UNIVERSITY OF TWENTE.

Formal Methods & Tools.

Scalable Multi-core Model Checking: Technology & Applications of Brute Force part II: Liveness & Timed Systems

Jaco van de Pol

30, 31 October 2014

VTSA 2014, Luxembourg

Table of Contents

Recall LTL

LTL formulae are built using temporal operators

ϕ and ψ are formulae, interpreted over infinite paths

- $\mathbf{X}\,\phi$: ϕ holds in the next state in this path . . . . neXt
- $\mathbf{F}\,\phi$: ϕ holds somewhere in this path . . . . Future
- $\mathbf{G}\,\phi$: ϕ holds everywhere on this path . . . . Global
- $\phi\,\mathbf{U}\,\psi$: ψ holds somewhere on this path, and ϕ holds in all preceding states . . . . Until
- $\phi\,\mathbf{R}\,\psi$: ψ holds as long as ϕ did not hold before . . . . . Releases

$$\phi ::= p \mid \neg\phi \mid \phi \wedge \phi \mid \phi \vee \phi \mid \mathbf{X}\,\phi \mid \mathbf{F}\,\phi \mid \mathbf{G}\,\phi \mid \phi\,\mathbf{U}\,\phi \mid \phi\,\mathbf{R}\,\phi$$

Sufficient basis for LTL:

$$\phi ::= p \mid \neg\phi \mid \phi \wedge \phi \mid \mathbf{X}\,\phi \mid \phi\,\mathbf{U}\,\phi$$

Kripke Structures and Paths

Kripke Structures . . . . (just labeled graphs)

A Kripke structure is a tuple $M = (S, S_0, R, AP, L)$, where

- S is a set of states
- $S_0 \subseteq S$ is set of initial states
- $R \subseteq S \times S$ is a (total) transition relation on S
- AP is a set of atomic proposition labels
- $L : S \to \mathcal{P}(AP)$ assigns to each state a set of labels

Infinite Paths . . . . (just sequences of connected states)

- A path π in M is an infinite sequence $(s_0, s_1, s_2, \ldots)$ through the Kripke structure M, so $\forall i.\, s_i \, R \, s_{i+1}$
- Notation: $\pi \in path(s)$ if π starts with s (i.e.: $s_0 = s$)
- Notation: π^i is the suffix from i, i.e.: $(s_i, s_{i+1}, \ldots)$

Formal CTL* semantics: $M, s_0 \models \phi$

Semantics of Path Formulas (given path π)

$$
\begin{array}{rcl}
\pi \models \phi & \Leftrightarrow & \pi(0) \models \phi \quad \text{if } \phi \text{ is a state formula} \\
\pi \models \mathbf{X}\,\phi & \Leftrightarrow & \pi^1 \models \phi \\
\pi \models \mathbf{F}\,\phi & \Leftrightarrow & \text{for some } i \geq 0,\ \pi^i \models \phi \\
\pi \models \mathbf{G}\,\phi & \Leftrightarrow & \text{for all } i \geq 0,\ \pi^i \models \phi \\
\pi \models \phi\,\mathbf{U}\,\psi & \Leftrightarrow & \exists i \geq 0.\, \pi^i \models \psi \wedge \forall j < i.\, \pi^j \models \phi \\
\pi \models \phi\,\mathbf{R}\,\psi & \Leftrightarrow & \forall j \geq 0.\, ((\forall i < j.\, \pi^i \not\models \phi) \Rightarrow \pi^j \models \psi)
\end{array}
$$

Some examples of LTL properties

- Every request will be acknowledged: $\mathbf{G}\,(req \implies req\,\mathbf{U}\,ack)$
- $\mathbf{G}\,\mathbf{F}\,p$: p happens infinitely often
- $\mathbf{F}\,\mathbf{G}\,p$: p is nearly always true
- Note duality: $\neg\mathbf{G}\,\mathbf{F}\,p \iff \mathbf{F}\,\mathbf{G}\,\neg p$

Basic Automata Theoretic Approach

Automata Theoretic Approach

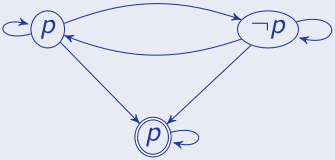
- Kripke Structure M (system); LTL formula ϕ (requirement)
- Construct an automaton A that recognizes violations of ϕ.
- In other words: A accepts a word $\pi \iff \pi \models \neg\phi$
- $M \models \phi$ iff $\mathcal{L}(M) \subseteq \mathcal{L}(\phi)$ iff $M \times A$ accepts $\emptyset$
- Problem: How to deal with infinite words?

