Multi-core On-the-fly Saturation

Tom van Dijk Jeroen Meijer Jaco van de Pol

TACAS 2019

Multi-core On-the-fly Saturation

Tom van Dijk Jeroen Meijer Jaco van de Pol

TACAS 2019

Overview

- TACAS 2001: Saturation
  original paper
- TACAS 2003: Saturation Unbound
  with on the fly transition learning
- QEST 2004: Saturation NOW
  network of workstations
- CAV 2007: Parallelising symbolic state-space generators
  multi-core, using work-stealing

Overview

- TACAS 2001: Saturation
  original paper
- TACAS 2003: Saturation Unbound
  with on the fly transition learning
- QEST 2004: Saturation NOW
  network of workstations
- CAV 2007: Parallelising symbolic state-space generators
  multi-core, using work-stealing
- PDMC 2009:

"Parallel symbolic state-space exploration is difficult,
but what is the alternative?"

Overview

- TACAS 2001: Saturation
  original paper
- TACAS 2003: Saturation Unbound
  with on the fly transition learning
- QEST 2004: Saturation NOW
  network of workstations
- CAV 2007: Parallelising symbolic state-space generators
  multi-core, using work-stealing
- PDMC 2009:

  "Parallel symbolic state-space exploration is difficult, but what is the alternative?"
- TACAS 2019: Multi-core On-the-fly Saturation (this work)
  almost 8× faster with 16 cores

Outline

Background

- On-the-fly model checking
- Using decision diagrams
- Saturation algorithm for transition relations
- The parallel decision diagram package Sylvan

Contribution

- How we parallelize saturation
- The tools that are available
- Our reproducible experiments

On-the-fly model checking

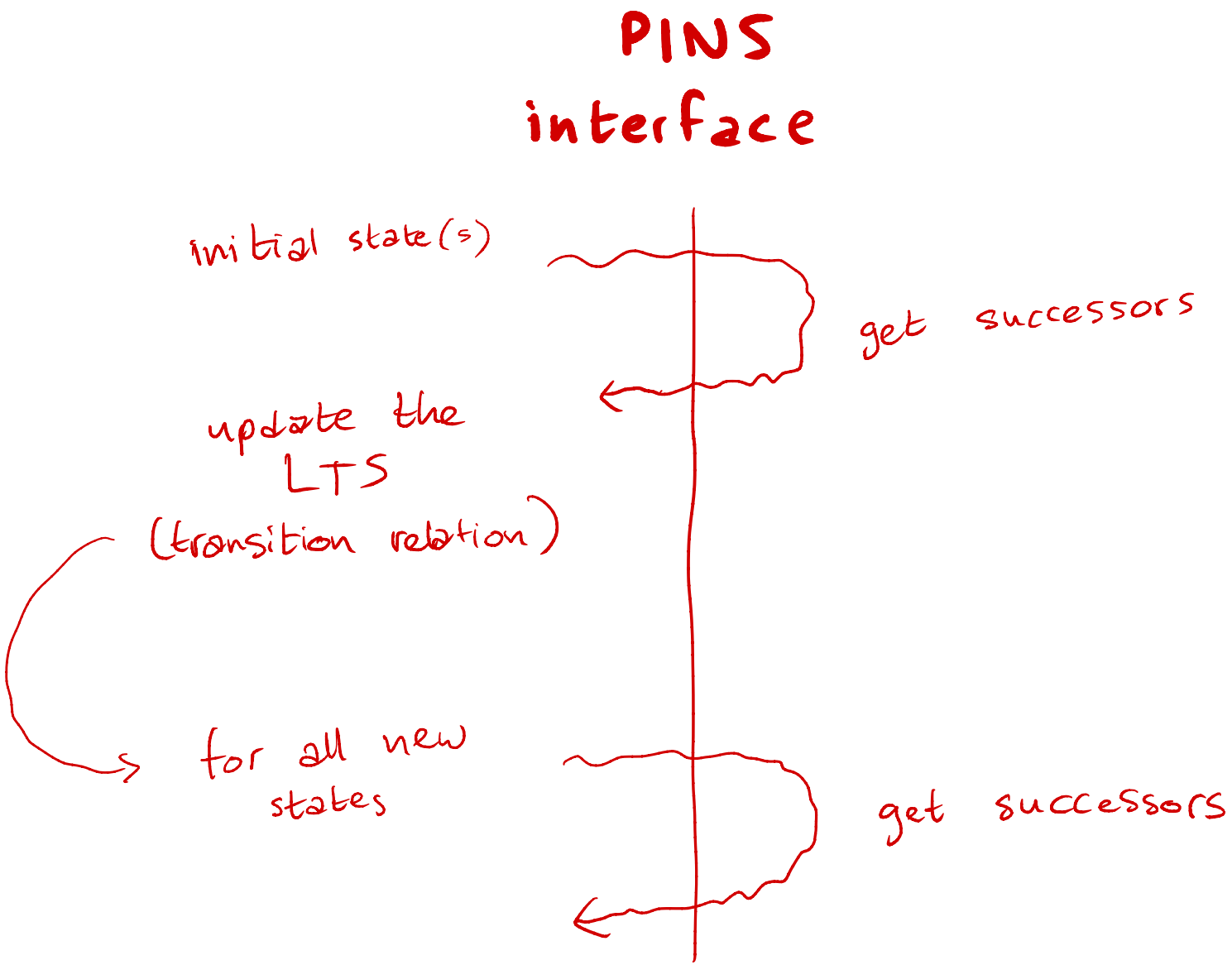
So what is on-the-fly model checking?

