Partial-Order Methods for Model Checking

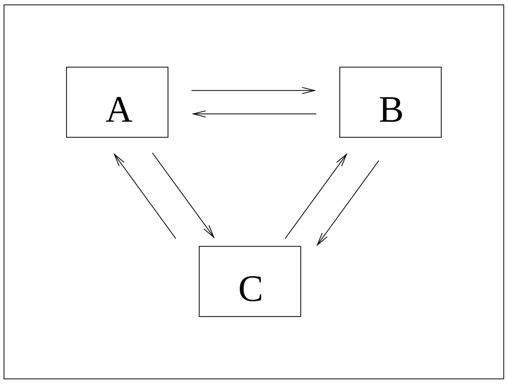
Patrice Godefroid

Bell Laboratories, Lucent Technologies

Overview:

- Model Checking
- Partial-Order Methods
- Main algorithms
- Implementation in SPIN and evaluation
- Implementation in VeriSoft and applications
- Conclusions and other related work

Concurrent Reactive System



Examples: network protocols, interconnected electronic circuits, telephone switches, plant controllers, etc.

Developing, testing and debugging concurrent reactive systems is hard!

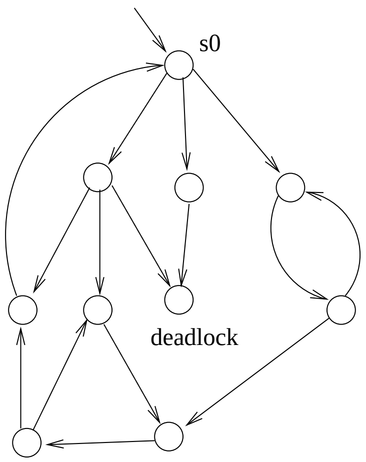
Simple Example: (Dekker's mutual-exclusion algorithm)

Initially: t=1, y1=F, y2=F

```
        Process 1                            Process 2

--> L0: skip;                        --> M0: skip;
    L1: y1:=T;                           M1: y2:=T;
    L2: if (y2=F) then goto L7           M2: if (y1=F) then goto M7
    L3: if (t=1) then goto L2            M3: if (t=2) then goto M2
    L4: y1:=F;                           M4: y2:=F;
    L5: loop until (t=1);                M5: loop until (t=2);
    L6: goto L1                          M6: goto M1
    L7: t:=2;  (critical section)        M7: t:=1;  (critical section)
    L8: y1:=F;                           M8: y2:=F;
    L9: goto L0                          M9: goto M0
```

Model Checking



The joint global behavior of all components can be represented by a global transition system A_G called its **state space**.

If A_G is finite, verification of properties can be done by exploring A_G exhaustively. (Verification means that *all* possible behaviors of the system satisfy a property).

Example of property: absence of deadlocks

Language for temporal properties: **Temporal Logic**

- Behavior described in terms of sequences of states/transitions.
- Example: $G(p \Rightarrow Fq)$ is an LTL formula.

Given a system A_G and a temporal-logic formula ϕ,
"does A_G satisfy ϕ?" = **Model Checking**

The State-Explosion Problem

Example:

Initially: v1=v2= ... =vn=0

Process 1	Process 2		Process n
$\longrightarrow$ s1: v1:=1;	$\longrightarrow$ s2: v2:=1;	— — —	$\longrightarrow$ sn: vn:=1;
s1': stop	s2': stop		sn': stop

$\rightarrow$ n! interleavings
2^n states

$\rightarrow$ **State Explosion**

Motivation of "Partial-Order Methods":
to develop verification methods that avoid the part of the combinatorial explosion due to the modeling of concurrency by interleaving.

Partial-Order Methods: Basic Idea

Exploring *all* interleavings of concurrent transitions is not *a priori* necessary for verification.

Indeed, interleavings corresponding to the same concurrent execution contain related information:

concurrent executions should be viewed as *partial orders* of occurrences of transitions, where concurrent independent (= non-interacting) transitions are left unordered.

Hence, it should be sufficient to explore only *some* of these interleavings for the verification of most properties ϕ.

Which interleavings are required may depend on ϕ.

$\rightsquigarrow$ explore only a part A_R of A_G such that:

$$A_G \text{ sat } \phi \text{ iff } A_R \text{ sat } \phi$$

How to generate A_R?