Büchi automata for accepting infinite words

- Just like an normal automaton (NFA), with accepting states
- Accept words that hit an accepting state infinitely often

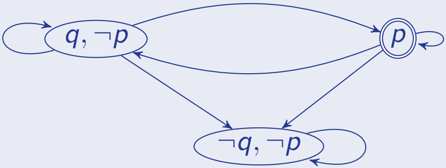
Examples of Büchi automata

almost always: $\mathbf{F}\,\mathbf{G}\,p$



infinitely often: $\mathbf{G}\,\mathbf{F}\,p$

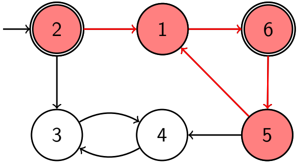
infinitely often with guarantee: $\mathbf{G}\,(q\,\mathbf{U}\,p)$



Model Checking by Accepting Cycles

LTL Model Checking

- A buggy run in a system can be viewed as an infinite word
- Absence of bugs: emptiness of some Büchi automaton
  - $\mathcal{S} \subseteq \mathcal{P}$ iff $\mathcal{S} \cap \overline{\mathcal{P}} = \emptyset$ iff $\mathcal{S} \times \neg\mathcal{P}$ has no accepting cycle
- Graph problem: find a reachable accepting state on a cycle
- Basic algorithm: Nested Depth First Search (NDFS)

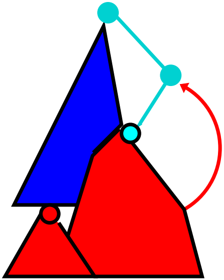


Properties of NDFS

- NDFS runs in linear time
- Inherently depends on post-order
- Post-order is P-complete [Reif'85]
- Not parallelizable (unless P=NC)

Recall: Nested Depth First Search

[CVWY'92] [HOLZMANN'92]



- Blue search: explore graph in DFS order
  - states on the blue search stack are cyan
  - on backtracking from an accepting state:
- Red search: find an accepting cycle
  - exit as soon as the cyan stack is reached
- Linear time, depends on post-order

Blue search

```
procedure dfsBlue(s)
    add s to Cyan
    for all successors t of s do
        if t ∉ Blue ∪ Cyan then
            dfsBlue(t)
        if s is accepting then
            dfsRed(s)
    move s from Cyan to Blue
```

Red search

```
procedure dfsRed(s)
    add s to Red
    for all successors t of s do
        if t ∈ Cyan then
            Exit: cycle detected
        if t ∉ Red then
            dfsRed(t)
```

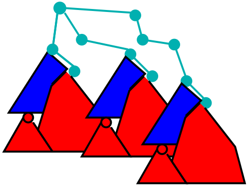
Table of Contents

Simple idea: Swarmed Nested Depth First Search

LAARMAN, LANGERAK, VAN DE POL, WIJS [ATVA'11]

Multi-core Swarmed NDFS

- W workers perform independent random NDFS
  - Visited states are stored in a shared hashtable
  - All workers use their own set of colors (2W bits per state)
  - Speeds up bug hunting only



Blue search

procedure *dfsBlue*(s, i)
    add s to $Cyan[i]$
    for all successors t of s **do**
        if $t \notin Blue[i] \cup Cyan[i]$ **then**
            dfsBlue(t, i)
        if s is accepting **then**
            dfsRed(s, i)
move s from $Cyan[i]$ to $Blue[i]$

Red search

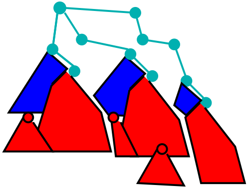
procedure *dfsRed*(s, i)
    add s to $Red[i]$
    for all successors t of s **do**
        if $t \in Cyan[i]$ **then**
            Exit: cycle detected
        if $t \notin Red[i]$ **then**
            dfsRed(t, i)

Multi-core Nested Depth First Search

Laarman, van de Pol,...[ATVA'11][PDMC'11]; Evangelista,L,vdP [ATVA'12]

Multi-core NDFS (several variations)

- Collaboration between NDFS workers
  - Share red and/or blue globally
  - Workers backtrack on parts finished by others
  - Correctness: Complicated to restore post-order
  - Performance: Reasonable scalability



Blue search

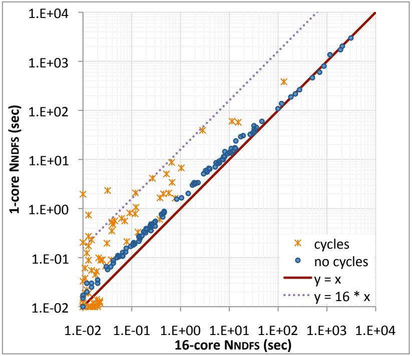
procedure *dfsBlue*(s, i)
  add s to $Cyan[i]$
  for all successors t of s **do**
    if $t \notin Blue \cup Cyan[i]$ **then**
      dfsBlue(t, i)
    if s is accepting **then**
      dfsRed(s, i)
move s from $Cyan[i]$ to $Blue$

