Lazy Reachability Analysis in Distributed Systems

Loïg Jezequel, Didier Lime

Lazy Reachability Analysis in Distributed Systems

Loïg Jezequel[1] **and Didier Lime**[2]

1 **Université de Nantes, IRCCyN UMR CNRS 6597, France**
Loig.Jezequel@irccyn.ec-nantes.fr
2 **École Centrale de Nantes, IRCCyN UMR CNRS 6597, France**
Didier.Lime@ec-nantes.fr

Abstract

We address the problem of reachability in distributed systems, modelled as networks of finite automata and propose and prove a new algorithm to solve it efficiently in many cases. This algorithm allows to decompose the reachability objective among the components, and proceeds by constructing partial products by lazily adding new components when required. It thus constructs more and more precise over-approximations of the complete product. This permits early termination in many cases, in particular when the objective is not reachable, which often is an unfavorable case in reachability analysis. We have implemented this algorithm in a first prototype and provide some very encouraging experimental results.

1998 ACM Subject Classification D.2.4 Software/Program Verification

Keywords and phrases Reachability analysis, compositional verification, automata

Digital Object Identifier 10.4230/LIPIcs.CONCUR.2016.17

1 Introduction

As distributed systems become more and more pervasive, the need for the verification of their correctness increases accordingly. The problem of verifying that some particular state of the system is reachable is a cornerstone in this endeavour. Yet even this simple problem is challenging in a distributed context, due to the exponential growth of the state-space of the system with the number of components, a problem often referred to as "state explosion".

To alleviate this issue several approaches have been proposed in the last two decades. In particular, partial order techniques, which allow to explore only part of the state-space while preserving completeness, have proved to be an efficient approach [12] and are implemented in some state-of-the-art tools, including LoLA [15]. Another technique called partial model-checking has been proposed in [1], which consists in incrementally quotienting temporal logic formulas by incorporating the behaviours of individual processes into that formula. This leads to a compositional verification scheme that has been recently extended and implemented in the PMC tool on top of the CADP toolbox [14]. This idea to incrementally take into account components of distributed systems is also present in works on compositional minimization [9, 6] and modular model checking [10, 7]. Recently, the IC3 algorithm [3] has been proposed to address the safety / reachability issue, with very promising results. This algorithm incrementally generates more and more precise over-approximations of the reachability relation, by computing stronger and stronger inductive assertions using SAT solving [3] or SMT solving [4]. Similar ideas of incremental refinements were also successfully used in AI planning [2, 11].

A common high-level scheme in the partial model-checking approach, the IC3 approach, and the hierarchical planning approach is to incrementally compute more and more precise approximate objects until sufficient precision permits to conclude. We propose a new approach

also based on this scheme. Contrarily to IC3, we deliberately use the classical tools of explicit state space exploration in finite automata-based models, with the motivation of ultimately combining our technique with some of the most successful improvements of these tools, like partial-order reduction techniques, and with well-known extensions to more expressive models like timed automata. In contrast to the partial model-checking approach, we take advantage of the simplicity of the reachability property we study, and adopt a lower-level approach, by focusing on the individual components of the system and their fine interactions.

Our contribution in this paper is therefore the following: first an algorithm, which projects a reachability property on the individual components, then lazily adds components to the projections and merges them together as interactions that are meaningful to the reachability property are discovered. We provide full proofs for completeness, soundness, and termination. Second, we report on a prototype implementation that permits a first evaluation of our algorithm. We have compared these performances with LoLA.

The paper is organized as follows: in Section 2 we give the basic definitions upon which our algorithm is built. In Section 3, we describe our algorithm and prove it. In Section 4 we report on experimental results, and we conclude in Section 5.

2 Preliminary definitions

2.1 LTSs and their products

We focus on labelled transition systems, synchronized on common actions, as models.

▶ **Definition 1.** A *labelled transition system* (LTS) is a tuple $\mathcal{L} = (S, \iota, T, \Sigma, \lambda)$ where S is a non-empty finite set of states, $\iota \in S$ is an initial state, Σ is an alphabet of transition labels, $T \subseteq S \times S$ is a finite set of transitions, and $\lambda : T \to \Sigma$ is a transition labeling function.

By a slight abuse of notations, we also denote by $\Sigma(\mathcal{L})$ the set of labels of $\mathcal{L}$ and by $\lambda(\mathcal{L})$ the set of labels effectively associated to at least one transition in $\mathcal{L}$.

▶ **Definition 2.** In such an LTS, a *path* is a sequence of transitions $\pi = t^1 \dots t^n$ such that: $\forall 1 \leq k \leq n, t^k = (s^k, s^{k+1}) \in T$ and $s^1 = \iota$. In this case we say that π *reaches* s^{n+1}. A state s is said to be *reachable* if there exists a path that reaches s.

▶ **Definition 3.** We say that two LTSs $\mathcal{L}_1 = (S_1, \iota_1, T_1, \Sigma_1, \lambda_1)$ and $\mathcal{L}_2 = (S_2, \iota_2, T_2, \Sigma_2, \lambda_2)$ are *isomorphic* if and only if there exists two bijections $f_S : S_1 \to S_2$ and $f_T : T_1 \to T_2$ so that: $f_S(\iota_1) = \iota_2$, and $\forall s_1, s_1' \in S_1, (s_1, s_1') \in T_1$ iff $(f_S(s_1), f_S(s_1')) \in T_2$, and $\lambda_1((s_1, s_1')) = \lambda_2((f_S(s_1), f_S(s_1')))$.

Our systems are built as parallel compositions of multiple LTSs.

▶ **Definition 4.** Let $\mathcal{L}_1, \dots, \mathcal{L}_n$ be LTSs such that $\forall 1 \leq i \leq n, \mathcal{L}_i = (S_i, \iota_i, T_i, \Sigma_i, \lambda_i)$. The *compound system* $\mathcal{L}_1 \parallel \dots \parallel \mathcal{L}_n$ is the LTS $(S, \iota, T, \Sigma, \lambda)$ such that $S = S_1 \times \dots \times S_n$, $\iota = (\iota_1, \dots, \iota_n)$, $\Sigma = \Sigma_1 \cup \dots \cup \Sigma_n$, and $t = ((s_1, \dots, s_n), (s_1', \dots, s_n')) \in T$ with $\lambda(t) = \sigma$ if and only if $\forall 1 \leq i \leq n$ if $\sigma \in \Sigma_i$ then $t_i = (s_i, s_i') \in T_i$ and $\lambda_i(t_i) = \sigma$ else $s_i = s_i'$.

Remark that $(\mathcal{L}_1 \parallel \mathcal{L}_2) \parallel \mathcal{L}_3$, $\mathcal{L}_1 \parallel (\mathcal{L}_2 \parallel \mathcal{L}_3)$, and $\mathcal{L}_1 \parallel \mathcal{L}_2 \parallel \mathcal{L}_3$ are isomorphic (they are identical up to renaming of states and transitions). It is thus possible to compute compound systems step by step, by adding LTSs to the composition one after the other.

▶ **Definition 5.** $\mathcal{L}_{id} = (S_{id}, \iota_{id}, T_{id}, \Sigma_{id}, \lambda_{id})$ with $S_{id} = \{id\}$, $\iota_{id} = id$, $T_{id} = \emptyset$, $\Sigma_{id} = \emptyset$, and λ_{id} is the unique function from $\emptyset$ to $\emptyset$.

