Spatial Logic Model Checker User's Guide
version 0.9

Hugo Vieira and Luís Caires

Departamento de Informática, FCT/UNL

March 2004

1 Introduction

Spatial logics support the specification not only of behavioral properties but also of structural properties of concurrent systems, in a fairly integrated way. Spatial properties arise naturally in the specification of distributed systems. In fact, many interesting properties of distributed systems are inherently spatial, for instance connectivity, stating that there is always an access route between two different sites, unique handling, stating that there is at most one server process listening on a given channel name, or resource availability, stating that a bound exists on the number of channels that can be allocated at a given location. Secrecy can also be sometimes understood in spatial terms, since a secret is a piece of data whose knowledge of is restricted to some parts of a system, and unforgeable by other parts. Spatial logics have been used in the definition of several core languages, calculi, and data models [1, 6, 9, 3, 5].

The Spatial Logic Model Checker is a tool allowing the user to automatically verify behavioral and spatial properties of distributed and concurrent systems expressed in a pi-calculus. The algorithm implemented (currently using on-the-fly model-checking techniques) is provably correct for all processes, and complete for the class of bounded processes [2], an abstract class of processes that includes the finite control fragment of the pi-calculus. The tool itself is written in OCAML, and runs on any platform supported by the OCAML distribution.

For background on spatial logics for concurrency, see [6, 3, 4, 2] and other references therein. Forthcoming releases of this manual will include a short tutorial on the subject, some examples on how to use the tool, and a presentation of the underlying algorithms.

In this report, we specify the syntax of the version of the pi-calculus currently supported in the tool, which is the synchronous polyadic pi-calculus, and the syntax of the spatial logic considered, which currently is in essence the logic described in [2], a spatial logic with behavioral and spatial operators and recursive formulas.

2 Syntax of Processes

Pi-calculus processes are specified according to the concrete syntax definition in Figure 1.

Understanding our syntax for the standard polyadic pi-calculus operators is straightforward. Note that restriction allows for the declaration of more than one restricted name in a row.

We adopt a CSP-like notation for input/output, so that in our syntax an output prefix $\overline{x}y_1y_2\ldots y_n$ is written `x!(`$\mathtt{y}_1$`,`$\mathtt{y}_2$`,...,`$\mathtt{y}_n$`)`, and an input prefix $x(y_1, y_2, \ldots, y_n)$ is written `x?(`$\mathtt{y}_1$`,`$\mathtt{y}_2$`,...,`$\mathtt{y}_n$`)`. The `select` construct refers

```
lower     ::=  ['a' − 'z']
upper     ::=  ['A' − 'Z']
letter    ::=  lower | upper
digit     ::=  ['0' − '9']
name      ::=  lower ( letter | digit | '_')*
namelist  ::=  ϵ | name (',' name)*
action    ::=  name!(namelist)
          |    name?(namelist)
process   ::=  0
          |    process | process
          |    new namelist in process
          |    action.process
          |    select {action.process (';' action.process)*}
          |    [name = name].process
          |    Id(namelist)
          |    ( process )
```

Figure 1: Syntax of Processes.

to the sum operator, being all alternatives either input or output prefixed processes. The test operator is present in it's usual form [*name* = *name*].

Priority of process operators is defined as usual (restriction is more binding that composition), so that *e.g* the process

```
new secret in hand!<secret>.0 | erase?(x).0
```

is parsed as

```
(new secret in hand!<secret>.0) | erase?(x).0
```

rather than as

```
new secret in (hand!<secret>.0 | erase?(x).0)
```

The form *Id*(*namelist*) refers to a process defined using the `defproc` command in the toplevel command interpreter of the model-checker. This command, described below, allows the definition of sets of mutually recursive parametric processes.

3 Syntax of Formulas

Formulas of the spatial logic are specified according to the concrete syntax definition presented in Figure 2. Several of the connectives available are not primitive from a logical viewpoint, but have been directly implemented for the sake of efficiency.

The boolean connectives are negation `not`, conjunction `and`, disjunction `or`, implication `=>`, and equivalence (bi-implication) `<=>`.

