Rewriting Models of Boolean Programs

Javier Esparza

University of Stuttgart

Joint work with Ahmed Bouajjani

Automatic verification using model-checking

Initiated in the early 80s in USA and France.

- 25 Years of Model Checking Workshop on August 16

Exhaustive examination of the state space using (hopefully) clever techniques to avoid state explosion.

Very successful for hardware or "low-level" software:

- Applied to commercial microprocessors, telephone switches launching protocols, brake systems, or the dutch Delta works.
- Model-checking groups at all major hardware companies.
- ACM Software System Award 2001, Gödel Prize 2000, Kannellakis Awards 1998 and 2005.

Software model-checking

Big research challenge of the 00s: extension to ‘high-level’ software.

Three main research questions:

- Integration of the tools in the software development process.
  - Users trust their hardware but may not trust their software: “post-mortem” verification, “backstage” verification tools . . .
- Automatic extraction of models from code.
- Algorithms for infinite-state systems.
  - Software systems “more often” infinite-state.

The lazy approach to software verification

Construct a sequence of increasingly faithful models that under- or overapproximate the code.

Underapproximations: 32-bit integer $\longrightarrow$ 2-bit integer, 500MB heap $\longrightarrow$ 10B heap.

Overapproximations using predicate abstraction:

- Define a set of predicates over the dataspace (e.g. $x < y$, “list is empty”).
- Partition the dataspace into $2^{\text{number of predicates}}$ abstract values.
- Execute symbolically: one boolean variable per predicate.

The lazy approach to software verification

Construct a sequence of increasingly faithful models that under- or overapproximate the code.

Underapproximations: 32-bit integer $\longrightarrow$ 2-bit integer, 500MB heap $\longrightarrow$ 10B heap.

Overapproximations using predicate abstraction:

- Define a set of predicates over the dataspace (e.g. $x < y$, “list is empty”).
- Partition the dataspace into $2^{\text{number of predicates}}$ abstract values.
- Execute symbolically: one boolean variable per predicate.

Both are boolean programs:

- Same control-flow structure as code + possibly nondeterminism.
- Only one datatype: booleans.
- Conceptually could also take any enumerated type but booleans are the bridge to SAT and BDD technology.

Rewriting models of boolean programs

Boolean programs are still pretty complicated objects:

- Procedures/methods and recursion.
- Concurrency and communication (threads, cobegin-coend sections).
- Object-orientation.

Must be “compiled” into simpler and formal models.

Rewriting models of boolean programs

Boolean programs are still pretty complicated objects:

- Procedures/methods and recursion.
- Concurrency and communication (threads, cobegin-coend sections).
- Object-orientation.

Must be “compiled” into simpler and formal models.

Use rewriting to model boolean programs. In a nutshell:

- Model program states as terms.
- Model program instructions as term-rewriting rules.
- Model program executions as sequences of rewriting steps.

Fundamental analysis problems

Reachability

- But reachability between two states not enough for verification purposes.
- Safety properties often characterized by an infinite set of dangerous states.

Symbolic reachability: Find a finite symbolic representation of the (possibly infinite) set of states reachable or backward reachable from a given (possibly infinite) set of states.

- $pre^*(S)$ denotes the set of predecessors of S.
  (states backward reachable from states in S)
- $post^*(S)$ denotes the set of successors of S.
  (states forward reachable from states in S)

Program for the rest of the talk

Rewriting models for:

- Procedural sequential programs.
- Multithreaded while-programs.
- Multithreaded procedural programs.
- Procedural programs with cobegin-coend sections.

For each of those:

- Complexity of the reachability problem.
- Finite representations for symbolic reachability.

A rewriting model of procedural sequential programs

State of a procedural boolean program: $(\, g, \ell, n, (\ell_1, n_1) \ldots (\ell_k, n_k) \,)$, where

- g is a valuation of the global variables,
- ℓ is a valuation of local variables of the currently active procedure,
- n is the current value of the program pointer,
- l_i is a saved valuation of the local variables of the caller procedures, and
- n_i is a return address.

