T–79.4301 Parallel and Distributed Systems (4 ECTS)

T–79.4301 Rinnakkaiset ja hajautetut järjestelmät (4 op)

Lecture 6

25th of February 2008

Keijo Heljanko

`Keijo.Heljanko@tkk.fi`

Adding Assertion Checks

```
search(state v) {
    state v’; transition t;
    if some_assert_fails_in(v) {
        print_counterexample(v); exit(1); /* Terminate */
    }
    forall t in enabled(v) {   /* evaluate all asserts  */
        v’ = fire(v,t); /* firing t at v results in v’  */
        if !RG.has_node(v’) {
            RG.add_node(v’); /* Add new state to V      */
            search(v’);      /* Process it later        */
        }
    }
}
```

Spin Example

```
$ spin -a peterson3
$ gcc -o pan pan.c
$ ./pan
hint: this search is more efficient if pan.c is compiled -DSAFET
(Spin Version 4.2.6 -- 27 October 2005)
        + Partial Order Reduction

Full statespace search for:
        never claim             - (none specified)
        assertion violations    +
        acceptance   cycles     - (not selected)
        invalid end states      +
```

Spin Example (cnt.)

```
State-vector 28 byte, depth reached 615, errors: 0
    2999 states, stored
     806 states, matched
    3805 transitions (= stored+matched)
       0 atomic steps
hash conflicts: 2 (resolved)

2.622   memory usage (Mbyte)

unreached in proctype user
        line 43, state 30, "-end-"
        (1 of 30 states)
```

Spin Example (cnt.)

- The line: “`State-vector 28 byte, depth reached 615, errors: 0`” tells us that each state requires 28 bytes, the DFS search stack depth was 615 at maximum, and that Spin found no errors in the model
- The line “`2999 states, stored`” gives the number of states in the reachability graph
- The text “`3805 transitions`” gives the number of arcs in the reachability graph
- The line “`2.622 memory usage (Mbyte)`” gives the total memory usage needed for the reachability graph generation

Bitstate Hashing

- For analyzing systems where it is not possible to store the states of the reachability graph in the memory, Spin contains additional algorithms
- These algorithms are probabilistic in the following sense: All bugs they report are real bugs but if they do not find bugs, there is still some probability that the system is incorrect
- The best known probabilistic method in Spin is called Bitstate Hashing

Bitstate Hashing (cnt.)

- In basic bitstate hashing the hash table storing the states is replaced with a bit-array a of, e.g., 1 Gigabyte of size. The bits are thus indexed $a[0], a[1], \ldots, a[8858993459 1]$, and are initially 0
- From each state v two hash functions are computed: $h_1(v)$ and $h_2(v)$, the domain of both is $0, 1, \ldots, 8858993459 1$.
- If both $a[h_1(v)] = 1$ and $a[h_2(v)] = 1$, then we assume the state v is already in the reachability graph, otherwise we are sure it has not been seen.
- The state v is added to the reachability graph by setting both $a[h_1(v)]$ and $a[h_2(v)]$ to 1.

Bitstate Hashing (cnt.)

- Bitstate hashing sometimes enables to find bugs in large systems
- If no bugs are found, the result is inconclusive.
- Bitstate hashing should be used as the last resort when all other ways of obtaining verification results have failed

Stateless Search

- A time-memory tradeoff
- Basic idea: Consider a variant of the DFS search algorithm where as the last line of `search(v)` the following line has been added:
  `RG.remove_node(v); /* V is no longer in DFS search stack, remove from RG to save memory */`
  - This variant will also eventually terminate, and will detect all assertion violations
  - In the reachability graph has $|V|$ nodes, the time needed to terminate might be $O(|V|^{|V|})$
  - Not feasible in practice

Statespace Caching

- Statespace caching: Variant of the above, where states are removed from the reachability graph only when running out of memory
- Still all states in the DFS search stack are stored fully to guarantee termination
- Works for some simple systems
- Very unpredictable runtime
- Not implemented in (main release version of) Spin