The Notion of Independent Transitions

Definition: Let $\mathcal{T}$ denote the set of system transitions. Let $D \subseteq \mathcal{T} \times \mathcal{T}$ be a binary, reflexive, and symmetric relation. The relation D is a *valid dependency relation* iff for all $t_1, t_2 \in \mathcal{T}$, $(t_1, t_2) \notin D$ (t_1 and t_2 are independent) implies that for all states $s \in S$ of A_G:

1. if t_1 is enabled in s and $s \xrightarrow{t_1} s'$, then t_2 is enabled in s iff t_2 is enabled in s'
   (independent transitions can neither disable nor enable each other); and
2. if t_1 and t_2 are enabled in s, then there is a unique state s' such that $s \overset{t_1 t_2}{\Rightarrow} s'$ and $s \overset{t_2 t_1}{\Rightarrow} s'$
   (commutativity of enabled independent transitions).

This definition (adapted from [Katz&Peled92]) characterizes the properties of possible "valid" dependency relations for the transitions of a given system.

Tracking (In)dependence

It is not practical to check these two properties for all pairs of transitions in all states s in A_G.

In practice, it is possible to give easily checkable *syntactic conditions* on program code that are *sufficient* for transitions to be *independent*.

Example:

For processes communicating via shared variables, a *sufficient* syntactic condition for transitions t_1 and t_2 in $\mathcal{T}$ to be independent is:

1. t_1 and t_2 do not belong to the same process, *and*
2. the set of variables that are accessed by t_1 is disjoint from the set of of variables that are accessed by t_2.

In practice, the more independence, the better!

Towards More Independence

1. By defining (in)dependence from operations on communication objects.

DEP.	$Write$	$Read$
$Write$	$+$	$+$
$Read$	$+$	$-$

Definition. Two transitions $t_1, t_2 \in \mathcal{T}$ are *independent* if:

1. process(t_1) is different from process(t_2), and
2. $\forall op_1 \in used(t_1)$ and $\forall op_2 \in used(t_2)$, if op_1 and op_2 are two operations on a same object, then op_1 and op_2 are independent.

2. By refining the operations themselves.

Example: for a boolean variable, add the operation $Compl$ such that $Compl(-,0) \to (-,1)$ and $Compl(-,1) \to (-,0)$.

DEP.	$Write$	$Compl$	$Read$
$Write$	$+$	$+$	$+$
$Compl$	$+$	$-$	$+$
$Read$	$+$	$+$	$-$

Towards More Independence (Continued)

3. By using conditional dependency.

Example: a bounded FIFO queue (channel) of size N with three operations:

- $Send(v, v_1 v_2 \ldots v_n) \rightarrow (-, v_1 v_2 \ldots v_n v)$ defined if $n < N$ and $v \in M$,
- $Receive(-, v_1 v_2 \ldots v_n) \rightarrow (v_1, v_2 \ldots v_n)$ defined if $n > 0$,
- $Length(-, v_1 v_2 \ldots v_n) \rightarrow (n, v_1 v_2 \ldots v_n)$ always defined.

The following tables give respectively a constant and a conditional dependency relation between these operations.

DEP.	*Send*	*Receive*	*Length*
Send	+	+	+
Receive	+	+	+
Length	+	+	–

DEP.	*Send*	*Receive*	*Length*
Send	$n < N$	$n = 0$ or $n = N$	$n < N$
Receive	$n = 0$ or $n = N$	$n > 0$	$n > 0$
Length	$n < N$	$n > 0$	–

Using Independence for Defining Equivalence of Transition Sequences

- Two sequences of transitions are *equivalent* if they can be obtained from each other by successively permuting adjacent independent transitions.
- *Equivalence classes* are called Mazurkiewicz traces. The trace containing a sequence of transitions w will be denoted $[w]$.
- The set of transition sequences in $[w]$ correspond to the set of linearizations (interleavings) of a partial order of transition occurrences.

Example: Consider $\mathcal{T} = \{t_1, t_2, t_3\}$, and $D = (\mathcal{T} \times \mathcal{T}) \setminus \{(t_2, t_3), (t_3, t_2)\}$. Then, the sequence $w = t_1 t_2 t_3 t_1$ defines the trace $[w] = \{t_1 t_2 t_3 t_1, t_1 t_3 t_2 t_1\}$.

$$\begin{tikzcd}
& t_2 \arrow[dr] & \\
t_1 \arrow[ur] \arrow[dr] & & t_1 \\
& t_3 \arrow[ur] &
\end{tikzcd}$$

- If $s \overset{w_1}{\Rightarrow} s_1$, $s \overset{w_2}{\Rightarrow} s_2$ in A_G, and $[w_1] = [w_2]$, then $s_1 = s_2$.
- Thus, to determine if a state is reachable by a trace, it is sufficient to explore *only one* transition sequence in this trace!

Thus, for deadlock detection, it is sufficient to explore *only one* interleaving per trace:

- for all deadlocks $d : \exists w : s_0 \stackrel{w}{\Rightarrow} d$
- $\forall w' \in [w] : s_0 \stackrel{w'}{\Rightarrow} d$

How to explore one interleaving per trace?