Modelled as a string $g \, \langle \ell, n \rangle \, \langle \ell_1, n_1 \rangle \, \ldots \, \langle \ell_k, n_k \rangle$

Instructions modelled as string-rewriting rules, e.g. $t \, \langle t, m_0 \rangle \longrightarrow f \langle f \, t \, f, p_0 \rangle \, \langle t, m_1 \rangle$

Prefix-rewriting policy:

$$\frac{u \longrightarrow w}{u \, v \xrightarrow{\; r \;} w \, v}$$

An example

bool function *foo*(ℓ)

f_0: **if** ℓ **then**

f_1: **return** $\mathbf{false}$

else

f_2: **return** $\mathbf{true}$

fi

procedure *main*$()$

global b

m_0: **while** b **do**

m_1: $b := foo(b)$

od;

m_2: **return**

$$
\begin{array}{rcl}
b\,\langle t, f_0\rangle & \longrightarrow & b\,\langle t, f_1\rangle \\
b\,\langle f, f_0\rangle & \longrightarrow & b\,\langle f, f_2\rangle \\
b\,\langle \ell, f_1\rangle & \longrightarrow & f \\
b\,\langle \ell, f_2\rangle & \longrightarrow & t
\end{array}
$$

$$
\begin{array}{rcl}
t\,m_0 & \longrightarrow & t\,m_1 \\
f\,m_0 & \longrightarrow & f\,m_2 \\
b\,m_1 & \longrightarrow & b\langle b, f_0\rangle\,m_0 \\
b\,m_2 & \longrightarrow & \epsilon
\end{array}
$$

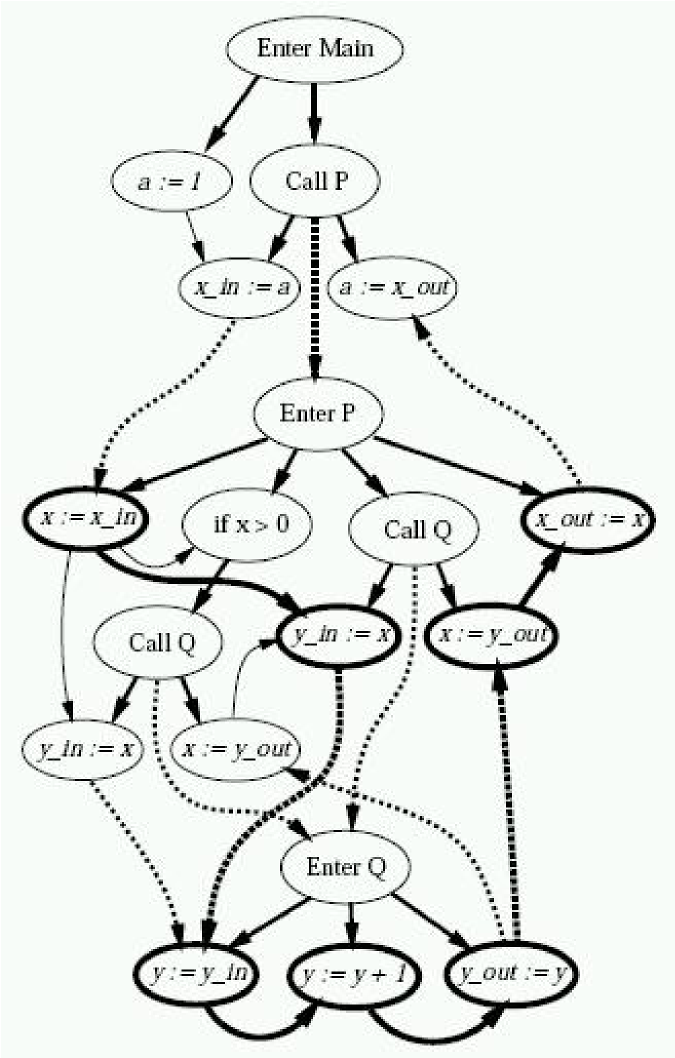
(b and ℓ stand for both t and f)