Symbolic Model Checking

- There are also model checking methods which use symbolic representations of the reachability graph instead of storing each state separately
- As a trivial example, if the system state vector contains three bits x_2, x_1, and x_0, a Boolean formula $x_2 \vee (x_1 \wedge \neg x_0)$ can be used to represent the reachable set of states: $\{010, 100, 101, 110, 111\}$
- *Ordered binary decision diagrams* (OBBDs) are often used to represent Boolean formulas in model checkers. Symbolic model checkers are the topic of the course: T–79.5302 Symbolic Model Checking `http://www.tcs.hut.fi/Studies/T-79.5302/`

Reachability Graph, Definition

Assume that we are give the following mathematical functions:

- $enabled(v)$: Given a global state v, it returns the set of global transitions t that are enabled in v
- $fire(v,t)$: Given a global state v, and a global transition $t \in enabled(v)$, it returns the global state v' reached from v by firing t

Reachability Graph, Definition (cnt.)

Reachability graph $G = (V, T, E, v^0)$ is the graph with the smallest sets of nodes V, global transitions T, and edges E such that:

- $v^0 \in V$, where v^0 is the initial state of the system, and
- if v in V, then for all $t \in enabled(v)$ it holds that $t \in T$, $fire(v,t) \in V$, and $(v, t, fire(v,t)) \in E$.

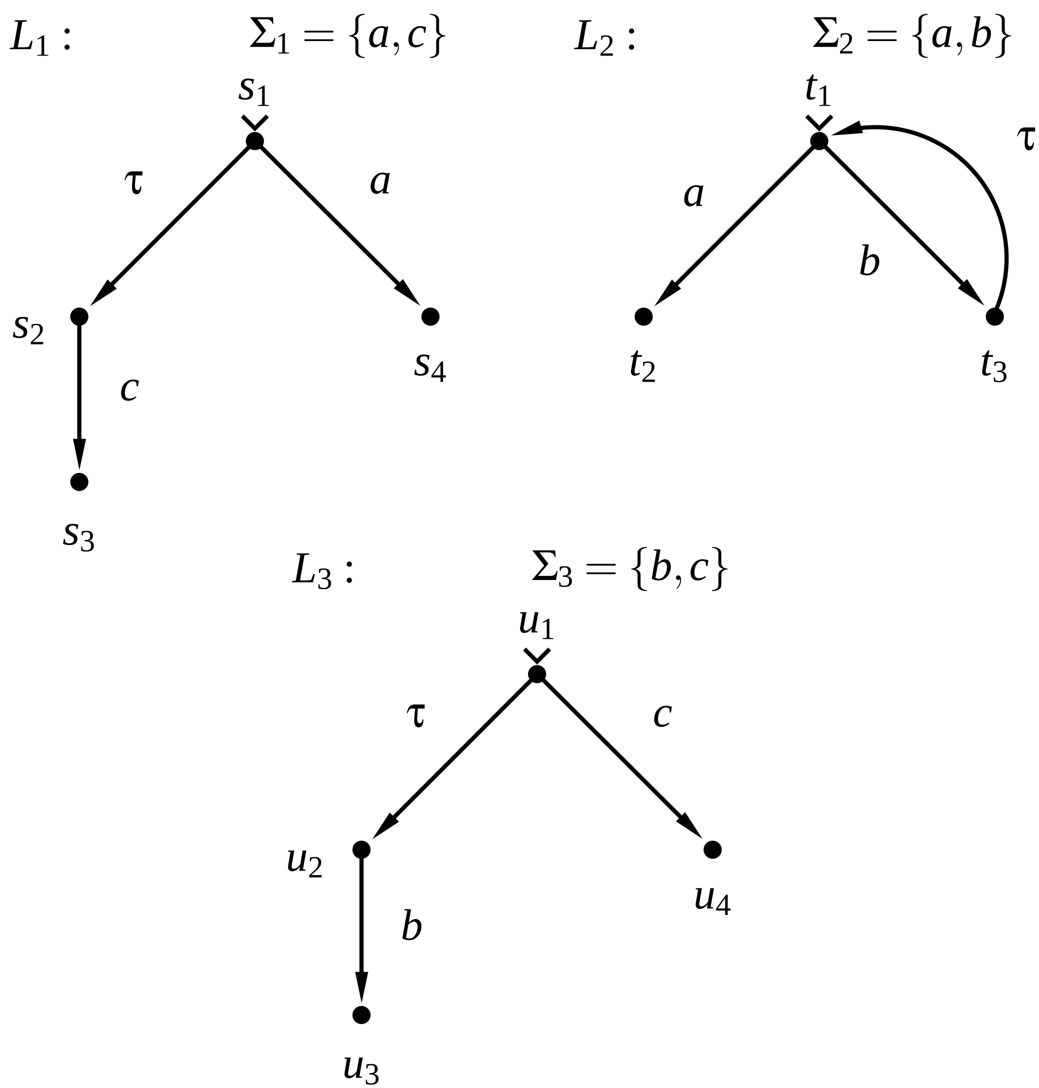
(Note: We could alternatively do the definition above by induction to obtain the same result.)

