# SLMC: A Tool for Model Checking Concurrent Systems against Dynamical Spatial Logic Specifications

Luís Caires and Hugo Torres Vieira

CITI and Departamento de Informática, Faculdade de Ciências e Tecnologia,
Universidade Nova de Lisboa, 2829-516 Caparica, Portugal

**Abstract.** The Spatial Logic Model Checker is a tool for verifying $\pi$-calculus systems against safety, liveness, and structural properties expressed in the spatial logic for concurrency of Caires and Cardelli. Model-checking is one of the most widely used techniques to check temporal properties of software systems. However, when the analysis focuses on properties related to resource usage, localities, interference, mobility, or topology, it is crucial to reason about spatial properties and structural dynamics. The SLMC is the only currently available tool that supports the combined analysis of behavioral and spatial properties of systems. The implementation, written in OCAML, is mature and robust, available in open source, and outperforms other tools for verifying systems modeled in $\pi$-calculus.

## 1 Introduction

Model-checking is one of the most widely used verification techniques in the analysis of software applications. The usual focus is on behavioral/temporal properties, which allow to check liveness and safety properties of systems, from the standpoint of their externally observable behavior. However, it is often the case that verification really needs to address on properties about (spatial) distribution, mobility, or resource usage. It is then crucial to be able to observe the structural/spatial configuration of systems. Examples of such properties include connectivity – there is always an access route between two sites – or unique handling – there is at most one server listening on a given channel name – or race absence – no simultaneous sends/writes to the same receiver/reader.

The Spatial Logic Model Checker [3] is a tool that allows the user to automatically verify behavioral and spatial properties of distributed and concurrent systems expressed in the $\pi$-calculus. Its base logic is a very rich dynamical spatial logic for concurrency, conveniently containing as a subset the logics supported by other model-checkers for $\pi$-calculi (e.g., [15, 11, 18]). The verification algorithm (using on-the-fly techniques) is provably correct for all expressible processes, and complete for the class of bounded processes [4], including the finite control $\pi$-calculus. In the next section, we present the SLMC by going through a simple example, which already illustrates the usefulness of the tool, briefly presenting the input languages in the meanwhile (see, e.g., [12, 14] and [4–7] for background on $\pi$-calculus and on dynamic spatial logics, respectively).

## 2 Checking a Topological Property of a Distributed Protocol

The example we now discuss models a protocol which allows a set of nodes to organize itself into a ring like structure. The basic idea of the protocol is that in each step

two rings (which includes the case of the singleton ring, i.e., a ring with one node) are merged into a larger ring. Then, regardless of the intermediate configurations, a sequence of such steps leads to the point in which the whole set of nodes is included in the ring. The correctness of such protocol may be verified by our tool, since we are able to observe the topology of the system and check if the protocol yields a ring configuration.

We start by the specification of the three possible states of a node in our system: state *Node* represents the initial state of a node, which has no connections; state *Link* represents a node which is in a ring, hence connected to its left and to its right (we use left and right for the sake of illustration); state *Leader* also represents a node which is in a ring, but is the only node in its ring that is willing to connect to other nodes.

The specification of the *Node* in SLMC syntax is as follows:

```
defproc Node(com) =
   new link,chan in select {
         com!(link,chan).chan?(right).Leader(com,link,right);
         com?(right,newch).newch!(link).Link(link,right)};
```

The `defproc` introduces a $\pi$-calculus process definition in the system, named `Node`. The parameter `com` is the name of a public channel used by nodes to connect to each other. The process specified creates names `link` and `chan` (cf., $\pi$-calculus name restriction) and then may `select` one of two possible behaviors: either it outputs on channel `com` the freshly created names `link` and `chan` or it receives some names `right` and `newch` in channel `com`. In the former case, the process proceeds by receiving `right` in `chan`, after which becomes the `Leader` of the ring. In the latter case, the process proceeds by sending `link` in the received `newch`, and then becomes a `Link` node.

We then specify a `Link` node as a process that either inputs from the node on its left or outputs to the node on its right, and after which proceeds as a `Link` node:

```
defproc Link(left,right) =
   select { left?().Link(left,right);
            right!().Link(left,right)};
```

Like the `Link`, a `Leader` also receives from its left node and outputs to its right:

```
defproc Leader(com,left,right) =
   new chan in select {
       left?().Leader(com,left,right);
       right!().Leader(com,left,right);
       com?(newr,newch).newch!(right).Link(left,newr);
       com!(right,chan).chan?(newr).Leader(com,left,newr)};
```

Furthermore, a `Leader` node is willing to connect to another ring via channel `com` (and a freshly created `chan`). Intuitively, two `Leader`s connect by swapping their right links, in such way merging two rings into one. This is the case both when the `Leader` receives or outputs on `com`, the difference is that the former implies yielding the `Leader` status (proceeding as `Link`), while the latter does not (proceeding as `Leader`).