Red search

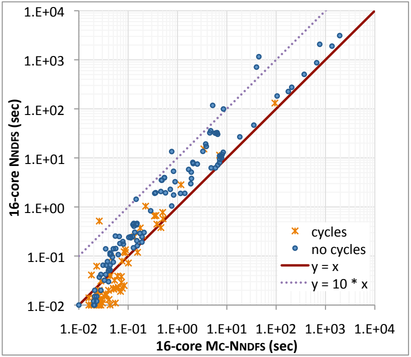
procedure *dfsRed*(s, i)
  add s to Red
  for all successors t of s **do**
    if $t \in Cyan[i]$ **then**
      Exit: cycle detected
    if $t \notin Red$ **then**
      dfsRed(t, i)

Swarmed NDFS versus Parallel NDFS

Experiments from [ATVA'11] on BEEM benchmarks on 16 cores



Swarmed versus
Sequential NDFS



Swarmed versus
Parallel NDFS

Conclusions

- Swarmed NDFS speeds up bug hunting
- Parallel NDFS also speeds up verification

Table of Contents

Nested Depth First Search [Courcoubetis, Vardi, etal.]

procedure DFSblue(s)
    s.blue := true
    for all t ∈ post(s) **do**
        if ¬t.blue **then** DFSblue(t)
    if s ∈ Accepting **then**
        seed := s
        DFSred(s)

procedure DFSred(s)
    s.red := true
    for all t ∈ post(s) **do**
        if t = seed **then** ExitCycle
        if ¬t.red **then** DFSred(t)

Nested DFS

- Blue search
    - Visits all reachable states
    - Starts Red search on accepting states (seed) in post order
- Red Search
    - Finds cycle through seed
    - Visits states at most once
- Linear time, on-the-fly
- Blue is inherently depth-first

Swarmed Multi-core Nested Depth First Search

code for worker i

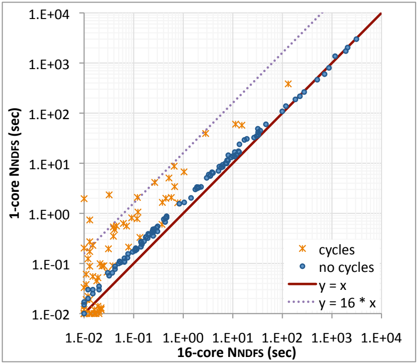
```
procedure DFSblue(s,i)
    s.blue[i] := true
    for all t ∈ post(s) do
        if ¬t.blue[i] then DFSblue(t,i)
    if s ∈ Accepting then
        seed[i] := s
        DFSred(s,i)

procedure DFSred(s,i)
    s.red[i] := true
    for all t ∈ post(s) do
        if t = seed[i] then ExitCycle
        if ¬t.red[i] then DFSred(t,i)
```

Multi-core Swarmed NDFS

- *N* workers perform parallel search independently [G. Holzmann etal.]
- Multi-core: store visited states in a shared hash table [FMCAD 2010, SPIN 2011]
- Scales well in the presence of accepting cycles (bugs)
- Otherwise, all workers traverse the whole graph

Approaches to Parallel LTL Model Checking

Speedup of Swarmed NDFS
(1 versus 16 cores)



[BEEM database]

Alternatives

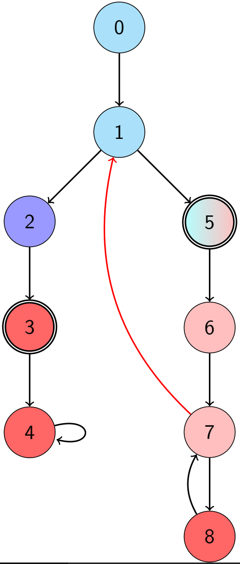
- Swarm verification with NDFS
  - Effective, only for bug finding
- Dual-core NDFS [Holzmann]
  - Red search on 2nd CPU
  - Speedup of at most factor 2
- Red Search as parallel reachability
  - Speedup still ≤ 2: $|G| + |G|/N$
- Can one do better?
  - Post-order is P-Complete, so
  - DFS not efficiently parallelizable
- Breadth-first based:
  - OWCTY, MAP [Brno]
  - Not linear ($|G| \cdot h$), not on-the-fly

