Design and Analysis of Communication Software

Part 4:

Inside VeriSoft

The Research Behind The Tool

Overview

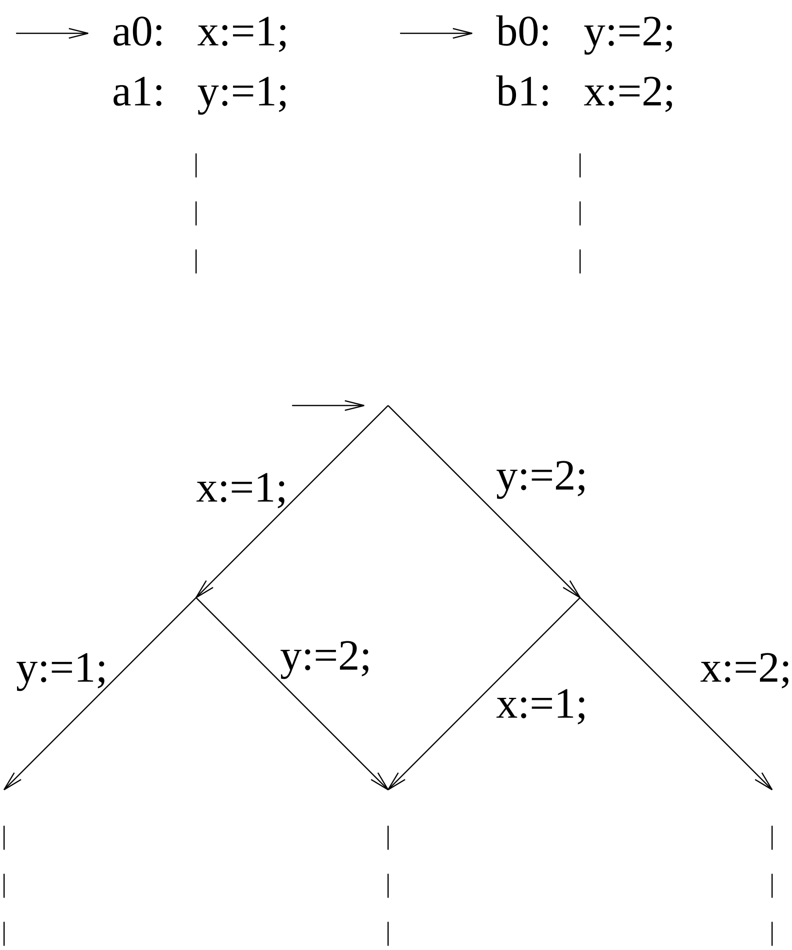
Part 4.2: (Oct 21)

- Sleep Sets
- Impact on State-Less Search
- Beyond Deadlock Detection
- Implementation in VeriSoft

Sleep Sets

Using persistent sets can lead to **independent** transitions simultaneously being selected.

Example:

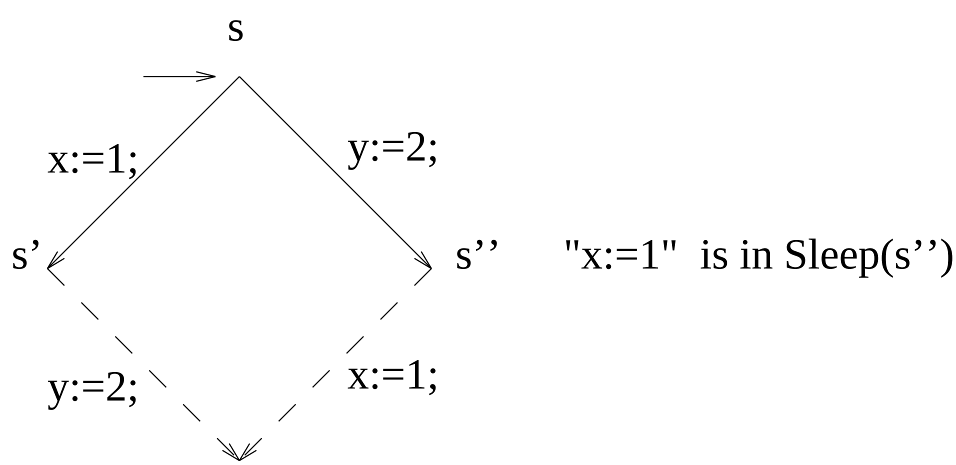


This can cause the wasteful exploration of several interleavings of these transitions.

