An Automata-Theoretic Approach to Software Model Checking

Javier Esparza

Software Reliability and Security Group

Institute for Formal Methods in Computer Science

University of Stuttgart

Automatic verification using model checking

Initiated in the early 80s in USA and France

Exhaustive examination of the state space using (hopefully) clever techniques

Very successful in hardware or “low-level” software

Automatic verification using model checking

Initiated in the early 80s in USA and France

Exhaustive examination of the state space using (hopefully) clever techniques

Very successful in hardware or “low-level” software

ACM’s 2001 Software System Prize awarded to SPIN

Automatic verification using model checking

Initiated in the early 80s in USA and France

Exhaustive examination of the state space using (hopefully) clever techniques

Very successful in hardware or “low-level” software

ACM’s 2001 Software System Prize awarded to SPIN
(other winners: Unix, TeX, PostScript, the World Wide Web, Java, . . .)

Automatic verification using model checking

Initiated in the early 80s in USA and France

Exhaustive examination of the state space using (hopefully) clever techniques

Very successful in hardware or “low-level” software

ACM’s 2001 Software System Prize awarded to SPIN
(other winners: Unix, TeX, PostScript, the World Wide Web, Java, . . .)

Floating-point bugs found in the design of the Pentium 4 processor
[Bentley, DAC2001]

An excerpt from [Bentley 2001]

The FADD instruction had a bug where, for a specific combination of source operands, the 72 bit FP-address was setting the carryout bit to 1 when there was no actual carryout

The FMUL instruction had a bug where, when the rounding mode was set to “round up”, the sticky bit was not set correctly for certain combinations of operand mantissa values, specifically:

$$\begin{aligned} src1[67:0] &:= X*2(i+15)+1*2i \\ src2[67:0] &:= Y*2(j+15)+1*2j \end{aligned}$$

where $i+j=54$ and X, Y are integers that fit in the 68-bit range

Either of these bugs could easily have gone undetected not just in pre-silicon environment but in post-silicon testing also. Had they done so, we would have faced the prospect of a recall similar to the Pentium processor’s FDIV problem in 1994.

Software model checking

Challenge: develop model-checking techniques for ‘higher-level’ software

Three main research questions:

Software model checking

Challenge: develop model-checking techniques for ‘higher-level’ software

Three main research questions:

Integration of the techniques in the system development process (requirements, refinement, testing)

Software model checking

Challenge: develop model-checking techniques for ‘higher-level’ software

Three main research questions:

Integration of the techniques in the system development process (requirements, refinement, testing)

- PathStar [Holzmann, Smith]: Checking Lucent’s PathStar access server
- Slam [Ball, Rajamani et al.]: Checking Windows XP drivers

Software model checking

Challenge: develop model-checking techniques for ‘higher-level’ software

Three main research questions:

Integration of the techniques in the system development process (requirements, refinement, testing)

- PathStar [Holzmann, Smith]: Checking Lucent’s PathStar access server
- Slam [Ball, Rajamani et al.]: Checking Windows XP drivers

Automatic extraction of models from code

Software model checking

Challenge: develop model-checking techniques for ‘higher-level’ software

Three main research questions:

Integration of the techniques in the system development process (requirements, refinement, testing)

- PathStar [Holzmann, Smith]: Checking Lucent’s PathStar access server
- Slam [Ball, Rajamani et al.]: Checking Windows XP drivers

Automatic extraction of models from code

- Work of the abstract interpretation community
- Bandera [Hatcliff et al.]: from Java code to model-checkable models through abstraction/static analysis

Software model checking

Challenge: develop model-checking techniques for ‘higher-level’ software

Three main research questions:

Integration of the techniques in the system development process (requirements, refinement, testing)

- PathStar [Holzmann, Smith]: Checking Lucent’s PathStar access server
- Slam [Ball, Rajamani et al.]: Checking Windows XP drivers

Automatic extraction of models from code

- Work of the abstract interpretation community
- Bandera [Hatcliff et al.]: from Java code to model-checkable models through abstraction/static analysis

Exploration of infinite-state spaces

Integration in the system development process

PathStar: Checking Lucent's Path-Star access server

- One system
- Verification interacts with design (300 versions)
- Highly concurrent, challenging code
- Complex specification (80/200 properties)

Slam: Checking Windows XP drivers

- Many systems
- Post-mortem verification
- Sequential, "straightforward" code
- Simple specification (correct locking/unlocking)

Sources of infinity in software systems

Data manipulation: integers, lists, trees, more general pointer structures, . . .