Selective Search

- **Idea:** Perform a classical search, but instead of executing systematically *all* enabled transitions in each state encountered during the search, execute *only some* of them.
- Only a subset A_R of the global state space A_G is explored.
- Which transitions should be selected?
  Two main (compatible) techniques:

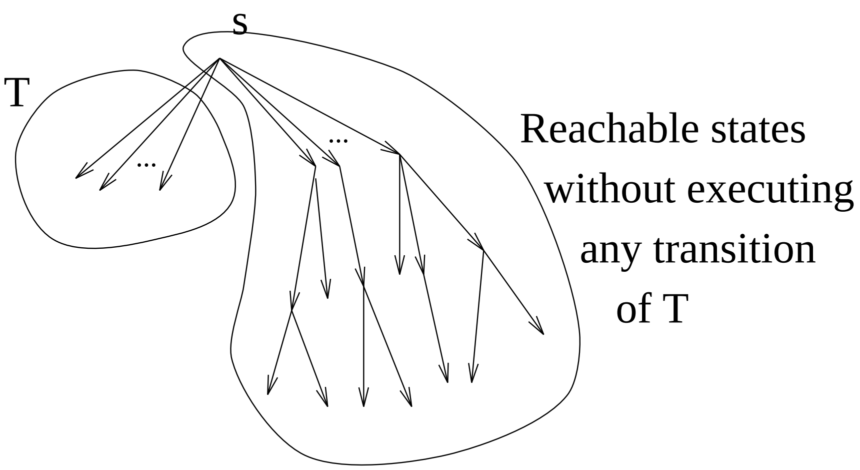
Persistent sets and **Sleep sets**

Persistent Sets

Definition: [Godefroid&Pirottin93]
A set T of transitions enabled in a state s is *persistent in s* if, for all nonempty sequences of transitions

$$s = s_1 \xrightarrow{t_1} s_2 \xrightarrow{t_2} s_3 \ldots \xrightarrow{t_{n-1}} s_n \xrightarrow{t_n} s_{n+1}$$

from s in A_G and including only transitions $t_i \notin T$, $1 \leq i \leq n$, t_n is independent with all transitions in T.



Theorem:
At each visited sate, it is sufficient to explore only the transitions of a nonempty persistent set rather than all enabled transitions in order to detect all deadlocks in the system.

Computing Persistent Sets: General Strategy

- $T = \{t\}$ start with a set that contains a single enabled transition.
- **Repeat**
  - Add all transitions that "can interfere" with some transition in T.
- **Until** no more transition need be added.

Worst case: return all enabled transitions.

"Can interfere" roughly means "is dependent with", and thus

- can disable, or
- can enable, or
- cannot be permuted with.

(Several definitions have been proposed with various level of precision...)

Computing Persistent Sets

Examples:

```
→ a0:  v:=1;         → b0:  x:=0;
  a1:  x:=1;           b1:  x:=2;
  a2:  stop            b2:  stop
```

Persistent set: {"v:=1"}

```
  a0:  x:=1;           b0:  y:=2;
→ a1:  y:=1;         → b1:  x:=2;
  a2:  stop            b2:  goto b0
```

Persistent set: {"x:=2"}

```
→ a0:  x:=1;                → b0:  y:=2;
  a1:  wait until (y=2)       b1:  wait until (x=1)
  a2:  y:=1;                  b2:  x:=2;
  a3:  stop                   b3:  stop
```

Persistent set: {"x:=1"}

Computing Persistent Sets

Several algorithms (proposed somewhat independently):

- "Conflicting Transitions" [Godefroid&Wolper91]
- "Overman's Algorithm" [Overman81]
- "Stubborn Sets" [Valmari88-91]
- "Conditional Stubborn Sets" [Godefroid&Pirottin93]

All these algorithms can be viewed as variants of Valmari's "stubborn set" algorithmic technique.

It can be shown that they all compute persistent sets.

Which Algorithm?

- Heuristic: small persistent sets are preferable;
- may require more computation time
- but efficient implementations exist!