Basic idea behind sleep sets

Aim: to avoid the wasteful exploration of all possible shufflings of independent transitions.

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.

How is the sleep set associated to a state computed?

$$
\begin{aligned}
Sleep &\xrightarrow{t_0} Sleep \setminus \{t\text{'s dependent with } t_0\} \\
&\xrightarrow{t_1} Sleep \cup \{t_0\} \setminus \{t\text{'s dependent with } t_1\} \\
&\xrightarrow{t_2} Sleep \cup \{t_0, t_1\} \setminus \{t\text{'s dependent with } t_2\} \\
&\xrightarrow{t_3} Sleep \cup \{t_0, t_1, t_2\} \setminus \{t\text{'s dependent with } t_3\} \\
&\ \vdots \\
&\xrightarrow{t_n} Sleep \cup \{t_0, t_1, t_2, \ldots t_{n-1}\} \setminus \{t\text{'s dependent with } t_n\}
\end{aligned}
$$

Algorithm: State-Less Depth-First Search Using Persistent Sets and Sleep Sets

Initialize: $Stack$ is empty;
Search() {
    DFS($\emptyset$);
    }
DFS(set: $Sleep$) {
    T = Persistent_Set()$\backslash Sleep$;
    while $T \neq \emptyset$ do {
        take t out of T;
        push (t) onto $Stack$;
        Execute(t);
        DFS($\{t' \in Sleep \mid t'$ and t are independent$\}$);
        pop t from $Stack$;
        Undo(t);
        $Sleep = Sleep \cup \{t\}$;
        };
    }

Note: see [G96] for algorithms using sleep sets in the context of a traditional (non state-less) search.

Proof of Correctness: The Previous Algorithm Preserves Deadlocks

Theorem.
Consider a concurrent system as previously defined, and let A_G denote its state space. Assume A_G is finite and acyclic. All deadlocks in A_G are visited by a state-less search using persistent sets and sleep sets.

Proof:

Imagine that we fix the order in which transitions selected in a given state are explored and that we first run a state-less search with persistent sets but *without* sleep sets.

Let A_R be the state space explored during this run. We know that A_R contains all the deadlocks in A_G (see Part 4.1).

Then, we run a state-less search with persistent sets *and* sleep sets while still exploring transitions in the same order.

For each deadlock d, we now prove that the very first path in A_R leading to d is still explored in the second run when using sleep sets.

Proof (Continued)

Let $p = s_0 \stackrel{t_0}{\rightarrow} s_1 \stackrel{t_1}{\rightarrow} s_2 \ldots s_{n-1} \stackrel{t_{n-1}}{\rightarrow} d$ be this path. The only reason why p might not be fully explored (i.e., until d is reached) by the second run using sleep sets is that some transition t_i of p is not taken because it is in the sleep set associated with s_i.

This means that t_i has been added to the sleep set associated with some previous state s_j, $j < i$, of the path p and then passed along p until s_i.

This implies that t_i has been explored *before* t_j from s_j since a transition is introduced in the sleep set after it has been explored.

Moreover, all transitions that occur between t_j and t_i in p, i.e., all t_k such that $j \leq k < i$, are independent with respect to t_i (otherwise, t_i would have been removed from the sleep set passed along p from s_j to s_i).

Consequently, $t_i t_j \ldots t_{i-1}$ (the sequence $t_j \ldots t_{i-1} t_i$ where t_i has been moved to the first position) is in $[t_j \ldots t_{i-1} t_i]$, and hence both sequences of transitions lead to the same state: $s_j \stackrel{t_i t_j \ldots t_{i-1}}{\Rightarrow} s_{i+1}$.

Since t_i is explored before t_j in s_j, this other path leading to d has been explored before p, and therefore p is not the very first path leading to p in the first run. A contradiction.

