Spatial Logic Model Checker User's Guide

version 1.15

Hugo Vieira and Luís Caires

Departamento de Informática, FCT/UNL

June 2009

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, 7, 14, 4, 6].

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 [7, 4, 5, 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 A Glimpse at Spatial Logics for Concurrency

In behavioral models of concurrency, a process is identified with its observable behavior, roughly, the sequence of actions it can perform in the course of time. Modalities of a purely behavioral logic support the specification of processes by allowing us to talk about their actions; logics of this kind [17, 10, 11] are extensions of those introduced by Mathew Hennessy and Robin Milner [15]. Besides the usual set of propositional connectives, HML offers action modalities allowing us to observe potentiality of interaction, essentially these are action-labeled modalities, where the labels are extracted from the canonical labeled transition system for the pi-calculus. We have thus the dual formulas $\langle\alpha\rangle\, A$ and $[\alpha]\, A$, whose satisfaction with relation to a process P is defined by

$$P \models \langle\alpha\rangle\, A \text{ if there is a process } P' \text{ such that } P \xrightarrow{\alpha} P' \text{ and } P' \models A$$

$$P \models [\alpha]\, A \text{ if for all processes } P' \text{ if } P \xrightarrow{\alpha} P' \text{ then } P' \models A$$

A crucial property of HML is that the logical equivalence it induces on processes coincides with bisimulation. Bisimulation is the standard notion of behavioral equivalence for

concurrent processes; depending on the particular modalities adopted by a given version of HML, one can capture various flavours of bisimulation [17]). Algorithms and tools for model-checking pi-calculus processes against several variants of HML have been already developed, for example, the Mobility Workbench, building on work by Bjorn Victor and Mads Dam.

Spatial logics offer an enhanced power of observation, when compared with purely behavioral logics, because they can distinguish between systems that differ on their distributed structure, although possibly not on their behavior.

Typically, a spatial logic adds to a given set of behavioral modalities a set of spatial operators, closely related to the static operators of the process calculus, as in [1]. For nominal process calculi, the static operators are the composition $P \mid Q$, its identity element $\mathbf{0}$ (denoting the empty system), and the name restriction $(\boldsymbol{\nu} n)P$. These process constructors give rise to the composition formula $A \mid B$, that holds of a process that can be separated into a process that satisfies formula A and a process that satisfies formula B, to the void formula $\mathbf{0}$, that holds of the void process, and to the hidden name quantifier $\mathsf{H}x.A$. Satisfaction for these connectives is defined thus:

$$\begin{array}{ll} P \models \mathbf{0} & \text{if } P \equiv \mathbf{0} \\ P \models A \mid B & \text{if there are } P' \text{ and } P'' \text{ such that} \\ & P \equiv P' \mid P'' \text{ and } P' \models A \text{ and } P'' \models B \\ P \models \mathsf{H}x.A & \text{if there is } P' \text{ and a fresh name } n \text{ such that} \\ & P \equiv (\boldsymbol{\nu} n)P' \text{ and } P' \models A\{x \leftarrow n\} \end{array}$$

The hidden name quantifier turns out to be definable by combining the revelation connective $n \circledR A$ [8] and the freshness quantifier $\mathsf{N}x.A$. Revelation allows us to quantify over restricted channel names, understood as local resources.

$$\begin{array}{ll} P \models \mathsf{N}x.A & \text{if there is a fresh name } n \text{ such that } P \models A\{x \leftarrow n\} \\ P \models n \circledR A & \text{if there is } P' \text{ such that } P \equiv (\boldsymbol{\nu} n)P' \text{ and } P' \models A \end{array}$$

Alternatively, a spatial logic can put a stronger emphasis on structure, and allow the observation of the behavior of processes in a more indirect way, using spatial adjuncts, and a minimal "next step" (corresponding to the $\langle \boldsymbol{\tau} \rangle A$ formula) or a "eventually" behavioral modality. The first proposal in this vein is the ambient logic of [7], also adopted in the π-calculus logic of [4, 3]. An advantage of this approach is its generality, moreover, it is easily adaptable to any process calculus whose operational semantics can be presented by a simple unlabeled reduction relation. Adjuncts are very expressive. For instance, composition adjunct $A \triangleright B$ supports an internal definition of validity, and makes it possible to express quite general context/system specifications. However, model-checking of logics with composition adjunct, and including either quantification over names [9] or revelation [13] turns out to be undecidable, even for the simplest process languages.

