Partial-Order Methods for Concurrent Program Verification

Pierre Wolper
University of Liège
Verifying Concurrent Programs: A Concrete Framework

Program: $n$ processes $P_i$, each described by a finite-state transition system
$A_i = (\Sigma_i, S_i, \Delta_i, s_{0i})$, where

- $\Sigma_i$ is an action alphabet,
- $S_i$ is a finite set of states,
- $\Delta_i \subseteq S_i \times \Sigma_i \times S_i$ is a transition relation, and
- $s_{0i} \in S_i$ is the initial state.

The problem is to check that the concurrent program $P$ satisfies a property $f$, for instance a temporal logic formula.
Eliminating Concurrency with interleaving semantics

The behavior of the concurrent program $P$ is defined by the reachable subset $A_G$ of the global transition system $A = (\Sigma, S, \Delta, s_0)$ where

- $\Sigma = \bigcup_{1 \leq i \leq n} \Sigma_i$,
- $S = \prod_{1 \leq i \leq n} S_i$ (Cartesian product), we denote the $i$th component of a state $s \in S$ by $s(i)$,
- $s_0 = (s_{01}, \ldots, s_{0n})$,
- $(s, a, s') \in \Delta$ if and only if
  - for all $1 \leq i \leq n$ such that $a \in \Sigma_i$, $(s(i), a, s'(i)) \in \Delta_i$, and
  - for all $1 \leq i \leq n$ such that $a \notin \Sigma_i$, $s(i) = s'(i)$. 
Example

$\Sigma_1 = \{a, b, c\}$

$\Sigma_2 = \{b, c, d, e\}$

$A_G$
The basic Idea

- The interleavings corresponding to the same concurrent execution are related and are not all necessary for the verification of most properties.

- It should thus be possible to verify properties on a semantical model of the program that only contains some interleavings of each concurrent execution.

- What interleavings are required could depend on the property to be checked.
Defining Concurrent Executions

The idea is that two interleaving sequences are part of the same concurrent execution if they can be obtained from each other by exchanging independent adjacent transitions.

Independent transition are transitions that can be exchanged without modifying the outcome of their execution.

We need to define precisely

- The concept of transition,

- The notion of independent transitions.
The concept of Transition

A *global* transition of $A_G$ is a triple $(s, a, s')$ and is unique. We need to view as identical transitions that represent the same program step.

Two global transitions $t_1 = (s_1, a_1, s'_1)$ and $t_2 = (s_2, a_2, s'_2)$ correspond to the same program transition if and only if

- $a_1 = a_2 = a$ (the action is the same) and,

- for all $1 \leq i \leq n$ such that $a \in \Sigma_i$, $s_1(i) = s_2(i)$ and $s'_1(i) = s'_2(i)$ (the state changes for the processes participating in the transition are identical).

In what follows, we only talk of program transitions.
The notion of Independent Transitions  
(semantical notion)

Two (program) transitions $t_1$ and $t_2$ are \textit{independent} if the following two conditions are true in all states $s$ of $A_G$ (otherwise they are said to be \textit{dependent}):

1. if $t_1$ ($t_2$) is enabled in $s$ and $s \xrightarrow{t_1} s'$ ($s \xrightarrow{t_2} s'$), then $t_2$ ($t_1$) is enabled in $s$ iff $t_2$ ($t_1$) is enabled in $s'$ (independent transitions cannot 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 both $s \xrightarrow{t_1 t_2} s'$ and $s \xrightarrow{t_2 t_1} s'$ (commutativity of enabled independent transitions).
It is not practical to check these two properties for all pairs of transitions in all states \( s \) in \( A_G \).

In practice, one uses *Sufficient* syntactic condition for two transitions to be independent.

**Example:**

In the context considered here, *Sufficient* syntactic condition for transitions \( t_1 \) and \( t_2 \) to be independent:

the set of processes that participate in \( t_1 \) is disjoint from the set of processes that participate in \( t_2 \).
Defining Concurrent Executions with Independence

- Two sequences of transitions are equivalent if they can be obtained from each other by successively permuting adjacent independent transitions.