LTSmin: a backend talks to a model via a Next-State(s) interface

- Start with some initial state and no transitions
- Ask the model for successors
- Update transition relation with every new transition
- Ask the model for more successors

- Allows e.g. checking safety properties, LTL, etc, "on-the-fly"

On-the-fly model checking



On-the-fly model checking

Partitioned transition relation

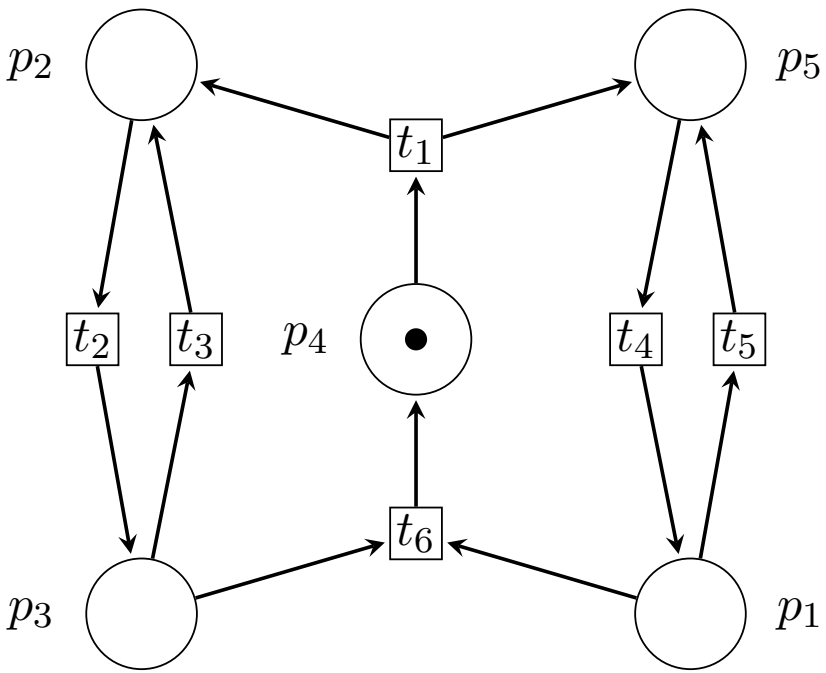
- Often transitions are local and independent
- Example: Petri nets, composed systems, etc
- Disjunctive partition into multiple transition relations
- "Short" transitions only affect some variables