Spatial connectives are void `void`, composition (or separation) `|`, decomposition `||` (de Morgan dual of composition), and revelation `reveal` (usually written ® [7]). We also include as a primitive connective the de Morgan dual of revelation `revealall`, and the occurrence connective `@`.

Names can be tested for equality and inequality by the `==` and `!=` operators.

We then have quantifiers over names; the universal quantifier `forall`, the existential quantifier `exists`, the freshness quantifier `fresh`, and the hidden name quantifier `hidden`.

Behavioral modalities are <*label*>, expressing possibility of action (*cf.*, diamond modality of Hennessy-Milner logic), and its dual [*label*], expressing necessity of action (*cf.*, the box modality of Hennessy-Milner logic). The argument *label* of the behavioral modalities specifies the (set of) actions considered. We have:

- `tau` | ϵ, that denote the silent action (an internal reduction step);
- *name*, that denotes any action (input or output) on subject *name*;
- `?`, that denotes any input action;
- `!`, that denotes any output action;
- *name*`!`, that denotes any output action on subject *name*;
- *name*`?`, that denotes any input action on subject *name*;
- *name*`?(`*namelist*`)`, that denotes a particular input action;
- *name*`!(`*namelist*`)`, that denotes a particular output action;
- `*`, that denotes any of the actions specified above.

It is also possible to define properties by recursion, as in the mu-calculus and the spatial logics of [3, 2]: `minfix` denotes the least fixpoint operator, and `maxfix` denotes the greatest fixpoint operator.

Other connectives that are considered as primitive are the k construct, being k an integer constant, that denotes processes that have k components, and `inside` that allows for the inspection of a formula under all restrictions, meaning that all restrictions are revealed using fresh names.

Two other primitive formulas are the `always` and the `eventually` constructs that can be expressed as 'for every possible configuration' and 'there will be a configuration', respectively, with regard to the system's internal evolution.

Last, but not least, formulas can be introduced by (non recursive) parametric definitions, by a mechanism described below (top level command `defprop`). Then $Id(namelist, formulalist)$ denotes a defined property.

```
formula ::= formula | formula
          | formula || formula
          | formula => formula
          | formula <=> formula
          | formula and formula
          | formula or formula
          | ( formula )
          | not formula
          | void
          | true
          | false
          | name == name
          | name != name
          | @ name
          | exists name . formula
          | forall name . formula
          | reveal name . formula
          | revealall name . formula
          | hidden name . formula
          | fresh name . formula
          | <label>formula
          | [label]formula
          | minfix Id.formula
          | maxfix Id.formula
          | k
          | inside formula
          | always formula
          | eventually formula
          | Id(namelist,formulalist)

label   ::= tau
          | name
          | ?
          | !
          | name?
          | name!
          | name?(namelist)
          | name!(namelist)
          | *
```

Figure 2: Syntax of Formulas.

4 Running the Tool

After installation, the tool can be executed by issuing the command

```
% sl-mc_<version>
```

in the operating system shell prompt. Currently, only a minimal command line interface is available.

5 Top level commands

In this section, we list the various commands that can be issued at the top level command prompt of the model checking tool.

Process definition

`defproc` *Id*(*namelist*) = *process* [`and` *Id*(*namelist*) = *process*]* `;`

Process identifiers always start with an upper case letter. An important remark is that the `and` construct enables mutually recursive definitions.

Example

```
> defproc
  EchoServer(chan) =
    chan?(data,reply).(reply!(data).0 | EchoServer(chan))
and
  Client(chan) =
    new callback in
      (chan!(data,callback) | callback?(x).Client(chan))
and
  System() =
    new private in       (Client(private) | EchoServer(private));
```

Property definition

`defprop` *Id*(*idlist*) = *formula*;

Formula identifiers start with a lower case letter. Note that parameters of property identifiers can be either name or formula parameters, but necessarily in that order and distinguished by lower and upper case letters, respectively. When given a *namelist* and a *formulalist*, in accordance to the specification, the formula is obtained through textual substitution of the parameters by the given arguments.

Example

```
> defprop sImp(A,B)= not (A | not B);
```

Checking

`check` *Id*(*namelist*) `|=` *formula*;

To make the check command the most user friendly possible two special constructs can be used in the formulas, `show_succeed` and `show_fail`, being their effect simply the listing of the process that holds or does not hold the formula defined within these special constructs.

