Lazy Reachability Analysis in Distributed Systems

Loïg Jezequel[1,3] and Didier Lime[2,3]

[1]Université de Nantes

[2]École Centrale de Nantes

[3]IRCCyN, UMR CNRS 6597

CONCUR
August 24, 2016
Québec

Overview of the problem

Distributed systems

- Components
- Communications

Reachability

- Component by component
- For a subset of components
- First step towards model checking

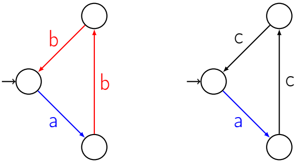
Solution

- Modular/compositional (component by component analysis)
- Lazy (add only the components needed along the analysis)

The formalism

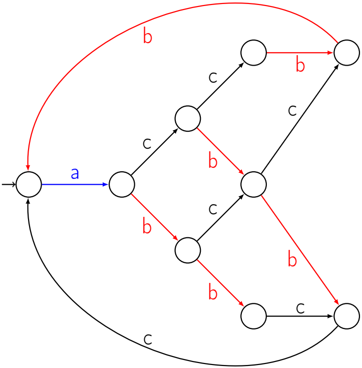
Components

Automata



Communication

Synchronous product



Reachability

Marked states

I. General principle of the algorithm

Initialization

Partition

Choose a partition of the LTSs involved in the reachability objective

Initial paths

For each element of the partition initialize a set of paths with only the empty path in it

Initialization

Partition

Choose a partition of the LTSs involved in the reachability objective

Initial paths

For each element of the partition initialize a set of paths with only the empty path in it

Finished?

Does this set of set of paths contains a solution?

Yes: we are done

No: add paths or merge sets

Completeness

Idea

A set of paths is not complete if no path reaches the (local) objective using only private actions

Solution to incompleteness: concretisation

- Add new paths to the incomplete set, or
- add new automata to the set of automata

Completeness

Idea

A set of paths is not complete if no path reaches the (local) objective using only private actions

Solution to incompleteness: concretisation

- Add new paths to the incomplete set, or
- add new automata to the set of automata

Finished?

Is this set of set of paths giving a solution?

Yes: we are done

No: add paths or merge sets

Consistency

Idea

A set of sets of paths is not consistent if two paths from different sets share actions

Solution to inconsistency: merging

- Select two sets of paths breaking consistency
- merge them (i.e. change the initial partition of the LTSs)

Consistency

Idea

A set of sets of paths is not consistent if two paths from different sets share actions

Solution to inconsistency: merging

- Select two sets of paths breaking consistency
- merge them (i.e. change the initial partition of the LTSs)

Finished?

Is this set of set of paths giving a solution?

Yes: we are done

No: add paths or merge sets

Backtracking

Concretisation: limiting exploration when adding paths to a set

In practice:

- only actions from the automata added at the very last concretisation step can be added,
- adding actions from other automata requires to backtrack (go back before previous concretisation steps),
- hence we record an *history* of concretisation steps

Backtracking

Concretisation: limiting exploration when adding paths to a set

In practice:

- only actions from the automata added at the very last concretisation step can be added,
- adding actions from other automata requires to backtrack (go back before previous concretisation steps),
- hence we record an *history* of concretisation steps

