Choose the search strategy by selecting (commenting or uncommenting) the appropriate lines in the file '`userconfig.H`'.

3.5.3 Checking Liveness

A state predicate is reachable live iff, from every reachable marking, there is a marking reachable where the given predicate is satisfied. The property corresponds to the CTL specification **AG EF** ϕ.

Edit file '`userconfig.H`' and select the option '`#LIVEPRPOP`' (by uncommenting the appropriate line and commenting all other lines in the properties section). Call '`make`' for generating an executable file. Call that file with your net specification and the specified marking to be analysed.

Example:

```
lola ph.net -a ph.statepredicate
```

In case of a non-live state predicate, LoLA may produce a witness state, i.e. a marking from which no marking is reachable that satisfies the predicate.

The following reduction techniques may be applied:

- Stubborn sets (always recommended)
- Invariant based state compression (always recommended)

3.5.4 Checking Fairness

A state predicate is fair iff it occurs infinitely often on every path that starts with the initial marking and treats all transitions fair w.r.t. their specified fairness assumption. The property corresponds to the LTL specification **GF** ϕ.

Edit file '`userconfig.H`' and select the option '`#FAIRPRPOP`' (by uncommenting the appropriate line and commenting all other lines in the properties section). Call '`make`' for generating an executable file. Call that file with your net specification and the specified marking to be analysed.

Example:

```
lola ph.net -a ph.statepredicate
```

The following reduction techniques may be applied:

- Stubborn sets (always recommended)
- Invariant based state compression (always recommended)

3.5.5 Checking Stabilization

A state predicate stabilizes iff, on every path that starts with the initial marking and treats all transitions fair w.r.t. their specified fairness assumption, the predicate is satisfied for all but finitely many states. The property corresponds to the LTL specification **FG** ϕ.

Edit file '`userconfig.H`' and select the option '`#STABLEPRPOP`' (by uncommenting the appropriate line and commenting all other lines in the properties section). Call '`make`' for generating an executable file. Call that file with your net specification and the specified marking to be analysed.

Example:

```
lola ph.net -a ph.statepredicate
```

The following reduction techniques may be applied:

- Stubborn sets (always recommended)
- Invariant based state compression (always recommended)

3.5.6 Checking Eventual Occurrence

A state predicate stabilizes iff, on every path that starts with the initial marking and treats all transitions fair w.r.t. their specified fairness assumption, the predicate is satisfied for all but finitely many states. The property corresponds to the LTL specification **F** ϕ.

Edit file '`userconfig.H`' and select the option '`#EVENTUALLYPRPOP`' (by uncommenting the appropriate line and commenting all other lines in the properties section). Call '`make`' for generating an executable file. Call that file with your net specification and the specified marking to be analysed.

Example:

```
lola ph.net -a ph.statepredicate
```

The following reduction techniques may be applied:

- Stubborn sets (always recommended)
- Invariant based state compression (always recommended)

3.6 Properties of a CTL Formula

3.6.1 Specification

The formula to be analysed can be specified in a separate file. The file name is passed to LoLA using the command line option '`-a`'. If no file name is provided, LoLA generates a file name by replacing the extension of the net input file with '`.task`'. If option '`-A`' is used instead of '`-a`', the specification is read from the standard input stream. Instead of a specification in a separate file, the specification can be immediately appended to the specification of the net. This approach is recommended if LoLA is integrated into another tool and communicates via standard input/output streams.

The specification starts with '`FORMULA`' followed by the description of the predicate. For place-transition nets, a formula can be recursively constructed as follows:

- Every comparison is a CTL formula. A comparison consists of a place name, one of the operators '`<`', '`>`', '`<=`', '`>=`', '`=`', '`<>`', '`#`', and a natural number. A marking satisfies a comparison if the number of tokens on the specified places is in the specified relation to the given number. Thereby, '`#`' and '`<>`' both represent inequality while all other operators have the obvious meaning.

  Example:
  ```
  p17 >= 28
  ```

- Boolean combinations of CTL formulas are CTL formulas. The Boolean operators '`AND`', '`OR`', and '`NOT`' can be used. Formulas may be enclosed in parenthesis for controlling precedence.

  Example:
  ```
  p1 > 3 AND NOT (p15 = 1 OR p17 <= 0) AND p3 # 2
  ```

- A CTL formula preceded by any of the following pairs of keywords is a CTL formula: '`ALLPATH ALWAYS`', '`EXPATH ALWAYS`'; '`ALLPATH EVENTUALLY`', '`EXPATH EVENTUALLY`', '`ALLPATH NEXTSTEP`', '`EXPATH NEXTSTEP`'. Thereby, '`ALLPATH`' represents the universal path quantifier **A** and requires validity of the subsequent formula on all paths, '`EXPATH`' represents the existential path quantifier **E** and requires validity of the subsequent formula on at least one path.

  '`ALWAYS`' represents the temporal operator **G** and requires validity of the subsequent formula on all states on a path, '`EVENTUALLY`' represents the temporal operator **F** and requires validity of the subsequent formula on at least one state on a path, '`NEXTSTEP`' represents the temporal operator **X** and requires validity of the subsequent formula on the second state of a path.

  Example:
  ```
  ALLPATH ALWAYS EXPATH EVENTUALLY (p1 = 3 OR EXPATH NEXTSTEP p2 > 5)
  ```

- If *phi* and *psi* are CTL formulas, so are '`EXPATH [` *phi* `UNTIL` *psi* `]`' and '`ALLPATH [` *phi* `UNTIL` *psi* `]`'. '`UNTIL`' represents the temporal operator **U** and requires that *psi* is valid on some state on a path such that *phi* is valid on all preceding states.