New NDFS with Cyan and Pink [à la Schwoon/Esparza]

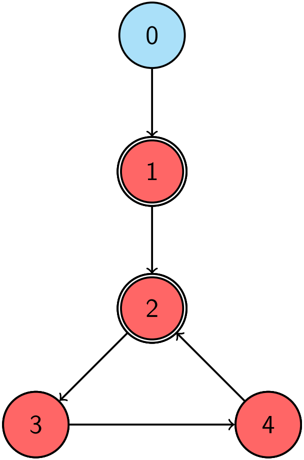
s.bc: white $\to$ cyan $\to$ blue
s.rc: white $\to$ pink $\to$ red

procedure DFSblue(s)
  s.bc := cyan
  for all t $\in$ post(s) **do**
    if t.bc=white **then** DFSblue(t)
  if s $\in$ Acc **then** DFSred(s)
  s.bc := blue

procedure DFSred(s)
  s.rc := pink
  for all t $\in$ post(s) **do**
    if t.bc=cyan **then** ExitCycle
    if t.rc=white **then** DFSred(t)
  s.rc := red



What goes wrong if the DFS order is violated?



What if:

- Red search starts from 1, no Cyan state is encountered
- On the backtrack, the states are colored red
- A new red search starts from 2, but terminates immediately

No accepting cycle is detected!

Parallel NDFS: share the red color (first try)

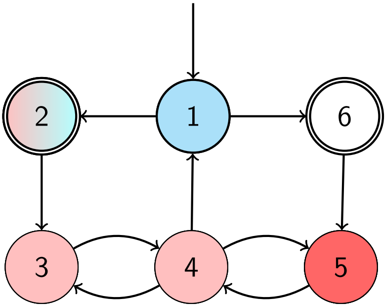
s.color[i] : white $\rightarrow$ cyan $\rightarrow$ blue
s.pink[i], s.red : Boolean

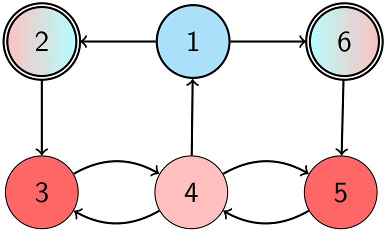
procedure DFSblue(s,i) — pruned by shared red color
    s.color[i] := cyan
    for all t $\in$ post(s) **do**
        if t.color[i]=white and $\neg$t.red **then** DFSblue(t,i)
    if s $\in$ Acc **then** DFSred(s,i)
    s.color[i] := blue

procedure DFSred(s,i) — pruned by shared red color
    s.pink[i] := true
    for all t $\in$ post(s) **do**
        if t.color[i]=cyan **then** ExitCycle
        if $\neg$t.pink[i] and $\neg$t.red **then** DFSred(t,i)
    s.red := true — (unfortunately incorrect)

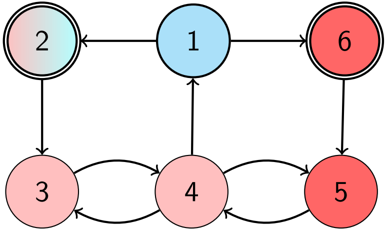
Example: what is the meaning of red? (2 workers)

All accepting cycles contain red:



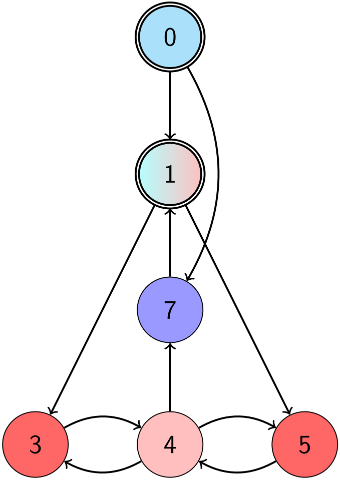


No problem: path pink→cyan

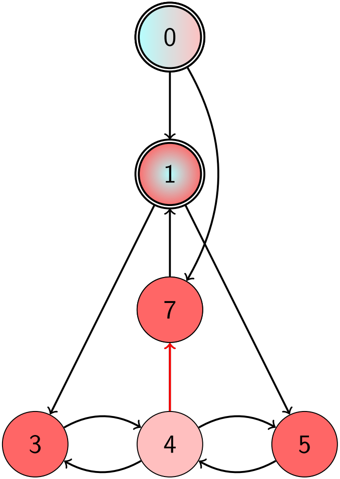


Accepting states on cycles get red:

Synchronisation is necessary: third worker strikes!



Workers 1,2 proceed as before



Worker 3 starts Red search in $1, 0$

No cycle will be detected!