Comparison with Horwitz, Reps, and Binkley 90

procedure $Q(y)$
q_0: $y := y + 1$
q_1: **return**

procedure $P(x)$
p_0: **if** $x > 0$ **then**
p_1: **call** $Q(x)$
fi;
p_2: **call** $Q(x)$
p_3: **return**

procedure *Main*();
local a
m_0: $a := 1$
m_1: **call** $P(a)$
m_2: **return**



$$\langle y, q_0\rangle \rightarrow \langle y+1, q_1\rangle$$
$$\langle y, q_1\rangle \rightarrow \epsilon$$

$$\langle x^+, p_0\rangle \rightarrow \langle x^+, p_1\rangle$$
$$\langle x^-, p_0\rangle \rightarrow \langle x^+, p_2\rangle$$
$$\langle x, p_1\rangle \rightarrow \langle x, q_0\rangle\langle x, p_2\rangle$$
$$\langle x, p_2\rangle \rightarrow \langle x, q_0\rangle\langle x, p_3\rangle$$
$$\langle x, p_3\rangle \rightarrow \epsilon$$

$$\langle a, m_0\rangle \rightarrow \langle 1, m_1\rangle$$
$$\langle a, m_1\rangle \rightarrow \langle a, p_0\rangle\langle a, m_2\rangle$$
$$\langle a, m_2\rangle \rightarrow \epsilon$$

Prefix string rewriting. From theory . . .

First studied by Büchi in 64 under the name regular canonical systems as a variant of semi-Thue systems.

Theorem: Given an effectively regular (possibly infinite) set S of strings, the sets $\mathit{pre}^*(S)$ and $\mathit{post}^*(S)$ are also effectively regular.

Prefix string rewriting. From theory . . .

First studied by Büchi in 64 under the name regular canonical systems as a variant of semi-Thue systems.

Theorem: Given an effectively regular (possibly infinite) set S of strings, the sets $\mathit{pre}^*(S)$ and $\mathit{post}^*(S)$ are also effectively regular.

Rediscovered by Caucal in 92.

Prefix string rewriting. From theory . . .

First studied by Büchi in 64 under the name regular canonical systems as a variant of semi-Thue systems.

Theorem: Given an effectively regular (possibly infinite) set S of strings, the sets $pre^*(S)$ and $post^*(S)$ are also effectively regular.

Rediscovered by Caucal in 92.

Polynomial algorithms by Bouajjani, E., Maler and Finkel, Willems, Wolper in 97.

- Saturation algorithms: the automata for $pre^*(S)$ and $post^*(S)$ are essentially obtained by adding transitions to the automaton for S.
  (Algorithms for similar models by Alur, Etessami, Yannakakis, and Benedikt, Godefroid, Reps and . . .)

. . . to applications

Efficient algorithms by E., Hansel, Rossmanith and Schwoon in 00.

Theorem (informal): Let Σ, R be the alphabet and set of rules of a 2-normalized prefix-rewriting system system and let A be a “small” NFA over Σ.

An NFA for $post^*(L(A))$ can be constructed in $O(|\Sigma||R|^2)$ time and space.

An NFA for $pre^*(L(A))$ can be constructed in $O(|\Sigma|^2|R|)$ time and $O(|\Sigma||R|)$ space.

. . . to applications

Efficient algorithms by E., Hansel, Rossmanith and Schwoon in 00.

Theorem (informal): Let Σ, R be the alphabet and set of rules of a 2-normalized prefix-rewriting system system and let A be a "small" NFA over Σ.
An NFA for $post^*(L(A))$ can be constructed in $O(|\Sigma||R|^2)$ time and space.
An NFA for $pre^*(L(A))$ can be constructed in $O(|\Sigma|^2|R|)$ time and $O(|\Sigma||R|)$ space.