  Example:
  ```
  EXPATH [ p1 = 3 UNTIL ALLPATH [p2 = 4 UNTIL ALLPATH EVENTUALLY p7 > 15]]
  ```

For high-level nets, there are the following additional features:

- A formula may be preceded by a variable quantification. A variable quantification starts with one of the keywords '`ALL`' or '`EXISTS`', followed by an identifier, a '`:`', the name of a sort which is defined in the net description, and a '`:`'. The subsequent formula comprises the scope of the quantification.
- A high level place name can be extended with a symbolic instance expression. The expression must be enclosed in parenthesis and is appended to the place name with a '`.`'. The expression must have a sort that is compatible to the sort of the involved place. It may use all constants and operations which are permitted in specifications of operations in the net description. An expression may contain those variables that have been introduced through those quantifications which have the specified expression in their scope.
- Every boolean expression (according to the syntax for operations in the net description, enclosed in brackets, is a CTL formula.

Example:

```
EXPATH EVENTUALLY ALL x : phil : ( [x = 3 ] OR [ hasleft . ( x ) > 0 )] )
```

specifies that there is a reachable marking where all instances of high level place '`hasleft`', except '`hasleft.3`', have at least one token.

3.6.2 Model Checking

A CTL formula is *valid* iff it is satisfied by the initial marking of the net.

Edit file '`userconfig.H`' and select the option '`#MODELCHECKING`' (by uncommenting the appropriate line and commenting all other lines in the properties section). Call '`make`' for generating an executable file. Call that file with your net specification and the specified marking to be analysed.

Example:

```
lola ph.net -a ph.ctlformula
```

In case of a satisfied existentially unquantified formula, or a violated universally quantified formula, LoLA may produce a witness path. The path does only concern the top level temporal operator, i.e., subformulas are treated as if they were atomic. The following reduction techniques may be applied:

- Stubborn sets (always recommended)
- Invariant based state compression (always recommended)

4 Reduction Techniques

The list of reduction techniques in LoLA includes

- Symmetries
- Stubborn Sets
- Sweep Line Method
- Cycle coverage
- Coverability graph
- Attracted Execution
- Invariant based compression

Most techniques may be applied in combination. LoLA shall always use a version of a reduction technique that preserves the analysed property. In many cases, variations of a technique are used which are particularly optimised for the analysed property.

4.1 Symmetries

Applicability

The symmetry method is available to all properties which concern single places, transitions, or markings. It is further applicable to the global verification problems of boundedness, reversibility, deadlock freedom. Symmetry reduction typically causes significant overhead in run-time and memory. It is recommended if the system under investigation exhibits a high degree of regularity such as a lot of identical components which interact in some systematically structured network. The reduction is typically linear in the number of graph automorphisms which in turn may be exponential in the size of the net.

Unlike many other tools, LoLA can determine symmetry in the system completely on its own. It neither needs an external specification of symmetries, nor the use of dedicated data types ("scalar sets"). Instead, it explores all graph automorphisms of the investigated Petri net which yields at least as strong reduction as alternative approaches.

Invocation

The symmetry method is invoked by uncommenting the line '`#SYMMETRY`' in the file '`userconfig.H`', prior to the generation of an executable file. The value '`#SYMMINTEGRATION`' selects a particular strategy of generating a symmetrically reduced state space. '`SYMMINTEGRATION`' can be given any value between 1 and 5. The values have the following meaning:

1 Compute a generating set of the automorphisms in advance. Then, for each marking, iterate the set of all symmetries (with some reasonable shortcuts) and check the symmetric image of the marking for presence in the set of known markings. This method yields maximum reduction but may be prohibitively inefficient for massively symmetrical systems.

2 Do not compute a generating set of the automorphisms in advance. Then, for each marking, iterate all known markings and try to compute a symmetry that maps between them. This method yields maximum reduction. It is only recommendable for massively symmetrical system. Currently, the use of the method is discouraged due to an unfixed bug.

3 Compute a generating set of the automorphisms in advance. Use this set for transforming a newly encountered marking into an approximated canonical representative

which is searched in, and inserted into, the set of known markings. Does not yield maximum reduction but has the by far best performance.

4 Do not compute a generating set of the automorphisms in advance. Compute an approximation of a canonical representative for each newly encountered marking. The method is an alternative to method 3 for massively symmetrical systems.

5 Do not compute a generating set of the automorphisms in advance. Compute a canonical representative for each newly encountered marking. The method is an alternative to method 4 for the case that the penalty of approximating the canonical representative (in terms of a larger state space) is significant.

We strongly recommend the use of method 3. It is the one we most frequently applied so far. Thus, it is the most stable option.

For method 4, the option '`MAXATTEMPT`' controls a trade-off between reduction and run-time. A large value refers to slow verification but close to maximum reduction while a small value refers to fast verification but a larger state space. We do not have much experience with a good choice of that value.

If one of the methods is used where a generating set of all graph automorphisms is generated in advance, LoLA can output this generating set using the '`-y`' option

Compatibility

The symmetry method can be applied in combination with the stubborn set method, the state compression based on invariants, the coverability graph construction, and the cycle coverage method. It is incompatible with the sweep-line method, and attracted execution. It can be applied with both depth-first and breadth-first search.

Further reading

The symmetry methods implemented in LoLA correspond to the publications:

- Tommi A. Junttila. **New canonical representative marking algorithms for place/transition-nets**. In J. Cortadella and W. Reisig, editors, *Application and Theory of Petri Nets 2004*, volume 3099 of Lecture Notes in Computer Science, pages 258-277. Springer, 2004.
- Karsten Schmidt. **How to Calculate Symmetries of Petri Nets**. *Acta Inf.*, 36(7):545-590, 2000. [DOI]
- 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, 2000. Springer-Verlag. [SpringerLink]

4.2 Stubborn Sets

Applicability

The stubborn set method is available for all properties supported by LoLA. Its use is always recommended as it does not produce significant overhead, neither in run time nor space. It performs best if the system under investigation exhibits a substantial amount of concurrency. Making more detailed predictions on its reduction power is rather difficult.

LoLA features a broad range of stubborn set methods. Each is optimised for the verification of a particular property. LoLA shall automatically incorporate a stubborn set method that preserves the class of properties that has been selected by the user in '`userconfig.H`'. Only for checking reachability and dead transition verification, the user may choose between a strict and a relaxed method, as explained below.