Example

```
> check System() |= hidden x.sImp(<x!>true,[x!]false);
   * yes *
```

Trace

`trace` [`on` | `off`] `;`

Switches the trace level on or off. When trace is on and a check command is executed a listing of the process representation is printed to standard output.

Parameter

`parameter` [*ParamId* [*new_value*]]`;`

Shows and defines the values for the model checker parameters. Currently there are three parameters: `max_threads` that bounds the size of processes being evaluated, defined through an integer; `show_time` that defines a mode where the time elapsed in the check procedure is shown, defined through `on` and `off`, that are also used to define parameter `check_counter`, again a mode definition, this one for printing the number of state visits.

Load

`load "`*`filename`*`";`

Executes the declarations and commands in the file whose pathname is obtained by the current path name by appending *`filename`*.

Change Path

`cd "`*`pathname`*`";`

Changes the current pathname to ***pathname***.

Show Path

`pd;`

Shows the current pathname.

Clear

`clear;`

Clears the current session, erasing all process and formula definitions.

List

`list` [`procs` | `props`]`;`

Lists the defined processes (`procs`) or properties (`props`).

Show

`show` *Id*;

Shows the process or formula assigned to the identifier *Id*.

Help

`help;`

Lists available commands.

Quit

`quit;`
Terminates the session.

6 Examples

In this section we illustrate loadable specifications.

Gossiping System

```
/* SYSTEM */

defproc Gossiper(info) = gossip!(info).Gossiper(info);

defproc Listener = gossip?(info).Gossiper(info);

defproc System =
        new secret in
        (
        Gossiper(secret)
        | Listener
        | Listener
        | Listener
        );

/* PROPERTIES */

defprop everywhere(A) = (false || (1 => A));

defprop everybody_knows(secret) = everywhere(@secret);

defprop everybody_gets_to_know =
        hidden secret.eventually everybody_knows(secret);

check System |= everybody_gets_to_know;

/* ---------- */

defprop gossiper_forever = maxfix X.(<gossip!> true and [*]X);

defprop all_gossipers =
        eventually inside everywhere(gossiper_forever);

check System |= all_gossipers;
```

Handover protocol (from Milner's book [10])

```
/* SYSTEM */

defproc Mobile(talk,switch)=
        select {
                talk?().Mobile(talk, switch);
```

```
                switch?(talkn, switchn).Mobile(talkn,switchn)
        };

defproc BaseStation(talk, switch, give, alert) =
  select {
    talk!().BaseStation(talk, switch, give, alert);
    give?(talkn, switchn).switch!(talkn, switchn).
      BaseStationIdle(talk,switch, give, alert)
    }
and
    BaseStationIdle(talk, switch, give, alert) =
    alert?().BaseStation(talk, switch, give, alert);

defproc Central1(talk1, talk2,
                switch1, switch2,
                give1, give2,
                alert1, alert2) =
        give1!(talk2, switch2).alert2!().
        Central2(talk1, talk2,
                switch1, switch2,
                give1, give2,
                alert1, alert2)
and
        Central2(talk1, talk2,
                switch1, switch2,
                give1, give2,
                alert1, alert2) =
        give2!(talk1, switch1).alert1!().
        Central1(talk1, talk2,
                switch1, switch2,
                give1, give2,
                alert1, alert2);

/* --- */

defproc System =  (new talk1, talk2,
                      switch1, switch2,
                      give1, give2,
                      alert1, alert2
                  in (
                      Mobile(talk1, switch1) |
                      BaseStation(talk1,switch1,give1,alert1) |
                      BaseStationIdle(talk2,switch2,give2,alert2) |
                      Central1(talk1, talk2,
```

```
                              switch1, switch2,
                              give1, give2,
                              alert1, alert2)
                    ));

/* PROPERTIES */

defprop deadLockFree = maxfix X. (<>true and []X);

check System |= deadLockFree;

/* ---------- */

defprop write(x) = (1 and <x!>true);

defprop read(x) = (1 and <x?>true);

defprop hasRace =
inside (exists x.( write(x) | write(x) | read(x) | true));

defprop raceFree = maxfix X.((not hasRace) and []X);

check System |= raceFree;
```