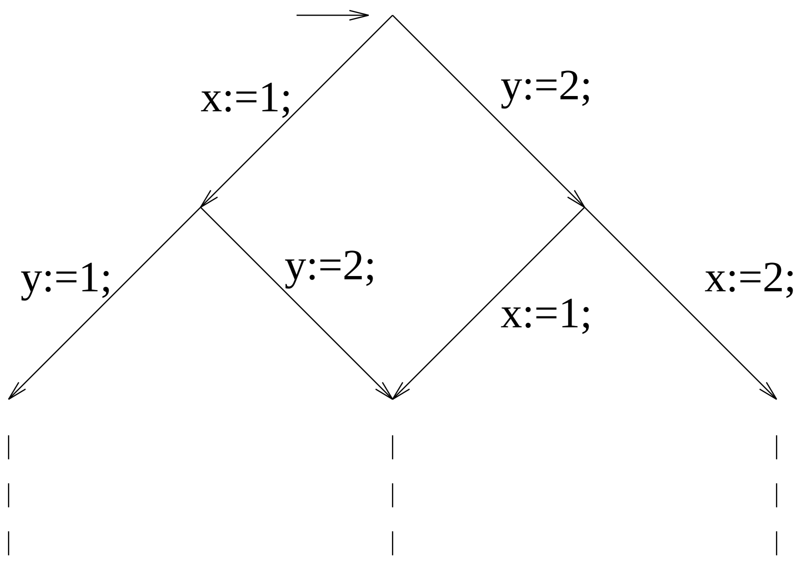
(Note: complexity varies from $O(|enabled|^2)$ to $O(\mathcal{T}^3)$.)

Sleep Sets

All previous selection strategies can lead to **independent** transitions simultaneously being selected.

Example:

```
→ a0:  x:=1;        → b0:  y:=2;
  a1:  y:=1;          b1:  x:=2;
```

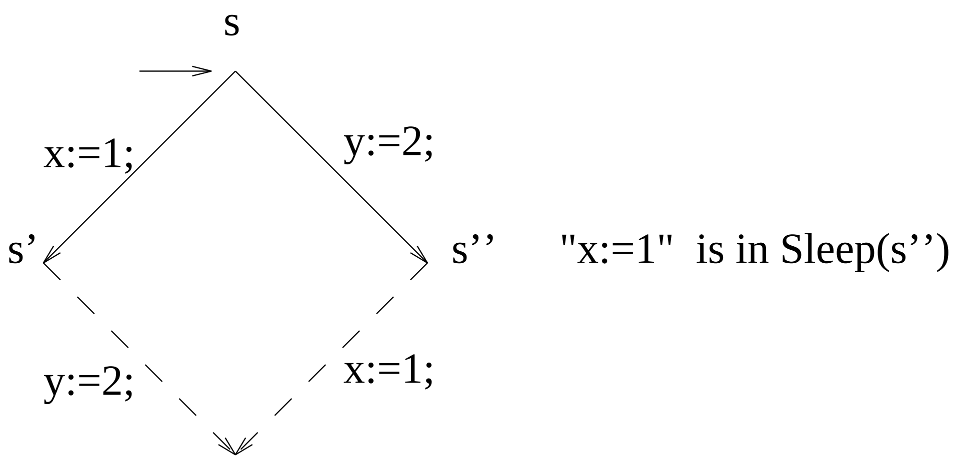


This can cause the wasteful exploration of several interleavings of these transitions.

Basic Idea Behind Sleep Sets

Aim: to avoid the wasteful exploration of all possible shufflings of independent transitions. [Godefroid&Wolper90-91]

Example:



- A sleep set is associated with each state s reached during the search.
- The sleep set associated with s is a set of transitions that are *enabled* in s but *will not be executed* from s.
- The sleep set associated with the initial state s_0 is the empty set.
- The sleep set associated with s' after $s \xrightarrow{t} s'$ is computed from the sleep set associated with s.

Sleep sets *combined* with persistent sets can further reduce the number of states and transitions explored.

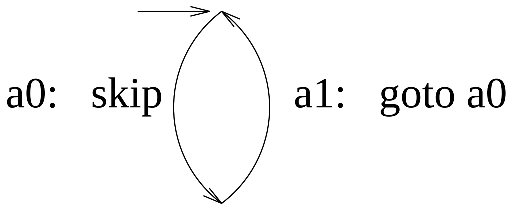
Beyond Deadlock Detection

To check for properties more elaborate than deadlocks, one needs to adapt the selective search algorithms.

Example:

```
        Process 1               Process 2

  → a0:  skip;            → b0:  x:=1;
    a1:  goto a0            b1:  stop
```



A_R

The behavior of some processes can be completely ignored during a selective search ("ignoring problem" [Valmari88-91]).

Solution: enforce additional conditions (a proviso) during the selective search in order to be "fair" in the choice of enabled independent transitions.

Example of Proviso for Safety Properties

Proviso compatible with persistent sets *and* sleep sets:

- Perform the selective search in Depth-first order.
- Enforce the following "proviso" [G.&Holzmann&Pirottin92]:
  - Only select persistent sets that contain at least one transition not in $s.Sleep$ and not leading to the current stack.
  - If there is no such persistent set, then all enabled transitions not in $s.Sleep$ are executed (they all lead to states in the stack).