Invocation

The stubborn set method is invoked by uncommenting the line '`#STUBBORN`' in the file '`userconfig.H`', prior to the generation of an executable file. The option '`#RELAXED`' toggles the choice between the strict and the relaxed version of the stubborn set method. It is only relevant for reachability and dead transition verification. We recommend to use the strict version in those cases where the investigated marking/state predicate is expected to be reachable or the investigated transition is expected not to be dead. Otherwise, we recommend to use the relaxed version.

Compatibility

The stubborn set method is compatible with all other reduction techniques, and both depth-first and breadth-first search.

Further reading

Most stubborn set methods implemented in LoLA correspond to the publications:

- Antti Valmari. **The State Explosion Problem**. In W. Reisig and G. Rozenberg, Eds, *Lectures on Petri Nets I: Basic Models, Advances in Petri Nets, the Volumes Are Based on the Advanced Course on Petri Nets*, volume 1491 of Lecture Notes in Computer Science, pages 429-528, 1998. Springer-Verlag.
- Rob Gerth, Ruurd Kuiper, Doron Peled, Wojciech Penczek: **A Partial Order Approach to Branching Time Logic Model Checking**. *Inf. Comput.* 150(2): 132-152 (1999)
- Lars Michael Kristensen, Karsten Schmidt, Antti Valmari: **Question-guided stubborn set methods for state properties**. *Formal Methods in System Design (FMSD)* 29(3):215-251 (2006). [DOI]
- Karsten Schmidt. **Stubborn Sets for Model Checking the EF/AG Fragment of CTL**. *Fundam. Inform.*, 43(1-4):331-341, August 2000.
- 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, 1999. Springer-Verlag. [SpringerLink]

4.3 Sweep-Line Method

Applicability

The sweep line method is available for all reachability, and dead transition problems. Use of the method is recommended for systems which exhibit substantial sequential subbehaviours in their components.

The sweep-line method is based on a so-called progress measure. LoLA generates such a progress measure automatically and is able to output it using the '`-y`' option.

Invocation

The sweep-line method is invoked by uncommenting the line '`#SWEEP`' in the file '`userconfig.H`', prior to the generation of an executable file.

Compatibility

The sweep-line method is compatible only with the stubborn set method. Use in combination with the stubborn set method is strongly recommended as otherwise only insignificant reduction can be obtained in many examples.

With the sweep-line method, it is impossible to produce a witness path for the checked property. It is also not possible to output the visited states. However, a witness state can still be generated.

Further reading

The sweep-line method as implemented in LoLA corresponds to the publications:

- Lars Michael Kristensen and Thomas Mailund. **A Generalised Sweep-Line Method for Safety Properties**. *Proc. Formal Methods Europe*, volume 2391 of Lecture Notes in Computer Science, 2002, pp. 549-567
- Karsten Schmidt. **Automated Generation of a Progress Measure for the Sweep-Line Method**. *STTT*, 8(3):195-203, June 2006. Also in: Kurt Jensen and Andreas Podelski, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 10th International Conference, TACAS 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Barcelona, Spain, March 29-April 2, 2004, Proceedings*, volume 2988 of Lecture Notes in Computer Science, pages 192-204, 2004. Springer-Verlag. [DOI]

4.4 Cycle Coverage

Applicability

The cycle coverage method is available only for the verification of reachability and dead transition problems. Its use is subject to a space/time trade-off which may be controlled in the configuration of LoLA.

Invocation

The cycle coverage method is invoked by uncommenting the line '`#CYCLE`' in the file '`userconfig.H`', prior to the generation of an executable file. The method basically consists of storing only as many states as necessary to have at least one state per cycle in the state space in the set of stored states. Other states are computed and processed but not stored. By the cycle coverage property, this method terminates but may compute one and the same state several times. The option '`#MAXUNSAVED`' controls the time/space trade-off through storing additional states. A large number leads to better reduction but increase of run-time while a small number leads to fast verification but weaker reduction. Another way of controlling the trade-off is to set (uncomment) the option '`#NONBRANCHINGONLY`' in the file '`userconfig.H`'. This method has a reasonable performance but may lead to less significant reduction than a good value for '`#MAXUNSAVED`'.

Compatibility

The cycle coverage method is compatible only with the stubborn set and symmetry methods. Use in combination with the stubborn set method is strongly encouraged as otherwise only insignificant reduction is obtained in many examples. The method requires the use of the depth-first search strategy.

Further reading

The method has been described in

- 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, 2003. Springer-Verlag. [SpringerLink]

4.5 Coverability Graph

Applicability

The coverability graph construction is available for the verification of boundedness (of a net or a particular place), and the dead transition problem. For the boundedness problems, its use is compulsory.

Invocation

The coverability graph method is invoked by uncommenting the line '`#COVER`' in the file '`userconfig.H`', prior to the generation of an executable file.

Compatibility

The coverability graph method is only compatible with the stubborn set method and the symmetry method. It can be used for both depth-first and breadth-first search.

Further reading

The coverability graph method implemented in LoLA corresponds to the publications:

- R.M. Karp, R.E. Miller: Parallel program schemata. J. Computer and System Sciences 4, 1969, pp. 147-195
- Karsten Schmidt. **Model-Checking with Coverability Graphs**. *Formal Methods in System Design*, 15(3):239-254, November 1999. [DOI]

4.6 Attracted Execution

Applicability

The attracted execution method is available for all reachability and dead transition properties as well as the existence of deadlocks.

It is not a complete verification technique. It rather generates random execution sequences and checks the visited markings for the property to be verified. In case of non-reachability or death of the investigated transition, LoLA runs forever.

Invocation

The attracted execution method is invoked by uncommenting the line '`#FINDPATH`' in the file '`userconfig.H`', prior to the generation of an executable file.

Compatibility

The attracted execution method is compatible only with the stubborn set method. Application in combination with the stubborn set method is strongly encouraged as only in this combination, execution is attracted towards witness states for the investigated property.

4.7 Invariant Based Compression

Applicability

The compression of states is applicable for all properties supported by LoLA. It does not reduce the number of generated states, but the amount of memory necessary for storing an individual state. It typically improves both run-time and memory consumption while the preprocessing is insignificant. It is thus recommended to always use this technique.