Link with merging

Do not merge sets of paths but *histories* of sets of paths

Laziness

Early finding of a solution

When completeness and consistency are achieved together

- not (necessarily) all automata involved

Laziness

Early finding of a solution

When completeness and consistency are achieved together

- not (necessarily) all automata involved

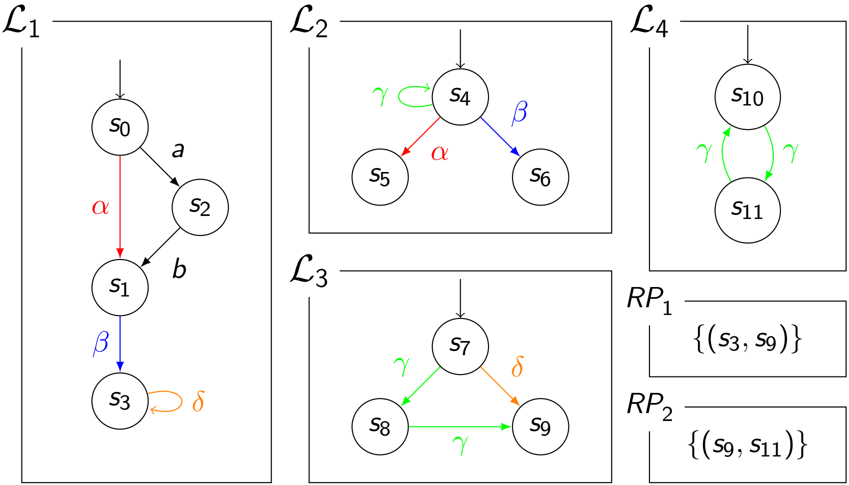
Early detection of absence of solution

When no path can be added at the beginning of an history

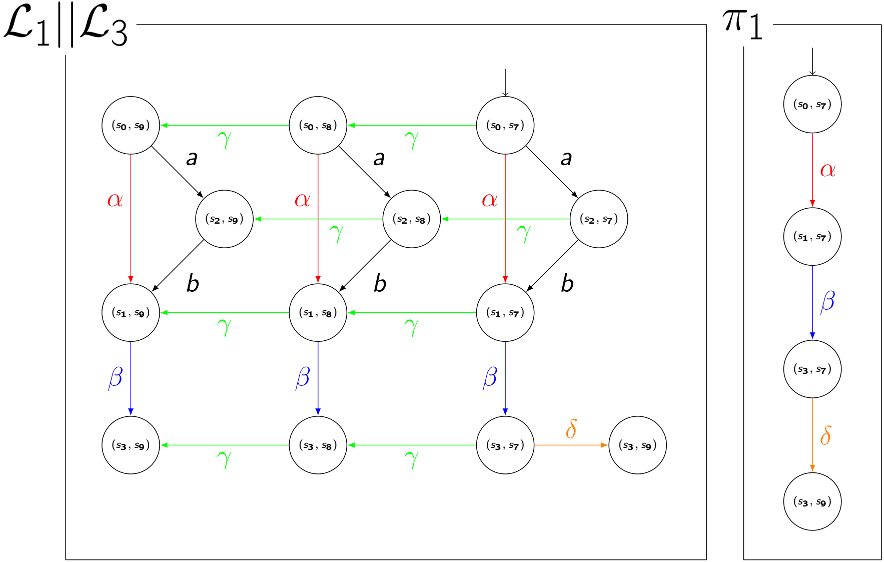
- not (necessarily) all automata involved
- not (necessarily) all merging done