Running Example - LTSs

Consider now our running LTS example from Lecture 5, reproduced on the next slide for convenience:

- We use as global transitions tuples $t = (t_1, t_2, t_3)$, where $t_i \in \Delta_i$ or $t_i =$ "$-$" in the case the LTS i does not take part in t
- In our running example $v^0 = (s_1, t_1, u_1)$
- $enabled(v^0) = \{ \ ((s_1, \tau, s_2), -, -), ((s_1, a, s_4), (t_1, a, t_2), -), \ (-, -, (u_1, \tau, u_2)) \ \}$
- $fire((s_1, t_1, u_1), ((s_1, a, s_4), (t_1, a, t_2), -)) = (s_4, t_2, u_1)$

Running Example - LTSs (recap)



Deadlocks

Let's now formally define some new concepts for LTSs.

- A *deadlock* is a global state v in the reachability graph such that $enabled(v) = \emptyset$
- Quite often (but not always) deadlocks are signs of "bad behavior" in the analyzed system. (Some behaviors of the system might be modelling "successful" termination of the system.)

Deadlocks (cnt.)

- The process of checking whether the reachability graph contains any deadlock states is called deadlock checking, and it can be easily added to the basic reachability analysis algorithm:
  - Report a deadlock if `enabled(v)` returns the empty list of enabled transitions for some reachable state `v`.

Livelock

- A *livelock* (also called divergence) exists in a state s in an LTS L, if $s \xrightarrow{\tau} s'$ and $s' \xrightarrow{\tau^*} s$ for some state s'.
- Intuitively a livelock (divergence) corresponds to a cycle in the LTS where the LTS performs only internal τ-transitions.
- As is the case with deadlocks, quite often (but not always) livelocks are signs of bad behavior in the analyzed system.
- Livelocks need another (simple) search algorithm to be detected, they cannot be detected with a single DFS or BFS.

Conflict

- A *conflict* occurs in a reachable global state v of $L = L_1 || L_2 || \cdots || L_n$ if there are (at least) two *conflicting transitions* t and t' in $enabled(v)$ such that
  - there is an LTS L_i with $1 \leq i \leq n$, such that $t = (\ldots, t_i, \ldots)$, $t' = (\ldots, t'_i, \ldots)$, and $t_i \neq t'_i$.

In other words, in the case of a conflict there is some component i which has at least two local transitions t_i and t'_i enabled, and it can fire either, having to choose between the two possibilities.

Conflict - Intuition

- If a reachability graph of a system has no conflicts in it, all non-determinism in it is caused by scheduling speeds of components
- Intuitively, conflict free systems are “concurrent, yet fully deterministic”, i.e., their behavior contains no “true non-deterministic choices”. This simplifies their analysis greatly.
- Unfortunately all real systems have conflicts: Whenever there is a resource shared between two components in a mutex manner, a conflict is going to happen when it is allocated to either one of the two components requesting access to it

Independence

- Two global transitions $t = (t_1, t_2, \ldots, t_n)$ and $t' = (t'_1, t'_2, \ldots, t'_n)$ are *independent* iff
  - $t \neq t'$, and
  - for all $1 \leq i \leq n$: if $t_i \neq$ "$-$" then $t'_i =$ "$-$" and if $t'_i \neq$ "$-$" then $t_i =$ "$-$".