The system is specified as a set of (e.g., four) `Node`s that share a public `com` channel:

```
defproc System = Node(com) | Node(com) | Node(com) | Node(com);
```

We may now present the spatial/behavioral properties that characterize the system. For starters, we describe a leader node:

```
defprop leader(a,b) =
  1 and (a != b) and (@com) and (<a?> true) and (<b!> true);
```

This `defprop` command defines property `leader` (with parameters `a,b`), which describes processes which are indivisible (`1`), that have `com` as a free name (`@com`) and that are able to input on a name (`<a?>` after which proceeding as processes that satisfy `true`, i.e., any) and output on another name (`<b!>` after which proceeding as any process). A link has a similar description, where `com` is not a free name:

```
defprop link(a,b) =
  1 and (a != b) and (not @com) and (<a?> true) and (<b!> true);
```

A node may be described as an indivisible process which is not a link nor a leader:

```
defprop node =
  1 and not exists a. exists b. (leader(a,b) or link(a,b));
```

Notice properties `link`, `leader` and `node` are specially suited for the node specification of this system in particular. However, testing for indivisibility (single-threaded) is a generic feature of a node, which is possible to observe thanks to the expressiveness of the logic. Property `1` may be taken as an abbreviation of "non-empty system which cannot be decomposed into two non-empty parts" – `not 0 and not (not 0 | not 0)`.

The separating composition `A|B` is a key operator of the dynamic spatial logic, characterizing systems that can be decomposed (via structural congruence) in two parts, one satisfying property `A` and the other satisfying property `B`. Using parallel composition, we may, e.g., specify the initial state of the system as a composition of four nodes:

```
defprop initial = inside ( node | node | node | node );
```

Property `inside` is used so as to *reveal* all name restrictions, i.e., open the scopes of all name restrictions, in such way allowing for spatial decomposition to split threads otherwise indivisible because of the sharing of some restricted name.

We now turn to the verification of the correctness of the protocol. In order to characterize rings, we first introduce the notion of a chain of connected link nodes:

```
defprop chain(c,d) = (minfix C(a,b).( link(a,b) or
                        (exists x. (link(a,x) | C(x,b)))))(c,d);
```

Intuitively, the least fixpoint (`minfix`), parameterized by `a,b` initially instantiated by `c,d`, characterizes a chain of linked nodes where the leftmost and rightmost links are `a` and `b`, respectively. Such chain may either be a single link node, or there `exists` (the existential quantifier) name `x` such that there is a `link(a,x)` in *parallel* with a chain from `x` to `b`. Then, a ring is a chain of `link`s in parallel with a `leader`:

```
defprop ring = exists a. exists b. (leader(a,b) | chain(b,a));
```

Notice the chain connects `b` to `a`, for some names `b,a`, which are the right and left link of the leader node, respectively. We may now ask the tool if all execution paths lead (`always` and `eventually`, defined as usual) to a ring configuration:

```
check System |= always (eventually (inside (ring)));
```

The success of this verification, which explores all possible execution paths of the system and exploits the unique combination of behavioral and spatial properties supported by the tool, guarantees the protocol always leads, regardless of intermediate steps, to a final configuration of a ring that connects all nodes.

## 3 Verification Algorithms and Implementation

In this section we discuss the verification algorithms, based on [4] and on a canonical representation of processes, and present some benchmark figures.

The model-checking procedure is based on an on-the-fly technique, which means the model state space is explored gradually, guided by the deconstruction of the formula. In our case, the verification comprises observing both structure and behavior of processes. Namely, model-checking relies on decompositions of processes – up to structural congruence – to check the composition formula $A \mid B$, on observing behaviors of processes – up to the labeled transition system which defines the operational semantics – to check action modalities, and, crucially to check fixpoints, on the ability to compare two processes – up to the identification of some *irrelevant* names for the purpose of the model-checking, i.e., names that are not referred by the formula [4]. A great deal of the reasoning performed by our algorithms is optimized by relying on equivariance [10] (working up to name permutations).