- Thus, given an independency relation, sequences of transitions can be grouped into equivalence classes corresponding to concurrent executions: Mazurkiewicz’s traces.

- The trace containing a sequence of transitions $w$ will be denoted $[w]$. 
• Sequences of transitions $w_1$ and $w_2$ belonging to the same trace lead to the same state of $A_G$:
  If $s \xrightarrow{w_1} s_1$, $s \xrightarrow{w_2} s_2$ 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 one transition sequence corresponding to that trace.
The Algorithms:
Selective Searches

- **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 of the global state-graph is explored.

- Which transitions should be selected?
  Two main (compatible) techniques:
  - Persistent sets
  - Sleep sets
Selective Search Algorithm

1. Initialize: \( \text{Stack is empty; } H \text{ is empty; } \)
2. \( \text{push } (s_0) \text{ onto } \text{Stack}; \)
3. Loop: while \( \text{Stack } \neq \emptyset \) do {
   4. \( \text{pop } (s) \text{ from } \text{Stack}; \)
   5. if \( s \) is NOT already in \( H \) then {
      6. enter \( s \) in \( H \)
   7. \( T = \text{Select\_Among\_Enabled\_Transitions}(s); \)
   8. for all \( t \) in \( T \) do {
      9. \( s' = \text{succ}(s) \) after \( t; /* \text{execution of } t */ \)
   10. \( \text{push } (s') \text{ onto } \text{Stack}; \)
   11. }
   12. }
   13. }
Persistent Sets

A set of transitions $T$ is persistent in a state $s$ if whatever one does from $s$, while remaining outside of $T$, does not interact with or affect $T$.

Definition:
A set $T$ of transitions enabled in a state $s$ is persistent in $s$ if and only if, for all transitions $t \notin T$ such that there exists a sequence in $A_G$

$$ s = s_0 \xrightarrow{t_0} s_1 \xrightarrow{t_1} s_2 \ldots \xrightarrow{t_{n-1}} s_n \xrightarrow{t_n=t} s_{n+1} $$

leading from $s$ to $t$ and including only transitions $t_i \notin T$, $t$ is independent with respect to all transitions in $T$. 

Reachable states without executing any transition of $T$
**Correctness:**
**Persistent Sets Preserve Deadlocks**

**Theorem:**
If \( s \) is reached in the search and if \( \exists w : s \xrightarrow{w} d \), \( d \) will also be reached.

**Proof:** Induction on the length of \( w \).

- \( |w| = 0 \): Immediate.
- \( |w| = n + 1 \): Thus \( w = tw' \) with \( |w'| = n \) and \( s \overset{t}{\rightarrow} s' \xrightarrow{w'} d \).
• $t$ is selected: immediate.

• $t$ not selected: use the fact that unselected transitions are independent w. r. t. selected transitions.

  – Some selected transition appears in $w'$ (if not, no deadlock). Let $t'$ be the first of these: $w' = w_1 t' w_2$.

  – $t$ and all transitions in $w_1$ are independent with respect to $t'$ since none of these is in the selected set. One has: $[tw_1 t' w_2] = [t' t w_1 w_2]$.

  – Thus $s \xrightarrow{t'} s'' \xRightarrow{t w_1 w_2} d$ and the result follows by the inductive hypothesis ($|tw_1 w_2| = n$).
Computing Persistent Sets

- Algorithms can be more or less complex depending on how much information about the structure of the program they use.

- The heuristic goal is to obtain a persistent set that is as small as possible.

- The set of all enabled transitions in a state $s$ is persistent in $s$: it is not possible to move from $s$ without executing a transition in the set.
Example:

\[
\Sigma_1 = \{a\} \quad \Sigma_2 = \{b\}
\]

Possible sets \( T \): \( \{a\}, \{b\}, \{a, b\} \)

Example:

\[
\Sigma_1 = \{a_1, a_2\} \quad \Sigma_2 = \{b_1, b_2\}
\]

Possible sets \( T \): \( \{a_1, a_2\}, \{b_1, b_2\}, \{a_1, a_2, b_1, b_2\} \)
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.
Can interfere?