Invocation

The state compression method is invoked by uncommenting the line '`#PREDUCTION`' in the file '`userconfig.H`', prior to the generation of an executable file.

Compatibility

The state compression technique is compatible with all other reduction techniques, except for the sweep-line method.

Further reading

The method corresponds to the publication:

- 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, 2003. Springer-Verlag. [SpringerLink]

5 Output

LoLA is able to produce some valuable output.

- Verification result: return value
- Witness/counterexample path
- Witness/counterexample state
- The generated portion of the state space
- The low level version of a HL net
- The generating set of net automorphisms
- The generated progress measure
- Status information

5.1 Return Value

The result of verification is written to the standard error stream. For a comfortable integration of LoLA into other tools, the result is also passed as the exit value of the program. It can thus be processed in a calling program or a wrapping shell script. The return values of the executable of LoLA has the following meaning:

0	specified state or deadlock found/net or place unbounded/home marking exists/net is reversible/predicate is live/CTL formula true/transition not dead/liveness property does not hold;
1	the opposite verification result as a thumb rule, if the outcome of a verification result can be supported by a counterexample or witness path, that case corresponds to return value 0;
2	Memory overflow during verification;
3	Syntax error in the net or property description
4	Error in accessing files (cannot open, no write permission etc.)
5	Maximal number of states (`MAXIMALSTATES` in '`userconfig.H`') exceeded
other	uncaught memory overflow, or bug

5.2 Witness Path

For some problems, LoLA is able to provide a witness or counterexample path for the verification problem. This path can be accessed using the '`-p`' or '`-P`' command line option when running LoLA. When the '`-p`' option is followed by a file name, the path is written to the specified file. If '`-p`' is used without a file name, a file name is created by replacing the suffix of the net specification file with '`.path`'. Using '`-P`', the path is written to the standard output stream which is convenient for integrating LoLA into other tools.

The path output starts with '`PATH`', followed by a white space separated list of transition names. Instances of high level net transitions come in their generated low level name.

If the coverability graph option is used, parts of the path may be enclosed in parenthesis '`(`' and '`)`'. In that case, the enclosed parts are to be executed "very often" in order to show that some places may have "many" tokens.

5.3 Witness State

For some problems, LoLA is able to provide a witness state for the verification problem. This state can be accessed using the '`-s`' or '`-S`' command line option when running LoLA. When the '`-s`' option is followed by a file name, the state is written to the specified file. If '`-s`' is used without a file name, a file name is created by replacing the suffix of the net specification file with

'`.state`'. Using '`-S`', the state is written to the standard output stream which is convenient for integrating LoLA into other tools.

The state output starts with '`STATE`', followed by the description of a marking in the same format as in the place/transition net description (but without the finalizing '`;`').

5.4 Computed Portion of the State Space

For most verification runs, LoLA is able to report on the computed portion of the state space. Exceptions concern the use of reduction techniques (like the sweep-line method or the goal-oriented execution) and the verification of some properties where advanced state space exploration strategies (other then normal depth-first or breadth-first search) are applied. Graph output can be activated using the '`-g`', '`-G`', '`-m`', or '`-M`' command line options when running LoLA. When the '`-g`' or '`-m`' option is followed by a file name, the graph is written to the specified file. If '`-g`' or '`-m`' is used without a file name, a file name is created by replacing the suffix of the net specification file with '`.graph`'. Using '`-G`' or '`-M`', the graph is written to the standard output stream which is convenient for integrating LoLA into other tools. The graph output consists of a list of states. If the '`-m`' or '`-M`' options are used, we write for each state, a header, the corresponding marking (in the same syntax as for the place/transition net initial marking), and information about successors. The first and third part of the description depend on the underlying search strategy. If the '`-g`' or '`-G`' options are used, only the first and third parts of the description (i.e., the mere graph structure) are written.

If depth first search is used, the header consists of the text '`STATE`' and a number. The number is the depth-first search number which is consecutively assigned to each state upon first visit. The order of appearance in the state output file corresponds to the order of completion of the states. Between '`STATE`' and number, there may occur one of the characters '`!`' or '`*`'. '`!`' identifies a state that proves the actual property (like an existing deadlock, the state satisfying the given predicate, etc.), and the states marked with '`*`' are those on a path from the initial state to the one marked '`!`'. These special marks occur only if the state space has not been traversed completely.

Subsequent to the (optional) marking description, a depth-first graph output lists the set transitions to be considered. For unreduced state space generation, this is the list of enabled transitions, otherwise a subset thereof. For each transition, we mention its name and, separated with '`->`' the number of the resulting successor state. If an incompletely traversed state space is printed, the '`->`' may be replaced by a '`=>`', and the number of the successor state may be replaced by a '`?`'. The single transition marked '`=>`' is the one on the witness or counterexample path for the property to be verified. '`?`' replaces numbers of those states which have not been visited during verification.

If breadth first search is used, the header has the form '`STATE` *`number1`* `FROM` *`number2`* `BY` *`transition-name`*'. '*`number1`*' is the consecutively assigned number of visit of the reported state. '*`number2`*' is the unique number of the predecessor state from which this state has been visited. '*`transition-name`*' is the name of the transition responsible for transforming state '*`number1`*' into state '*`number2`*'.

Subsequent to the (optional) marking description, the list of considered transitions (without information on the reached state number) is listed as a white-space separated list of transition names.

5.5 Place/Transition Net

This feature is only useful if the original net specification is a high-level net. Then, it is possible to generate a complete net description file in the place/transition net syntax, containing the semantically equivalent low level counterpart of the given high-level net. This output option is triggered by the '`-n`' or '`-N`' command line option. When the '`-n`' option is followed by a file

name, the net is written to the specified file. If '`-n`' is used without a file name, a file name is created by replacing the suffix of the net specification file with '`.llnet`'. Using '`-N`', the net is written to the standard output stream which is convenient for integrating LoLA into other tools.

The names used in the generated description correspond to the internally used names for place and transition instances.