A persistent-set selective search with such a proviso does not "ignore" transitions. The resulting A_R is such that:

Theorem: For all w in A_G from the initial state s_0, there exists a w' in A_R from s_0 such that w' is a permutation of an extension of w: $w \in Pref([w'])$.

A_R is then called a *trace automaton*.

Corollaries

- All system transitions that occur in A_G also occur in A_R. Hence can detect assertion violations, invariant violations, unreachable code.
- Local state reachability is preserved: all *local properties* (involving only one process) can be checked.
- The sequences of transitions in A_G projected on a single process are preserved in A_R.

Checking for General Safety Properties:

A selective search with proviso can be used to check for safety properties as follows.

- A safety property can be represented by a prefix closed automaton A_S on finite words. [Alpern&Schneider87]
- Build the complement $A_{\neg S}$ of A_S:
  $A_{\neg S}$ has one accepting state X.
- Check if the local state X of $A_{\neg S}$ is reachable in the system composed of the processes P_i combined with $A_{\neg S}$.

Note: $\exists$ other provisos for persistent sets only [Valmari91]...

Verifying Temporal Logic Formulas

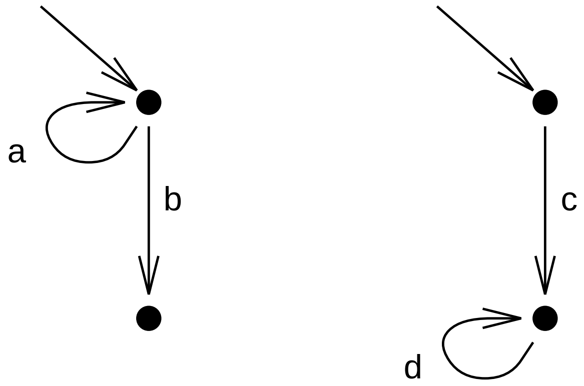
The "Model-Checking" Problem:
do all infinite computations of A_G satisfy ϕ?

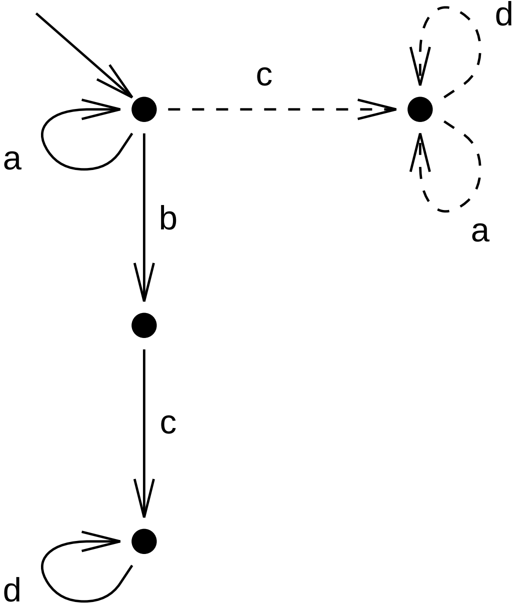
One needs to preserve in A_R at least one *infinite computation* of A_G violating ϕ, if any.

Can one use A_R as previously defined? No:

- A temporal formula can be sensitive to the order of independent actions. Then:
  1. Consider actions that appear in the formula as dependent [Valmari90].
  2. Use an automata-theoretic approach: the (LTL) formula is represented by a process [Godefroid&Wolper91].
  3. Restrict to insensitive formulas [Peled93].
- Infinite computations involving more than one process are not preserved in A_R.

Infinite Computations not Preserved?





A_R

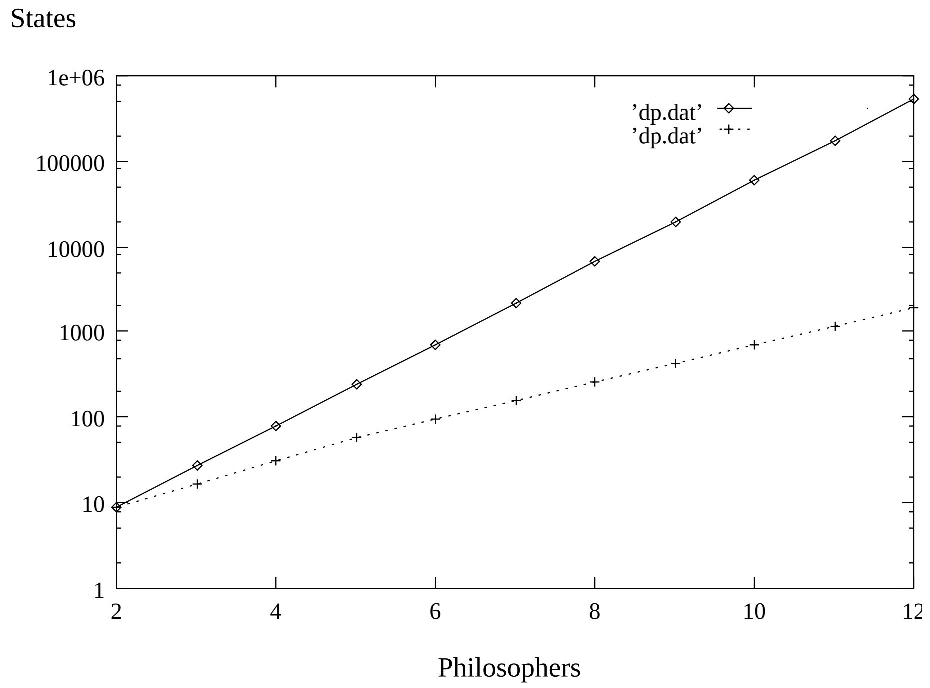
The infinite computation $c; (ad)^\omega$ of A_G is not represented in A_R, even allowing for the exchange of independent actions.