Remark that $\mathcal{L}_{id}$ can be considered as the neutral element for the composition of LTSs: $\forall \mathcal{L}, \mathcal{L} \parallel \mathcal{L}_{id}$ and $\mathcal{L}$ are isomorphic. Also remark that for any LTS $\mathcal{L}$, there is an LTS containing only the initial state of $\mathcal{L}$ which is isomorphic to $\mathcal{L}_{id}$. We denote it by $\mathsf{id}(\mathcal{L})$.

2.2 Partial products and reachability of partial states

We define a notion of extension of LTSs, using a partial order relation.

▶ **Definition 6.** An LTS $\mathcal{L} = (S, \iota, T, \Sigma, \lambda)$ *extends* an LTS $\mathcal{L}'$, noted $\mathcal{L}' \sqsubseteq \mathcal{L}$, if and only if $\mathcal{L}'$ is isomorphic to some LTS $(S', \iota', T', \Sigma', \lambda')$ with $S' \subseteq S$, $T' \subseteq T$, $\Sigma' \subseteq \Sigma$, $\iota = \iota'$, and $\lambda' = \lambda|_{T'}$. If, moreover, $S' \neq S$, or $T' \neq T$, or $\Sigma' \neq \Sigma$, $\mathcal{L}$ is said to *strictly extend* $\mathcal{L}'$, noted $\mathcal{L}' \sqsubset \mathcal{L}$.

We define $\mathsf{ini}(\mathcal{L})$ as the LTS containing only the initial state of $\mathcal{L}$ but, contrarily to $\mathsf{id}(\mathcal{L})$, with $\Sigma(\mathsf{ini}(\mathcal{L})) = \Sigma(\mathcal{L})$. Note that we clearly have $\mathsf{ini}(\mathcal{L}) \sqsubseteq \mathcal{L}$.

Given a set of LTSs, we now define partial products as products in which some parts of the LTSs, or possibly some LTSs altogether, are not used:

▶ **Definition 7.** An LTS $\mathcal{L}'$ is a *partial product* of a compound system $\mathcal{L} = \mathcal{L}_1 \parallel \cdots \parallel \mathcal{L}_n$ if there exists m LTSs $\mathcal{L}'_{k_1}, \ldots, \mathcal{L}'_{k_m}$ (with $\{k_j \ : \ j \in [1..m]\} \subseteq [1..n]$) such that $\mathcal{L}'$ is isomorphic to $\mathcal{L}'_{k_1} \parallel \cdots \parallel \mathcal{L}'_{k_m}$ and $\forall j \in [1..m], \mathcal{L}'_{k_j} \sqsubseteq \mathcal{L}_{k_j}$.

Note that in the algorithm we propose in Section 3, we will actually always have, by construction, $\mathsf{ini}(\mathcal{L}_{k_j}) \sqsubseteq \mathcal{L}'_{k_j} \sqsubseteq \mathcal{L}_{k_j}$, which implies that all three LTSs have the same alphabet.

We focus on solving particular reachability problems where one is interested in reaching partial states.

▶ **Definition 8.** In a compound system $\mathcal{L}_1 \parallel \ldots \parallel \mathcal{L}_n$ we call any element from $S_1 \times \cdots \times S_n$ a *global state*. A *partial state* is an element from $(S_1 \cup \{\star\}) \times \cdots \times (S_n \cup \{\star\}) \setminus \{(\star, \ldots, \star)\}$.

We say that a partial state $(s'_1, \ldots, s'_n)$ *concretises* a partial state $(s_1, \ldots, s_n)$ if $\forall i, s_i \neq \star$ implies $s'_i = s_i$.

A partial state is therefore in some sense the specification of the set of global states that concretise it, i.e., that share the same values on dimensions not equal to $\star$ in the partial state. We use partial states to specify our reachability objectives.

▶ **Definition 9.** In a compound system $\mathcal{L}$, a partial state is said *reachable*, if there exists a reachable global state that concretises it. Given a set $\mathcal{R}$ of partial states, we call *reachability problem* ($RP^{\mathcal{R}}_{\mathcal{L}}$) the problem of deciding whether or not some element from $\mathcal{R}$ is reachable.

Given a reachability problem $RP^{\mathcal{R}}_{\mathcal{L}}$ we denote by $\mathbb{L}_g$ the set of indices of the LTSs involved in $\mathcal{R}$: for each $i \in \mathbb{L}_g$, there exists at least one partial state in which the element corresponding to $\mathcal{L}_i$ is not $\star$. In a reachability problem, if there exists a reachable global state in $\mathcal{L}$ that concretises an element from $\mathcal{R}$, we write $\mathcal{L} \to \mathcal{R}$. If this is not the case, we write $\mathcal{L} \nrightarrow \mathcal{R}$.

We conclude this section by establishing two basic results on partial products of LTSs that will be instrumental in proving that our approach is sound and complete.

Lemma 10 formalizes the fact that, due to the synchronization by shared labels mechanism, removing an LTS from a product produces an over-approximation of the reachability property projected on the remaining LTSs.

▶ **Lemma 10.** *Let $\mathcal{L} = \mathcal{L}_1 \parallel \cdots \parallel \mathcal{L}_n$ be a compound system in which some global state $s = (s_1, \ldots, s_n)$ is reachable. Let $\mathcal{L}'$ be the partial product of $\mathcal{L}$ obtained by removing $\mathcal{L}_i$ for some i. Similarly, let s' be the state of $\mathcal{L}'$ obtained from s by removing the i^{th} component s_i.*

Then s' is reachable in $\mathcal{L}'$.

Lemma 11 works in the opposite direction to Lemma 10: If we can find a subset of the LTSs, in which we can reach some given state, and that does not make use of any label appearing in an LTS not in this subset – condition (i) –, then we can add the missing LTSs to get the full product, while still ensuring the reachability of our given state. The result we prove is actually a bit stronger: one can preserve the reachability found in a *partial product* of our subset of LTSs, provided that if some label appears on a transition in the partial product, then it is present in the alphabets of all the components of the partial product that can be extended into an LTS that uses this label – condition (ii). Condition (i) ensures that we do not add any synchronization constraint to existing transitions when adding new LTS, while condition (ii) does the same but when extending the LTSs already in the subset.

▶ **Lemma 11.** *Let $\mathcal{L} = \mathcal{L}_1 \parallel \cdots \parallel \mathcal{L}_n$ be a compound system. Let $H \subseteq [1..n]$. Suppose to simplify the writing that $H = [1..h]$ and let $\mathcal{C} = \mathcal{C}_1 \parallel \cdots \parallel \mathcal{C}_h$ be a partial product of $(\parallel_{i \in H} \mathcal{L}_i)$ such that: (i) for all $i \notin H$, $\Sigma(\mathcal{L}_i) \cap \lambda(\mathcal{C}) = \emptyset$, and (ii) for all $i \in H$, $\Sigma(\mathcal{L}_i) \cap \lambda(\mathcal{C}) \subseteq \Sigma(\mathcal{C}_i)$.*

If some global state $s = (s_1, \ldots, s_n)$ is reachable in $\mathcal{C}$ with path π then the global state $s^* = (s_1^*, \ldots, s_n^*)$ defined by $s_i^* = s_i$ if $i \in H$ and s_i^* is an arbitrary state of $\mathcal{L}_i$ otherwise is reachable in $\mathcal{L}$ with the same path from the state $s^0 = (s_1^0, \ldots, s_n^0)$ such that s_i^0 is the initial state of $\mathcal{L}_i$ if $i \in H$ and $s_i^0 = s_i^*$ otherwise.