Parallel NDFS: share the red color (correct version)

procedure DFSblue(s,i)
    s.color[i] := cyan
    for all t $\in$ post(s) **do**
        if t.color[i]=white and $\neg$t.red **then** DFSblue(t,i)
    if s $\in$ Acc **then** DFSred(s,i)
    s.color[i] := blue

procedure DFSred(s,i)
    s.pink[i] := true
    for all t $\in$ post(s) **do**
        if t.color[i]=cyan **then** ExitCycle
        if $\neg$t.pink[i] and $\neg$t.red **then** DFSred(t,i)
    pink[i] := false
    if s $\in$ Acc **then await** $\forall j$: $\neg$s.pink[j]
    s.red := true

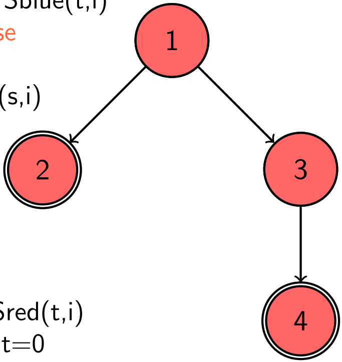
[ATVA 2011]

Optimization 1: Early detection and 2N+1+log(N) bits

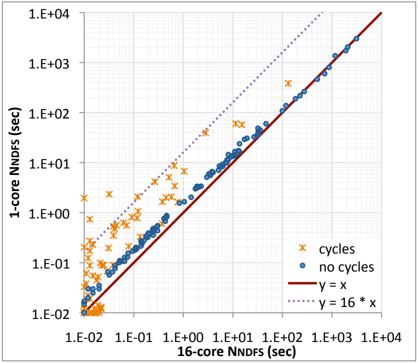
```
procedure DFSblue(s,i)
    s.color[i] := cyan
    for all t ∈ post(s) do
        if t.color[i]=cyan and s or t ∈ Acc then ExitCycle
        if t.color[i]=white and ¬t.red then DFSblue(t,i)
    if s ∈ Acc then s.count++; DFSred(s,i)
    s.color[i] := blue

procedure DFSred(s,i)
    s.color[i] := pink
    for all t ∈ post(s) do
        if t.color[i]=cyan then ExitCycle
        if t.color[i]≠pink and ¬t.red then DFSred(t,i)
    if s ∈ Acc then s.count−−; await s.count=0
    s.red := true
```

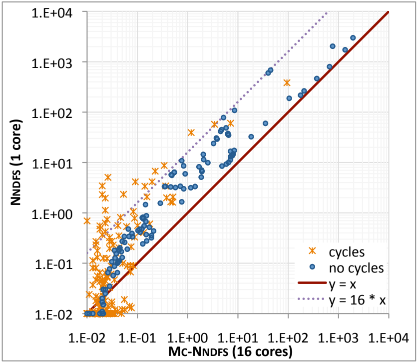
Optimization 2: Sprinkle red paint [Gaiser/Schwoon]

```
procedure DFSblue(s,i)
    s.color[i] := cyan
    all_successors_red := true
    for all t ∈ post(s) do
        if t.color[i]=cyan and s or t ∈ Acc then ExitCycle
        if t.color[i]=white and ¬t.red then DFSblue(t,i)
        if ¬t.red then all_successors_red := false
    if all_successors_red then s.red := true
    else if s ∈ Acc then s.count++; DFSred(s,i)
    s.color[i] := blue

procedure DFSred(s,i)
    s.color[i] := pink
    for all t ∈ post(s) do
        if t.color[i]=cyan then ExitCycle
        if t.color[i]≠pink and ¬t.red then DFSred(t,i)
    if s ∈ Acc then s.count−−; await s.count=0
    s.red := true
```



Swarmed NDFS versus Parallel NDFS

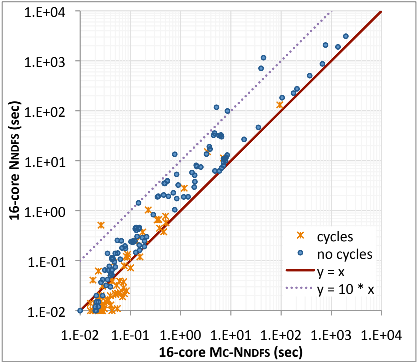




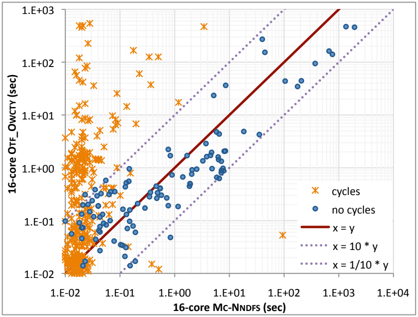
Swarmed NDFS
(1 versus 16-core)