II. Overview on an example

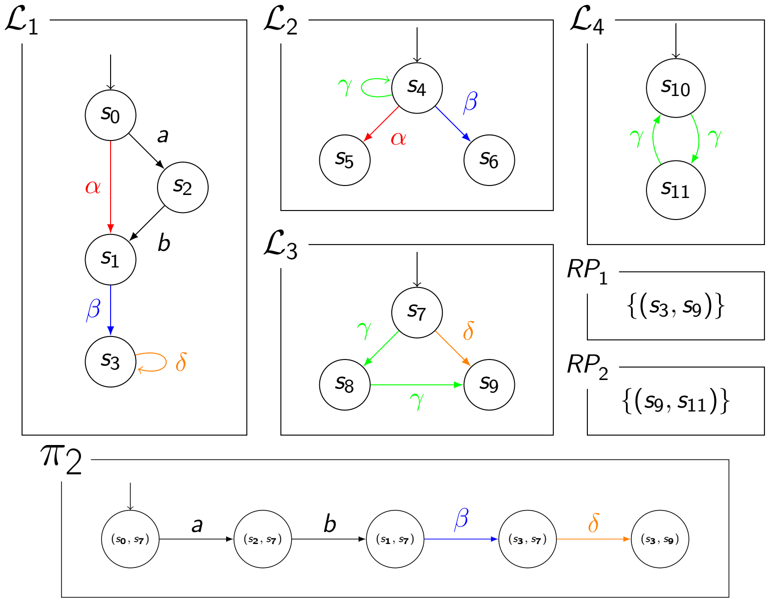
A distributed system and two problems



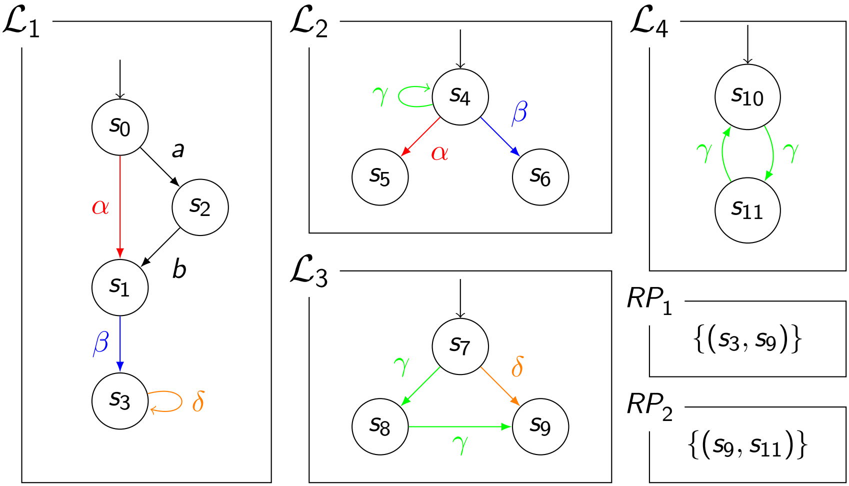
Lazily solving a first problem (with a solution)



Lazily solving a first problem (with a solution)



Lazily solving a second problem (with no solution)



III. Experimental results

LaRA: Lazy Reachability Analyzer

About LaRA

- About 500 lines of Haskell code

Implementation choices

- Immediately take full sets of paths
- Add only one automaton at a time

Try it!

lara.rts-software.org

The rivals

Three tools

- PMC: Partial model checking[1]
- On the fly model checking with CADP[2]
- LoLA: Model checking Petri nets[3]

Preliminary results

LoLA clearly outperforms the other tools on the particular problems we consider

[1] Lang and Mateescu. Partial Model Checking Using Networks of Labelled Transition Systems and Boolean Equation Systems. LMCS, 2013.

[2] `http://cadp.inria.fr/`

[3] `http://service-technology.org/lola/`

Benchmarks

Selected from a standard set of benchmarks[4].

Model	Description	Size	Property	Verified?
Cyclic	Milner's cyclic scheduler.	Number of tasks.	One task in two in waiting state together.	Yes.
DAC	Divide and conquer computation.	Maximal number of processes.	A process can finish the task alone.	Yes.
Philo	Dinning philosophers.	Number of philosophers.	One philosopher in two can eat together.	Yes for even sizes. No for odd sizes.
PhiloDico	Variation of Philo.	idem.	idem.	idem.
PhiloSync	Variation of Philo.	idem.	idem.	idem.

[4] Corbett. Evaluating Deadlock Detection Methods for Concurrent Software. IEEE Trans. Software Eng. 1996.

Test setting

- Runtime comparison
- Problems from size 5 to 50000
- 24 Core computer with 128GB of memory
- 20 minutes time limit for each instance of each problem

Promising results

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	

To conclude

Conclusion and future work

What we have done

- An “as generic as possible” algorithm for lazy reachability analysis in distributed systems
- An early prototype giving promising results

What we are doing now

- Distributed systems with time (i.e. networks of timed automata)
- Compute incomplete partial products in our prototype

What we plan to do next

- Parametric timed systems
- Parallel implementation