3 The Lazy Reachability Analysis Algorithm

In the following subsections, we present our algorithm. It makes use of the classical abstract list data-structure, with the usual operations: $\mathsf{hd}(), \mathsf{tl}(), \mathsf{len}()$ give respectively the *head*, *tail*, and *length* of a list. Operator : is the list constructor (prepend) and ++ is concatenation. $\mathsf{rev}()$ reverses a list. [ ] is the empty list and $L[i]$ is the i^{th} element of list L.

Algorithm 1 Algorithm solving $RP_{\mathcal{L}}^{\mathcal{R}}$ ($\mathbb{L}_g$: indices of LTSs involved in $\mathcal{R}$)

```
1:  function Solve(L,R)
2:      choose a partition {I_1, ..., I_p} of L_g
3:      ∀k ∈ [1..p], let ID_k = ||_{i∈I_k} id(L_i) and INI_k = ||_{i∈I_k} ini(L_i).
4:      Ls ← [([ ], (ID_1, INI_1, I_1, ∅, I_1), [ ]), ..., ([ ], (ID_p, INI_p, I_p, ∅, I_p), [ ])]
5:      Complete ← False
6:      Consistent ← False
7:      while not Complete or not Consistent do
8:          Complete ← ∀k, Ls[k] is complete
9:          if not Complete then
10:             optional unless Consistent
11:                 mayHaveSol ← Concretise(Ls)
12:                 if not mayHaveSol then return False
13:                 end if
14:             end option
15:         end if
16:         Consistent ← Ls is consistent
17:         if not Consistent then
18:             optional unless Complete
19:                 Merge(Ls)
20:             end option
21:         end if
22:     end while
23:     return True
24: end function
```

Algorithm 1 is the main function solving our reachability problems. It starts from a partition of the LTSs involved in the reachability objective. The idea here is to decompose

this objective and verify it separately on each involved component with the hope that they do not interact. List Ls has (initially) one element per part of the initial partition. Each of those elements is a list of tuples $(\mathsf{A}, \mathsf{C}, I, J, K)$, described in details in the next subsection, that represent more and more concrete partial products (in the sense that they include more and more LTSs) of the system built around the LTSs in each partition, as we go towards the end of the list. In our algorithm, this list is walked along using the functional programming idiom of Huet's Zipper by decomposing it in Left, the current element, and Right, but it could as well be represented as a big array with a current index, etc.

We start with partial products consisting of only the initial states of each involved LTSs. The algorithm then performs two main tasks: concretisation and merging.

3.1 Concretisation

Concretisation consists in extending the partial products in two different directions: by computing more and more states and transitions for a given number of LTSs, and by adding LTSs. The (indices of the) LTSs currently used in the products are in the set J, and those being added are in set K. Set I serves as a memory of the initial partition of LTSs involved in the objective. LTS C is the current partial product we have computed and A represents what we had computed at the previous level (before we added the LTSs in K).

Algorithm 2 Auxiliary function CONCRETISE() for Algorithm 1

```
1:  function CONCRETISE(Ls)
2:      choose k ∈ [0..len(Ls) − 1] such that Ls[k] is not complete
3:      (Left, (A, C, I, J, K), Right) ← Ls[k]
4:      if there exists C* s.t. C ⊏ C* ⊑ (‖ i∈K L_i) ‖ A and C* → R_|J∪K  then
5:          choose such a C*
6:          N ← {i ∉ J ∪ K : Σ(L_i) ∩ λ(C*) ≠ ∅}
7:          case Right ≠ [ ]
8:              (_, C', _, J', K') ← hd(Right)
9:              Ls[k] ← ((A, C*, I, J, K) : Left, (C*, C', I, J', K'), tl(Right))
10:         case Right = [ ] and N ≠ ∅
11:             choose ∅ ⊂ K' ⊆ N
12:             Ls[k] ← ((A, C*, I, J, K) : Left, (C*, ‖_{i∈J∪K∪K'} ini(L_i), I, J ∪ K, K'), [ ])
13:         case Right = [ ] and N = ∅
14:             Ls[k] ← (Left, (A, C*, I, J, K), [ ])
15:     else
16:         if Left = [ ] then return False
17:         else
18:             Right' ← FORGET((A, C, I, J, K) : Right)
19:             Ls[k] ← (tl(Left), hd(Left), Right')
20:         end if
21:     end if
22:     return True
23: end function
```

The goal of the CONCRETISE() function (Algorithm 2) is to find a partial product reaching the objective and using only LTSs that are either in J or in K. This is what we call *complete* and is formalized as follows:

▶ **Definition 12** (Completeness). The tuple $(\mathsf{Left}, (\mathsf{A}, \mathsf{C}, I, J, K), \mathsf{Right})$ is *complete* if $\exists \mathsf{C}^* \sqsubseteq \mathsf{C}$ such that $\mathsf{C}^* \to \mathcal{R}_{|J\cup K}$ and $\{i \notin J \cup K \ : \ \Sigma(\mathcal{L}_i) \cap \lambda(\mathsf{C}^*) \neq \emptyset\} = \emptyset$.

To ensure completeness, we have to add LTSs that share actions with our current partial product: this is the role of the case in line 10.

LTS A serves as a limit as to what we are allowed to compute at a given level. If we need to compute more, because we cannot find the (partial) objective, we have to backtrack to the previous level to increase this limit (line 19). If we cannot backtrack (line 16), then we have explored the whole product only missing some components, so we have an over-approximation,

which implies that the property is false. If we successfully backtrack, and when we have extended again our partial product at that "lower" level, we can go forth again using the memory of what we had already computed (line 7).

Note the use of the FORGET() function in line 19 that permits to "forget" part of that memory when we backtrack. This is useful if we are able to determine that we had made bad moves: for instance in the choice of the LTSs to add to the product. Many implementations of this function are possible provided they have the following property:

▶ **Property 13** (Good FORGET() implementation)**.** Let Lt be a list $[(\mathsf{A}_1, \mathsf{C}_1, I_1, J_1, K_1), \ldots, (\mathsf{A}_n, \mathsf{C}_n, I_n, J_n, K_n)]$ of tuples. Then FORGET(Lt) must be a list $[(\mathsf{A}'_1, \mathsf{C}'_1, I'_1, J'_1, K'_1), \ldots, (\mathsf{A}'_m, \mathsf{C}'_m, I'_m, J'_m, K'_m)]$ of tuples with $0 \leq m \leq n$ and $f : [1..m] \to [1..n]$ a non-decreasing function, such that:

- $\mathsf{A}'_1 = \mathsf{A}_1$, $J'_1 = J_1$,
- $\forall j \in [1..m]$,
  - $I'_j = I_1$ (remark that, by construction, $I_1 = I_2 = \cdots = I_n$),
  - $\emptyset \subset K'_j \subseteq (J_{f(j)} \cup K_{f(j)}) \setminus J'_j$
  - $\mathsf{C}'_j \sqsubseteq C_{f(j)}$ and $\| _{k \in J'_j \cup K'_j} \mathsf{ini}(\mathcal{L}_k) \sqsubseteq \mathsf{C}'_j \sqsubseteq (\|_{k \in K'_j} \mathcal{L}_k) \parallel \mathsf{A}'_j$
- $\forall j \in [2..m]$,
  - $\mathsf{A}'_j = \mathsf{C}'_{j-1}$
  - $J'_j = J'_{j-1} \cup K'_{j-1}$

In some sens, a good FORGET() implementation can be seen as a restriction of Lt to a subset of its elements. Each of these elements being itself restricted to a less concrete tuple (by taking subsets of C, J, and/or K).