5.6 Net Automorphisms

In most cases where the symmetry method is applied, LoLA is able to report the computed set of net automorphisms which describe the symmetries in the net. This information can be accessed using the '`-y`' or '`-Y`' command line option when running LoLA. When the '`-y`' option is followed by a file name, the automorphisms are written to the specified file. If '`-y`' is used without a file name, a file name is created by replacing the suffix of the net specification file with '`.symm`'. Using '`-Y`', the automorphisms are written to the standard output stream which is convenient for integrating LoLA into other tools.

Each automorphism description starts with '`GENERATOR`', followed by numbers which are separated by a '`.`'. These numbers describe the structure of the generating set. The first number is a family number, the second one a consecutive number within each family. Each automorphism can be obtained from generators by composing at most one generator per family. Thereby, the composition of 0 generators is supposed to yield the identity mapping.

Subsequent to the discussed numbers, the actual automorphism (a bijection on the places) is reported. It is description in the so-called cycle notation. It consists of a list of cycles where is cycle is a list of place names, enclosed in parenthesis '`(`' and '`)`'. The corresponding mapping is defined as follows: If a place name does not appear in any cycle, it is mapped to itself. If a place name occurs as the last entry of a cycle, it is mapped to the first entry of this cycle Otherwise, the place is mapped to the respective next entry of its cycle

Example:

```
(a b c) (d f)
```

represents the mapping

```
a->b, b->c, c->a, d->f, e->e, f->d.
```

5.7 The Generated Progress Measure

If the sweep-line method is among the reduction techniques to be used, LoLA calculates a progress measure which is an important ingredient to that technique. This measure can be accessed using the '`-y`' or '`-Y`' command line option when running LoLA. When the '`-y`' option is followed by a file name, the measure is written to the specified file. If '`-y`' is used without a file name, a file name is created by replacing the suffix of the net specification file with '`.state`'. Using '`-Y`', the progress measure is written to the standard output stream which is convenient for integrating LoLA into other tools.

The output starts with the text '`PROGRESS MEASURE`', followed by a white space separated list where each entry consists of a transition name (or the name of a transition instance), a '`:`', and a number. From this information, the used progress measure is defined as follows: The initial marking has progress value 0. If some marking m has progress value p, firing transition t in m leads to marking m', and we report value x for t, then m' has progress value p + x. The design of the measure takes care that progress values are independent of the path on which we reach them.

5.8 Status information

During the execution of LoLA, status information is generated and printed to the standard error output stream. During calculation of net automorphisms, LoLA reports traversal of certain levels in a search tree (which has a size that is equal to the number of places and transitions in the place-transition net). During standard state space exploration, LoLA reports the number of visited states and explored state changes. During plain execution, LoLA reports the number of fired transition. Using the sweep-line method, LoLA reports the number of fired transition, the currently processed round and progress value as well as current and peak number of stored states. The amount of produced information can be controlled through the directive '`#REPORTFREQUENCY`' in the file '`userconfig.H`'. The value refers in most cases to the number of fired transitions after which a message is produced. For the calculation of automorphisms, it refers to the depth at which a message is emitted.

5.9 State Limit

The number of states to be generated can be controlled through the directive '`#MAXIMALSTATES`' in the file '`userconfig.H`'. As soon as this number of states was generated, LoLA terminates with return value 5.

6 Download

The use of LoLA is free under the GNU General Public License which is part of the distribution.

After downloading and unpacking LoLA, there will be a directory called lola containing a number of C++ source files. For running LoLA,

- edit the file '`userconfig.H`'. Most parts of this file concern properties to be verified or available reduction techniques. The effect of editing those parts of '`userconfig.H`' is explained in the respective section of the online documentation. The only remaining option is '`#HASHSIZE`'. Its value controls the size of the hash table for storing visited states. A larger value speeds up state space exploration a little bit at the price of requiring more memory for the table itself. For most users, the original value should be satisfactory.
- Create an executable file by calling the shell tool '`make`'. As a result, there will be an executable called '`lola`'.
- Call '`lola`' with a file containing the net description, a file containing information about the verified property (if applicable – refer to the documentation for details), and some command line options controlling the desired output information (also explained in the documentation).
- The configuration in which LoLA has been generated can be accessed calling '`lola`' with the command line option '`-h`'.

In a subdirectory of LoLA, you can find a number of example Petri net descriptions as well as examples for additional information about verification problems.

7 First Steps

7.1 Setup and Installation

1. Go to http://service-technology.org/files/lola and download the latest release version of LoLA, say 'lola-1.15-unreleased.tar.gz'. To setup and compile LoLA, change into your download directory and type

```
tar xfz lola-1.15-unreleased.tar.gz
cd lola-1.15-unreleased
./configure
make
```

   After compilation, a binary 'src/lola' is generated.[1] If you experience any compiler warnings, don't panic: LoLA contains some generated or third party code that we cannot influence.

2. To test whether everything went fine, type

```
make check
```

   to execute the testcases located in 'tests'. If everything went fine, you should see something like:

```
==================
All 9 tests passed
==================
```

   If an error occurs, please send the output to lola@service-technology.org.

3. To install the binary, the manpage, and the documentation, type

```
make install
```

   You might need superuser permissions to do so.

If you need any further information, see file 'INSTALL' for detailed instructions.

7.2 Contents of the Distribution

The distribution contains several directories:

'doc' The Texinfo documentation of LoLA and a PDF file 'background.pdf' with a short description of the setting in which LoLA should be used. The documentation can be created using 'make pdf'. Note you need to have LoLA properly installed before (see Installation description above).

'src' The source code of LoLA.

'tests' Testcases for LoLA which check the generated binary.

[1] On Microsoft Windows, the file will be called 'lola.exe'.

8 Additional Utilities

The directory '`utils`' contains several small helper tools to postprocess outputs from LoLA.

8.1 Drawing Reachability Graphs: graph2dot

The Dining Philosophers can deadlock if every philosopher takes his left fork. LoLA can find this deadlock. Enter

```
lola-deadlock phils.llnet
```

which returns

```
15 Places
12 Transitions

dead state found!

>>>>> 4 States, 3 Edges, 4 Hash table entries
```

To visualize the generated state space and the deadlock trace, execute

```
lola-deadlock phils.llnet -m
graph2dot -g phils.graph -d phils-deadlock.dot
dot phils-deadlock.dot -Tpng -O
```

The resulting graph '`phils-deadlock.dot.png`' should look as the graph in Fig. 1.