Parallel NDFS
(1 versus 16-core)

OWCTY and Swarmed NDFS versus Parallel NDFS



Swarmed versus Parallel NDFS (both 16 cores)

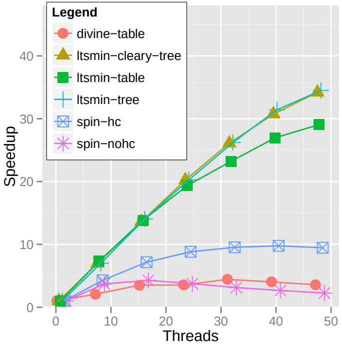


OWCTY versus Parallel NDFS (both 16 cores)

Experiments extended to 48 cores

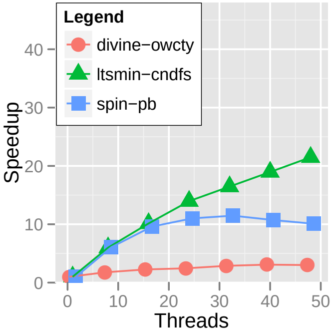
FROM [PDMC'12]. See `fmt.cs.utwente.nl/tools/ltsmin/performance/`

Reachability



Promela: Bakery protocol

LTL model checking



Promela: Elevator controllor

Table of Contents

Interim Evaluation: what did we learn?

Reachability: Implementation matters, keep it simple

- Leave workers alone when possible; load balancing
- Rely on randomness to avoid "duplicate work"
- Careful design of concurrent data structures

LTL model checking

- Previous parallel algorithms (OWCTY) used BFS: $\mathcal{O}(N^2)$
- Now: linear, speedups . . . $P = NC$, or what did we do?
  - $W \to \infty$ versus $W = 48$
  - Worst case $\mathcal{O}(N \cdot W)$, no speedup

Remaining theoretical questions

- Average (randomized) runtime/scalability analysis
- Why doesn't this work for Strongly Connected Components?

Practical Evaluation: Solved multi-core model checking?

Multi-core MC is compatible

- On-the-fly
- Partial-order reduction
- State compression
- Symbolic model checking

Quite general

- Arbitrary state/edge labels
- mCRL2, Promela, DVE, GSPN,
- LLVM, C, xUML, POOSL, ??
- Domain Specific Languages?

Remaining Questions

- Even better speedup – especially for symbolic model checking
- Quite restricted to explicit state model checking
- Infinite state systems? data, recursion, time, BDDs, . . .

Table of Contents

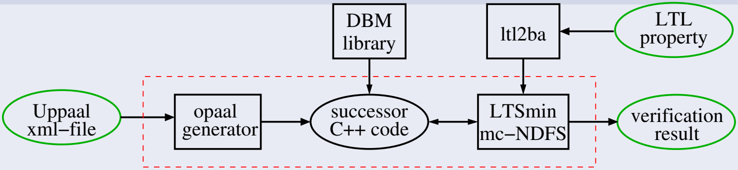
Model checking LTL for Timed Automata

LAARMAN, OLESEN, DALSGAARD, KIM LARSEN, VDPOL [FORMATS'12] [CAV'13]

Handling Timed Automata

- Work with timed zones (DBM) for Timed Büchi Automata
- Checking LTL properties for Uppaal timed automata
  - Use subsumption to prune Nested DFS where possible
  - Multi-core NDFS algorithm for Timed Büchi Automata

Tool support

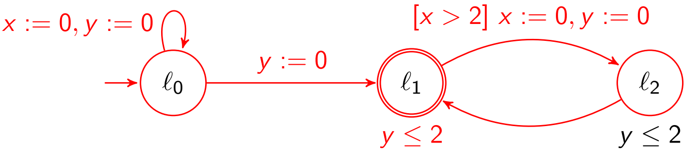


- Open source through OPAAL and LTSMIN
  - `opaal-modelchecker.com/`
  - `fmt.cs.utwente.nl/tools/ltsmin/`

Table of Contents

Timed Büchi Automata

[Alur,Dill'94]