LTSmin implements a "Partitioned Next-State Interface" and models provide LTSmin with a dependency matrix relating state variables and transitions

Advantage: few Next-State calls to learn exponentially larger model

On-the-fly model checking

Example Petri net



$$
\begin{array}{c} \\ t_1 \\ t_2 \\ t_3 \\ t_4 \\ t_5 \\ t_6 \end{array}
\begin{array}{c}
\begin{array}{ccccc} p_1 & p_2 & p_3 & p_4 & p_5 \end{array} \\
\left[\begin{array}{ccccc}
0 & \mathbf{1} & 0 & \mathbf{1} & \mathbf{1} \\
0 & \mathbf{1} & \mathbf{1} & 0 & 0 \\
0 & \mathbf{1} & \mathbf{1} & 0 & 0 \\
\mathbf{1} & 0 & 0 & 0 & \mathbf{1} \\
\mathbf{1} & 0 & 0 & 0 & \mathbf{1} \\
\mathbf{1} & 0 & \mathbf{1} & \mathbf{1} & 0
\end{array}\right]
\end{array}
$$

- Each transition is local and independent
- Few NEXT-STATE calls to learn all reachable transitions

On-the-fly model checking

LTSmin

- On-the-fly compute product with automata of specifications
- On-the-fly check safety and reachability
- LTSmin support various languages and backends via the interface
  - Promela, Petri nets (PNML), DVE, mCRL2, ProB languages, etc

TACAS 2015: Gijs Kant, Alfons Laarman, Jeroen Meijer, Jaco van de Pol, Stefan Blom and Tom van Dijk, LTSmin: High-Performance Language-Independent Model Checking

Model checking and sets

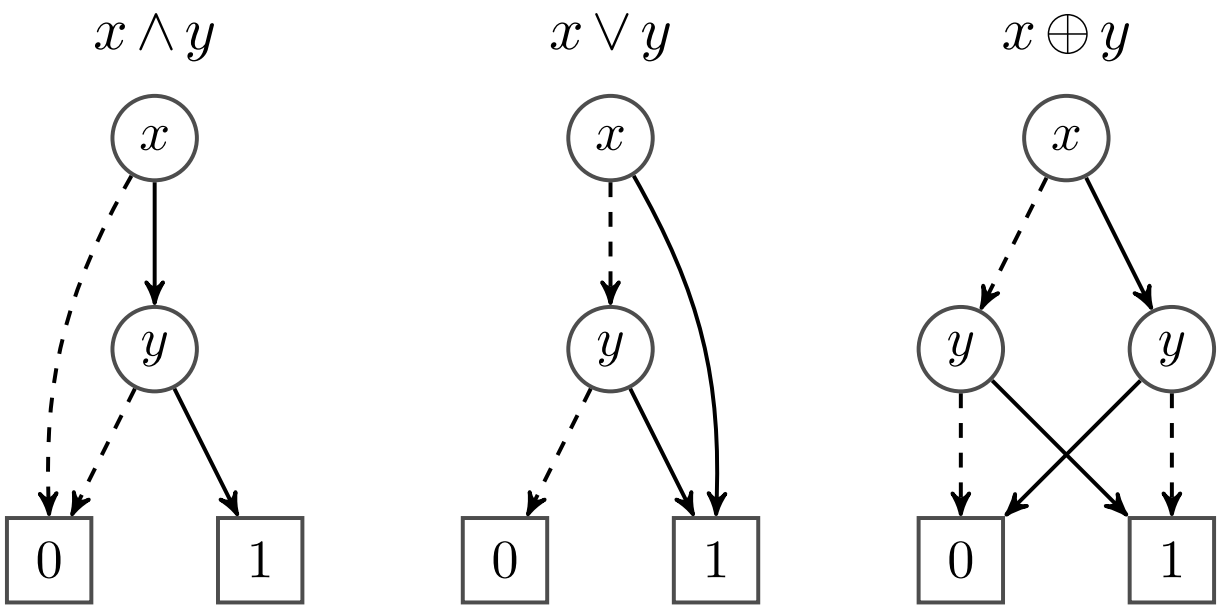
- States and transitions are sets of Boolean vectors $\mathbb{B}^k$ and $\mathbb{B}^{2k}$
- State $x \in \mathbb{B}^k$ is in the set if it is reachable
- Transition $(x, y) \in \mathbb{B}^{2k}$ is in the set if $x \to y$ is a transition