Intuitively the set of LTSs which participate in t and t' need to be disjoint for the two transitions to be independent.

Independence (cnt.)

Note that independence (as defined in this course) is a static property of global transitions (independent of the current state v), while conflicts are a dynamic property (both conflicting transitions need to be enabled in v).

Independence (cnt.)

- Two independent transitions t and t' can never be in conflict, and two conflicting transitions t and t' can never be independent.
- If two transitions t and t' are independent and they are both in $enabled(v)$ then they are said to be concurrent. In this case both of the sequences of transition t, t' and t', t can be fired from v, and the two states reached by doing so will be the same.

Independence (cnt.)

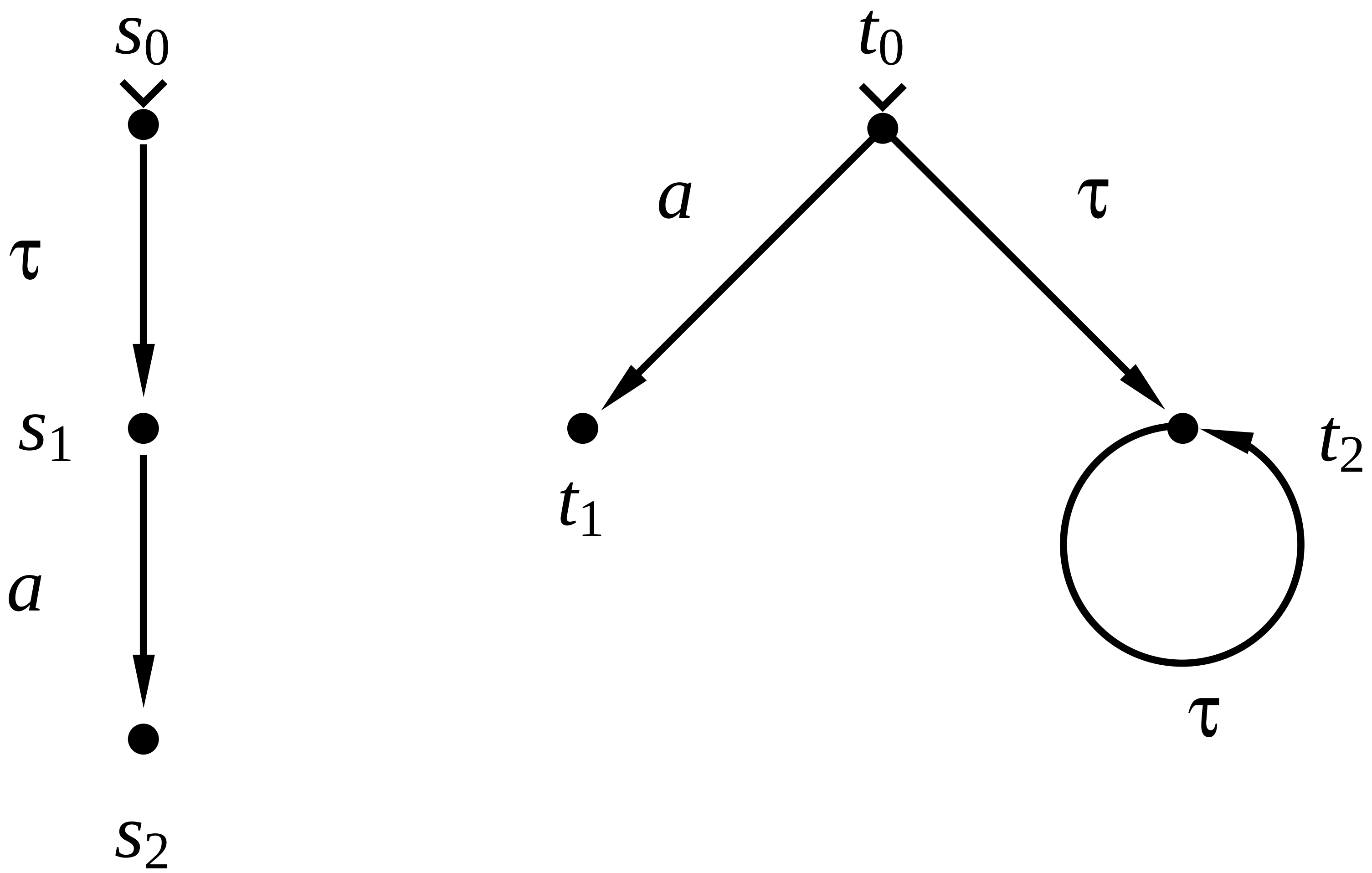
- If a state v has n pairwise independent transitions in $enabled(v)$, then any reachability graph containing v has at least 2^n nodes and $n \cdot 2^{(n-1)}$ edges. There are $n! = 2^{O(n \log_2 n)}$ possible orders of firing the independent transitions.
- Such a structure is called a "diamond", and it can be seen as an n-dimensional hypercube (hint: 1-dimensional hypercube is the line, 2-dimensional hypercube is the square, and 3-dimensional hypercube is the cube) with a single entry vertex, where all edges are directed arcs which are directed towards a single exit vertex