The two most obvious implementations are either to never forget anything (then FORGET() is just the identity function) or to always forget everything (that is, take $m = 0$ in the above property – then FORGET() always returns the empty list). It is clear that these two choices satisfy Property 13.

Finally, in any element of Ls, we have a list of partial products that concern more and more LTSs. To sum up, a call to CONCRETISE() at least adds new LTSs to the current tuple $(\mathsf{A}, \mathsf{C}, I, J, K)$, or adds paths in C.

3.2 Merging

Merging occurs when two parts of a partition concretised independently use common LTSs. We have to ensure that these common LTSs behave consistently and we therefore merge the two partitions by computing the product of the two partial products. This use of common LTSs in different partitions we call inconsistency and it is formals as follows:

▶ **Definition 14** (Consistency)**.** The list of triples $\mathsf{Ls} = [(\mathsf{Left}_1, (\_, \_, \_, J_1, K_1), \mathsf{Right}_1), \ldots, (\mathsf{Left}_n, (\_, \_, \_, J_n, K_n), \mathsf{Right}_n)]$ is *consistent* if $\forall i \neq j \in [1..n]$, $(J_i \cup K_i) \cap (J_j \cup K_j) = \emptyset$

Merging itself is done by the MERGE() function. As for the FORGET() function, many implementations are possible provided they satisfy Property 15:

▶ **Property 15** (Good MERGE() implementation)**.** Let Ls be a non-consistent list of triples $[\mathcal{T}_1, \ldots, \mathcal{T}_n]$. For any triple $\mathcal{T}_k$, note $\mathcal{T}_k = (\mathsf{Left}_k, (\mathsf{A}_k, \mathsf{C}_k, I_k, J_k, K_k), \mathsf{Right}_k)$ and $\mathsf{Full}(\mathcal{T}_k) = \mathsf{rev}(\mathsf{Left}_k) +\!+ [(\mathsf{A}_k, \mathsf{C}_k, I_k, J_k, K_k)] +\!+ \mathsf{Right}_k$. Then MERGE($\mathsf{Ls}$) is any list $[\mathcal{T}_1, \ldots, \mathcal{T}_{i-1}, \mathcal{T}_{i+1}, \ldots, \mathcal{T}_{j-1}, \mathcal{T}_{j+1}, \ldots, \mathcal{T}_n, \mathcal{T}]$ of triples such that:

- $(J_i \cup K_i) \cap (J_j \cup K_j) \neq \emptyset$ (such i, j always exist because Ls is not consistent)

- There exist two non-decreasing functions $f_i : [1..m] \to [1..n_i]$ and $f_j : [1..m] \to [1..n_j]$ so that $\mathsf{Full}(\mathcal{T}) = [(\mathsf{A}''_1, \mathsf{C}''_1, I''_1, J''_1, K''_1), \ldots, (\mathsf{A}''_m, \mathsf{C}''_m, I''_m, J''_m, K''_m)]$, $\mathsf{Full}(\mathcal{T}_i) = [(\mathsf{A}_1, \mathsf{C}_1, I_1, J_1, K_1), \ldots, (\mathsf{A}_{n_i}, \mathsf{C}_{n_i}, I_{n_i}, J_{n_i}, K_{n_i})]$, and $\mathsf{Full}(\mathcal{T}_j) = [(\mathsf{A}'_1, \mathsf{C}'_1, I'_1, J'_1, K'_1), \ldots, (\mathsf{A}'_{n_j}, \mathsf{C}'_{n_j}, I'_{n_j}, J'_{n_j}, K'_{n_j})]$ with $m \leq n_i + n_j$ verify:
  - $\mathsf{A}''_1 = \mathsf{A}_1 \parallel \mathsf{A}'_1$, $J''_1 = J_1 \cup J'_1 = \emptyset$ (remark that, by construction, $J_1 = J'_1 = \emptyset$),
  - $I_1 \cup I'_1 \subseteq K''_1 \subseteq J_{f_i(1)} \cup K_{f_i(1)} \cup J'_{f_j(1)} \cup K'_{f_j(1)}$
  - $\forall k \in [1..m]$,
    - $I''_k = I_1 \cup I'_1$ (remark that, by construction, $I_1 = \cdots = I_{n_i}$ and $I'_1 = \cdots = I'_{n_j}$),
    - $\mathsf{C}''_k \sqsubseteq \mathsf{C}_{f_i(k)} \parallel \mathsf{C}_{f_j(k)}$ and $\parallel_{\ell \in J''_k \cup K''_k} \mathsf{ini}(\mathcal{L}_\ell) \sqsubseteq \mathsf{C}''_k \sqsubseteq (\|_{\ell \in K''_k} \mathcal{L}_\ell) \parallel \mathsf{A}''_k$
  - $\forall k \in [2..m]$,
    - $\emptyset \subset K''_k \subseteq (J_{f_i(k)} \cup K_{f_i(k)} \cup J'_{f_j(k)} \cup K'_{f_j(k)}) \setminus J''_k$
    - $\mathsf{A}''_k = \mathsf{C}''_{k-1}$
    - $J''_k = J''_{k-1} \cup K''_{k-1}$

Intuitively, a good Merge() implementation should select two triples breaking the consistency of Ls and merge them. These two triples are in fact two histories of concretisations (i.e. the result of a sequence of calls to Concretise()). Merging them basically consists in interleaving these two histories and then apply a Forget()-like construct on this interleaving. This produces an history that could have been produced by a sequence of calls to Concretise() starting from $([\,], (\parallel_{i \in I_1 \cup I'_1} \mathsf{id}(\mathcal{L}_i), \parallel_{i \in I_1 \cup I'_1} \mathsf{ini}(\mathcal{L}_i), I_1 \cup I'_1, \emptyset, I_1 \cup I'_1), [\,])$, i.e if $I_1 \cup I'_1$ had been an element of the initial partition chosen in Algorithm 1.

At the lowest level of implementation, the basic merging operation of two tuples $h_1 = (\mathsf{A}_1, \mathsf{C}_1, I_1, J_1, K_1)$ and $h_2 = (\mathsf{A}_2, \mathsf{C}_2, I_2, J_2, K_2)$ gives the tuple $(\mathsf{A}_1 \parallel \mathsf{A}_2, \mathsf{C}_1 \parallel \mathsf{C}_2, I_1 \cup I_2, J_1 \cup J_2, (K_1 \setminus J_2) \cup (K_2 \setminus J_1))$. Let us denote it by $h_1 * h_2$. Building on that, a minimal implementation satisfying Property 15 would be to select i and j such that $\mathsf{Ls}[i] = (\mathsf{Left}_i, h_i, \mathsf{Right}_i)$, with $h_i = (\mathsf{A}_i, \mathsf{C}_i, I_i, J_i, K_i)$ and $\mathsf{Ls}[j] = (\mathsf{Left}_j, h_j, \mathsf{Right}_j)$, with $h_j = (\mathsf{A}_j, \mathsf{C}_j, I_j, J_j, K_j)$, and $(J_i \cup K_i) \cap (J_j \cup K_j) \neq \emptyset$, remove them from Ls and replace them by the singleton list $([\,], \mathsf{hd}(\mathsf{Left}_i) * \mathsf{hd}(\mathsf{Left}_j), [\,])$. Another relevant choice, which is easy to compute and also trivially satisfies Property 15, would be to also keep the current elements by replacing both $\mathsf{Ls}[i]$ and $\mathsf{Ls}[j]$ by $([\mathsf{hd}(\mathsf{Left}_i) * \mathsf{hd}(\mathsf{Left}_j)], h_i * h_j, [\,])$.