Processes are modeled by data structures representing sets of equations in a normal form. Each equation describes a ***flat*** state of the process, where only immediate actions are represented. Using $\mathcal{X}$ to range over equation identifiers and $\alpha$ to range over actions we then write $\mathcal{X}(\overline{x}) \mapsto (\boldsymbol{\nu}\overline{a})\ (\alpha_1 \mid \ldots \mid \alpha_k)$ to represent the equation identified by variable $\mathcal{X}$, parameterized by the $\overline{x}$ variable set ($\overline{x}$ abbreviates $x_1, \ldots, x_j$), specifying a flat configuration with restricted names $\overline{a}$ and consists of the composition of $k$ actions $\alpha_1$ to $\alpha_k$. Action prefixes, denoted by $p$, and actions, denoted by $\alpha$, are given by:

$$\begin{array}{rcllll} p & ::= & n!(\overline{m}) & \text{(Output)} & \mid\ n?(\overline{x}) \ \text{(Input)} & \qquad \alpha ::= \ \alpha + \alpha \quad \text{(Sum)} \\ & \mid & [n = m] & \text{(Test)} & \mid\ \tau \qquad \text{(Internal)} & \qquad \mid\ \ p.\,\mathcal{X}(\overline{n}) \ \text{(Prefix)} \end{array}$$

An action prefix may be an output $n!(\overline{m})$ (read "send names $\overline{m}$ on channel $n$"), an input $n?(\overline{x})$ (read "instantiate variables $\overline{x}$ with the names received on channel $n$"), a test $[n = m]$ (read "if $n$ is the same as $m$ proceed") and $\tau$ which represents a process internal action. Actions are either the non-deterministic choice of two actions $\alpha + \alpha$, or a prefix and its respective continuation. Continuations in our setting are specified by the corresponding equation variable, hence $p.\,\mathcal{X}(\overline{n})$ represents a process which after $p$ behaves as specified in the equation identified by $\mathcal{X}$, instantiating its parameters with $\overline{n}$.

Abstracting continuation states with equation variables is crucial to quickly verify if two processes are the same, since we only need to check if the immediate actions and their respective continuations are the same. This simplification is crucial for the verification of fixpoints, which rely on an approximation of the fixpoint which is updated and consulted throughout unfolding, via the process comparison mechanism.

A process model may then be represented by a set of equations, together with an entry point. To further optimize the verification we model active (top-level) processes considering the set of *connected components* [9]: two processes that share a restricted name and thus cannot be decomposed via $\pi$-calculus structural congruence. Hence, for the sake of verifying the composition formula, which involves exploring all possibilities for decomposing a process, it is vital that the representation clearly identifies which threads are not decomposable, identifying the basic units of decomposition. The SLMC top-level process representation is then given by:

$$(\boldsymbol{\nu}\overline{a}_1)\ (\alpha_1^1 \mid \ldots \mid \alpha_k^1) \mid \ldots \mid (\boldsymbol{\nu}\overline{a}_j)\ (\alpha_1^j \mid \ldots \mid \alpha_k^j)$$

| | SLMC | Petruchio | MWB (`prove`) | MWB (`check`) | MMC |
|---|---|---|---|---|---|
| Handover | 0.0005 | 0.2 | 0.002 | 0.015 | 0.01 |
| Arrow (a) | 0.01 | 0.8 | 0.115 | – | – |
| Arrow (b) | 0.3 | 4.3 | 6.2 | – | – |

**Table 1.** Model-Checking Deadlock Absence (in seconds).

| Ring | Handover | Arrow (a) | Arrow (b) |
|---|---|---|---|
| 0.08 | 0.02 | 0.11 | 7.76 |

**Table 2.** Model-Checking Spatial properties (in seconds).

where each $(\boldsymbol{\nu}\overline{a}_i)$ $(\alpha_1^i \mid \ldots \mid \alpha_k^i)$ piece (for some $i$) is an indivisible process because the $\alpha_t^i$ actions share between them the $\overline{a}_i$ restricted names. This way, the several possibilities for decomposing a process in two pieces are obtained by the possible combinations of gathering these basic indivisible blocks.

At the level of the optimizations, the main challenge we address is the expedite (re-)building of the process normal form, i.e., updating the top-level process representation as the consequence of observing an action/transition. For example, observing an output action entails updating the top-level process with the continuation of the output which then becomes active. So, actions and name restrictions specified in the continuation configuration (given by the equation identified by the variable prefixed by the output) must be integrated in the top-level representation, and the set of connected components must be updated considering the "new" actions and due to restricted names scope changes.

We now present some benchmark figures, comparing with other existing $\pi$-calculus model-checkers: the Petruchio tool [11], the Mobility Workbench (MWB) [15], and the Mobility Model Checker (MMC) [18]. The comparison is established for the verification of a fundamental behavioral property: *deadlock absence*. We consider two challenging systems: Milner's implementation of the Handover protocol [12] and a $\pi$-calculus implementation of the Arrow Distributed Directory Protocol [8], both available in the SLMC homepage [3].