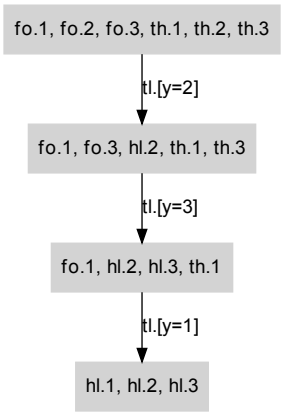

Figure 1. Deadlock trace of the Dining Philosophers

To draw the whole reachability graph of the Dining Philosophers, execute the following commands:

```
lola-full phils.llnet -m
graph2dot -g phils.graph -d phils.dot
dot phils.dot -T png -O
```

The resulting graph '`phils.dot.png`' should look as the graph in Fig. 2.

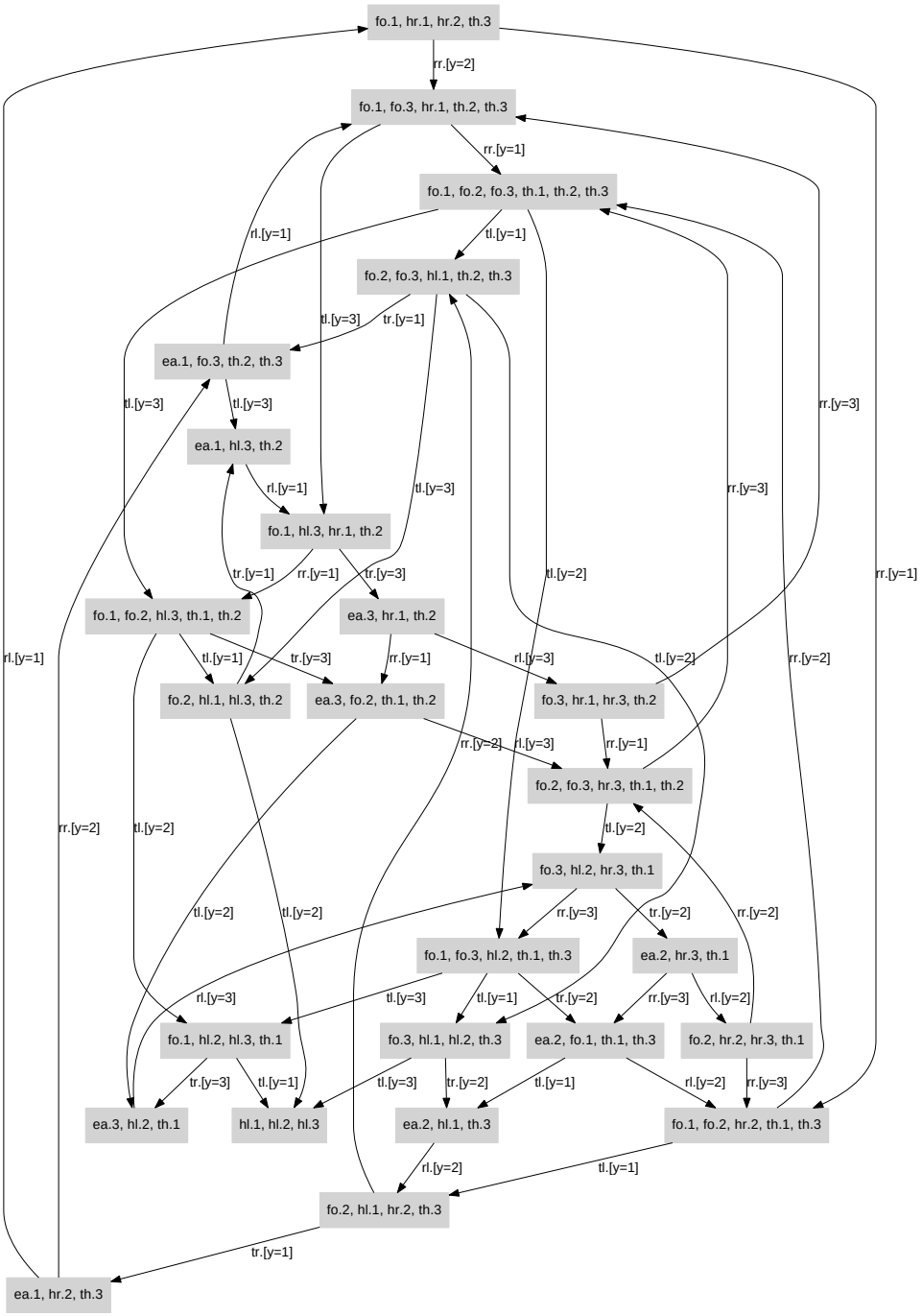

Figure 2. Reachability graph of the Dining Philosophers

To display all command-line parameters of graph2dot, enter

```
graph2dot 1.15-unreleased

Converts a reachability graph into a Graphviz dot notation

Usage: graph2dot [-h|--help] [-V|--version] [-gFILE|--graph=FILE]
         [-dFILE|--dot=FILE] [-cINT|--columns=INT] [-e|--emptyStates]
         [-p|--pathOnly] [-fPLACE|--filter=PLACE] [-xPLACE|--exclude=PLACE]

  -h, --help           Print help and exit
  -V, --version        Print version and exit

Input/output:
  -g, --graph=FILE     Read a reachability graph from file. (If option is
                         omitted, graph2dot reads from stdin)
  -d, --dot=FILE       Write the dot representation to file. (If option is
```

```
                        omitted, graph2dot writes to stdout)

Options:
  -c, --columns=INT    Number of places of a marking to be printed in a state
                         before a newline is printed.
  -e, --emptyStates    Do not print the marking in a state, but only a small
                         circle (useful for large graphs).  (default=off)
  -p, --pathOnly       Only print the states on the witness/counterexample path
                          (default=off)
  -f, --filter=PLACE   Print only the marking of the given places in the
                         states. Multiple places can be given, either as
                         comma-separated list or with multiple '--filter'
                         options.
  -x, --exclude=PLACE  Do not print the marking of the given places in the
                         states. Multiple places can be given, either as
                         comma-separated list or with multiple '--exclude'
                         options.

Examples:
   graph2dot -c1 -fp1,p2,p3 -g net.graph -d net.dot

   lola net.lola -M | graph2dot | dot -Tpng -o net.png
```

9 Version History

LoLA is developed under the "Release Early, Release Often" maxime (see http://catb.org/~esr/writings/cathedral-bazaar/cathedral-bazaar/ar01s04.html): Whenever enough integrated or a non-trivial changes have summed up, a new version is published. Though this releases might now always mark significant changes, they at least allow to quickly fix bugs and avoid infinite procrastination.