Independence (cnt.)

- So called *partial order reductions* use independence between transitions to remove reachable states from the reachability graph while still preserving, e.g., the existence of deadlocks
- It is a *common beginners mistake* to assume that going through each “diamond” induced by n independent transitions at v by taking the independent transitions of $enabled(v)$ in exactly one order will preserve the existence of, e.g., deadlocks in the reachability graph. This is NOT the case!

Counterexample: Independence

$L_1: \qquad \Sigma_1 = \{a\} \qquad\qquad L_2: \qquad \Sigma_2 = \{a\}$



Example: Independence (cnt.)

- In the initial state $v^0 = (s_0, t_0)$, $enabled(v^0) = \{((s_0, \tau, s_1), -), (-, (t_0, \tau, t_2))\}$
- Now the two enabled transitions are independent
- If we only fire $t' = (-, (t_0, \tau, t_2))$ in v^0, the deadlock state (s_2, t_1) reachable by first firing $t = ((s_0, \tau, s_1), -)$ at v^0 and then firing $t'' = ((s_1, a, s_2), (t_0, a, t_1))$ is no longer reachable

Example: Independence (cnt.)

- Thus by removing some “internal nodes of the diamond” deadlocks of the system can be missed. (Note: This phenomenon is sometimes called “confusion” in the concurrency literature.)
- The partial order reduction methods know how to deal with this problem while still being able to remove some unnecessary interleavings of independent transitions of the system

Partial Order Reductions Disabled

```
$ spin -a peterson3
$ gcc -o pan -DNOREDUCE pan.c
$ ./pan
hint: this search is more efficient if pan.c is compiled -DSAFET
(Spin Version 4.2.6 -- 27 October 2005)

Full statespace search for:
        never claim             - (none specified)
        assertion violations    +
        acceptance   cycles     - (not selected)
        invalid end states      +
```

Partial Order Reductions Disabled

```
State-vector 28 byte, depth reached 5837, errors: 0
   25362 states, stored
   44425 states, matched
   69787 transitions (= stored+matched)
       0 atomic steps
hash conflicts: 791 (resolved)

Stats on memory usage (in Megabytes):
... stuff removed ...
3.236   total actual memory usage
```

Comparison

- The partial order reductions in Spin are on by default but can be disabled by the "`-DNOREDUCE`" compile time option
- Compared to the results in Lecture 5, disabling the partial order reductions results in:
  - Number of stored states rose from `2999` to `25362`
  - Number of transitions rose from `3805` to `69787`
  - The effect was modest (just one order of magnitude) because the example only has three parallel processes. Usually the differences are even larger.