$\rightsquigarrow$ need for specialized model-checking algorithm [G.&Wolper91] or additional proviso [Valmari90-93], [Peled93] ("ample sets").

Evaluating Partial-Order Methods

1. Worst case complexity analysis tells us nothing.
2. Analyzing the behavior of the algorithms on academic examples is only of limited interest.

Example: dining philosophers



To be evaluated, the methods need to be implemented and widely experimented with.

$\leadsto$ Meaningful performance evaluation on a variety of examples, including "real" protocol examples.

From Theory to Practice

"Real" examples

$\leadsto$ **Partial-Order Package** for **SPIN**

SPIN: code available, well documented, examples available.

SPIN accepts **PROMELA** as validation language:

- defines systems of sequential processes that are executed asynchronously
- interactions via shared global variables
- interactions via message channels,
  synchronous (by rendez-vous) or asynchronous (buffered)

Two implementations:

- The ULg Partial-Order Package [92-94].
- Part of the official SPIN distribution [94-present].

(Both can be downloaded from the web.)

Experimental Results

- **MULOG3**: mutual exclusion algorithm
  (6 processes + FIFO channels, 97 lines of Promela)

Tool	States	Trans.	Time(sec)
SPIN	38,181	111,668	25.3
SPIN + P.O. Pack.	17,984	18,057	26

- **DTP**: Data Transfer Protocol
  (3 processes + FIFO channels, 406 lines of Promela)

Tool	States	Trans.	Time(sec)
SPIN	251,409	648,467	200.2
SPIN + P.O. Pack.	9,904	10,351	11.5

Observation

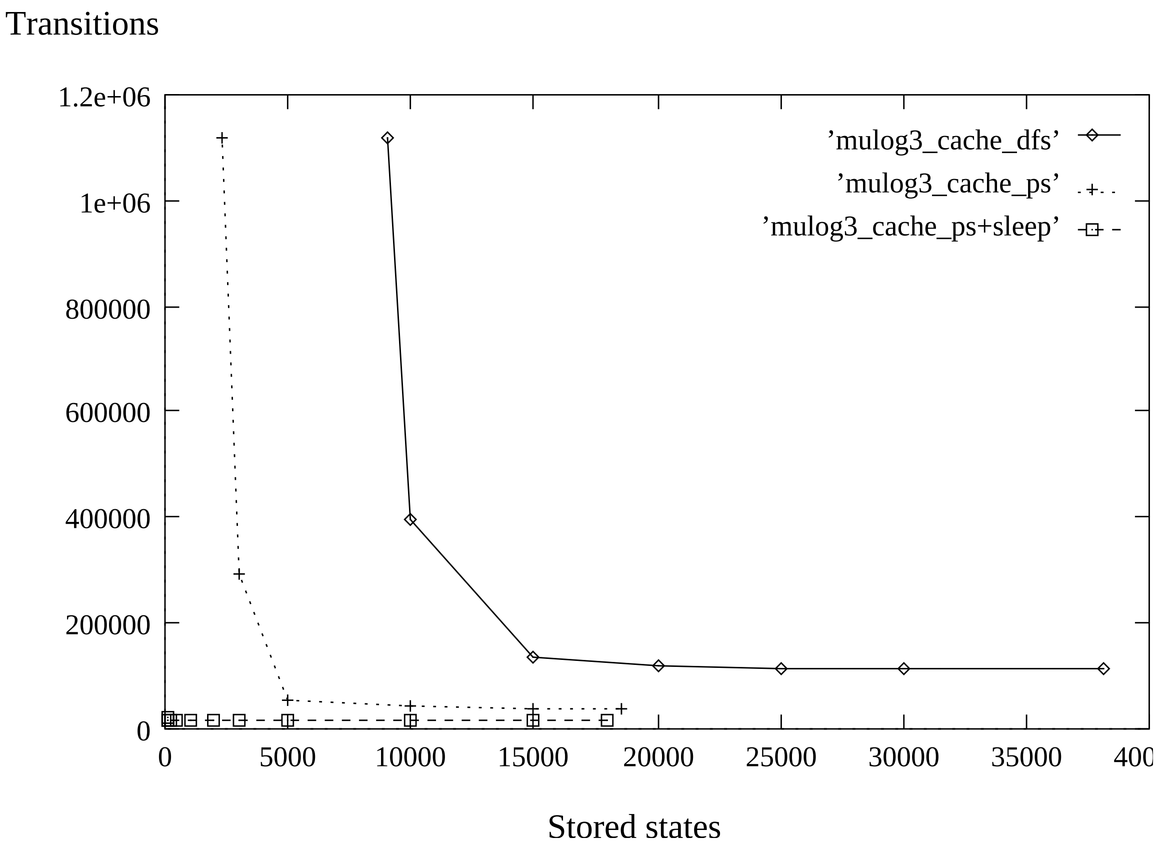
When using partial-order methods, the number of state matchings strongly decreases [Godefroid&Holzmann&Pirottin92]