Version 1.15-unreleased

- fixed bug #15223 (https://gna.org/bugs/?15223): the coverability graph is now not used in the '`HOME`' mode
- added a parameter '`-r`'/'`--resultFile`' to write all analysis results into a single file in the file format of libconfig (see http://www.hyperrealm.com/libconfig)
- fixed bug #12910 and bug #15282 (https://gna.org/bugs/?12910 and https://gna.org/bugs/?15282): file output for '`-g`' and '`-m`' parameter was broken
- added user configurations for the binaries used by the ProM plugin
- fixed bug #15678 (https://gna.org/bugs/?15678): '`HOME`' now switches on '`TWOPHASE`' if '`STUBBORN`' is also used

Version 1.14

- no statistics output is printed when using directive '`STATESPACE`'
- fixed bug #14295 (see https://gna.org/bugs/?14295): Place is ignored in marking output; isolated places are not removed when using directive '`STATESPACE`'
- whenever a SCC representative is found, all members (excluding the representative) of the current SCC are now printed out when directive '`STATESPACE`' is used

Version 1.13

- fixed a small bug in the '`PREDUCTION`' mode: nets without significant places are now analyzed instantly
- added new scripts to build binary releases
- a binary release consists of the following pre-configured LoLAs: lola-full, lola-full1, lola-deadlock, lola-deadlock1, lola-modelchecking, lola-boundednet, lola-liveprop, lola-liveprop1, lola-statepredicate, and lola-statepredicate1
- changed the way tests are run (using GNU Autotest now)

Version 1.12

- addressed bug #13538 (https://gna.org/bugs/?13538): '`make install`' installs *all* binaries with the name '`lola`' or '`lola-xxx`' for a standard configuration '`userconfig.H.xxx`' that are present in the '`src`' folder. Likewise, '`make uninstall`' removes all installed binaries with the name '`lola`' or '`lola-xxx`' for a standard configuration '`userconfig.H.xxx`'
- adjusted the parser to cope with different line endings (CR, LF, CRLF)
- licensed LoLA under the GNU Affero General Public License (Affero GPL), see file '`COPYING`'

Version 1.11

- fixed bug #12903 (https://gna.org/bugs/?12903): fixed problems regarding '`BOUNDEDNET`' mode
- fixed bug #12907 (https://gna.org/bugs/?12907): fixed problems regarding '`COVER`' option
- completed documentation
- adapted code to avoid deprecation warnings of GCC 4.2
- updated '`-h`' output
- removed old manual '`lola-old.ps`' from documentation
- adapted documentation to fix bug #12090 (https://gna.org/bugs/?12090): syntax description deviated from implementation
- small changes to the test scripts and the lexer to make LoLA compilable on FreeBSD
- added a directory '`src/configs`' containing some example configurations for LoLA; for each file '`userconfig.H.xxx`' in directory '`src/configs`' exists a Makefile target '`lola-xxx`' which compiles LoLA with that configuration
- added Makefile target '`all-configs`' to compile all configurations in directory '`src/configs`'
- added maintainer scripts to create source and binary releases; the latter contain binaries of all configurations in directory '`src/configs`'
- added a directory '`utils`' for small helper tools: graph2dot creates a graphical representation of a reachability graph created by LoLA (using option '`-G`' or '`-M`')
- command line options are now handled by GNU Gengetopt (see http://www.gnu.org/software/gengetopt)
- added a manpage for LoLA using `help2man` tool (see http://www.gnu.org/software/help2man)
- added '`--enable-mpi`' command line parameter for the '`configure`' script to use MPI compiler wrappers instead of GCC (disabled by default)
- added an option `MAXIMALSTATES` to '`userconfig.H`': by defining this option to a value, say 100000, LoLA will abort as soon as more than 100000 states are processed; the exit code will be 5
- canonized LoLA's error messages
- removed directory '`patches`': these options (show number of states after capacity excess; limited state space generation) are now built into LoLA
- added '`--offspring`' command line parameter that creates a file containing all necessary information to compile a new binary with the same configuration used for the calling binary. To compile this new binary, copy the resulting file '`userconfig.H.offspring`' into the '`src/configs`' directory of the source distribution and run '`make lola-offspring`'.

Version 1.10

- this is an official source release by Karsten Wolf – removed warning after executing the configure script
- added a generic Doxygen (http://www.doxygen.org) configuration file '`Doxyfile.in`'
- completed task #6267 (http://gna.org/task/?626): `TWOPHASE` is only set when it makes sense, i.e. only in `LIVEPROP` and `HOME`
- actually using command line parameters ('`--enable-win32`', '`--enable-64bit`', and '`--enable-universal`') from the configure script