Control structures: procedures , process creation, . . .

Asynchronous communication: unbounded FIFO queues

Parameters: number of processes, duration of delays . . .

Real-time: discrete or dense domains

Current approach of most of (?) the ISMC community

Model data abstractions of the program by means of (networks of) extended automata

(For the purpose of this talk, all but one source of infinity must be abstracted)

Using the automata theoretic-approach to model checking, reduce the verification problem to reachability or repeated reachability problems

Develop algorithms or semi-algorithms for these problems using symbolic search and accelerations

Reintroduce the abstracted data incrementally by means of predicate abstraction and counterexample-guided abstraction refinement

Extended automata

Automata with transitions guarded by and operating on data structures

Systems	Automata	Data structure	Transition
Procedures	Pushdown automata	stack	$q \xrightarrow[a/ba]{top=a} q'$
Multithreads	(Ext. of) Petri nets	counters	$q \xrightarrow[x_2:=x_2+x_3]{x_1=0} q'$
Timed systems	Timed automata	clocks	$q \xrightarrow[c_1:=0]{c_1-c_2>1} q'$
Protocols	FIFO automata	queues	$q \xrightarrow[l?x]{l\neq\epsilon} q'$

First example: Drawing skylines

```
void m() {
    if (?) {
        s(); right();
        if (?) m();
    } else {
        up(); m(); down();
    }
}
```

```
void s() {
    if (?) return;
    up(); m(); down();
}

main() {
  s();
}
```

Model

```
void s() {
    s0: if (?) s1: return;
    s2: up();
    s3: m();
    s4: down(); s5:
}
```

var st:**stack** of $\{s_0, \ldots, s_5, \ldots\}$

$$q \xrightarrow[s_0/s_1(s_2)]{top=s_0} q \qquad q \xrightarrow[s_1/\varepsilon]{top=s_1} q$$

$$q \xrightarrow[s_2/up_0\, s_3]{top=s_2} q$$

$$q \xrightarrow[s_3/m_0\, s_4]{top=s_3} q$$

$$q \xrightarrow[s_4/down_0\, s_5]{top=s_4} q \qquad q \xrightarrow[s_5/\varepsilon]{top=s_5} q$$

Local variables $(l_1, \ldots, l_k)$ $\longrightarrow$ stack symbols $(s_i, vl_1, \ldots, vl_k)$

Global variables $(g_1, \ldots, g_m)$ $\longrightarrow$ control states $(vg_1, \ldots, vg_m)$

Second example: Fischer's mutex protocol

A simplified version (so that the analysis can be visualized in one slide . . .)

```
var v:{1,2} init 1;
```