The logic supported by the Spatial Logic Model Checker is a π-calculus logic based on purely structural spatial and behavioral observations. By "purely structural" we mean observations that can be determined by inspection of the local structure of the processes; therefore the logic does not include adjuncts operators. As a consequence, we obtain decidability of model-checking on interesting and pragmatically useful classes of processes,

while preserving the ability to express context-dependent behavioral and spatial properties of distributed systems.

For the spatial fragment we consider the connectives of composition, void, and revelation. For the behavioral fragment we consider a few simple modalities. A behavioral modality is defined either from the label $\boldsymbol{\tau}$, that denotes an internal communication, or from one of the labels $n\langle m\rangle$ and $n(m)$, denoting respectively the action of sending name m on channel n, and the action of receiving name m on channel n. To this basic set of connectives, we then add propositional operators, first-order and freshness quantifiers, and recursive definitions, along the lines of [4].

To illustrate in an informal way the expressiveness of the logic, we run through a sequence of examples. First, we show that by combining the fresh and hidden name quantifiers with the behavioral operators we can define modalities for name extrusion and intrusion (*cf.*,[17]).

$$\begin{array}{lcll} n\langle\boldsymbol{\nu} x\rangle.A & \triangleq & \mathsf{H}x.n\langle x\rangle.A & \text{(Bound Output)} \\ n(\boldsymbol{\nu} x).A & \triangleq & \mathsf{N}x.n(x).A & \text{(Bound Input)} \end{array}$$

The definition of bound output uses the hidden name quantifier [1, 4]. The hidden name quantifier is derived ([8]) from the fresh name quantifier and the revelation operator: $\mathsf{H}x.A \triangleq \mathsf{N}x.x\circledR A$. Using these two operators we can define the formula *Comm* below:

$$\begin{array}{lcl} \mathit{Comm} & \triangleq & m\langle\boldsymbol{\nu} x\rangle.A \mid m(\boldsymbol{\nu} x)B \Rightarrow \boldsymbol{\tau}.\mathsf{H}x.(A \mid B) \\ \mathit{Pair} & \triangleq & ((\boldsymbol{\nu} n)m\langle n\rangle.n\langle m\rangle.\mathbf{0}) \mid m(q).q\langle q\rangle.\mathbf{0} \end{array}$$

The formula *Comm* talks about name extrusion: it says that after interacting, two separate parts of the system can become "connected" by a shared secret. For example, the process *Pair* defined above satisfies the formula *Comm*. It also satisfies the formula $(\neg\mathbf{0} \mid \neg\mathbf{0}) \wedge \boldsymbol{\tau}.\neg(\neg\mathbf{0} \mid \neg\mathbf{0})$: this formula says that the process has two separate threads initially, that become tied by a private shared channel after a reduction step. This illustrates the fact that the logic has the power to count resources (*e.g.*, threads, restricted names). Combining spatial operators and recursive formulas we can define other useful operators, *e.g.*, $\mathsf{H}^*A \triangleq \boldsymbol{\mu} X.(A \vee \mathsf{H}x.X)$. The formula H^*A means that A holds under a (finite) number of restricted names [4]. For example, the formula

$$\neg\mathsf{H}^*\exists y.(\exists x.y(x).\mathbf{T} \mid \exists x.y(x).\mathbf{T})$$

expresses a unique handling property [16]: it says that there are no separate processes listening on the same (public or private) channel name. Using recursive formulas and behavioral modalities, we can also define a formula $\Box A$ stating that A will hold in all future states of the system, even when it interacts with an arbitrary environment (*e.g.*, an attacker). Then, the formula

$$\Box\neg\exists y.\mathsf{H}x.((P(x) \mid \mathbf{T}) \wedge y\langle x\rangle.\mathbf{T})$$

expresses a secrecy property: it asserts that it will never be the case that the system reaches a state where the identity x of a secret resource that satisfies $P(x)$ is about to be sent to the environment.