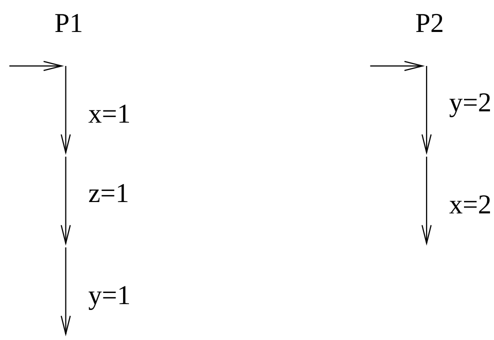
Notes

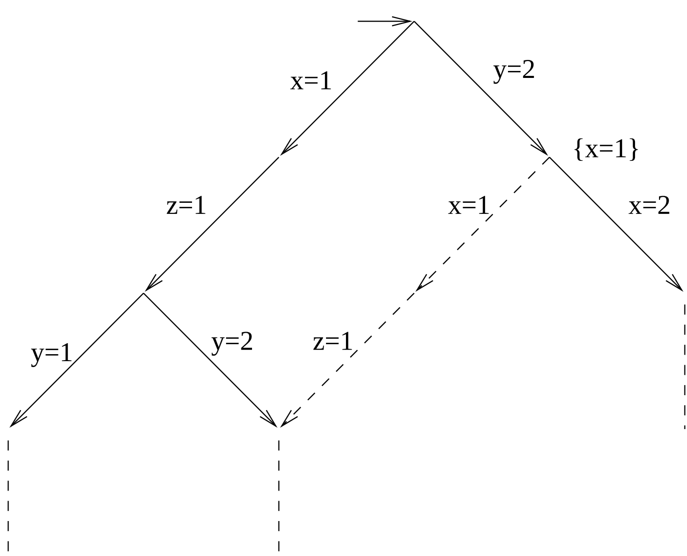
- The previous results hold whether or not the valid dependency relation used is conditional.
- The set "$T = \text{Persistent\_Set}() \backslash Sleep$" is not necessarily a persistent set in s (see last example).
- Hence, sleep sets makes it possible to go *beyond* persistent sets in computing the transitions that need be explored in a selective search.
- Technical remark: proving the correctness of sleep sets with a traditional (non state-less) search does not use the "very first path" argument... (see [G96])

Properties of Sleep Sets

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

Example:

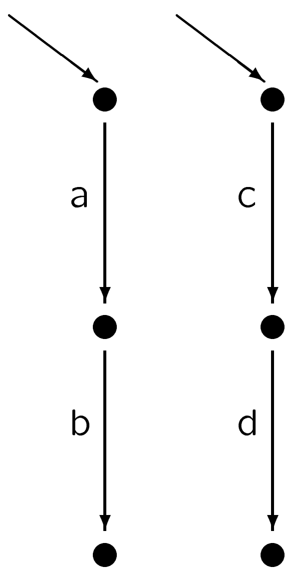




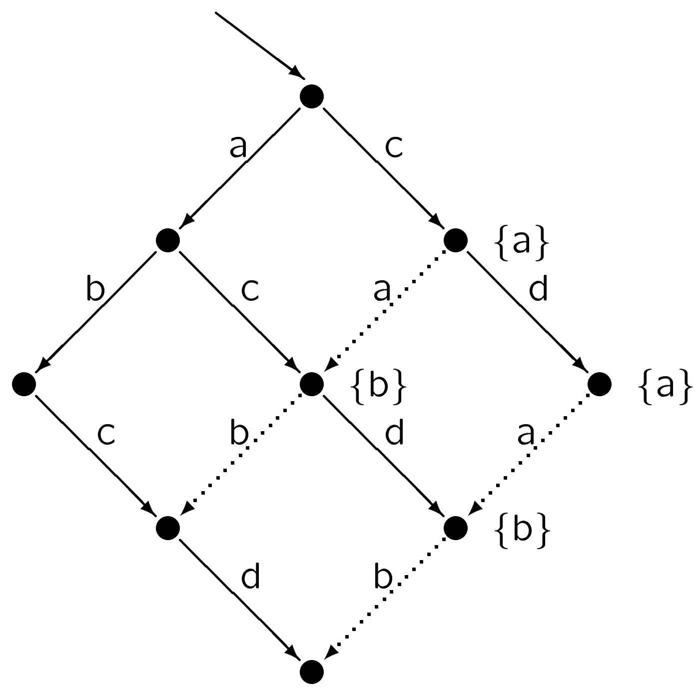
Properties of Sleep Sets (Continued)

Sleep sets alone only remove transitions, not states.