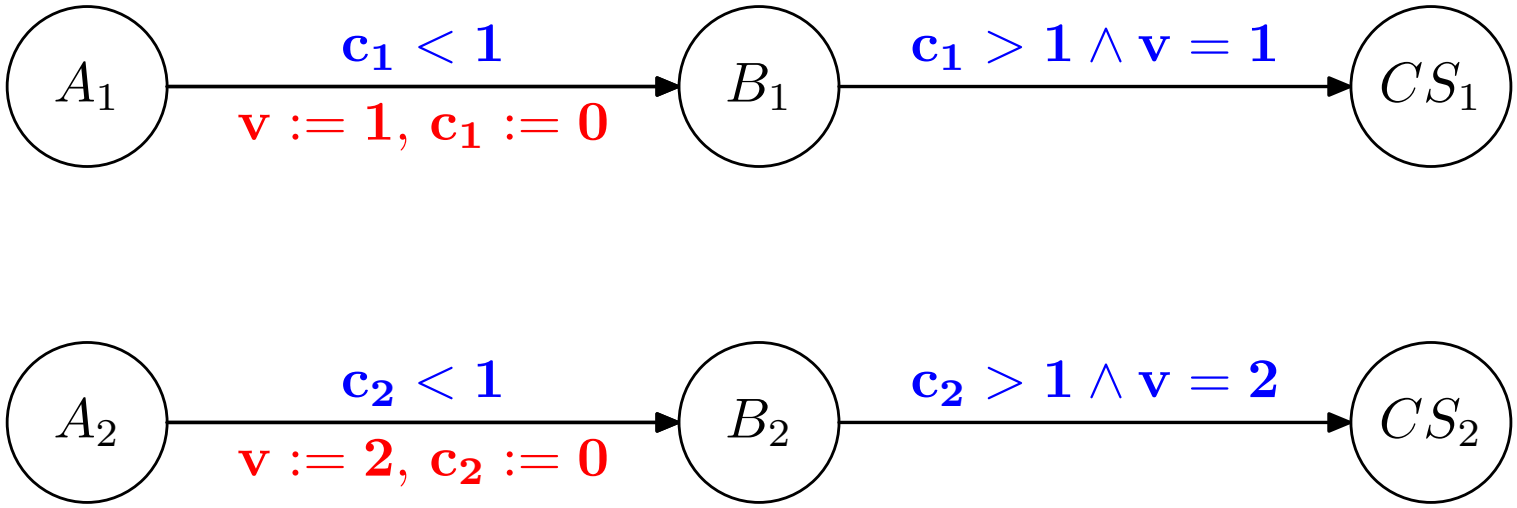
```
delay < 1;
v:= 1;
delay > 1;
if v = 1 then goto cs1
```

```
delay < 1;
v:= 2;
delay > 1;
if v = 2 then goto cs2
```