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

Figure 1: Syntax of Processes.

3 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!(`y_1`,`y_2`,...,`y_n`)`, and an input prefix $x(y_1, y_2, \ldots, y_n)$ is written `x?(`y_1`,`y_2`,...,`y_n`)`. The test operator allows to test name equality `[`*name* `=` *name*`]` and name inequality `[`*name* `!=` *name*`]`. We also consider the τ prefix that, as usual, stands for an internal invisible computation step. To note that in the `select` construct, which refers to the sum operator, all alternative branches are prefixed processes.

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 *CapsId*`(`*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.

4 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 ® [8]). 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 [4, 2]: `minfix` denotes the least fixpoint operator, and `maxfix` denotes the greatest fixpoint operator. We also consider parametric recursive properties.

Other built-in connectives 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` *CapsId.formula*
| `(minfix` *CapsId* `(`*namelist*`).`*formula*`)(`*namelist*`)`
| `maxfix` *CapsId.formula*
| `(maxfix` *CapsId* `(`*namelist*`).`*formula*`)(`*namelist*`)`
| *CapsId*
| *CapsId*`(`*namelist*`)`
| *k*
| `inside` *formula*
| `always` *formula*
| `eventually` *formula*
| *Id*`(`*namelist*`,`*formulalist*`)`

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

Figure 2: Syntax of Formulas.

5 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.

6 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` *CapsId*(*namelist*) = *process* [`and` *CapsId*(*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` *CapsId*`(`*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.

7 Examples

In this section we illustrate loadable specifications.

Gossiping System

```
/* GOSSIPING 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 */

check System |= 4 and (<> 3) and (<><> 2) and (<><><>1);

/***/

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;
```

This toy example considers a system that illustrates how name passing can influence the evolution of the structure of a system, and shows how one can specify simple properties using the spatial logic to observe this structural evolution.

We start off by defining the simpler kinds of behavior in our system. Firstly a Gossiper process that is always able to spread some information around, by means of taking this information as a name parameter and trying to emit it on public channel gossip, after which continuing with the dissemination intent considering the same piece of information.

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

Then we define the Listener process that is eager to learn some information, using a reception on channel gossip to that end, after which wanting to spread this received name around and hence becoming a gossiper.

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

Finally we define the System which is made of three Listeners and one Gossiper, considering a restricted name to be the piece of information to be disseminated by the Gossiper which initially is the only process to know this secret.

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

This system will clearly evolve at each step by a synchronization between a Listener and a Gossiper being this secret disclosured to one more Listener at a time. Since the secret is a restricted name each time that happens the Listener joins in the circle of trust of the secret, and the spatial bound created by the sharing of a restricted name makes the processes indivisible. Hence if at the starting point there where 4 distinct processes after one internal synchronization there are only 3, then 2 after two steps and finally 1 after three steps when all processes share the secret. We check to see if our system specification satisfies this description.

```
check System |= 4 and (<> 3) and (<><> 2) and (<><><>1);
```

We then specify concretely this name sharing property, starting by defining a formula that states that the argument property is true in every single point of the system.

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

We reuse this last property to define another that states that a given name is present everywhere.

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

After that we define a property that says that there is a restricted name which will be at some point in time known everywhere in the system. The hidden quantification reveals the restricted name, i.e. opens up the spatial bound induced by it, therefore allowing the observation of the inner distinct components that were bound together.

