CS 267: Automated Verification

Lecture 11: Partial Order Reduction

Instructor: Tevfik Bultan

Partial Order Reduction: Main Idea

- Statements of concurrently executing threads can generate many different interleavings
  - all these different interleavings are allowable behavior of the program
- A model checker has to check that: For all possible interleavings, the behavior of the program is correct
  - However different interleavings may generate equivalent behaviors
- In such cases it is sufficient to check just one interleaving without exhausting all the possibilities
  - This is called partial order reduction

The following discussion on partial order reduction is based on the book:

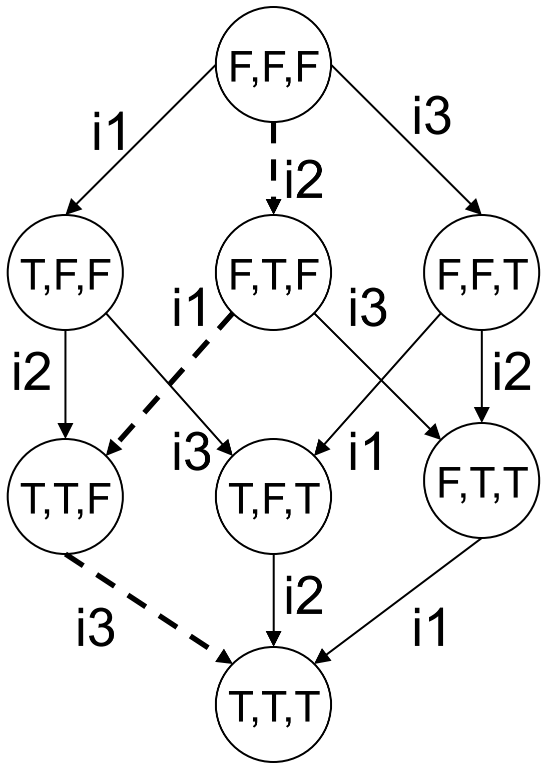
Model Checking by E. M. Clarke, Orna Grumberg, Doron Peled

Concurrency In Asynchronous Systems

- Concurrent execution of multiple threads or processes can be modeled as interleaved execution of atomic instructions
- Given n atomic instructions that can be executed concurrently there are n! different execution sequences
- If these n atomic instructions are independent, all n! execution sequences will end at the same global state
  - However, they will generate 2^n intermediate states (one for each subset of the n atomic instructions)
  - If the property we are interested in verifying does not distinguish among the n! possible execution sequences, then we can use a single execution sequence to represent all n! of them and visit only n states instead of 2^n states

An Example: Three Independent Transitions

```
a, b, c: boolean
Initial: a=b=c=false
i1: a:=true || i2: b:=true || i3: c:=true
```



- Can we pick one representative execution (e.g., i2,i1,i3) rather than trying all 6 (3!) executions and visiting all 8 (2^3) states?
- When is it safe to pick only one representative execution?
- Can we do this on the fly without generating the whole state space?

Another Example: Partial vs. Total Order

```
a, b, c, d: integer
Initial: a=b=c=d=0
P1:                           P2:
     P11 : a := a+1;               P21 : c := d+1;
     P12 : b := b+1;               P22 : d := c*2;

Program: P1 || P2
```

The order of execution: $P_{11} < P_{12}$ and $P_{21} < P_{22}$

- Note that this is a partial order for P_{11}, P_{12}, P_{21}, P_{22}
- Any execution sequence which obeys this partial order is a valid execution sequence
- An interleaved execution sequence gives a total order
  - We want to pick one representative interleaved execution sequence and avoid generating all of them

Some Definitions

- Given a transition system (S, I, R) and a labeling function $L: S \rightarrow 2^{AP}$
- We assume that $R = t_1 \cup t_2 \cup \ldots \cup t_n$ where each $t_i \subseteq S \times S$ is called a transition
- Let $T = \{ t_1 , t_2 , \ldots , t_n \}$ denote the set of transitions
- Given a transition $t \in T$, following notations are equivalent:
  - $(s,s') \in t$
  - $t(s,s')$
  - $s' = t(s)$ (here we are assuming that t is deterministic, i.e. for any state s, there is at most one state s' s.t. $(s,s') \in t$)
- A transition t is ***enabled*** at state s, if there exists an s' such that $t(s,s')$ holds, otherwise it is ***disabled***
- enabled(s) is the set of transitions that are enabled at state s

Independence Relation

- The ***independence relation*** $I \subseteq T \times T$ is a symmetric, anti-reflexive relation such that, for each $s \in S$ and for each $(t, t') \in I$, the following conditions hold:
  - enabledness: If $t, t' \in \mathrm{enabled}(s)$ then $t \in \mathrm{enabled}(t'(s))$
    - Independent transitions do not disable each other (but they can enable each other)
  - commutativity: If $t, t' \in \mathrm{enabled}(s)$ then $t(t'(s)) = t'(t(s))$