Model

var $v : \{1, 2\}$ **init** 1

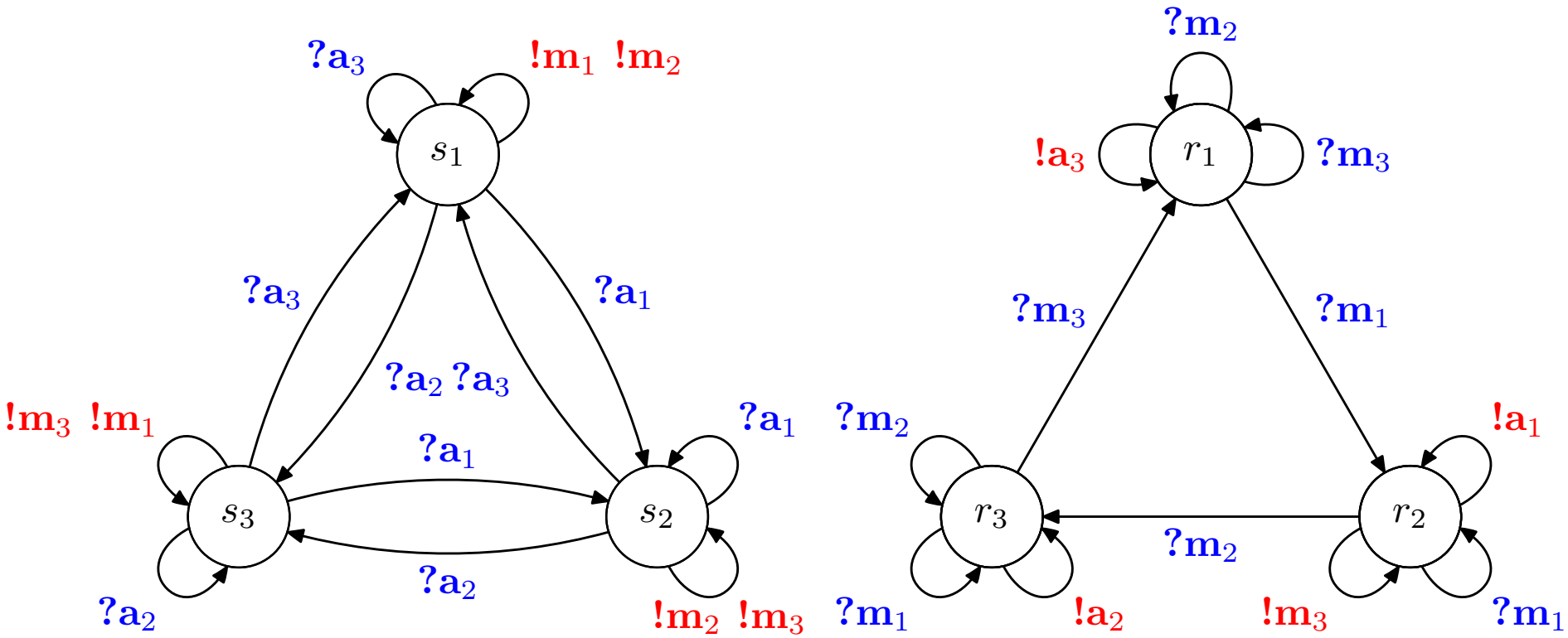
var $c_1, c_2 :$ clock **init** 0



Delays can be modelled with nondeterministic selfloops

Reducible to one single automaton with 9 states of the form (l_1, l_2)

Third example: A sliding window protocol



Other models

Multithread programs $\longrightarrow$ automata extended with counters

- Guards: $c > 0$
- Operations: $c := c + 1$, $c := c - 1$, $c_1 := c_1 + c_2$

Dynamic networks $\longrightarrow$ automata extended with graph transformation rules

- Guards: graph contains fixed subgraph
- Operation: replace the subgraph by another fixed graph

The automata-theoretic approach to specifications

Define the executions of an extended automaton as sequences of states/transitions, hiding the information about variables

Model a property as a finitary (safety) or infinitary (liveness) regular language D of dangerous executions

System S	$\longrightarrow$	Extended automaton $\mathcal{A}_S$
Dang. exec. D	$\longrightarrow$	Automaton $\mathcal{A}_D$
$\mathcal{L}(\mathcal{A}_S) \cap D = \emptyset$	iff	$\mathcal{L}(\mathcal{A}_S \times \mathcal{A}_D) = \emptyset$

So the model-checking problem is reduced to the emptiness problem of extended automata [Vardi, Wolper]

In turn, the emptiness problem is reducible to:

- Reachability

  Given: system S, sets I and F of initial and final configurations of $\mathcal{A}_S$
  To decide: if F can be reached from I,
  i.e., if there exist $i \in I$ and $f \in F$ such that $i \rightarrow^* f$

- Repeated reachability

  Given: System S, sets I and F of initial and final configurations of $\mathcal{A}_S$
  To decide: if F can be repeatedly reached from I,
  i.e. if there exist $i \in I$ and $f_1, f_2, \ldots \in F$ such that $i \rightarrow^* f_1 \rightarrow^* f_2 \cdots$

  Observe: I and F can be infinite

Symbolic search

A general framework for the reachability problem

Let $post(C)$ denote the immediate successors of a (possibly infinite!) set C of configurations

Forward symbolic search

Initialize $C := I$

Iterate $C := C \cup post(C)$ until

$C \cap F \neq \emptyset$; return “reachable”, or

a fixpoint is reached; return “non-reachable”

Backward search defined similarly

Question: when is symbolic search effective?

Symbolic search effective if . . .

. . . a class $\mathcal{C}$ of (possibly infinite) sets of configurations can be found such that:

1. each $C \in \mathcal{C}$ has a finite representation

2. all operations and guard evaluations (i.e., $C := C \cup post(C)$, $C \cap F = \emptyset$, set containment) can be effectively computed

3. any chain $C_1 \subseteq C_2 \subseteq C_3 \ldots$ reaches a fixpoint after finitely many steps

(1)–(2) for semi-algorithm, (3) guarantees termination

Symbolic reachability for timed automata

Two clock vectors $\mathbf{t} = (t_1, \ldots, t_n)$ and $\mathbf{u} = (u_1, \ldots, u_n)$ of a timed automaton are time-equivalent if they satisfy the same conditions of the form

$$x_i \leq n \qquad \text{and} \qquad x_i - x_j \leq n$$

for every n less than or equal to the maximal delay in the syntactic description of the automaton (delays are assumed to be integer)