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

Finally we check our system specification to see if it satisfies this specification.

```
check System |= everybody_gets_to_know;
```

Going one step further we now intend to specify that our system will be, at some point in time, made exclusively out of Gossipers. We start by defining what is a gossiper and that amounts to say that it is a process that is always able to emit on channel gossip.

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

Then we specify systems that internally, i.e. underneath all restricted names, are made exclusively out of Gossipers.

```
defprop all_gossipers = eventually
                            inside everywhere(gossiper_forever);
```

Finally we check to see if our system satisfies this specification.

```
check System |= all_gossipers;
```

Token Ring System

```
/* TOKEN RING SYSTEM */

defproc Exit(inCh,outCh) =
        select {[outCh=inCh].0; outCh!(inCh).0};

defproc IdleNode(inCh,outCh) =
        inCh?(newInCh).TokenOwner(newInCh,outCh)

and TokenOwner(inCh,outCh) =
        select {
             tau.Exit(inCh,outCh);
             outCh!(outCh).IdleNode(inCh,outCh)};

defproc System =
        (new l1,l2,l3,l4,l5 in
             (IdleNode(l1,l2) |
              IdleNode(l2,l3) |
              IdleNode(l3,l4) |
              IdleNode(l4,l5) |
              TokenOwner(l5,l1)));

/* PROPERTIES */

check System |= always eventually 0;

/***/

defprop exiting(inl,outl) =
        1 and (((inl != outl) or <>0) and < outl!(inl) > 0);
```

```
defprop node(inl,outl) =
        1 and
        (exiting(inl,outl) or
        (maxfix X(inLnk).
                ((< inLnk?(newInLnk) > X(newInLnk))
                or ((<> exiting(inLnk,outl))
                   and (< outl!(outl) > X(inLnk)))))
        (inl));

defprop ring =
        0 or
        (hidden lnk.
        (minfix Y(x).
                (node(x,lnk) or
                (hidden y. (node(x,y) | Y(y)))))
        (lnk));

check System |= always ring;
```

This simple example shows a system holding a set of nodes that communicate in a token ring fashion, hence they are connected circularly and they are either idle and waiting for the token or active while holding the token. Being active in this case just means that they are able to exit the system or pass along the token.

The structure of the system is determined by the names that are shared between the nodes, being these processes responsible for updating the links when they intend to leave the system. Each node receives information in one channel - the input channel - and emits in another - the output channel. Getting the token is represented by the reception while emitting stands for passing the token to the next node.

We start off by defining the exit procedure for the nodes. There are two possible alternate behaviors: if the output channel is the same as the input channel that means that this is the only node in the ring and therefore there is nothing left to be done; the other possibility occurs when there is a node waiting for the token, so there is a possible communication. The emitted information consists of the entry link of the node that is exiting so that the next node can update it's entry point, something like a short circuit in the links that existed for the exiting node.

```
defproc Exit(inCh,outCh) =
        select {[outCh=inCh].0; outCh!(inCh).0};
```

We now define the behavior of the nodes starting by the IdleNode that consists in a process waiting on a reception on it's input channel, being the received name considered as the new input channel. After the reception the process symbolically holds the token and hence becomes a TokenOwner. This process can either make an internal silent action and then exit the system or just pass the token along, maintaining the linkage information, after which losing the token and becoming an IdleNode.

```
defproc IdleNode(inCh,outCh) =
        inCh?(newInCh).TokenOwner(newInCh,outCh)

and TokenOwner(inCh,outCh) =
        select {
             tau.Exit(inCh,outCh);
             outCh!(outCh).IdleNode(inCh,outCh)};
```

Finally we define the system consisting in a set of five nodes, four idle and obviously one holding the token, linked in a circular fashion by restricted names known only to the nodes that hold them as links.

```
defproc System =
        (new l1,l2,l3,l4,l5 in
             (IdleNode(l1,l2) |
              IdleNode(l2,l3) |
              IdleNode(l3,l4) |
              IdleNode(l4,l5) |
              TokenOwner(l5,l1)));
```