Version 1.03

- fixed bug #12302 (https://gna.org/bugs/?12302): LoLA: Exit Codes in `MODELCHECKING`: LoLA now returns the result via the '`_exit`' command
- added directory '`patches`' collecting some unpublished adjustments
- fixed bug #12063 (https://gna.org/bugs/?12063): `LIVEPROP` now works without crashing
- added a test case '`umlprocess`' to avoid regression of bug #12063 (https://gna.org/bugs/?12063)
- out-commented '`TWOPHASE`' in '`userconfig.H`' for liveprop test
- fixed bug #12061 (https://gna.org/bugs/?12061): `BOUNDEDNET` now works without crashing
- added a test case '`unbounded`' to avoid regression of bug #12061 (https://gna.org/bugs/?12061)
- updated the documentation – integrated first parts of http://wwwteo.informatik.uni-rostock.de/ls_tpp/lola/
- the files '`ChangeLog`' and '`NEWS`' are now generated from the file '`doc/ChangeLog.texi`' as it is done in Rachel or BPEL2oWFN
- renamed Makefile target '`cvs-clean`' to '`svn-clean`'
- tidied the configure script and removed unnecessary checks
- set bug reporting address to lola@service-technology.org
- added command line parameters for the '`configure`' script:
  - '`--disable-assert`' to disable assertions (enabled by default)
  - '`--enable-64bit`' to build for a 64 bit architecture such as x86_64 or ppc64 (disabled by default)
  - '`--enable-universal`' to build a Mac Universal binary which is executable on Intel and Power PC platforms (disabled by default)
  - '`--enable-win32`' to build a Windows binary that is independent of a local Cygwin installation (disabled by default)
- updated documentation – took text from http://wwwteo.informatik.uni-rostock.de/ls_tpp/lola

Version 1.02

- fixed bug #12089 https://gna.org/bugs/?12089
- added Makefile target '`win-bin`' that produces a Cygwin independent binary when compiling under Cygwin, helps to avoid bug #12071 http://gna.org/bugs/?12071
- added a testcase '`choreography`' from a BPEL4Chor choreography using symmetries (see file '`tests/nets/choreography.tar.gz`' on how to create this file with BPEL2oWFN)
- fixed bug #12097 https://gna.org/bugs/?12097
- fixed bug #12109 https://gna.org/bugs/?12109

Version 1.01

- fixed bug #12062 http://gna.org/bugs/?12062
- added Makefile target '`cvs-clean`' that removes all files that can be rebuilt by '`autoreconf -iv`'
- the information gathered by the configure script is now collected in a header file '`src/config.H`'

- tidied Makefiles
- added a testsuite (invoked by Makefile target '`check`') consisting of:
  - the Dining Philosophers
  - the Echo Algorithm (currently not used)
  - a business process translated from a UML specification
  - an AI planning problem
- added the documentation from http://www.informatik.hu-berlin.de/~kschmidt/doku.ps
- added a (undocumented) command line parameter '`--bug`' for debug purposes

Version 1.00

- code of a version of Karsten Wolf that has not been officially released; this version is not a completely tested version and is only intended for internal purposes
- minor adjustments (only affecting the frontend) to use the GNU Autotools

The most recent change log is available at LoLA's website at http://service-technology.org/files/lola/ChangeLog.

Index

#

#BOUNDEDNET ... 16
#BOUNDEDPLACE ... 19
#CAPACITY *k* ... 5, 12
#CHECKCAPACITY *k* ... 5, 12
#COVER ... 30
#CYCLE ... 29
#DEADLOCK ... 15
#DEADTRANSITION ... 20
#EVENTUALLYPRPOP ... 23
#FAIRPRPOP ... 23
#FINDPATH ... 30
#FULL ... 17
#HASHSIZE ... 36
#HOME ... 16
#LIVEPRPOP ... 22
#MAXATTEMPT, part of #SYMMETRY ... 26
#MAXIMALSTATES ... 35
#MAXUNSAVED, part of #CYCLE ... 29
#MODELCHECKING ... 25
#NONBRANCHINGONLY, part of #CYCLE ... 29
#NONE ... 17
#PREDUCTION ... 30
#REACHABILITY ... 18
#RELAXED, part of #STUBBORN ... 28
#REPORTFREQUENCY ... 35
#REVERSIBILITY ... 15
#STABLEPRPOP ... 23
#STATEPREDICATE ... 22
#STUBBORN ... 28
#SWEEP ... 28
#SYMMETRY ... 26
#SYMMINTEGRATION, part of #SYMMETRY ... 26

-

--analysis option ... 19, 20, 24
--Analysis option ... 19, 20, 24
--automorphisms option ... 28, 34
--Automorphisms option ... 28, 34
--graph option ... 33
--Graph option ... 33
--marking option ... 33
--Marking option ... 33
--net option ... 19, 33
--Net option ... 19, 33
--path option ... 32
--Path option ... 32
--state option ... 32
--State option ... 32
--userconfig option ... 36
-a option ... 19, 20, 24
-A option ... 19, 20, 24
-g option ... 33
-G option ... 33
-h option ... 36
-m option ... 33
-M option ... 33
-n option ... 19, 33
-N option ... 19, 33
-p option ... 32
-P option ... 32
-s option ... 32
-S option ... 32
-y option ... 28, 34
-Y option ... 28, 34

.

.graph files ... 33
.llnet files ... 33
.path files ... 32
.state files ... 32, 34
.symm files ... 34
.task files ... 17, 19, 20, 21, 24

A

ALL ... 24
ALLPATH ... 21, 24
ALWAYS ... 21, 24
ANALYSE MARKING ... 17
ANALYSE PLACE ... 19
ANALYSE TRANSITION ... 20
AND ... 10, 24
ARRAY ... 6

B

BOOLEAN ... 6

C

CONSUME ... 6, 13

E

ENUMERATE ... 6
EVENTUALLY ... 21, 24
EXISTS ... 24
EXIT operation ... 8
EXPATH ... 21, 24

F

FALSE ... 6
FOR operation ... 8
FORMULA, CTL specification ... 24
FORMULA, state predicate specification ... 21
FUNCTION ... 8

G

GENERATOR ... 34
graph2dot utility ... 38
GUARD ... 13

I

IF operation ... 8

M

MARKING 5, 13

N

NET 4
NEXTSTEP 21, 24
NOT 10, 24

O

OR 10, 24

P

PATH 32
PLACE 5, 12
PRODUCE 6, 13
PROGRESS MEASURE 34

R

RECORD 6
REPEAT operation 8
RETURN operation 8

S

SAFE 5, 12
SORTS 6
STATE 32, 33
STRONG FAIR 6, 13
SWITCH operation 8

T

TRANSITION 6, 13
TRUE 6

U

UNTIL 21, 24

V

VAR 8

W

WEAK FAIR 6, 13
WHILE operation 8