Two configurations $c = \langle q, \mathbf{t} \rangle$ and $c' = \langle q', \mathbf{u} \rangle$ are equivalent if $q = q'$ and $\mathbf{t}$ and $\mathbf{q}$ are time-equivalent

An equivalence class of configurations is called a region

We choose $\mathcal{C}$ as the powerset of the set of regions

Observe: the number of regions is exponential in the number of clocks and on the number of digits of the maximal delay, but finite

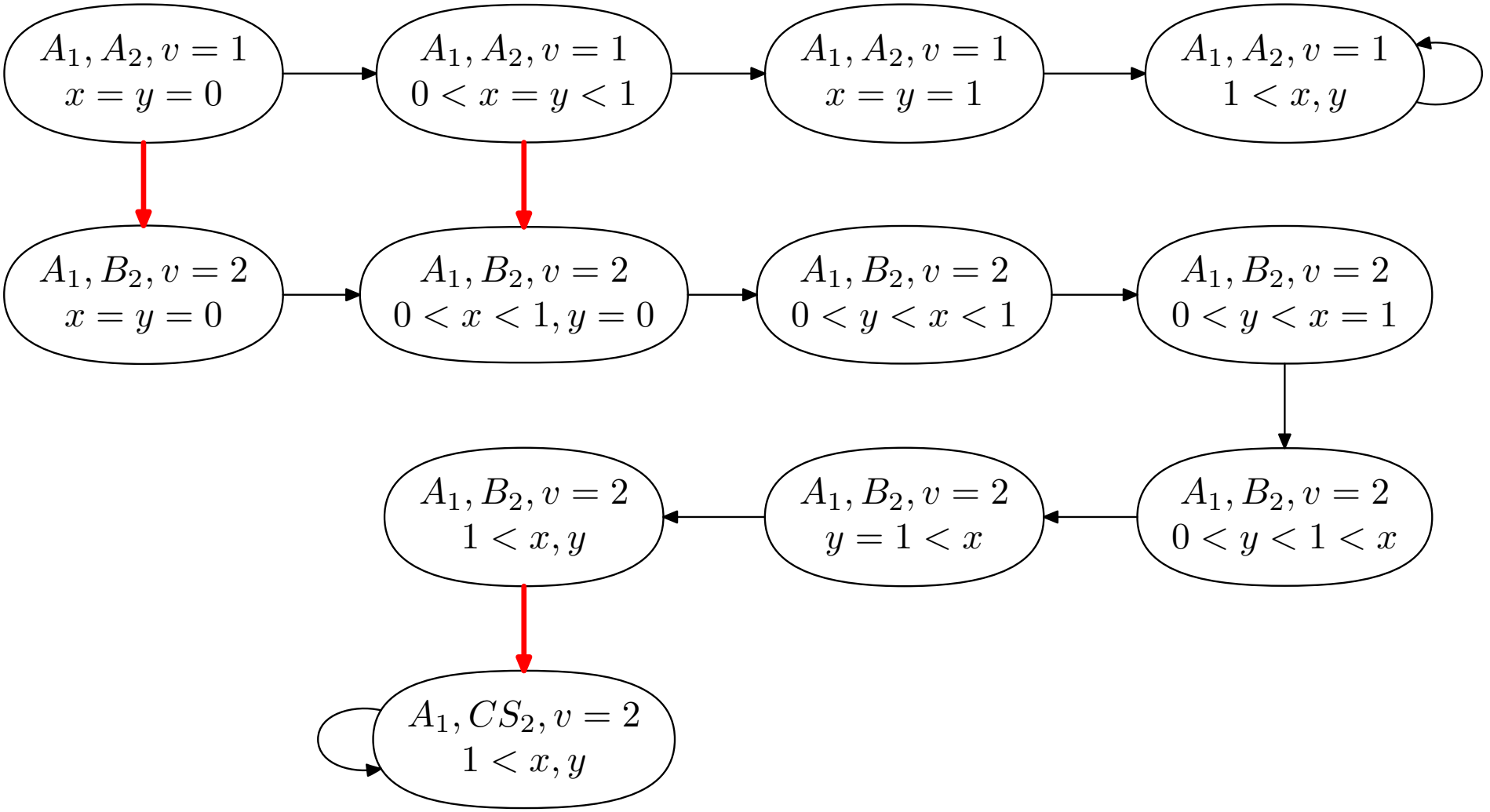
Theorem [Alur, Dill 90]: The powerset of regions satisfies conditions (1), (2), (3)