$\leadsto$ most of the states are visited *only once* during the search

$\leadsto$ not necessary to store all of them!

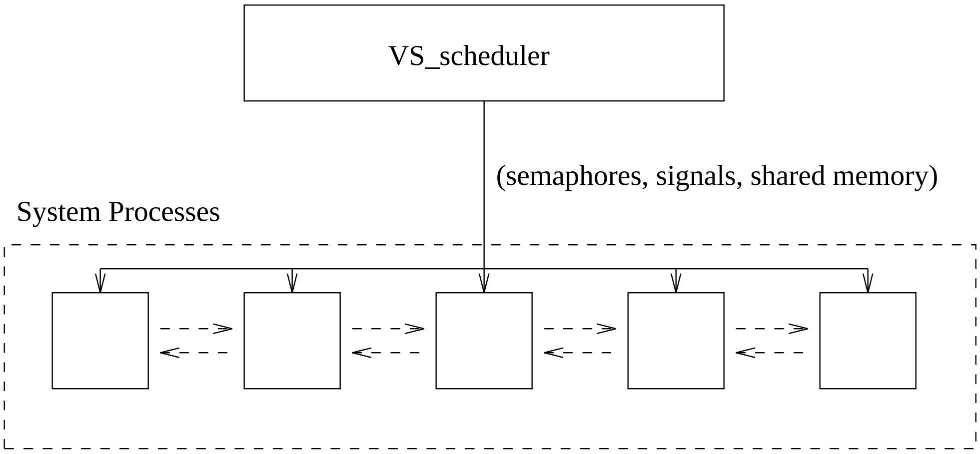
State-Space Caching

Mulog3



VeriSoft

VeriSoft is a tool for systematically exploring the state space of concurrent reactive software systems composed of (Unix-like) processes executing *arbitrary* code (e.g., C, C++,...).