  Recall that:

  symmetric: for all $t, t' \in T$, $(t, t') \in I \Rightarrow (t', t) \in I$

  anti-reflexive: for all $t \in T$, $(t, t) \notin I$

- Dependence relation is defined as the complement of the independence relation

Invisible Transitions

- A transition t is ***invisible*** with respect to propositions AP' $\subseteq$ AP if for each pair of states s, s' such that s' = t(s)
  - $L(s) \cap AP' = L(s') \cap AP'$

  i.e., execution of the transition t from any state does not change the value of the propositions in AP'

- A transition is visible if it is not invisible

Stuttering Equivalence

- Two infinite paths $x = s_0, s_1, s_2, \ldots$ and $y = r_0, r_1, r_2, \ldots$ are stuttering equivalent (denoted as $x \sim_{st} y$), if there are two infinite sequences of positive integers:
  - $0 = i0 < i1 < i2 < \ldots$ and $0 = j0 < j1 < j2 < \ldots$ such that for every $k \geq 0$

  $L(s_{ik}) = L(s_{ik+1}) = \ldots = L(s_{i(k+1)-1})$

  $= L(r_{jk}) = L(r_{jk+1}) = \ldots = L(r_{j(k+1)-1})$

- A property is invariant under stuttering if for each stuttering equivalent path either both satisfy the property or neither satisfy the property.

LTL_{-X}

- LTL_{-X} is the temporal logic LTL without the next state operator X
- Any LTL_{-X} formula is invariant under stuttering
- Two transition systems TS_1 and TS_2 are stuttering equivalent
  - if they have the same initial states
  - for each path x in TS_1 that starts at the initial state s, there exists a path y in TS_2 which starts from the same initial state and $x \sim_{st} y$
- If two transitions systems are stuttering equivalent, then they satisfy the same set of LTL_{-X} properties

Partial Order Reduction

- We can check LTL_{-X} properties using partial order reduction techniques

- While visiting a state s, instead of exploring all the transitions in enabled(s), we will just explore the transitions in ample(s) where ample(s) $\subseteq$ enabled(s)

- Using ample(s) instead of enabled(s)
  - should reduce the number of states visited,
  - but it should not change the result of the verification,
  - and the cost of computing ample(s) should be small.

Modified Depth First Search

```
main()  {
  StateSpace = {$s_0$};
  expand_state($s_0$);
}

expand_state(s)  {
  work_set(s) := ample(s);
  while work_set(s) is not empty {
     let t $\in$ work_set(s);
     work_set(s) := work_set(s) – {t};
     s’ := t(s);
     if s’ $\notin$ StateSpace {
        StateSpace := StateSpace $\cup$ s’;
        expand_state(s’);
     }
     create_edge(s, t, s’);
   }
   set completed(s);
}
```

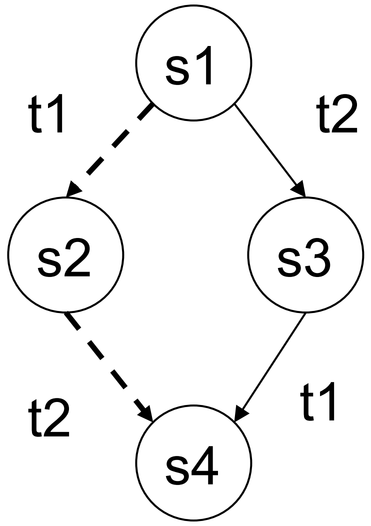
Conditions for Partial Order Reduction

- Now we will discuss four conditions for ample sets
- If the ample sets satisfy these conditions then the result of the state space search without using partial order reduction and with using partial order reduction should be the same for LTL_{-X} properties
- Let's call the state graph generated with state space search without using partial order reduction the *full state graph* and the state graph generated with state space search with using partial order reduction the *reduced state graph*
- If the ample sets satisfy these four conditions then the full state graph and the reduced state graph are stuttering equivalent
  - Hence, they satisfy the same set of LTL_{-X} properties

Conditions for Partial Order Reduction

- C0: ample(s) = $\emptyset$ if and only if enabled(s) = $\emptyset$
  - If a state has at least one successor in the full state graph, then it should also have at least one successor in the reduced state graph
- C1: Along every path in the full state graph that starts at s, a transition that is dependent on a transition in ample(s) cannot be executed without a transition in ample(s) occuring first
  - This implies that the transitions in enabled(s) – ample(s) are all independent of those in ample(s)
- C2: If s is not fully expanded (i.e., ample(s) $\neq$ enabled(s)), then every t $\in$ ample(s) is invisible
- C3: A cycle is not allowed if it contains a state in which some transition t is enabled but is never included in ample(s) for any state s on the cycle.