(1): Regions are finitely represented by their defining equations

(2): If C is a set of regions, then $C \cup post(C)$ is also a set of regions

(3): Every increasing chain of sets of regions reaches a fixpoint, because the number of regions is finite

(One half of) The region graph of Fischer's protocol



Other positive results

Model	Source of inf.	Class $\mathcal{C}$	For./Back.
Timed automata	Time	Regions (or zones)	For./Back.
Ext. Petri nets	Parameters	Upward-closed sets	Backward
Lossy channels	asynchronous comm.	Upward-closed sets	Backward

Timed automata: Alur, Dill, Henzinger, Larsen, Sifakis, Yovine . . . .

Ext. Petri nets: Abdulla, Bouajjani, Delzanno, E. , Finkel, Raskin . . .

Lossy channels: Abdulla, Bouajjani, Cecé, Finkel, Jonsson, Schnoebelen . . .

Symbolic reachability for pushdown automata

Configurations are pairs $\langle q, w \rangle$, where w is a word of stack symbols

Idea: describe (possibly infinite) regular sets of stack words by finite automata

We choose $\mathcal{C}$ as the regular sets of configurations

Symbolic reachability satisfies (1) and (2)

Property (3) fails: not every chain of regular sets reaches a fixpoint

Observe: Since a configuration only has a finite number of successors, during the computation the current regular set C is always finite

However, the fixpoint is regular:

If C is a regular set of configurations, $post^*(C)$ is also regular [Büchi 64]

Accelerations

A loop is a sequence $C \xrightarrow{\sigma} post[\sigma](C)$ such that

$$C \xrightarrow{\sigma} post[\sigma](C) \xrightarrow{\sigma} post[\sigma^2](C) \xrightarrow{\sigma} post[\sigma^3](C) \cdots$$

Examples: $\langle q, \gamma \rangle \xrightarrow{\sigma} \langle q, \gamma v \rangle$ in pushdown automata

$M \xrightarrow{\sigma} M' \geq M$ in Petri nets

Acceleration: given a loop $C \xrightarrow{\sigma} post[\sigma](C)$, replace $post[\sigma](C)$ by

$$X = post[\sigma^*](C) = C \cup post[\sigma](C) \cup post[\sigma^2](C) \cup \ldots$$

Approach:

- find a suitable class of loops such that $post[\sigma^*](C)$ belongs to $\mathcal{C}$
- find algorithms to compute $post[\sigma^*](C)$