Example: (a, b, c, d execute purely local operations)



Σ_1={a,b} Σ_2={c,d}



This can still be very useful!

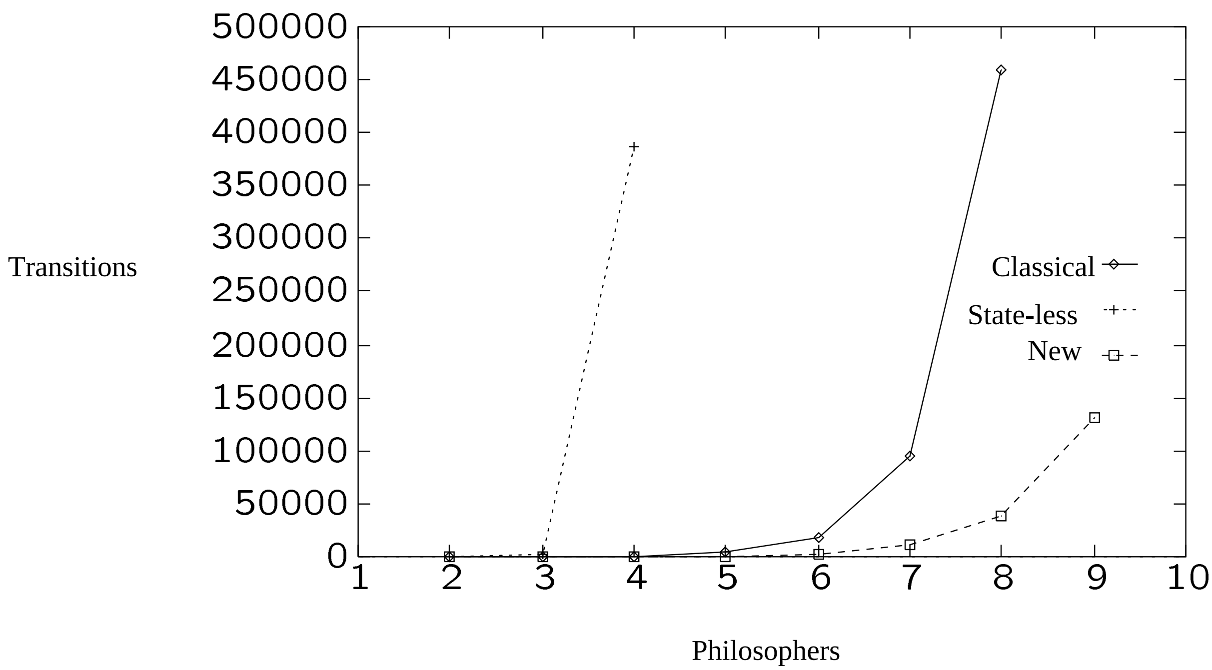
Impact on State-Less Search

Observation: [GHP92]
With partial-order methods (and sleep sets in particular), the number of state matchings when exploring the state space of a concurrent system strongly decreases.

⇝ Most of the states are visited *only once* during the search.

⇝ Not necessary to store these!

Example:



Without partia-order methods, a state-less search in the state space of a concurrent system would be untractable!

Summary

- Concurrent reactive system composed of a finite set of processes communicating by executing operations on a finite set of communication objects.
- Global behavior represented by a global state space A_G.
- Algorithms for computing a reduced state space A_R.
  - Deadlocks are preserved.
  - Two algorithmic techniques used: Persistent sets and Sleep sets.
  - Prune the state space *and* transform its shape!