- Set of reachable states $\mathcal{S}(x)$
- Transition relations: $\mathcal{T}_1(r_1, w_1')$ $\mathcal{T}_2(r_2, w_2')$ ... $\mathcal{T}_M(r_M, w_M')$

With x all state variables, and r_i the "read" variables of transition i, w_i' the "write" variables

Binary decision diagrams

- A BDD is a directed acyclic graph encoding a $\mathbb{B}^k \to \mathbb{B}$ function or a $S \subseteq \mathbb{B}^k$ set
- Every node has label x and two children then and else
- Semantics: **if** x **then** follow then **else** follow else
- Paths represent valuations of $\mathbb{B}^k$
- For sets: path to $\boxed{1}$, then valuation is in the set

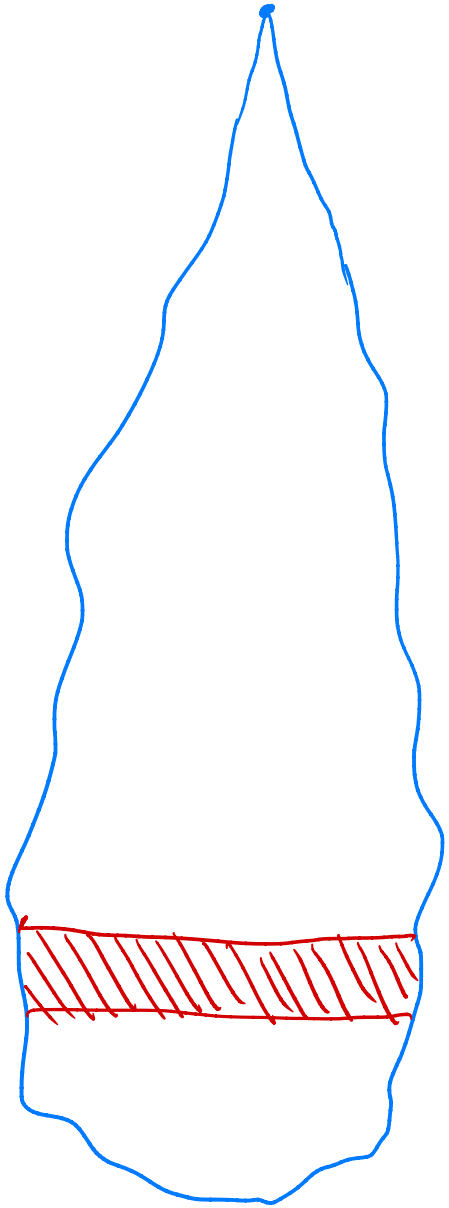


Saturation

Motivation

- Transitions are local, only affect a part of the BDD
- Changing a node affects all ancestors
- Possibly waste of effort to recompute these ancestors!