When we have found a list of complete partial products that is consistent then we know that our objective is reachable.

3.3 Example

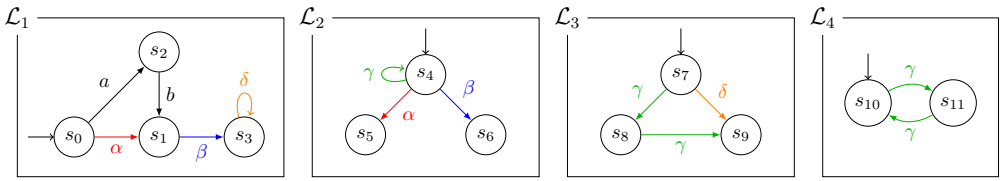

Figure 1 A compound system with four LTSs ($\mathcal{L}_1$ to $\mathcal{L}_4$).

Let us perform a sample execution of the algorithm on the system described in Figure 1. Suppose we want to reach the partial state $(s_3, \star, s_9, \star)$. Therefore we have the set of indices of the LTSs involved in the objective $\mathbb{L}_g = \{1, 3\}$ and we choose to partition it, for instance, as $\{\{1\}, \{3\}\}$.

Then, $\mathsf{Ls}[1]$ is the list $[([\,], (\mathsf{id}(\mathcal{L}_1), \mathsf{ini}(\mathcal{L}_1), \{1\}, \emptyset, \{1\}), [\,])]$ and, similarly, $\mathsf{Ls}[2]$ is the list $[([\,], (\mathsf{id}(\mathcal{L}_3), \mathsf{ini}(\mathcal{L}_3), \{3\}, \emptyset, \{3\}), [\,])]$. None of those elements is complete because they do not reach their part of the objective so we call Concretise().

In that function, we choose for instance $k = 1$ and compute an extension of $\mathsf{ini}(\mathcal{L}_1)$ that reaches the objective. Say we compute the extension C_1^* made of the path s_0, s_1, s_3 (Figure 2). Then, since labels α and β are shared with $\mathcal{L}_2$, and $2 \notin K_1 = \{1\}$, we have $\mathcal{N} = \{2\}$. Since $\mathsf{Right}_1 = [\,]$, we get to line 10, replace $\mathsf{ini}(\mathcal{L}_1)$ by C_1^*, add the tuple $(\mathsf{C}_1^*, \mathsf{ini}(\mathcal{L}_1) \parallel \mathsf{ini}(\mathcal{L}_2), \{1\}, \{1\}, \{2\})$ to the list represented by $\mathsf{Ls}[1]$, and set that tuple as the current element of that list. Finally, we return true.

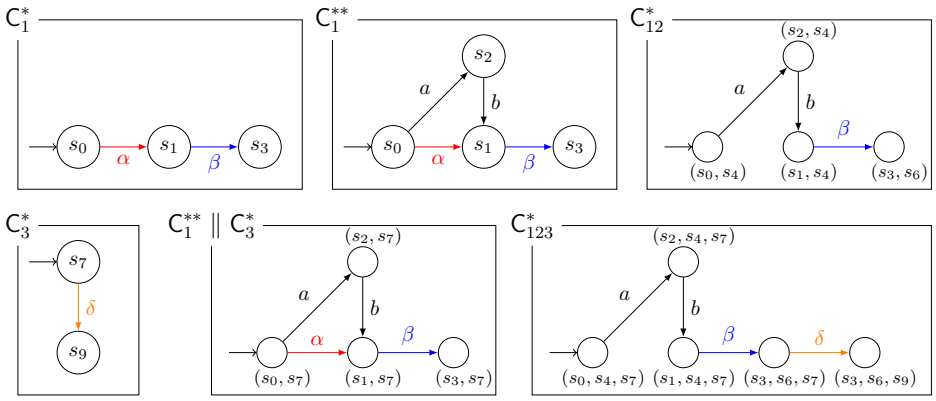

Figure 2 Some LTSs appearing during an execution of our algorithm on the example of Figure 1.

Back in the main function, Ls is consistent for now, so we move to the next iteration. Again both $\mathsf{Ls}[1]$ and $\mathsf{Ls}[2]$ are not complete because their current elements do not reach their objective. Let us say we concretise again $\mathsf{Ls}[1]$. This time we cannot find an extension of $\mathsf{ini}(\mathcal{L}_1) \parallel \mathsf{ini}(\mathcal{L}_2)$ in the product of $\mathcal{L}_1$, $\mathcal{L}_2$ and C_1^*, which is empty. Then, since Left_1 is not empty, we go to line 18, call Forget() (suppose here it forgets nothing), and set again the head of the list represented by $\mathsf{Ls}[1]$ (which is also the head of Left_1) as the current element. Then we return true again.

Back in the main function, Ls is still consistent and both its element are not complete. We then call Concretise() and for the example choose again $k = 1$. This time we extend C_1^* by taking C_1^{**} as the whole of $\mathcal{L}_1$ except the δ transition (Figure 2). Since Right_1 is not empty this time, we go to line 7, update the tuples with C_1^{**}, and move the current element of the list right, then return true.

Back in the main function we still call Concretise() and choose again $k = 1$. This time, we can extend $\mathsf{ini}(\mathcal{L}_1) \parallel \mathsf{ini}(\mathcal{L}_2)$ with the LTS C_{12}^* made only of the path $(s_0, s_4), (s_2, s_4), (s_1, s_4), (s_3, s_6)$ (Figure 2). Set $\mathcal{N}$ is empty because no LTS other than $\mathcal{L}_1$ and $\mathcal{L}_2$ has labels β, a or b (used in this path). Right_1 is also empty so we just update the current element in $\mathsf{Ls}[1]$ with C_{12}^* (line 14) and return true.

Back in the main function, $\mathsf{Ls}[1]$ is now complete but not $\mathsf{Ls}[2]$. So, we call Concretise() and choose $k = 2$. We extend $\mathsf{ini}(\mathcal{L}_3)$ by C_3^* made of the path s_7, s_9 (Figure 2). Then $\mathcal{N} = \{1\}$ because $\mathcal{L}_1$ shares label δ with C_3^* and, as before, Right_2 being empty, we go to line 10, replace $\mathsf{ini}(\mathcal{L}_3)$ by C_3^*, add the tuple $(\mathsf{C}_3^*, \mathsf{ini}(\mathcal{L}_3) \parallel \mathsf{ini}(\mathcal{L}_1), \{3\}, \{3\}, \{1\})$ to the list represented by $\mathsf{Ls}[2]$, and set that tuple as the current element of that list. Then we return true.