BDD-based algorithms by E. and Schwoon in 01.

. . . to applications

Efficient algorithms by E., Hansel, Rossmanith and Schwoon in 00.

Theorem (informal): Let Σ, R be the alphabet and set of rules of a 2-normalized prefix-rewriting system system and let A be a "small" NFA over Σ.
An NFA for $post^*(L(A))$ can be constructed in $O(|\Sigma||R|^2)$ time and space.
An NFA for $pre^*(L(A))$ can be constructed in $O(|\Sigma|^2|R|)$ time and $O(|\Sigma||R|)$ space.

BDD-based algorithms by E. and Schwoon in 01.

MOPED model checker by Schwoon in 02.

. . . to applications

Efficient algorithms by E., Hansel, Rossmanith and Schwoon in 00.

Theorem (informal): Let Σ, R be the alphabet and set of rules of a 2-normalized prefix-rewriting system system and let A be a “small” NFA over Σ.
An NFA for $post^*(L(A))$ can be constructed in $O(|\Sigma||R|^2)$ time and space.
An NFA for $pre^*(L(A))$ can be constructed in $O(|\Sigma|^2|R|)$ time and $O(|\Sigma||R|)$ space.

BDD-based algorithms by E. and Schwoon in 01.

MOPED model checker by Schwoon in 02.

MOPS checker by Chen and Wagner in 02.

. . . to applications

Efficient algorithms by E., Hansel, Rossmanith and Schwoon in 00.

Theorem (informal): Let Σ, R be the alphabet and set of rules of a 2-normalized prefix-rewriting system system and let A be a "small" NFA over Σ.
An NFA for $post^*(L(A))$ can be constructed in $O(|\Sigma||R|^2)$ time and space.
An NFA for $pre^*(L(A))$ can be constructed in $O(|\Sigma|^2|R|)$ time and $O(|\Sigma||R|)$ space.

BDD-based algorithms by E. and Schwoon in 01.

MOPED model checker by Schwoon in 02.

MOPS checker by Chen and Wagner in 02.

"Model Checking an Entire Linux Distribution for Security Violations"
by Schwarz et al. at ACSAC 05.

Büchi did it

Moshe Vardi, 25MC Workshop:

Büchi automata, introduced by Büchi in the early 60s to solve problems in second-order number theory, have been translated, unlikely as it may seem, into effective algorithms for model checking tools.

Büchi did it twice

Moshe Vardi, 25MC Workshop:

Büchi automata, introduced by Büchi in the early 60s to solve problems in second-order number theory, have been translated, unlikely as it may seem, into effective algorithms for model checking tools.

This talk:

Regular canonical systems, introduced by Büchi in the early 60s because he liked them, have been translated, unlikely as it may seem, into effective algorithms for software model checking tools.

A rewriting model of multithreaded while-programs

Communication through global variables.

State determined by: $\{\, g, (\ell_0, n_0), (\ell_1, n_1) \ldots (\ell_k, n_k) \,\}$ where

- g is a valuation of the global variables,
- ℓ_i is a valuation of the local variables of the i-th thread, and
- n_i is the value of the program pointer of the i-th thread.

Modelled as a multiset

$$g \parallel \langle \ell_0, n_0 \rangle \parallel \langle \ell_1, n_1 \rangle \parallel \ldots \parallel \langle \ell_k, n_k \rangle$$

Instructions modelled as multiset-rewriting rules, e.g.

$$t\, f \parallel m_0 \rightarrow f\, f \parallel m_1 \parallel \langle f, p_0 \rangle$$

Multiset rewriting, or rewriting modulo assoc. and comm. of $\parallel$.