On the verification side, we start off by checking to see if our system, for every configuration, will at some point become the empty system, something along the lines of is it always possible for the system to terminate.

```
check System |= always eventually 0;
```

We now intend to verify that the set of nodes in our system are always connected circularly. We start off by characterizing the behavior of a node that is exiting, something that can be precisely expressed by the fact that it is a single point in space, and it holds either two different names or it can perform an internal action and become the empty system, and it is willing to emit it's input link on it's output link and become the empty system.

```
defprop exiting(inl,outl) =
        1 and (((inl != outl) or <>0) and < outl!(inl) > 0);
```

We now characterize a node by saying that it is a single point in space and it is either an exiting process or it has the possible infinite behavior of receiving a name and continuing considering this new name as the new input channel, which captures the behavior of the idle node, or of being able to perform an internal action after which exiting and an output in the output link carrying the output channel name and continuing with the same input link, which describes the active node.

```
defprop node(inl,outl) =
        1 and
        (exiting(inl,outl) or
        (maxfix X(inLnk).
```

```
                ((< inLnk?(newInLnk) > X(newInLnk))
                or ((<> exiting(inLnk,outl))
                   and (< outl!(outl) > X(inLnk)))))
        (inl));
```

Finally we define the circularity property by stating that it is either the empty system or it consists of only one node that has it's output connected to it's input, or there spatially co-exist a node and a chain of nodes with one ore more elements, being the node connected to the chain which in turn leads back to the initial node.

```
defprop ring =
        0 or
        (hidden lnk.
        (minfix Y(x).
                (node(x,lnk) or
                (hidden y. (node(x,y) | Y(y)))))
        (lnk));
```

We then check to see if the system holds this property in all possible configurations.

```
check System |= always ring;
```

Handover protocol (from Milner's book [16])

```
/* HANDOVER PROTOCOL */

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
    Central(talk, talkNxt,
```

```
            switch, switchNxt,
            give, giveNxt,
            alert, alertNxt) =
        give!(talkNxt, switchNxt).alertNxt!().Central(talkNxt, talk,
                                                      switchNxt, switch,
                                                      giveNxt, give,
                                                      alertNxt, alert);

/* --- */

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) |
                        Central(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;
```

This example simulates the interaction going on between cell phones, the stations that directly communicate to the cell phones and the centrals that regulate which station is

to be used for the communications. The general idea is to represent cell phone mobility, considering the shifting connectivity to the stations.

Starting by the definition of the behavior of a cell phone, it consists in either talking and continuing with the initial communication links or switching to new communication links, which represents a change of the station used for communication.

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

Now it is possible to define the station's behavior simply by saying that it can either communicate to the cell phone or pass away the connectivity to another base station, by means of a communication with the central, and become idle after which waiting for the next cell phone assignment coming from the central.

```
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);
```

Finally we have the central whose job is simply to inform a base station that it has to give the connectivity to other station after which alerting this newly appointed station that it has to operate this connection, alternating the roles of the stations in the next step.

```
defproc
    Central(talk, talkNxt,
            switch, switchNxt,
            give, giveNxt,
            alert, alertNxt) =
        give!(talkNxt, switchNxt).
              alertNxt!().Central(talkNxt, talk,
                                  switchNxt, switch,
                                  giveNxt, give,
                                  alertNxt, alert);
```

To finish off the process definitions a very simple system is defined having just one cell phone, the two alternating base stations and a central.

```
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) |
                      Central(talk1, talk2,
                              switch1, switch2,
                              give1, give2,
                              alert1, alert2)
                    ));
```

We now define a standard behavioral property that expresses that the system is never deadlocked, i.e., has a possible reduction in every possible configuration.