Ingredients

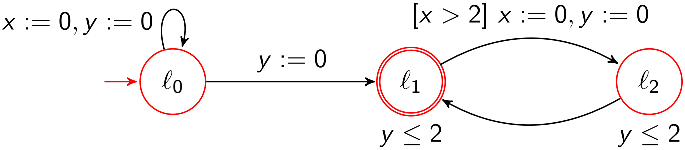
- locations (ℓ_0, ℓ_1, ℓ_2), can be initial or accepting
- transitions, governed by real-valued clocks (x, y)
- timed runs should respect clock guards, resets, invariants

$$\ell_0, \begin{pmatrix}0\\0\end{pmatrix} \xrightarrow{2.7} \ell_0, \begin{pmatrix}0\\0\end{pmatrix} \xrightarrow{1.8} \ell_1, \begin{pmatrix}1.8\\0\end{pmatrix} \xrightarrow{0.5} \ell_2, \begin{pmatrix}0\\0\end{pmatrix} \xrightarrow{2.0} \ell_1, \begin{pmatrix}2.0\\2.0\end{pmatrix} \not\rightarrow$$

Question: is the Büchi language empty? . . . . no counterexample

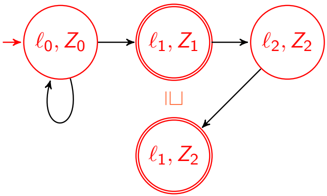
Does a (non-zeno) timed run exist that visits an accepting state infinitely often?

Finite representation: zone abstraction, extrapolation



Finite representation by zones (DBM) [Dill'89] [Daws,Tripakis'98]

- A zone is a set of constraints
- finite by taking into account the lower/upperbounds



$$\begin{aligned} Z_0 &:= y = x \\ Z_1 &:= y \leq x \wedge y \leq 2 \\ Z_2 &:= y = x \wedge y \leq 2 \end{aligned}$$

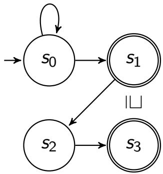
Subsumption:

$Z_2 \subseteq Z_1$, so $(\ell_1, Z_2) \sqsubseteq (\ell_1, Z_1)$

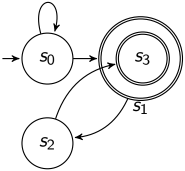
No accepting run!

Subsumption, or inclusion abstraction

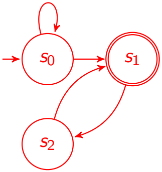
Why explore a state again, if it is subsumed by a previous state?



Zone abstraction



$s_3 \sqsubseteq s_1$



subsumption

Known results

[Behrmann et al'04] [Tripakis'09] [Li'09]

- finite zone abstraction preserves reachability of locations
- finite zone abstraction also preserve Büchi emptiness
- subsumption preserves reachability of locations as well

Open problem

posed in [Tripakis'09]

Is emptiness of Timed Büchi Automata preserved by subsumption? NO

Table of Contents

Extension to Multi-core Reachability . . . [FORMATS'12]

- Timed zones captured in Difference Bound Matrices (DBM)
- For LTSmin, extend discrete state vector s with a pointer to a DBM (s, σ)
- Extend the PINS API with a function $\mathit{Covers}(\sigma, \tau)$
- Hash based on discrete parts, keep list of maximal zones
- Can be generalized to other symbolic domains (lattice model checking)

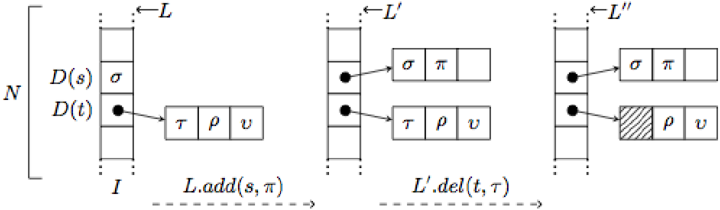


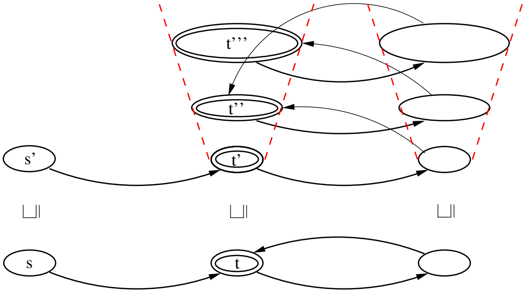
Table of Contents

Analysis of accepting spirals with subsumption

[CAV'13]

$\sqsubseteq$ is a simulation relation:

$$\begin{array}{ccc} s' & \rightarrow & t' \\ \sqcup\!\!\mid & & \sqcup\!\!\mid \\ s & \rightarrow & t \end{array}$$