An example

thread $p()$
p_0: **if** ? **then**
p_1: $b := \mathrm{true}$
else
p_2: $b := \mathrm{false}$
fi;
p_3: **end**

$$
\begin{array}{lcl}
b \parallel p_0 & \rightarrow & b \parallel p_1 \\
b \parallel p_0 & \rightarrow & b \parallel p_2 \\
b \parallel p_1 & \rightarrow & t \parallel p_3 \\
b \parallel p_2 & \rightarrow & f \parallel p_3 \\
b \parallel p_3 & \rightarrow & \epsilon
\end{array}
$$

thread $main()$
global b
m_0: **while** b **do**
m_1: **fork** $p()$
od;
m_2: **end**

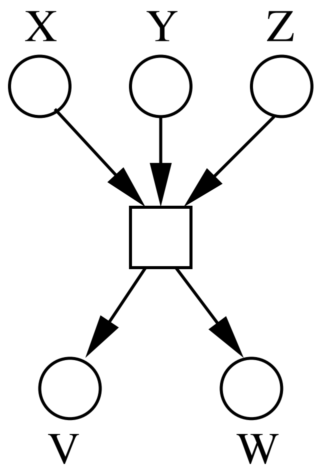
$$
\begin{array}{lcl}
t \parallel m_0 & \rightarrow & t \parallel m_1 \\
f \parallel m_0 & \rightarrow & f \parallel m_2 \\
b \parallel m_1 & \rightarrow & b \parallel m_0 \parallel p_0 \\
b \parallel m_2 & \rightarrow & \epsilon
\end{array}
$$

Multiset rewriting

Theorem [Mayr, Kosaraju, Lipton, 80s]: The reachability problem for multiset-rewriting is decidable but EXPSPACE-hard.

- Equivalent to the reachability problem for Petri nets.
- A place for each alphabet letter.
- A Petri net transition for each rewrite rule.

$$X \parallel Y \parallel Z \longrightarrow V \parallel W$$



Algorithms (not only proofs) quite complicated.

Negative results for $pre^*(\{s\})$ and $post^*(\{s\})$.

Symbolic reachability for *pre** and upward-closed sets

Upward-closed set: if some multiset t belongs to the set, then $t \parallel t'$ also belongs to the set for every t'.

Finitely representable e.g. by the its of minimal elements.

Upward-closed sets capture properties that can be decided by inspecting a bounded number of threads (e.g. mutual exclusion).

Theorem [Abdulla et al. 96]: Given a multiset-rewriting system and an upward-closed set of states S, the set $pre^*(S)$ is upward-closed and effectively constructible.

- Very simple algorithm: compute $pre(S), pre^2(S), pre^3(S) \ldots$.

Extensions applied to multithreaded Java [Delzanno, Raskin, Van Begin 04].

Monadic multiset-rewriting

Monadic rules $\equiv$ no global variables $\equiv$ no communication

Monadic multiset-rewriting

Monadic rules $\equiv$ no global variables $\equiv$ no communication

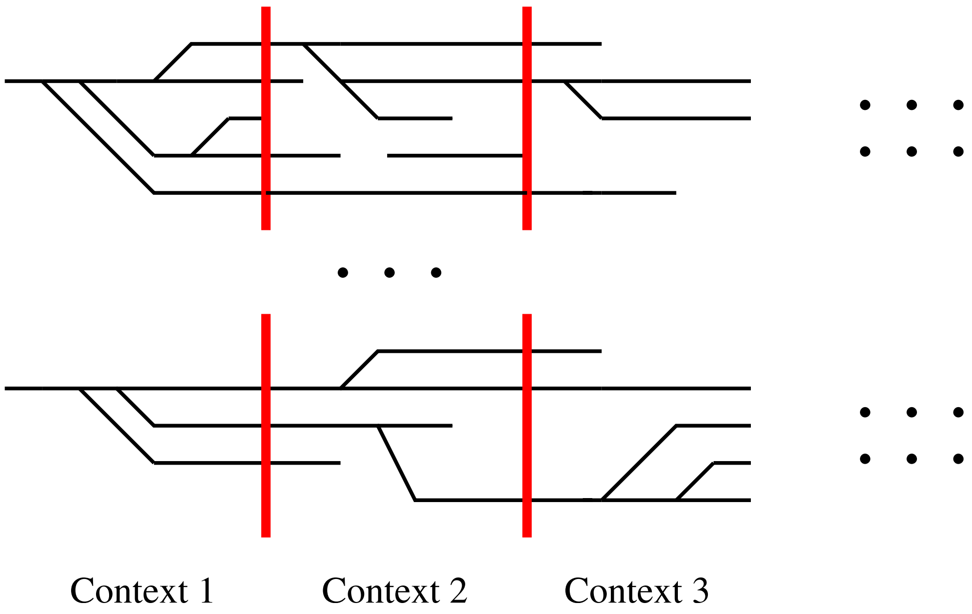
. . . but what are threads that cannot communicate with each other good for?!!!