Now, we have the index 1 in $J \cup K$ for both the current elements of $\mathsf{Ls}[1]$ and $\mathsf{Ls}[2]$, so we can choose to merge them. Let us do it: we use the second of the simple strategies outlined

above: we keep and merge only the first and current elements of each list. After the call to MERGE(), $\mathsf{Ls} = [([(\mathsf{id}(\mathcal{L}_1) \parallel \mathsf{id}(\mathcal{L}_3), \mathsf{C}_1^{**} \parallel \mathsf{C}_3^*, \{1,3\}, \emptyset, \{1,3\})], (\mathsf{C}_1^{**} \parallel \mathsf{C}_3^*, \mathsf{C}_{12}^* \parallel \mathsf{ini}(\mathcal{L}_3) \parallel \mathsf{ini}(\mathcal{L}_1), \{1,3\}, \{1,3\}, \{2\}), [\,])]$ and $\mathsf{C}_{12}^* \parallel \mathsf{ini}(\mathcal{L}_3) \parallel \mathsf{ini}(\mathcal{L}_1)$ consists only of state (s_0, s_4, s_7), because $\mathsf{ini}(\mathcal{L}_1)$ restricts C_{12}^* to its initial state and δ is not in C_{12}^*. That product has no path to a state $(s_3, \star, s_9)$, so the new $\mathsf{Ls}[1]$ is not complete and we need to call CONCRETISE() one last time.

We will then be able to extend $\mathsf{C}_{12}^* \parallel \mathsf{ini}(\mathcal{L}_3) \parallel \mathsf{ini}(\mathcal{L}_1)$ to an LTS C_{123}^* containing state (s_3, s_6, s_9) but with no transition γ (Figure 2). Then $\mathcal{N}$ is empty, as well as Right_1, so we go through line 14 to update the current LTS, and return true. Finally, in the main function, we now have that Ls is consistent, since it contains only one element, and that element is complete because C_{123}^* contains only labels not shared with $\mathcal{L}_4$. So we terminate and return true.

Note that we never needed to consider products involving $\mathcal{L}_4$ – which is the reason why we call our analysis "lazy".

3.4 Soundness, completeness, termination

We now proceed to proving that our algorithm is sound and complete and that it terminates. We first state two utility lemmas.

▶ **Lemma 16.** *Let* $(\mathit{Left}, h, \mathit{Right})$ *be an element of* Ls *in* SOLVE$(\mathcal{L}, \mathcal{R})$ *and let* $\mathcal{L} = (\|_{i\in[1..n]} \mathcal{L}_i)$.

Let (A, C, I, J, K) *be either* h*, or an element of* Left*, or an element of* Right*. Then* C *is a partial product of* $(\|_{i\in J\cup K} \mathcal{L}_i)$ *and, if we write* $C = (\|_{i\in J\cup K} C_i)$*, then* $\Sigma(C_i) = \Sigma(\mathcal{L}_i)$.

▶ **Lemma 17.** *Let* $(\mathit{Left}, (A, C, I, J, K), \mathit{Right})$ *be an element of* Ls *in* SOLVE$(\mathcal{L}, \mathcal{R})$*. If* Left *is empty then* $I \subseteq K$*,* $J = \emptyset$ *and* $A = \|_{i\in I} \mathsf{id}(\mathcal{L}_i)$.

And finally the main results:

▶ **Proposition 18.** If SOLVE$(\mathcal{L}, \mathcal{R})$ returns False then $\mathcal{R}$ is not reachable in $\mathcal{L}$.

Proof. The only way SOLVE$(\mathcal{L}, \mathcal{R})$ can return False is through line 12 in Algorithm 1, and in turn this means that the call to CONCRETISE() in line 11 returned False. Now CONCRETISE() will only return False through line 16 in Algorithm 2. To get there, there must exist some k such that $\mathsf{Ls}[k]$ can be decomposed as the triple $(\mathsf{Left}, (\mathsf{A}, \mathsf{C}, I, J, K), \mathsf{Right})$, with (1) Left being empty, and (2) there is no C^* s.t. $\mathsf{C} \sqsubset \mathsf{C}^* \sqsubseteq (\|_{i\in K} \mathcal{L}_i) \parallel \mathsf{A}$ and $\mathsf{C}^* \to \mathcal{R}_{|J\cup K}$.

By Lemma 16 we know that C is a partial product of $\mathcal{L}$. With (1) and Lemma 17, we have that $I \subseteq K$, $J = \emptyset$ and $\mathsf{A} = \|_{i\in I} \mathsf{id}(\mathcal{L}_i)$. So $(\|_{i\in K} \mathcal{L}_i) \parallel \mathsf{A} = (\|_{i\in K} \mathcal{L}_i)$. Then, with (2), we can deduce that $\mathcal{R}_{|K}$ is not reachable in $(\|_{i\in K} \mathcal{L}_i)$, and finally with the contrapositive of Lemma 10, we get that $\mathcal{R}$ is not reachable in $\mathcal{L}$. ◀

▶ **Proposition 19.** If SOLVE$(\mathcal{L}, \mathcal{R})$ returns True then $\mathcal{R}$ is reachable in $\mathcal{L}$.

Proof. The only way SOLVE$(\mathcal{L}, \mathcal{R})$ can return True is through line 23 in Algorithm 1. This can only happen when Ls is such that (1) for all k, $\mathsf{Ls}[k]$ is complete and (2) Ls is consistent.

If we denote by $(\mathsf{A}_k, \mathsf{C}_k, I_k, J_k, K_k)$ the second component of $\mathsf{Ls}[k]$, and by H_k the union $J_k \cup K_k$, (1) translates to $\forall k$, there exists $\mathsf{C}_k^* \sqsubseteq \mathsf{C}_k$ such that C_k^* can reach $\mathcal{R}_{|H_k}$ and $\{i \notin H_k : \Sigma(\mathcal{L}_i) \cap \lambda(\mathsf{C}_k^*) \neq \emptyset\} = \emptyset$. Similarly, (2) translates to $\forall i, j, H_i \cap H_j = \emptyset$.

By Lemma 16, each C_k^* is a partial product of $(\|_{i\in H_k} \mathcal{L}_i)$ (and thus also of $\mathcal{L}$). Now consider some $i \in H_k$ and $\sigma \in \Sigma(\mathcal{L}_i) \cap \lambda(\mathsf{C}_k^*)$. By Lemma 16, we know that there exist some C_i such that $\mathsf{C}^* = (\|_{i\in H_k} \mathsf{C}_i)$ and $\Sigma(\mathsf{C}_i) = \Sigma(\mathcal{L}_i)$. Consequently, for all $i \in H_k$,

$\Sigma(\mathcal{L}_i) \cap \lambda(\mathsf{C}) \subseteq \Sigma(\mathsf{C}_i)$. From (2), we also have that $\forall i \notin H_k$, $\Sigma(\mathcal{L}_i) \cap \lambda(\mathsf{C}) = \emptyset$ and we can thus use Lemma 11 and obtain that $\mathcal{R}_{|H_k}$ is reachable in $\mathcal{L}$, whatever the states of the components not in H_k (and leaving them unchanged). Therefore, by finally putting all components together, $\mathcal{R}$ is reachable in $\mathcal{L}$. ◀

Relation $\sqsubseteq$ is not sufficient to reflect progress in our algorithm. We therefore introduce a new relation $<_\ell$, built on top of $\sqsubseteq$, as a partial order over lists Ls (as the ones appearing in Algorithm 1). Relation $<_\ell$ does reflect progress in concretisation (by advancing in the history of concretisations (2), or by adding LTSs (3), or by adding paths in partial products (4,5)), and merging (by reducing the length of the list (1)).