Is dependent with, and thus

1. can disable, or

2. can enable, or

3. cannot be permuted with.

In the concrete framework we have outlined,

\[(3) \rightarrow (1) \lor (2)\]
Computing Persistent Sets
(Valmari: Stubborn sets)

- $T = \{t\}$ start with a set that contains a single enabled transition.

- **Repeat**
  
  - Add all transitions that can can disable some transition in $T$,
  
  - If $T$ contains some disabled transition, add at least one transition that needs to be executed for that transition to become enabled.

  **Until** no more transition need be added.

- Restrict $T$ to enabled transitions.
Example:

\[
\begin{align*}
\Sigma = \{a, b, g, x\} & \quad \Sigma = \{b, g, e, g, x\} \quad \Sigma = \{b, d, e, h\} \quad \Sigma = \{f, h\} \\
\end{align*}
\]

\[
\begin{align*}
\{a\} & \\
\{a, b\} & \\
\{a, b, x\} & \\
\{a, b, x, g\} & \\
\text{Restrict:} & \quad \{a\}
\end{align*}
\]
Sleep Sets

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

Example:

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

Thus Sleep sets are introduced
Basic idea behind sleep sets

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

Example:

- A sleep set is defined as a set of transitions.
- One 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.
How is the sleep set associated to a state computed?

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

1. Initialize: Stack is empty; $H$ is empty;
   
2. \( s_0.Sleep = \emptyset \);
3. push \((s_0)\) onto Stack;
4. Loop: while Stack \( \neq \emptyset \) do {
   
5. pop \((s)\) from Stack;
6. if \(s\) is NOT already in $H$ then {
   
7. \( T = \text{Persistent Set}(s) \setminus s.Sleep; \)
8. if \( T = \emptyset \land s.Sleep = \emptyset \) then print “Deadlock!”;
9. enter \( s \) in $H$
10. }
11. else {
12. \( T = \{t \mid t \in H(s).Sleep \land t \notin s.Sleep\} \);
13. \( s.Sleep = s.Sleep \cap H(s).Sleep; \)
14. \( H(s).Sleep = s.Sleep \)
15. }
16. for all \(t\) in \(T\) do {
17. \( s' = \text{succ}(s) \) after \(t\); // execution of \(t\) */
18. \( s'.Sleep = \{t' \mid t' \in s.Sleep \land (t,t') \text{ are independent }\} \);
19. push \((s')\) onto Stack;
20. \( s.Sleep = s.Sleep \cup \{t\} \)
21. }
22. }

26
Proof of Correctness: Sleep Sets + Persistent Sets Preserve Deadlocks

Lemma:
if $\exists w : s \xrightarrow{w} d$, and if, for all $w_i \in [w]$, the first transition of $w_i$ is not in $s.Sleep$, then, if $s$ is pushed onto Stack (i.e., is visited), $d$ is visited.

Proof: Induction on the length of $w$.

$|w| = 0$: Immediate.

$|w| = n + 1$: Thus $s \xrightarrow{w} d$ and

$\forall w_i \in [w], t_i \not\in s.Sleep$, where $t_i$ is the first transition of $w_i$.

- If $s$ is not already in $H$:
  - at least one of the $t_i$ is selected by $\text{Persistent\_Set}(s)$
  - let $t_1$ be the first of these which is explored
    - $s \xrightarrow{t_1} s' \xrightarrow{w'} d$ and $\forall w_i' \in [w'], t_i' \not\in s'.Sleep$
• If $s$ is already in $H$: let $H(s).\text{Sleep}$ be the sleep set stored in $H$ with $s$

  − If some of the $t_i$ are in $H(s).\text{Sleep}$: they are executed and the first of these leads to an $s'$ in which the inductive hypothesis can be applied.

  − If none of the $t_i$ are in $H(s).\text{Sleep}$: either none of the $t_i$ were ever in $H(s).\text{Sleep}$ or they were some, but they have been removed later:

    in both cases, inductive hypothesis can be applied to a successor of $s$
What Do Sleep Sets Do for Us?