Arrow Distributed directory protocol [8]

```
/* SYSTEM */

defproc
    TerminalOwner(find,move,obj) =
        find?(mymove,myfind).Owner(find,move,myfind,mymove,obj)
and
    Owner(find,move,link,queue,obj) =
        new iask in ( iask!() |
        select {
          find?(mymove,myfind).iask?().
                (Owner(find,move,myfind,queue,obj)
                | link!(mymove,find));
          iask?().(Idle(find,move,link) | queue!(obj))
        })
and
    Idle(find,move,link) =
        new iask in ( iask!() |
        select {
```

```
          find?(mymove,myfind).iask?().(Idle(find,move,myfind)
                                       | link!(mymove,find));
          iask?().(TerminalWaiter(find,move) | link!(move,find))
        })
and
    TerminalWaiter(find,move) =
        select {
        find?(mymove,myfind).Waiter(find,move,myfind,mymove);
        move?(obj).TerminalOwner(find,move,obj)
      }
and
    Waiter(find,move,link,queue) =
        select {
          find?(mymove,myfind).(Waiter(find,move,myfind,queue)
                               | link!(mymove,find));
          move?(obj).Owner(find,move,link,queue,obj)
    }
;

/* --- */

defproc Dir =
   new find1,move1,find2,move2,find3,move3,obj in
     ( obj!() |
                TerminalOwner(find1,move1,obj) |
                Idle(find2,move2,find1) |
                Idle(find3,move3,find2));

/* PROPERTIES */

defprop deadlockfree = always(<>true);

check Dir |= deadlockfree;

/* ---------- */

defprop object(s) = <s!>0;

defprop node(f) = 1 and (<> fresh a. fresh b. <f?(a,b)>true);

defprop owns(i,obj) = (node(i) and @obj);

defprop exclusive(i,obj) = (owns(i,obj) | not @obj);
```

```
defprop live = hidden obj.
        inside (object(obj) | forall i. ((node(i) | true) =>
                eventually exclusive(i,obj)));

check Dir |= always(live);
```

Acknowledgements Work on the Spatial Logic Model Checker is funded by the EU project FET IST-2001-33100 Profundis.

References

[1] L. Caires. *A Model for Declarative Programming and Specification with Concurrency and Mobility*. PhD thesis, Dept. de Informática, FCT, Universidade Nova de Lisboa, 1999.

[2] L. Caires. Behavioral and spatial properties in a logic for the pi-calculus. In Igor Walukiwicz, editor, *Proc. of Foundations of Software Science and Computation Structures'2004*, Lecture Notes in Computer Science. Springer Verlag, 2004.

[3] L. Caires and L. Cardelli. A Spatial Logic for Concurrency (Part I). *Information and Computation*, 186(2):194–235, 2003.

[4] L. Caires and L. Cardelli. A Spatial Logic for Concurrency (Part II). *Theoretical Computer Science*, to appear.

[5] L. Cardelli, P. Gardner, and G. Ghelli. Manipulating Trees with Hidden Labels. In A. D. Gordon, editor, *Proceedings of the First International Conference on Foundations of Software Science and Computation Structures (FoSSaCS '03)*, Lecture Notes in Computer Science. Springer-Verlag, 2003.

[6] L. Cardelli and A. D. Gordon. Anytime, Anywhere. Modal Logics for Mobile Ambients. In *27th ACM Symp. on Principles of Programming Languages*, pages 365–377. ACM, 2000.

[7] L. Cardelli and A. D. Gordon. Logical Properties of Name Restriction. In S. Abramsky, editor, *Typed Lambda Calculi and Applications*, number 2044 in Lecture Notes in Computer Science. Springer-Verlag, 2001.

[8] M. J. Demmer and M. P. Herlihy. The Arrow Distributed Directory Protocol. In *Proceedings of the 12th International Symposium on Distributed Computing (DISC'98)*, volume 1499 of *LNCS*, 1998.

[9] S. Ishtiaq and P. O'Hearn. BI as an Assertion Language for Mutable Data Structures. In *28th ACM Symp. on Principles of Programming Languages*, 2001.

[10] R. Milner. *Communicating and Mobile Systems: the π-calculus*. Cambridge University Press, 1999.