Monadic multiset-rewriting

Monadic rules $\equiv$ no global variables $\equiv$ no communication

. . . but what are threads that cannot communicate with each other good for?!!!

They are good for underapproximations [Qadeer and Rehof 05]



Reachability

Theorem [Huyhn 85, E.95]: The reachability problem for monadic multiset-rewrite systems is NP-complete.

- Membership in NP not completely trivial.
- Hardness very easy, reduction from SAT:

  A thread for each variable x_i that (a) nondeterministically chooses $l_i \in \{x_i, \overline{x}_i\}$ and (b) spawns a clause thread for each clause satisfied by l_i.

  The thread for a clause does nothing and terminates.

  Formula satisfiable iff there is state at which one thread per clause is active.

Symbolic reachability for semi-linear sets

Semi-linear sets usually defined as subsets of $\mathbb{N}^n$.

- Finite union of linear sets.
- $\{r + \lambda_1 p_1 + \ldots + \lambda_n p_n \mid \lambda_1, \ldots, \lambda_n \in \mathbb{N}\}$.

Language interpretation: "commutative closure" of the regular languages.

Similar properties to regular languages: closure under boolean operations, decidable (but no longer polynomial) membership problem, etc.

Theorem [E.95]: Given a monadic multiset-rewriting system and a semi-linear set of states S, the sets $post^*(S)$ and $pre^*(S)$ are semi-linear and effectively constructible.

Multithreaded procedural programs

Two-counter machines can be simulated by a program with two recursive threads communicating over two global (boolean) variables:

- Tops of the recursion stacks contains two copies of the machine's control point.
- Depths the two recursion stacks model the values of the counters.
- Calls and returns model increasing and decrementing the counters.
- One variable to ensure alternation of moves.
- One variable to keep the two copies of the control point "synchronized".

If communication takes place by rendezvous the two variables are no longer needed: programs without variables are still Turing powerful.

Multithreaded procedural programs

Two-counter machines can be simulated by a program with two recursive threads communicating over two global (boolean) variables:

- Tops of the recursion stacks contains two copies of the machine's control point.
- Depths the two recursion stacks model the values of the counters.
- Calls and returns model increasing and decrementing the counters.
- One variable to ensure alternation of moves.
- One variable to keep the two copies of the control point "synchronized".

If communication takes place by rendezvous the two variables are no longer needed: programs without variables are still Turing powerful.

Communication-free case: [Bouajjani, Müller-Olm and Touili 05]

Communication through nested locks: [Kahlon and Gupta 06]

A rewriting model for the communication-free case

State of a multithreaded procedural program without global variables: multiset $\{s_1, s_2 \ldots, s_k\}$ of states of procedural programs, where

$$s_i = (\ell_{i0}, n_{i0})\,(\ell_{i1}, n_{i1}) \ldots (\ell_{im}, n_{im})$$

Modelled as a string $\# w_k \# w_{k-1} \# \ldots \# w_1$ where

$$w_i = \langle \ell_{i0}, n_{i0} \rangle \, \langle \ell_{i1}, n_{i1} \rangle \, \ldots \, \langle \ell_{im}, n_{im} \rangle$$

Instructions modelled as string-rewriting rules. A new thread is inserted to the left of its creator, e.g.