An acceleration for pushdown automata

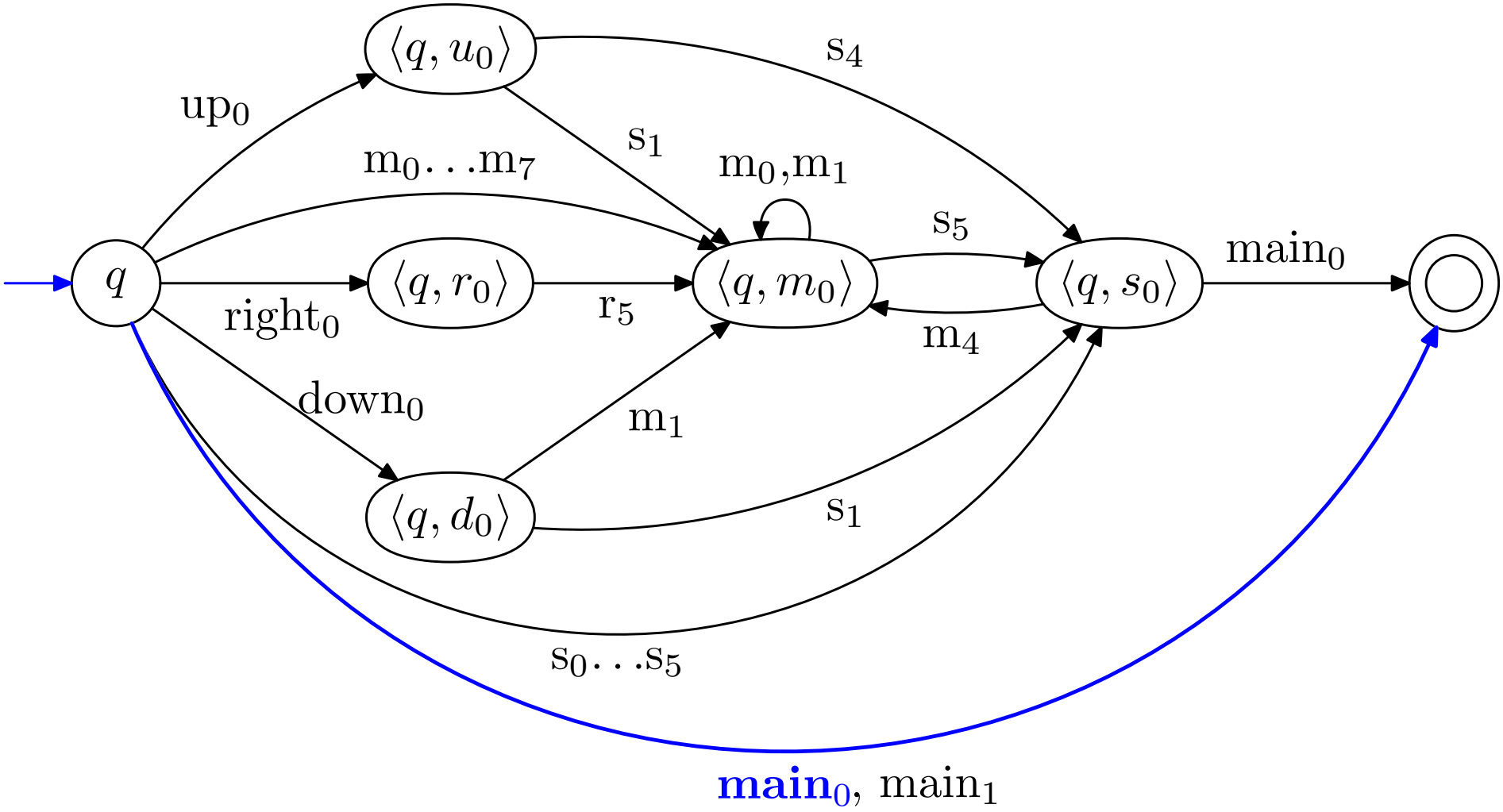
Without acceleration the automata for $C_1 \subseteq C_2 \subseteq C_3 \ldots$ remain acyclic and the number of states can grow unboundedly

Class of loops for the acceleration: all of the form $\langle q, \gamma \rangle \xrightarrow{\sigma} \langle q, \gamma v \rangle$

Special features of pushdown automata lead to a 'magic' acceleration algorithm [Book,Otto 83][E.,Hansel,Rossmanith,Schwoon 2000]

- start with the automaton accepting the initial set of configurations (often a singleton);
- add a fixed number of new states (one per procedure);
- add new transitions following a simple rule and "reusing" the old states

Reachable configurations of the plotter program



Complexity

Complexity of the computation of post*:

- Linear in the size of the control-flow
- Linear in the number of procedures
- A new local boolean variable can multiply the running time by a factor 4
- A new global boolean variable can multiply the running time by a factor 8

Other terminating accelerations

Model	Source of inf.	Class $\mathcal{C}$	Forw./Backw.
PA-systems	Procedures + process creation, no global variables, no sync.	Reg. sets (trees)	Forw./Backw.
PAD-systems	PA + global variables	Reg. sets (trees)	Forw./Backw.

PA-systems: [E., Podelski 00]

PAD-systems: [Bouajjani, Touili 03]

Symbolic reachability for lossy channel systems

Configurations are tuples $\langle q, \mathbf{w} \rangle$ where q is a control state and $\mathbf{w} = (w_1, \ldots, w_n)$ is a vector of queue contents

We choose $\mathcal{C}$ as the set of simple regular expressions (SREs)

Atomic expression:	$(a + \epsilon) \mid (a_1 + \ldots + a_m)^*$
Product:	$e_1 e_2 \ldots e_n$
SRE:	$p_1 + \ldots + p_n$

SREs satisfy conditions (1) and (2), but not (3)

The fixpoint is an SRE, but this time it cannot be effectively computed!