- Sleep sets alone only remove transitions, not states (this can still be very useful).

\[
\Sigma_1 = \{a, b\} \quad \Sigma_2 = \{c, d\}
\]
• Sleep sets combined with persistent sets can further reduce the size of the state space being explored.
Checking General Properties

- Local state reachability is preserved by a minor modification of the algorithms we have presented.
- For checking more general properties, one makes them local!
- This forces the dependencies that are necessary for checking the property.
Evaluating “Partial-Order” verification techniques

1. Worst case complexity analysis tells us nothing.

2. Analyzing the behavior of the algorithms on academic examples is only of limited interest: let dining philosophers eat and think!

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

1. Meaningful performance evaluation on a variety of examples, including industrial size ones.

2. A rational basis for evaluating the trade-off between the complexity of the selection algorithms and the state-space reduction that is obtained.
Implementation context

- To handle “real” examples, one needs a real system.

- A **Partial-Order Package** has been implemented for **SPIN**

- Why SPIN?
  - code available
  - well documented
  - examples available
SPIN

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)
- dynamic process creation/deletion allowed: not with (current) P.O. package
SPIN

Errors Caught:

- system deadlock (i.e. improper termination)
- unreachable code (i.e. dead code)
- buffer overrun
- violations of correctness assertions (e.g. `assert(condition);` user-specified)
- violations of LT Temporal Logic formulae (user-specified): not (yet) with P.O. Package

Error Diagnosis: sequence of transitions
Experimental Results

- **PFTP**: file transfer protocol, 206 lines of code, 3 processes communicating via FIFO channels.

- **MU3**: mutual exclusion algorithm, 97 lines of code, 6 processes communicating via FIFO channels and shared variables.

- **ABRA**: Abracadabra protocol, 168 lines of code, 4 processes communicating via FIFO channels.

- **DTP**: data transfer protocol, 406 lines of code, 3 processes communicating via FIFO channels.
**Experimental Results (continued)**

<table>
<thead>
<tr>
<th>Protocol</th>
<th>Alg.</th>
<th>States</th>
<th>Transitions</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>PFTP</td>
<td>DFS</td>
<td>446,982</td>
<td>1,257,317</td>
<td>478.2</td>
</tr>
<tr>
<td></td>
<td>PS</td>
<td>276,722</td>
<td>482,722</td>
<td>662.7</td>
</tr>
<tr>
<td>MU3</td>
<td>DFS</td>
<td>38,181</td>
<td>111,668</td>
<td>25.3</td>
</tr>
<tr>
<td></td>
<td>PS</td>
<td>18,537</td>
<td>38,906</td>
<td>25.8</td>
</tr>
<tr>
<td>ABRA</td>
<td>DFS</td>
<td>149,816</td>
<td>372,010</td>
<td>494.2</td>
</tr>
<tr>
<td></td>
<td>PS</td>
<td>32,289</td>
<td>40,931</td>
<td>166.3</td>
</tr>
<tr>
<td>DTP</td>
<td>DFS</td>
<td>251,409</td>
<td>648,467</td>
<td>200.2</td>
</tr>
<tr>
<td></td>
<td>PS</td>
<td>9,904</td>
<td>10,351</td>
<td>11.3</td>
</tr>
</tbody>
</table>

SPARC2 workstation (64 Megabytes of RAM).
Conclusions

- Partial-Order verification methods do provide substantial gains.
- they are compatible with other techniques (hashing without collision detection, state-space caching, ...)
- however, they only tackle one cause of the state explosion problem: the exploration of all interleavings of concurrent actions.
- Future: combine these techniques with methods that tackle other causes...
Another approach: Net unfolding (McMillan)

- Program described by a Petri net.

- Nontrivial algorithm for unfolding the Petri net just enough to be able to check for reachable markings.

- Algorithm for checking for reachable terminal states (deadlocks).

- Semantics that are preserved are standard Petri net reachability.