Example of a transition that only affects a part of the BDD



BDD of the reachable states

} Affected by the local transition

Saturation

State space exploration strategies

- Breadth-first: apply all transition relations once to current states, add all results to set of states
- Chaining: apply transition 1 and add to set of states, apply transition 2 and add to set of states, etc
- (and learn new transitions before applying each transition relation)

Saturation

State space exploration strategies

- Breadth-first: apply all transition relations once to current states, add all results to set of states
- Chaining: apply transition 1 and add to set of states, apply transition 2 and add to set of states, etc
- (and learn new transitions before applying each transition relation)

Saturation strategy [Gianfranco Ciardo et al, since 2001]

- Always apply "deepest" transition until "saturated"
- Whenever we find new states, saturate deeper transitions first
- Saturate BDD nodes in-situ
- Considered an "optimal strategy"

Our solution

How we parallelize saturation

- SIMPLIFY the saturation idea:
  - no in-situ updates of BDD nodes
  - no mutual recursion of "saturate" and "fire event"
  - implement a single BDD operation "saturate"
- Leverage the multi-core BDD framework Sylvan
  - Already parallelizes many BDD operations
    - relational product
    - other popular BDD operations required for model checking
  - Already offers all the infrastructure for new BDD operations
- Implement for standard BDDs but also for LDDs (multi-way decision diagrams implemented as linked list structures)

Parallel BDDs

Sylvan

- `https://www.github.com/trolando/sylvan`
  (google "github sylvan", mirror at `utwente-fmt` repository)
- BDD but also framework for MTBDD with any type of terminal
  For example in probabilistic model checkers Storm, IscasMC, ePMC, with floating points, rational numbers, parameterised functions, etc
- Can also extend for other types of decision diagrams like LDDs
- Implements all common MTBDD operations internally parallelized
  Your program is sequential, automatically parallelized
- Relies on the work-stealing framework Lace to do scalable fine-grained load balancing (every suboperation is a task)
- Main datastructures (nodes table, operation caches) lock-free scalable
- With Sylvan, parallel saturation is easy!

Algorithm

`Saturate` set of states S given current transition i (of k sorted transitions)

- Leaf cases: return S if $S = 0$, $S = 1$ or no more relations left ($i = k$)
- Check if result is in the cache
- Root BDD node of S: $\langle x, \text{Then}, \text{Else} \rangle$
- Is x accessed by transition i
- Yes?
  - First saturate deeper: $S \leftarrow \texttt{saturate}(S, i+1)$
  - Then apply relation i to S with multi-core operation `relprod`
  - Repeat until no change
- No?
  - Run in parallel: $\texttt{saturate}(\text{Then}, i)$ and $\texttt{saturate}(\text{Else}, i)$
  - Compute new BDD node of the result
- Store final result in the cache

Algorithm

`Saturate` set of states S given current transition i (of k sorted transitions)

- Leaf cases: return S if $S = 0$, $S = 1$ or no more relations left ($i = k$)
- Check if result is in the cache
- Root BDD node of S: $\langle x, \text{Then}, \text{Else} \rangle$
- Is x accessed by transition i
- Yes?
  - First saturate deeper: $S \leftarrow \texttt{saturate}(S, i+1)$
  - Then apply relation i to S with multi-core operation `relprod`
  - Repeat until no change
- No?
  - Run in parallel: $\texttt{saturate}(\text{Then}, i)$ and $\texttt{saturate}(\text{Else}, i)$
  - Compute new BDD node of the result
- Store final result in the cache

Learning: update transition just before running `relprod`

Algorithm

Parallel is easy!

- Task parallelism in Sylvan: spawn 2 tasks, then wait until they are done!
- Implemented with the work-stealing framework Lace
- Scalable parallel datastructures nodes tables and operation cache

Empirical evaluation

Evaluation based on Petri nets of the Model Checking Contest

- Compare with state-of-the-art MEDDLY (sequential)
  (Algorithms are fundamentally different, but are we at least close?)
- Measure parallel speedup on 16 cores and 48 cores
- Compare with/without learning (on-the-fly or offline)
- (and compare with BDDs + compare to chaining and bfs)