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

And we check to see if the system satisfies this property.

```
check System |= deadLockFree;
```

We now intend to verify that our system is race free, or in other words, does not have two processes trying to write on a channel while another one is trying to read on it. We start by saying that a writer is a single point in space and is willing to do an output in a determined channel. Analogously for the reader considering a read instead of a write.

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

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

We now state that the existence of a race can be specified by looking underneath all restrictions and finding a channel name such that the system can be divided in three our more parts, being two of these parts writers on that channel and a third part a reader that uses that channel.

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

Finally we state that race freeness is not having a race in any possible configuration.

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

We check to see if our system satisfies this property.

```
check System |= raceFree;
```

Arrow Distributed directory protocol [12]

```
/* THE ARROW DISTRIBUTED DIRECTORY PROTOCOL */

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

This interesting example describes a protocol used in systems that share a unique resource, like for instance a token that gives privileges to it's owner. Working on a determined connectivity structure, a minimal spanning tree, the nodes in the system evolve by requesting, waiting, getting and releasing the object. The connectivity structure evolves along with the nodes in such a way that the links give the direction to either the object or the nodes that are waiting for it. Since there can be several nodes requesting the object in parallel a waiting queue can be established, distributed throughout the system.

First off the nodes behavior is defined: A TerminalOwner that waits for requests for the object, being that no request has arrived to it; an Owner that either accepts requests for the object or releases it to the head of the waiting queue; An Idle that passes along requests for the object or makes it's own request for it; A TerminalWaiter that either passes along requests for the object or receives it and a Waiter that has the same choice, differing only on the necessary update of the connectivity links.

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

```
and
    Idle(find,move,link) =
        select {
            find?(mymove,myfind).(Idle(find,move,myfind) |
                                  link!(mymove,find));
            tau.(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)
        };
```

A very simple system is then defined, considering only three nodes.

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

We then check to see if the system has always a reduction in every possible configuration.

```
defprop deadlockfree = always(<>true);

check Dir |= deadlockfree;
```

We now intend to verify that the object will be possibly acquired, in exclusive mode, by any node. First off we specify that an object has as behavior a possible emission on a channel after which it becomes the empty process. We also define a node as being a single point in space that is able to receive two names in a channel, since all nodes are able to accept requests for the object.

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

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

We then state that for a node to own the object it just has to contain an occurrence of the object's name, and to own the object exclusively the node has to own it while the rest of the system has no occurrences of the object's name.

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

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

Finally we specify our liveness property by stating that all nodes in the system can come to acquire exclusive access to the object at some point in time.

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

We check to see if the system satisfies this property for every possible configuration.

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

Acknowledgements All ongoing work on the Spatial Logic Model Checker has been funded by IP Sensoria, and POSI/EIA/55582/2004 Space-Time Types, and 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 II). In *CONCUR 2002 (13th International Conference)*, Lecture Notes in Computer Science. Springer-Verlag, 2002.

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

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

[6] 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.

[7] 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.

[8] 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.

[9] W. Charatonik and J.-M. Talbot. The decidability of model-checking mobile ambients. In D. Metayer, editor, *11th European Symposium on Programming (ESOP 2002)*, number 2305 in Lecture Notes in Computer Science. Springer-Verlag, 2001.

[10] M. Dam. Model checking mobile processes. *Information and Computation*, 129(1):35–51, 1996.

[11] M. Dam. Proof systems for π-calculus logics. In de Queiroz, editor, *Logic for Concurrency and Synchronisation*, Studies in Logic and Computation. Oxford University Press, To appear.

[12] 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.

[13] G. Ghelli and G. Conforti. Decidability of freshness, undecidability of revelation. Technical Report 03–11, Dipartimento di Informatica, Universita di Pisa, 2003.

[14] 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.

[15] R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.

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

[17] R. Milner, J. Parrow, and D. Walker. Modal logics for mobile processes. *Theoretical Computer Science*, 114:149–171, 1993.