An acceleration for lossy channel systems

Theorem [Abdulla, Bouajjani, Jonsson 98]: For any loop σ of a lossy channel system and any SRE r, the set $post[\sigma^*](r)$ is an SRE that can be computed in quadratic time in the size of r

Use in verification algorithms:

- Preselect a set of loops (in our case, those corresponding to simple cycles in the syntactic description of the lossy channel system)
- Given a set of configurations, compute first the effect of executing each of the loops infinitely often, and then compute for each transition the effect of computing it
- Pray for termination or apply widening techniques losing precision

Channel contents of the sliding window protocol

States	Mess. channel	Ack. channel
s_1, r_1	$(m_2 + m_3)^*(m_1 + m_3)^*(m_1 + m_2)^*$	a_3^*
s_1, r_2	$(m_1 + m_3)^*(m_1 + m_2)^*$	$a_3^* a_1^*$
s_1, r_3	$(m_1 + m_2)^*$	$a_3^* a_1^* a_2^*$
s_2, r_1	$(m_2 + m_3)^*$	$a_1^* a_2^* a_3^*$
s_2, r_2	$(m_1 + m_3)^*(m_1 + m_2)^*(m_2 + m_3)^*$	a_1^*
s_2, r_3	$(m_1 + m_2)^*(m_2 + m_3)^*$	$a_1^* a_2^*$
s_3, r_1	$(m_2 + m_3)^*(m_1 + m_3)^*$	$a_1^* a_2^*$
s_3, r_2	$(m_1 + m_3)^*$	$a_2^* a_3^* a_1^*$
s_3, r_3	$(m_1 + m_2)^*(m_2 + m_3)^*(m_1 + m_3)^*$	a_2^*

Other accelerations without termination guarantee

Model	Source of inf.	Class $\mathcal{C}$	Forw./Backw.
Ext. Petri nets	Parameters	UC sets	Forw.
Perfect FIFO channels	asynchronous comm.	CREs	Forw.
Graph trans.systems	process creation + mobility	NoNameYet	Forw.

Ext. Petri nets: Delzanno, Raskin et al.

Perfect channels: Abdulla, Bouajjani, Finkel, Godefroid, Habermehl, Wolper et al.

Graph transformation systems: Baldan, Corradini, König

Repeated reachability in our examples

Easy for timed automata

Requires some more effort for pushdown automata

Undecidable for lossy channel systems, even though reachability decidable

Counterexample-guided abstraction refinement

Initially: abstract model (all behaviors of the system and many more)

Iterate:

- Check if property holds for the current model
- If it holds, report “PROPERTY HOLDS”

  Otherwise:

  - Get counterexample (model checking) and try to execute it in system
  - If execution succeeds: report “PROPERTY FAILS”

    Otherwise:

    - Identify guard *G* at which execution can’t be continued (symbolic execution!)
    - Add a boolean variable that “keeps track of *G*’s value” (theorem proving)

Based on static analysis/ abstract interpretation ideas, but adds iterative refinement of abstractions

Exploits the main strength of model checking: finding counterexamples

“Fault tolerant” with respect to the theorem prover

At the basis of

- SLAM Microsoft Redmond
- BLAST Berkeley/Lausanne
- MAGIC CMU
- SAL SRI

Tools and experiments on Stuttgart's results

Technique for pushdown automata implemented in

- the MOPED tool [Schwoon], and
- the library on Weighted Pushdown Systems (WPD) [Reps, Schwoon]

and by Chen, Wagner in the MOPS tool

Case studies and applications of MOPED/WPD

- MOPED in Slam
- Used by IST-Project VerifiCard to verify Smart Cards (Java)
- Used at CMU for verification of exception constructs in Java programs
- WPD used as support for the SPKI/SDSI authorization scheme

Le moyen de ennuyer est de vouloir tout dire.

The secret of being a bore is to tell everything.

Voltaire