Tools

- LTSMIN: on-the-fly transition learning; using LDDs
- LDDMC: offline (pre-learned) using LDDs
- MEDMC+MEDDLY: offline non-parallel version of Babar, Miner
- (and BDDMC + implementations of LDD chaining and bfs)

Empirical evaluation

Procedure

- Take Petri nets of MCC 2016 (491 input files)
- Use two good variable orders Sloan and Force for BDD variable ordering
- Use LTSmin with *generous timeout* to generate transition system
  413 out of 982 potential transition systems
- All tools use precisely the same model and variable order
- Compare runtimes only for inputs that all solvers can handle
  301 inputs

- Please reproduce our results: (Apache 2.0)
  `https://www.github.com/trolando/ParallelSaturationExperiments`
  Artifact Evaluation Accepted

- Please download the tools: (Apache 2.0)
  `https://www.github.com/trolando/sylvan`
  `https://www.github.com/utwente-fmt/ltsmin`

Results

Method		Number of solved models with # workers					
		1	2	4	8	16	Any
LTSmin	otf, par	387	397	399	404	407	408
LDDmc	par	388	393	399	402	402	404
Meddly	seq	375	–	–	–	–	375

Table: Number of benchmarks (out of 413) solved within 20 minutes with each method with the given number of workers.

Results

Method	Order	Total time (sec) with # workers					Total speedup			
		1	2	4	8	16	2	4	8	16
LTSmin	Sloan	1850	1546	698	398	313	1.2	2.7	4.6	5.9
LDDmc	Sloan	932	609	311	194	151	1.5	3.0	4.8	6.2
Meddly	Sloan	572	–	–	–	–	–	–	–	–
LTSmin	Force	2704	1162	712	401	343	2.3	3.8	6.8	7.9
LDDmc	Force	856	602	348	216	180	1.4	2.5	4.0	4.7
Meddly	Force	1738	–	–	–	–	–	–	–	–

Table: Cumulative time and parallel speedups for each method-#workers combination on the models where all methods solved the model in time. These are 301 models in total: 151 models with Force, 150 models with Sloan.

Results

Model (with ldd-sat)	**Order**	**Time (sec)**			**Speedup**	
		1	24	48	24	48
Dekker-PT-015	Sloan	77.3	4.7	2.4	16.3	32.5
PhilosophersDyn-PT-10	Force	273.8	16.8	12.4	16.3	22.1
Angiogenesis-PT-10	Sloan	333.2	28.5	16.5	11.7	20.2
SwimmingPool-PT-02	Force	25.0	2.1	1.4	11.6	17.8
BridgeAndVehicles-PT-V20P10N20	Force	1035.8	101.8	60.7	10.2	17.1
Model (with otf-ldd-sat)						
Dekker-PT-015	Sloan	174.5	7.4	3.3	23.6	52.2
SwimmingPool-PT-07	Sloan	1008.0	69.2	42.0	14.6	24.0
SmallOperatingSystem-PT-MT0256DC0064	Sloan	957.3	52.9	40.0	18.1	23.9
Kanban-PT-0050	Sloan	940.6	78.7	48.9	11.9	19.2
TCPcondis-PT-10	Force	68.4	5.7	3.8	11.9	17.8

Table: Parallel speedup for a selection of benchmarks on the 48-core machine (only top 5 shown)

Conclusions

- Saturation is parallelized in LTSmin and Sylvan
  - Parallel saturation now available for many modeling languages
  - Promela, Petri nets (PNML), DVE, mCRL2, ProB languages, etc
- Online available under Apache 2.0 License
  - `github.com/utwente-fmt/ltsmin`
  - `github.com/trolando/sylvan`
  - `github.com/trolando/ParallelSaturationExperiments`
- Competitive: often as good as or better than Meddly, especially with 2 or more cores
- Scalable: up to 7.9× on-the-fly on a 16-core machine (with the FORCE variable ordering); even some superlinear speedups
- Similar scalability with BDDs, with on-the-fly learning
- Parallel saturation is easy with Sylvan!