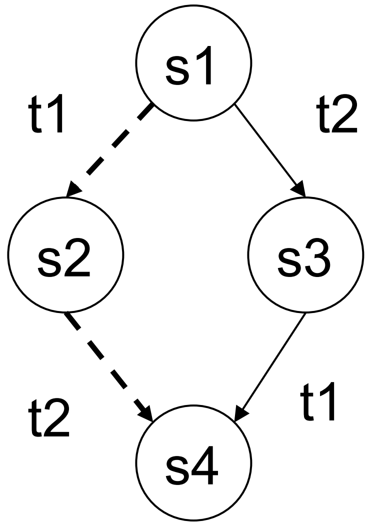
Simplest Example


$$\begin{tikzcd} & s1 \arrow[dl, dashed, "t1"'] \arrow[dr, "t2"] & \\ s2 \arrow[dr, dashed, "t2"'] & & s3 \arrow[dl, "t1"] \\ & s4 & \end{tikzcd}$$

- Basic question: Can we explore just the path s1, s2, s4 and ignore the path s1, s3, s4?
- If we can we can, then we will set
  - ample(s1) = {t1}
- This will allow us to ignore the transition t2 at state s1
- Note that

  ample(s1) ⊆ enabled(s1) = {t1, t2}

When Can We Do It?



- In order to set ample(s1) = {t1} we need certain conditions to hold
- First we need t1 and t2 to commute

  t2(t1(s1)) = t1(t2(s1))

  – This is true in the figure

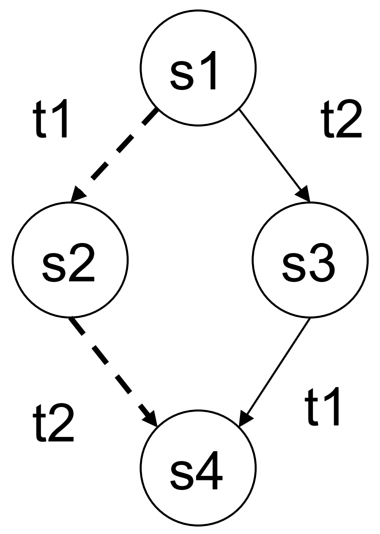
    t2(t1(s1)) = t1(t2(s1)) = s4
- Second, we need to make sure that

  t2 $\in$ enabled(t1(s1))

  – This is also true in the figure

    t2 $\in$ enabled(t1(s1)) = enabled(s2)
- These two conditions basically state that the transitions that are in ample(s1) and enabled(s1) - ample(s1) should be independent which is covered by condition C1

Possible problems

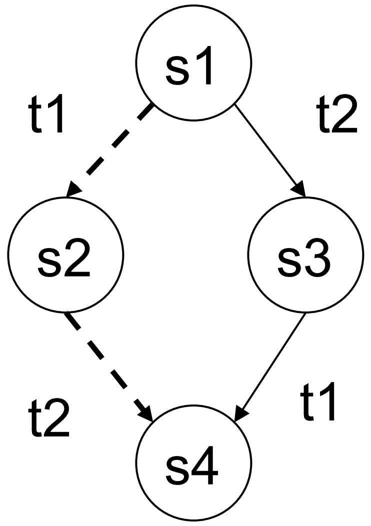


If we ignore the path s1, s3, s4 what can happen?

Problem 1: The truth value of the property we are interested in may be different for the paths s1, s2, s4 and s1, s3, s4

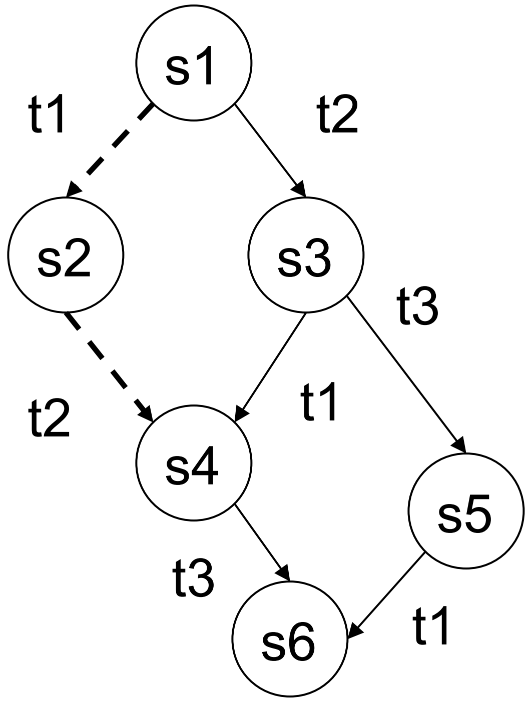
Problem 2: State s3 may have other successors which may not be reachable if we do not visit s3

When Can We Do It?