$\sqsubseteq$ is a finite abstraction

Lemma: If s has an accepting cycle then any $s' \sqsupseteq s$ has it as well

Lemma: If t' has an accepting spiral then t' has an accepting cycle

Preservation of accepting cycles — Proof Sketch

$$\begin{array}{ccccccccccc} s' & \rightarrow^* & t' & \rightarrow^+ & t'' & \rightarrow^+ & \cdots x \cdots & \rightarrow^+ & t''' & \rightarrow^+ & x \\ \sqcup\!\!\mid & & \sqcup\!\!\mid & & \sqcup\!\!\mid & & & & \sqcup\!\!\mid & & \sqcup\!\!\mid \\ s & \rightarrow^* & t & \rightarrow^+ & t & \rightarrow^+ & \cdots\cdots & \rightarrow^+ & t & \rightarrow^+ & t \end{array}$$

Subsumption in Nested Depth First Search [CAV'13]

Blue search — find accepting states in post order

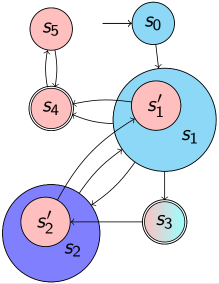
procedure $dfsBlue(s)$
    $Cyan := Cyan \cup \{s\}$
    for all successors t of s **do**
        if $t \notin Blue \cup Cyan \wedge t \not\sqsubseteq Red$ **then** — Prune the blue search
            $dfsBlue(t)$
    if s is accepting **then**
        $dfsRed(s)$
    $Blue, Cyan := Blue \cup \{s\}, Cyan \setminus \{s\}$

Red search — find cycles on accepting states

procedure $dfsRed(s)$ — Postcondition: no accepting spiral reachable
    $Red := Red \cup \{s\}$
    for all successors t of s **do**
        if $t \in Cyan$ $t \sqsupseteq Cyan$ **then** — Accepting spiral found!
            Exit: cycle detected
        if $t \notin Red$ $t \not\sqsubseteq Red$ **then** — Spiral on t would give spiral from Red
            $dfsRed(t)$

Subsumption on Blue is Unsound

Assume we would backtrack on t as soon as $t \sqsubseteq Blue$:

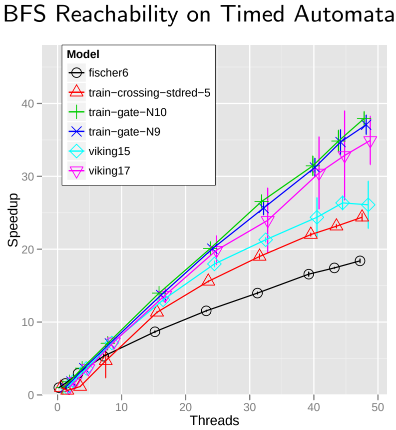


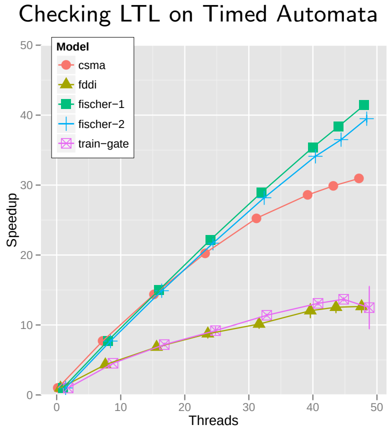
Accepting cycle s_4–s_5 not detected

- The blue search proceeds via s_0, s_1, s_2, then backtracks via s_1 to s_3
- Now since $s_2' \sqsubseteq Blue$, the blue search is pruned at s_3
- $s_3 \in Acc$, so a red search is started: s_3, s_2', s_1', s_4, s_5
- The only accepting cycle s_4–s_5 is erroneously made red
- Note: accepting states are not visited in post-order

Experiments: speedup up to 48 cores

Reachability: [FORMATS'12]. LTL model checking: [CAV'13]





Experiments with OPAAL and LTSMIN – open source

hours ⟶ minutes ⟶ seconds

Literature on LTSmin (liveness - LTL model checking)

LTL model checking

- Alfons Laarman, Rom Langerak, Jaco vd Pol, Michael Weber, A. Wijs, Multi-Core Nested Depth-First Search. .... (ATVA 2011)
- Alfons Laarman, Jaco van de Pol, Variations on Multi-Core Nested Depth-First Search ...... (PDMC 2011)
- Sami Evangelista, Alfons Laarman, Laure Petrucci and Jaco van de Pol, Improved Multi-Core Nested Depth-First Search .... (ATVA 2012)

Timed Automata

- A. Dalsgaard, A.W. Laarman, K.G. Larsen, M. Olesen, J. van de Pol, Multi-Core Reachability for Timed Automata .... (FORMATS'12)
- Alfons Laarman, M. Olesen, A. Dalsgaard, K.G. Larsen, J. van de Pol, Multi-core emptiness checking of timed Büchi automata using inclusion abstraction .... (CAV'13)