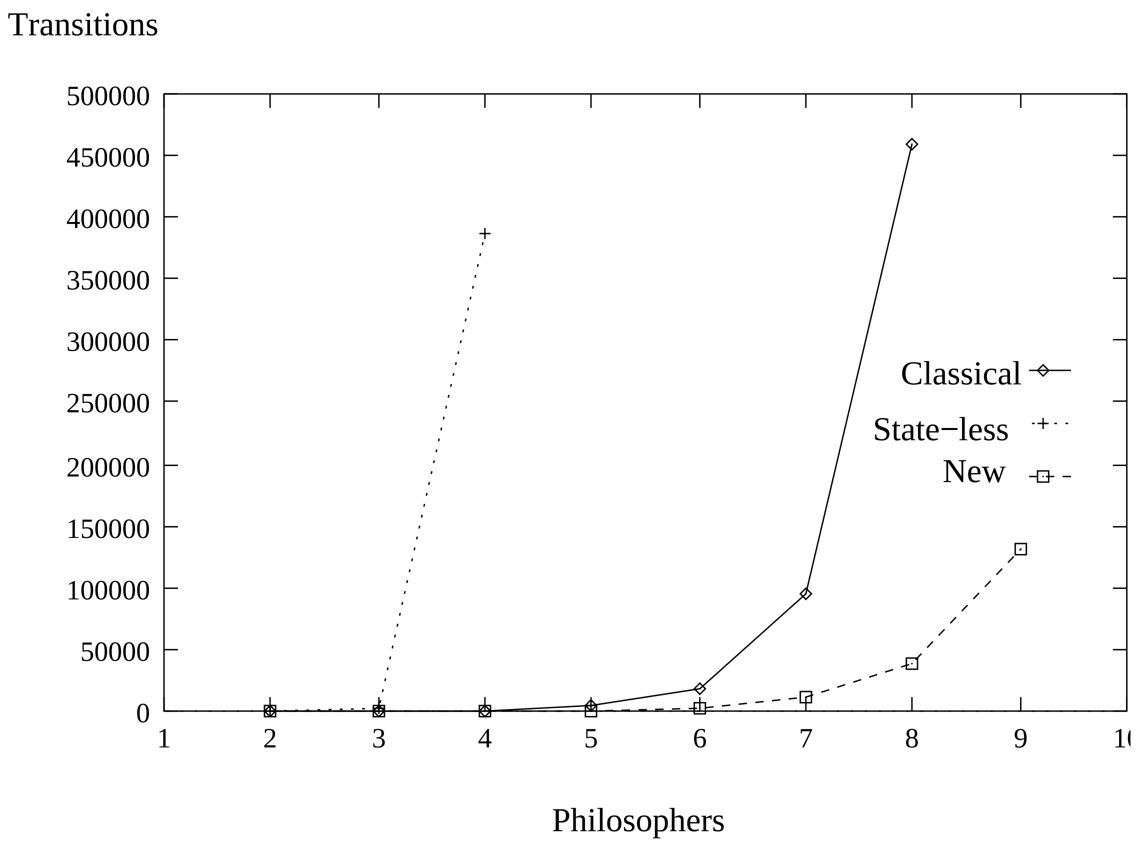


- Intercepts (suspends/resumes) all visible operations.
- Uses a *state-less search* with persistent sets and sleep sets.
- The semantics of visible operations is described in *libraries of communication objects*:
  - defines when enabled/disabled,
  - dependency relation (for sleep sets),
  - $\triangleright_s$ "can interfere" relation (for persistent sets).
- VeriSoft is limited to checking safety properties.

An Efficient State-Less Search

Theorem: (simplified)
For finite acyclic state spaces, the above algorithm can be used for the detection of deadlocks and assertion violations without incurring the risk of any incompleteness in the verification results.

Observation:
The key to make tractable a state-less search in the state space of a concurrent system is to use partial-order algorithms!



VeriSoft – Summary

VeriSoft is the first tool for systematically exploring the state spaces of systems composed of several concurrent processes executing arbitrary (e.g., C or C++) code. [Godefroid97]

Originality:

- does not require any specific programming language
- performs a state-less search (with partial-order methods)

In practice, the search is typically incomplete.

From a given initial state, VeriSoft can always guarantee a *complete coverage* of the state space *up to some depth.*

Impact:

- Since 1999, hundreds of non-commercial licences to industry and academia in 25+ countries.
  See `www.bell-labs.com/projects/verisoft`.
- Numerous industrial applications, such as:
  - Lucent's 4ESS Heart-Beat Monitor [GJH98]
  - Lucent's CDMA Call-Processing Library [CGP02]

Conclusions

Partial-Order Methods for Model Checking:

- address the main limitation of MC: state explosion
- transparent for the user
- basically "no risk" improvement
- can be much more efficient *both in time and memory*
- good results with typical "real" examples (e.g., containing only few processes...)
- compatible with other techniques: state-space caching, hashing without collision detection, symmetry reduction, symbolic (BDD-based) techniques, etc.
- broaden the applicability of classic state-space exploration tools (SPIN)
- broaden the applicability of model checking to software (VeriSoft)

Other Related Work

- "partial-order models of concurrency" (event structures [Winskel...], traces [Mazurkiewicz...], pomsets [Pratt...], [Lamport]...): more expressive models
- "partial-order temporal logics" ([Pinter&Wolper, Katz&Peled, Penczek,...]): more expressive logics
- partial-order methods for theorem proving (rewrite system) [Francez&al, Katz&Peled, Pnueli...]: same idea but different techniques (proof rules for confluence)
- "net unfoldings" technique [McMillan, Esparza,...]: use explicit partial-order representations for model checking
- how to define independence on code fragments automatically (commutativity analysis) [Diniz&Rinard, Stoller...]
- etc. (see references on next page for more)

Main References

- "Partial-Order Methods for the Verification of Concurrent Systems – An Approach to the State-Explosion Problem", P. Godefroid, LNCS 1032, Springer-Verlag.
- "Model Checking for Programming Languages using VeriSoft", P. Godefroid, POPL'97.
- Lecture notes of "Design and Analysis of Communication Software", course given at Stanford University, Fall 1999.
- See `http://www.bell-labs.com/~god`