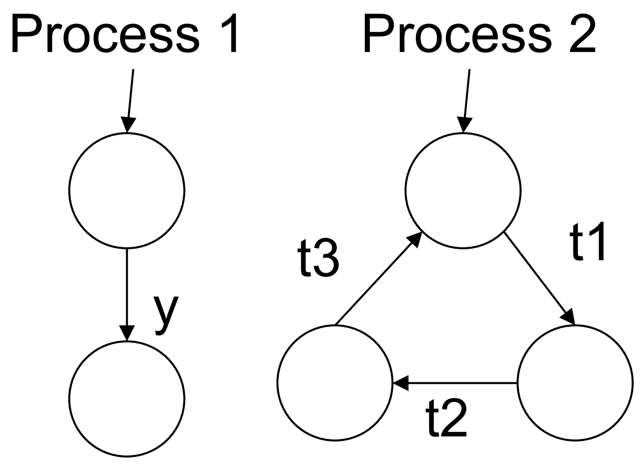
- Condition C2 resolves Problem 1
- Due to condition C2, since s1 is not fully explored, every transition in ample(s1) must be invisible
- This means that as far as the property of interest is concerned the paths
  - s1, s2, s4 and s1, s3, s4 are stuttering equivalent
- This means that s1, s2, s4 will satisfy the property of interest if and only if s1, s3, s4 satisfies the property of interest
- Hence ignoring the path s1, s3, s4 will not change the result of the verification

When Can We Do It?



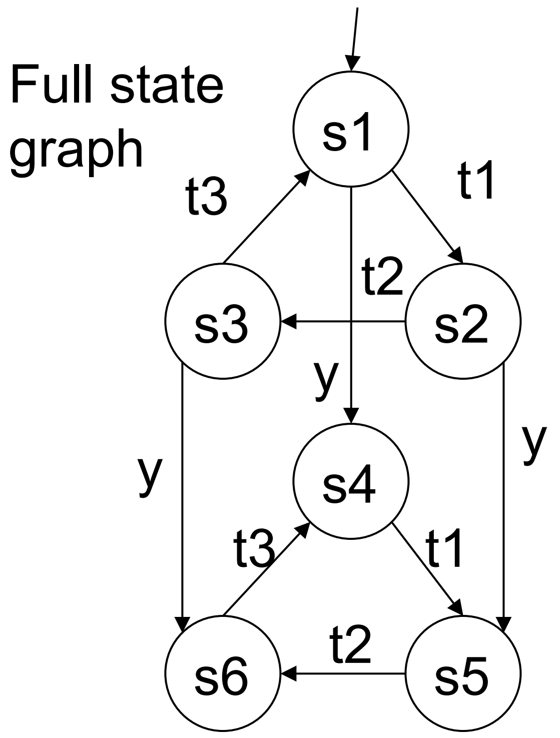
- Condition C1 resolves the Problem 2
- Assume that there is a state s5 reachable from state s3 with transition t3
- Due to condition C1, t3 and t1 must be independent
  - Then, t3 is enabled in s4
- Since t3 and t1 are independent we also have t3(t1(s3)) = t1(t3(s3))
- Since t1 is invisible the following paths are stuttering equivalent:
  - s1, s2, s4, s6 and
  - s1, s3, s5

What about condition C3?



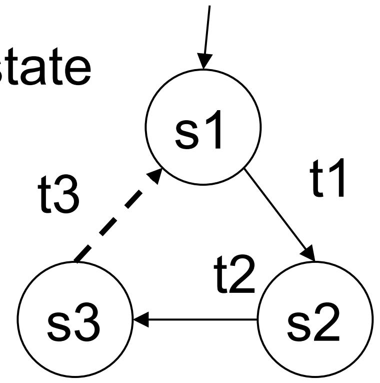
Assume that:

- y is independent of t1, t2 and t3
- t1, t2 and t3 are interdependent
- y is visible
- t1, t2 and t3 are invisible



What about condition C3?

Reduced state graph



This reduction is not sound since we ignored a visible transition: y

- If we ignore condition C3 we can do the following:
  ample(s1) = {t1}
  ample(s2) = {t2}
  ample(s3) = {t3}
- Each state defers the execution of y to a future state.
- When the cycle is closed the depth first search terminates (since the only successor is already visited)
- Transition y is never executed although it is visible!
- Condition C3 prevents this scenario