▶ **Definition 20.** Given two lists of tuples Ls_1 and Ls_2 with the same type as Ls in Algorithm 1, we define $<_\ell$ such that $\mathsf{Ls}_1 <_\ell \mathsf{Ls}_2$ if and only if $\mathsf{Ls}_1 \neq \mathsf{Ls}_2$ and:

- $\mathsf{len}(\mathsf{Ls}_1) > \mathsf{len}(\mathsf{Ls}_2)$, or (1)
- $\mathsf{len}(\mathsf{Ls}_1) = \mathsf{len}(\mathsf{Ls}_2)$ and $\forall k, \mathsf{Ls}_1[k] \neq \mathsf{Ls}_2[k] \implies \mathsf{Ls}_1[k] <_t \mathsf{Ls}_2[k]$;

where $(\mathsf{Left}_1, h_1, \mathsf{Right}_1) <_t (\mathsf{Left}_2, h_2, \mathsf{Right}_2)$ if and only if, for $\mathsf{hLr}_i = \mathsf{rev}(\mathsf{Left}_i) +\!\!+ [h_i]$,

- hLr_2 is a prefix of $\mathsf{rev}(\mathsf{Left}_1)$, or (2)
- $\exists \ell, \mathsf{hLr}_1[\ell] \neq \mathsf{hLr}_2[\ell]$ and, for the smallest such ℓ one has $\mathsf{hLr}_1[\ell] <_a \mathsf{hLr}_2[\ell]$;

where $(\mathsf{A}_1, \mathsf{C}_1, I_1, J_1, K_1) <_a (\mathsf{A}_2, \mathsf{C}_2, I_2, J_2, K_2)$ if and only if:

- $J_1 \cup K_1 \subset J_2 \cup K_2$, or (3)
- $J_1 \cup K_1 = J_2 \cup K_2$ and $\mathsf{A}_1 \sqsubset \mathsf{A}_2$, or (4)
- $J_1 \cup K_1 = J_2 \cup K_2$, $\mathsf{A}_1 = \mathsf{A}_2$, and $\mathsf{C}_1 \sqsubset \mathsf{C}_2$. (5)

If $\mathsf{Ls}_1 <_\ell \mathsf{Ls}_2$ or $\mathsf{Ls}_1 = \mathsf{Ls}_2$ we write $\mathsf{Ls}_1 \leq_\ell \mathsf{Ls}_2$.

▶ **Proposition 21.** The calls to Solve$(\mathcal{L}, \mathcal{R})$ always terminate (and return only True or False).

Sketch of the proof. The fact that, if a call to Solve$(\mathcal{L}, \mathcal{R})$ terminates, it can only return True or False, comes from lines 23 (returning True) and 12 (returning False) of Algorithm 1 which are the only return statements of the Solve$(,)$ function.

In order to prove the termination one can show that (1) $\leq_\ell$ is an order relation over the Ls used in Algorithm 1, (2) the set of such lists appearing in any instance of Algorithm 1 is finite (and so, there are lists which are greater or incomparable to any other lists with respect to $\leq_\ell$), and (3) any step of the while loop of Algorithm 1 terminates and if the return of line 12 is not used, strictly increases Ls with respect to $\leq_\ell$. From (1) and (2) one then gets that, in any instance of Algorithm 1, there cannot exist an infinite strictly increasing chain of Ls with respect to $\leq_\ell$. Hence, from (3), Algorithm 1 always terminates. ◀

4 Experimental analysis

In order to get insight on the practical efficiency of our algorithm we developed a tool[1] (LaRA, for Lazy Reachability Analyzer) using it. We then compared the time efficiency of LaRA with that of other tools on several reachability analysis tasks in distributed systems. We originally selected three other tools for these experiments:

- LoLA[2]: A Petri net analyzer (it is straightforward to convert the compound systems we consider in this paper into (safe) Petri nets) which efficiently implements many techniques

[1] For reproducibility of experiments, LaRA is available at `http://lara.rts-software.org`
[2] `http://service-technology.org/lola/`

for model checking Petri nets. LoLA is arguably very effective for reachability analysis in Petri nets as it won the reachability track at the last model checking contest [13].
- PMC [14]: A tool for partial model checking that uses incremental techniques for dealing with the verification of distributed systems.
- The on the fly model checking capabilities of the CADP toolbox [8]

Early preliminary experiments revealed that, on all our benchmarks, LoLA outperformed PMC and CADP. For the larger experiments on which we report here we thus focused on comparing LoLA with our tool.

4.1 Implementation choices

LaRA consists of approximately 500 lines of Haskell code, using the standard Parsec parsing library, and the fgl graph library. Note that memory management in Haskell is automatic.

We have presented our algorithm in a manner as generic as possible including the possibility for many heuristic choices. For our first prototype presented here, we have chosen to completely compute each partial product before adding more components. This eliminates the need for backtracking, which greatly simplifies the code and, to some extent, favors the case when the desired state is not reachable. This choice is rather drastic and probably not optimal when exploring big partial products, in which a more on-the-fly approach would usually give better results. However, we believe it is reasonable, as our objective was to evaluate the influence of the laziness feature of our approach.

Before adding LTSs to the partial product, we trim it by keeping only the reachable and coreachable states. We add only one automaton each time, chosen arbitrarily in the set of LTSs synchronized on the path that synchronizes as few LTSs as possible.

4.2 Benchmarks

Our benchmarks where taken from a set of benchmarks proposed by Corbett in the 90's [5]. Among these, we selected the ones where scaling increases the number of components but does not change the size of individual components. The reason for this choice is that our early implementation is not made for dealing with large state spaces of individual components – as, again, our goal is to evaluate the impact of its laziness feature. Not embedding efficient search techniques it was hopeless to cope with finely tuned model checking tools. This left us with six models, described in Table 1. For each model we define a simple reachability property and state if it is verified by the model.

Both tools were used in a similar setting: on a machine with four Intel® Xeon® E5-2620 processors (six cores each) with 128GB of memory. Though this machine has some potential for parallel computing, all the experiments presented here are actually monothreaded. We put a time limit of 20 minutes for computations. For each experiment, each tool had as input a file in its own format: file generation and conversion are not taken into account in the processing times.

4.3 Positive results.

In most of the cases (namely Cyclic, Philo, PhiloDico, and PhiloSync) our tool outperformed LoLA with increasing size of models. On DAC, our results are comparable to those obtained with LoLA, and for very large instances we slightly outperform it. The experimental results for these cases are summarized in Table 2. For each model, timeout indicates the first

Table 1 Benchmarks description

Model	Description	Size	Property	Verified?
Cyclic	Milner's cyclic scheduler, a set of tasks have to be scheduled in a cyclic order.	Number of tasks to be scheduled.	One task in two can be in their waiting state together.	Yes.
DAC	Divide and conquer computation, a task has to be completed by a set of processes. Each one can complete the task alone or fork.	Maximal number of processes involved in the solving of the task.	A given process can be involved in the solving and decide to complete it alone.	Yes.
Philo	Dining philosophers, in its eating cycle a philosopher takes and releases his left fork first.	Number of philosophers.	One philosopher in two can eat together.	Yes for even sizes. No for odd sizes.
PhiloDico	Variation of Philo, a dictionary turns around the table, preventing the philosopher holding it to take forks.	Number of philosophers.	One philosopher in two can eat together.	Yes for even sizes. No for odd sizes.
PhiloSync	Variation of Philo, philosophers take and release both their forks in a single step.	Number of philosophers.	One philosopher in two can eat together.	Yes for even sizes. No for odd sizes.
TokenRing	Classical mutual exclusion algorithm with a token circulating on a ring of processes.	Number of processes.	Two given processes can reach their critical section together.	No.

instance of this model for which a tool reached the time limit of 20 minutes. Notice that, in the variants of the dining philosophers, there are two timeouts: one for instances of odd size and the other one for instances of even size. This is because the property we verify is false for odd sizes and true for even sizes, which makes a significant difference for LoLA.