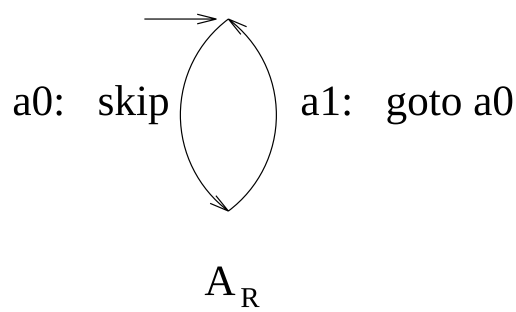
Beyond Deadlock Detection

In Principle: (true at abstract level)

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



The behavior of some processes can be completely ignored during a selective search ("ignoring problem").

Solution: enforce additional conditions (a proviso) during the selective search in order to be "fair" in the choice of enabled independent transitions (based on cycle/SCC detection; see [G96]).

In Practice: (true at implementation level)

- Cycles are rare at implementation level...
- One can often force the state space to be acyclic (by forcing the termination of sequences of inputs driving the system, etc.).

Impact of Cycles on State-Less Search

If the state space contains cycles, a state-less search will not terminate.

If the state space is finite and acyclic, termination is guaranteed, and the following properties hold:

- Assertion violations are preserved in A_R.
- All system transitions that occur in A_G also occur in A_R.
- Local state reachability is preserved in A_R.
- The sequences of transitions in A_G projected on a single process are preserved in A_R.

Other Properties

How can one check global properties?

- Make them *local* by adding synchronization or by adding a process embodying the property.
- Be careful, this introduces more dependency!

Any *safety property* can be checked this way ("any safety property can be represented by a prefix closed automaton on finite words" [AS87]).

How can one check liveness properties?

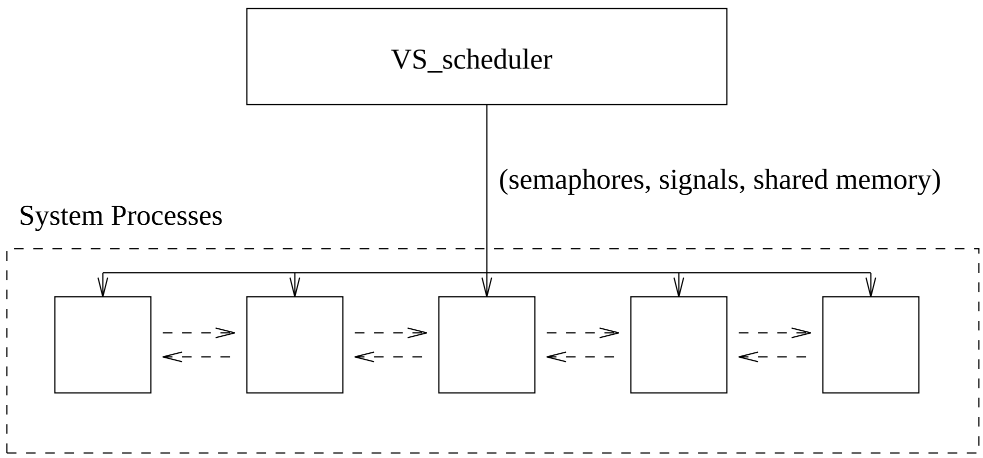
Such properties cannot be checked with a state-less search because cycles cannot be detected.

But approximations can be checked:

- Livelock: no enabled transition for a process during x successive transitions (test "*progress*").
- Divergence: a process does not communicate with the rest of the system during more than x seconds (test "*responsiveness*").

Implementation in VeriSoft

VeriSoft explores (automatically or interactively) the state space of a concurrent reactive system.



- 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),
  - $\triangleright_s$ relation (for persistent sets).
- Structural properties ("which process may or may not execute which visible operation on which communication object") can be described in the file `system_file.VS` (optional but recommended when possible, used for persistent sets).

Summary of Part 4

- Introduction to Partial Order Methods
- Concurrent Systems and Dynamic Semantics
- Using Partial Orders to Tackle State Explosion
- Towards More Independence
- Persistent Sets
- How to Compute Persistent Sets
- Discussion
- Sleep Sets
- Impact on State-Less Search
- Beyond Deadlock Detection
- Implementation in VeriSoft