The numbers shown in Table 1 list the amount of time needed for each tool to verify the systems are deadlock free, obtained running the tools on a Mac OS X 10.5.8, 2.4GHz Intel Core 2 Duo. For the Petruchio tool in particular, the figures indicate the time needed to translate $\pi$-calculus specifications into petri-nets [13], since Petruchio exports such petri-nets to external verification engines to carry out the model-checking. In the case of the MWB, we distinguish between the `prove` and `check` procedures. We consider the arrow system as available in [3] – (a) – and also a small variation obtained by adding one node to the system – (b). Notice that both the `check` procedure available in the MWB and the MMC did not provide results for checking deadlock absence for the arrow system, due to timeout and memory overflow, respectively. The comparison with Petruchio and the MWB `prove` procedure is favorable to the SLMC, where the figures obtained hint on the complexity of the arrow system itself.

Notice however none of the tools mentioned above supports the verification of structural properties, and there exists none, to the best of our knowledge, which allows for the combined analysis of behavioral and spatial specifications as the ones expressible in dynamic spatial logic. Table 2 reports on figures obtained for verifying spatial properties over the same systems, namely the verification shown in the previous section in the Ring system, race-freedom in the Handover system, and a complex correctness property of the Arrow system (see [3]): it is always the case that every node may eventually gain

exclusive access to the shared object. As stated above, no comparison is possible in this case, since no other tool can handle spatial properties as the SLMC does.

## 4 Concluding Remarks

The SLMC is publicly available online, in open source, and is often downloaded. The tool, in development since 2004, has reached a very high maturity and robustness level, and is very fast in practical use. It has been routinely used for teaching purposes in our department, and we would like to further promote its use elsewhere. The development of the first version of the tool was supported by the FET Profundis project [1]. The development continued under the support of project IP Sensoria [2], where the tool was included in the Sensoria tool suite, and extended for the verification of service-oriented systems, as described in [17]. In particular, we have also concluded recently further extensions to the tool, namely an extension to the applied $\pi$-calculus (for security), and another for checking choreography conformance of service-oriented applications, based on an encoding of the Conversation Calculus [16]. Further information about the Spatial Logic Model Checker may be found in `http://ctp.di.fct.unl.pt/SLMC/`.

## References


1. FET Profundis Project. `http://www.it.uu.se/profundis/`.
2. IP Sensoria Project. `http://www.sensoria-ist.eu/`.
3. Spatial Logic Model Checker. `http://ctp.di.fct.unl.pt/SLMC/`.
4. L. Caires. Behavioral and Spatial Observations in a Logic for the $\pi$-Calculus. In *FOSSACS 2004, Proceedings*, volume 2987 of *LNCS*, pages 72–89. Springer, 2004.
5. L. Caires. Dynamical Spatial logics: A Tutorial Survey. In *Bulletin of the EATCS*, 2008.
6. L. Caires and L. Cardelli. A Spatial Logic for Concurrency (Part I). *Information and Computation*, 186(2):194–235, 2003.
7. L. Cardelli and A. Gordon. Anytime, Anywhere: Modal Logics for Mobile Ambients. In *POPL 2000, Proceedings*, pages 365–377. ACM Press, 2000.
8. M. Demmer and M. Herlihy. The Arrow Distributed Directory Protocol. In *DISC 1998, Proceedings*, volume 1499 of *LNCS*, pages 119–133. Springer, 1998.
9. J. Engelfriet and T. Gelsema. Multisets and Structural Congruence of the $\pi$-Calculus with Replication. *Theor. Comput. Sci.*, 211(1-2):311–337, 1999.
10. M. Gabbay and A. Pitts. A New Approach to Abstract Syntax with Variable Binding. *Formal Aspects of Computing*, 13(3–5):341–363, 2002.
11. R. Meyer and T. Strazny. Petruchio: From Dynamic Networks to Nets. In *CAV 2010, Proceedings*, volume 6174 of *LNCS*, pages 175–179. Springer, 2010.
12. R. Milner. *Communicating and Mobile Systems: the $\pi$-Calculus*. CUP, 1999.
13. C. Petri and W. Reisig. Petri net. *Scholarpedia*, 3(4):6477, 2008.
14. D. Sangiorgi and D. Walker. *The $\pi$-Calculus: A Theory of Mobile Processes*. CUP, 2001.
15. B. Victor and F. Moller. The Mobility Workbench - A Tool for the $\pi$-Calculus. In *CAV 1994, Proceedings*, volume 818 of *LNCS*, pages 428–440. Springer, 1994.
16. H. Vieira, L. Caires, and J. Seco. The Conversation Calculus: A Model of Service-Oriented Computation. In *ESOP 2008*, volume 4960 of *LNCS*, pages 269–283. Springer, 2008.
17. M. Wirsing and M. Hölzl, editors. *Rigorous Software Engineering for Service-Oriented Systems*. Springer-Verlag, 2011.
18. P. Yang, C. Ramakrishnan, and S. Smolka. A Logical Encoding of the $\pi$-Calculus: Model-Checking Mobile Processes Using Tabled Resolution. *STTT*, 6(1):38–66, 2004.