$$\# \, \langle b, m_1 \rangle \longrightarrow \# \, p_0 \, \# \, \langle f, m_3 \rangle$$

Threads “in the middle” of the string should also be able to “move”: back to ordinary rewriting

$$\frac{u \longrightarrow w}{v_1 \, u \, v_2 \xrightarrow{r} v_1 \, w \, v_2}$$

An example

process $p()$;

p_0:	**if** (?) **then**		$\# p_0$	$\longrightarrow$	$\# p_1$
p_1:	**call** $p()$		$\# p_0$	$\longrightarrow$	$\# p_2$
	else		$\# p_1$	$\longrightarrow$	$\# p_0 \, p_3$
p_2:	**skip**		$\# p_2$	$\longrightarrow$	$\# p_3$
	fi;		$\# p_3$	$\longrightarrow$	$\#$
p_3:	**return**				

process $main()$

m_0:	**if** (?) **then**		$\# m_0$	$\longrightarrow$	$\# m_1$
m_1:	**fork** $p()$		$\# m_0$	$\longrightarrow$	$\# m_2$
	else		$\# m_1$	$\longrightarrow$	$\# p_0 \# m_3$
m_2:	**call** $main()$		$\# m_2$	$\longrightarrow$	$\# m_0 \, m_3$
	fi;		$\# m_3$	$\longrightarrow$	$\# \epsilon$
m_3:	**return**		$\# \#$	$\longrightarrow$	$\#$

Analysis

Theorem [BMOT05]: For every effectively regular set S of states, the set $pre^*(S)$ is regular and a finite-state automaton recognizing it can be effectively constructed in polynomial time.

- Similar to pre^* for monadic string-rewriting [Book and Otto 93].

Theorem [BMOT05]: For every effectively context-free set S of states, the set $post^*(S)$ is context-free and a pushdown automaton recognizing it can be effectively constructed in polynomial time.

Counterexample to regularity: P that spawns a copy of Q and calls itself.

The number of threads is equal to the depth of the recursion.

Reachability set: $\{(\#q)^n \#p^{(n+1)} \mid n \geq 0\}$.

Cobegin-coend sections

Difference with threads: implicit synchronization induced by the coend.

- “Threads have to wait for its siblings to terminate.”
- Corresponds to calling procedures in parallel.

Rewriting model only works well for the communication-free (monadic) case.

States modelled as terms with both $\|$ and $\cdot$ as infix operators e.g

$$(\langle t, p_1 \rangle \parallel q_2) \cdot \langle t\, f, m_1 \rangle$$

Rewriting modulo assoc. of $\cdot$ and assoc. and comm. of $\|$.

This model is called monadic process rewrite systems (monadic PRS) [Mayr 00].

Analysis

Symbolic reachability with commutative hedge automata (CHA) [Lugiez 03].

Theorem [Bouajjani and Touili 05]: Given a monadic PRS, for every CHA-definable set of terms T, the sets $post^*(T)$ and $pre^*(T)$ are CHA-definable and effectively constructible.

Weaker approach: construct not the sets $post^*(T)$ or $pre^*(T)$ themselves, but representatives w.r.t. the equational theory.

Sufficient for control reachability problems.

Theorem [Lugiez and Schnoebelen 98, E. and Podelski 00]:
Let R be a monadic PRS and let A be a bottom-up tree automaton.
One can construct in $O(|R| \cdot |A|)$ time bottom-up tree automata recognizing a set of representatives of $post^*(L(A))$ and $pre^*(L(A))$.

Conclusions

Rewriting concepts can be used to give elegant semantics to programming languages.

- String/multiset rewriting correspond to sequential/parallel computation.
- Monadic/non-monadic rewriting correspond to absence or presence of communication.
- Rewriting modulo useful for combining concurrency and procedures.

Symbolic reachability is the key problem to solve.

Comparison with process algebras:

- Process algebras have a notion of hiding or encapsulation.
- Rewriting much closer to automata theory $\longrightarrow$ algorithms.