4.4 Focus on TokenRing.

Table 3 presents the results obtained with various modeling of TokenRing. It compares runtimes of LoLA and our tool on Corbett's modeling. It appears that LaRA is far from efficient on this particular example. This is due to the fact that, without taking into account all the components, it is not possible for our tool to figure out that only one token exists in the system. So, for deciding that no two processes can be in their critical sections together, LaRA cannot be lazy and has to explore the full state space of the system.

Table 3 Runtimes on TokenRing.

Size	TokenRing	
	LaRA	LoLA
7	0.514s	<0.01s
8	1.716s	<0.01s
9	6.713s	<0.01s
10	25.810s	<0.01s
11	70.370s	<0.01s
12	322.440s	<0.01s
13	**Timeout**	<0.01s
1000		0.15s

5 Conclusion

We have presented a new approach for the verification of reachability in distributed systems. It builds on both decomposing the goal state into its projection on the different components of the system and lazily adding components in an iterative fashion to produce more and more precise over-approximations. This notably allows for early termination both when the state is reachable and when it is not. We have presented an algorithm based on this approach, together with proofs for completeness, soundness, and termination. We have also

Table 2 Comparison of runtimes of LoLA and LaRA on instances of increasing size of Cyclic, DAC, Philo, PhiloDico, and PhiloSync.

Size	Cyclic		DAC		Philo		PhiloDico		PhiloSync	
	LaRA	LoLA	LaRA	LoLA	LaRA	LoLA	LaRA	LoLA	LaRA	LoLA
15	0.01s	<0.01s	0.01s	<0.01s	0.04s	28.47s	0.10s	30.92s	0.02s	<0.01s
16	0.01s	<0.01s	0.01s	<0.01s	0.04s	<0.01s	0.05s	<0.01s	0.02s	<0.01s
17	0.01s	<0.01s	0.01s	<0.01s	0.05s	327.55s	0.10s	349.38s	0.02s	0.02s
18	0.01s	<0.01s	0.02s	<0.01s	0.04s	<0.01s	0.06s	<0.01s	0.03s	<0.01s
19	0.01s	<0.01s	0.01s	<0.01s	0.05s	**Timeout**	0.10s	**Timeout**	0.02s	0.05s
24	0.02s	<0.01s	0.01s	<0.01s	0.05s	<0.01s	0.08s	<0.01s	0.03s	<0.01s
25	0.02s	<0.01s	0.01s	<0.01s	0.06s		0.13s		0.03s	0.97s
35	0.03s	<0.01s	0.02s	<0.01s	0.08s		0.15s		0.04s	182.54s
45	0.03s	<0.01s	0.02s	<0.01s	0.11s		0.17s		0.06s	**Timeout**
1000	0.57s	2.55s	0.35s	0.56s	1.90s	2.44s	2.34s	2.50s	1.11s	2.38s
3000	2.68s	64.32s	1.08s	1.15s	6.87s	64.84s	8.56s	64.55s	4.82s	64.31s
6000	8.07s	514.89s	2.25s	1.62s	17.86s	520.86s	21.32s	523.54s	13.83s	519.21s
8000	13.37s	**Timeout**	2.97s	2.79s	27.63s	**Timeout**	32.21s	**Timeout**	22.15s	**Timeout**
10000	20.86s		3.72s	3.14s	39.73s		44.69s		33.10s	
30000	234.97s		11.24s	9.46s	334.79s		346.36s		319.15s	
50000	687.68s		19.10s	19.75s	1063.69s		1072.71s		946.86s	

implemented this into an early prototype named LaRA. This rather naive implementation already gives very promising results, on which we report together with comparisons to LoLA, a state-of-art model-checker for Petri nets.

Further note that, when a reachability property is true, LaRA has computed, and can output, a consistent list of complete LTSs satisfying that property. An interesting plus-value is that adding whatever number of new components to that list would not change the outcome provided that none of those new components shares actions that are used in the list. Therefore in systems with a particular synchronization structure, like rings for instance, we can generalize the reachability result to any number of components in the ring : for instance to prove that a philosopher can eat we need to add the two forks around her and the other two philosophers that could also use those forks. Now, any additional fork or philosopher beyond those do not share any action required to establish that the first philosopher can eat. We can then deduce that she can eat regardless of the number of philosophers around the table.

We have proposed and proved our algorithm in a generic and extendable way. In particular, it seems very likely that partial order or Decision Diagram-based symbolic techniques could be incorporated in this approach. The algorithm we propose also offers several opportunities for parallelisation. First, between two merge operations all concretisations in the different partitions can clearly be performed in parallel. Second, the different choices left open in the algorithm, such as the choice of a particular path to concretise, or a specific automaton to add to the product, can be resolved by some heuristics but may also better be handled by testing several of the different choices in parallel.

In addition to studying theses issues, further work includes extensions to more expressive formalisms, in particular (parametric) timed automata and time Petri nets, and to more complex properties.

Acknowledgments. We gratefully thank Frédéric Lang for the time he spent helping us to use his partial model checking tool. We also thank Karsten Wolf for offering help with LoLA. Finally, we thank the anonymous reviewers for their valuable comments.

References

1 H. R. Andersen. Partial model checking. In *LICS*, pages 398–407, 1995.
2 F. Bacchus and Q. Yang. Downward refinement and the efficiency of hierarchical problem solving. *Artificial Intelligence*, 71(1):43–100, 1994.
3 A. R. Bradley. SAT-based model checking without unrolling. In *VMCAI*, pages 70–87, 2011.
4 A. Cimatti and A. Griggio. Software model checking via IC3. In *CAV*, pages 277–293, 2011.
5 J. C. Corbett. Evaluating deadlock detection methods for concurrent software. *IEEE Trans. Software Eng.*, 22(3):161–180, 1996.
6 P. Crouzen and F. Lang. Smart reduction. In *FASE*, pages 111–126, 2011.
7 C. Flanagan and S. Qadeer. Thread-modular model checking. In *SPIN*, pages 213–224, 2003.
8 H. Garavel, F. Lang, R. Mateescu, and W. Serwe. CADP 2011: a toolbox for the construction and analysis of distributed processes. *STTT*, 15(2):89–107, 2013.
9 S. Graf and B. Steffen. Compositional minimization of finite state systems. In *CAV*, pages 186–196, 1990.
10 O. Grumberg and D. E. Long. Model checking and modular verification. *TOPLAS*, 16(3):843–871, 1994.
11 J. Hoffmann, J. Porteous, and L. Sebastia. Ordered landmarks in planning. *JAIR*, 22(1):215–278, 2004.
12 G. J. Holzmann and D. Peled. An improvement in formal verification. In *FORTE*, pages 197–211, 1994.
13 F. Kordon, H. Garavel, L. M. Hillah, F. Hulin-Hubard, A. Linard, M. Beccuti, A. Hamez, E. Lopez-Bobeda, L. Jezequel, J. Meijer, E. Paviot-Adet, C. Rodriguez, C. Rohr, J. Srba, Y. Thierry-Mieg, and K. Wolf. Complete Results for the 2015 Edition of the Model Checking Contest. http://mcc.lip6.fr/2015/results.php, 2015.
14 F. Lang and R. Mateescu. Partial model checking using networks of labelled transition systems and boolean equation systems. *LMCS*, 9(4), 2013.
15 A. Lehmann, N. Lohmann, and K. Wolf. Stubborn sets for simple linear time properties. In *ICATPN*, pages 228–247, 2012.