Ample Sets

- The partial order reduction algorithm implemented in Spin is based on a method called ample sets. (Similar methods: persistent and stubborn sets.)
- The most upto-date description of the Spin algorithm can be found from Chapter 10 of the book:
  - Edmund M. Clarke, Jr., Orna Grumberg, and Doron Peled: Model Checking, MIT Press, 1999.
  - `http://mitpress.mit.edu/book-home.tcl?isbn=0262032708`
- The algorithms inside Spin and other explicit state model checkers are a main topic of the course: T–79.5301 Reactive Systems
  `http://www.tcs.hut.fi/Studies/T-79.5301/`

Traces

- The set of traces of an LTS L is defined to be set of sequences of visible actions of L:
  $traces(L) = \{\sigma \in \Sigma^* \mid L \stackrel{\sigma}{\Rightarrow}\}$.
  (Recall: $\tau \notin \Sigma$.)
- Intuitively: $traces(L)$ is the language of all executions of L projected on Σ, thus removing all τ-transitions.
- Another intuition: See L as a non-deterministic FSA $A = lts2fsa(L)$ where all τ-transitions have been replaced with ε-moves, and all states are accepting. Now $traces(L)$ is the language accepted by A.

Traces (cnt.)

- Two LTSs L_1 and L_2 are called trace equivalent iff $traces(L_1) = traces(L_2)$.
- The trace preorder $\leq_{tr}$ is defined as follows: $L_1 \leq_{tr} L_2$ iff $traces(L_1) \subseteq traces(L_2)$.
- Hint: A preorder is a just a relation which is reflexive and transitive.

Traces (cnt.)

- An LTS L deadlocking in the initial state has $traces(L) = \{\varepsilon\}$ (where ε denotes the empty word), and therefore $L \leq_{tr} L'$ for any L'.
- An LTS L with $traces(L'') = \Sigma^*$ is the maximal element of the trace preorder, i.e., $L' \leq_{tr} L''$ for any LTS L'.
  - It is easy to construct L'' such that $traces(L'') = \Sigma^*$: the LTS has one state s^0, and a transition $s^0 \xrightarrow{a} s^0$ for all $a \in \Sigma$.

Checking Trace Containment

- To check whether $L \leq_{tr} L'$ we proceed as follows:
  - Create a FSA $A' = lts2fsa(L')$.
  - Create a FSA $A'' = det(A')$, the determinized version of A' with a total transition relation. (Requires changing the definition of $det(\cdot)$ slightly to also handle ε-moves.)
  - Create a FSA $A''' = \overline{A''}$ by swapping the final and non-final states of A''.

Checking Trace Containment (cnt.)

- To check whether $L \leq_{tr} L'$ (cnt.):
  - See A''' as an LTS L'''.
  - Compute the product $P = L || L'''$.
  - Check if any state (s,t) is reachable in P, where t is a final state of A'''.
    - No: $L \leq_{tr} L'$ holds.
    - Yes: $L \leq_{tr} L'$ does not hold, and there is a sequence $\sigma \in \Sigma^*$ such that $(s^0,t^0) \stackrel{\sigma}{\Rightarrow} (s,t)$, and thus σ is a sequence in $traces(L)$ which does not exist in $traces(L')$.

FSA Determinization with ε-moves

Definition 1 Let $A_1 = (\Sigma, S_1, S_1^0, \Delta_1, F_1)$ be a non-deterministic automaton with ε-moves. We define a deterministic automaton $A = (\Sigma, S, S^0, \Delta, F)$, where

- $S = 2^{S_1}$, the set of all sets of states in S_1,
- $S^0 = \{s' \mid s \xrightarrow{\varepsilon^*} s', \text{ for some } s \in S_1^0\}$,
- $(Q, a, Q') \in \Delta$ iff $Q \in S, a \in \Sigma$, and $Q' = \{s'' \in S_1 \mid \text{ there is } (s, a, s') \in \Delta_1 \text{ such that } s \in Q \text{ and } s' \xrightarrow{\varepsilon^*} s''\}$; and
- $F = \{s \in S \mid s \cap F_1 \neq \emptyset\}$, those states in S which contain at least one accepting state of A_1.

Checking Trace Containment (cnt.)

- Typical application: $I \leq_{tr} S$, where I is an implementation and S is a specification.
- To check the trace containment $I \leq_{tr} S$ one has to determinize S.
- As usual, determinization requires worst-case exponential space in S (hopefully the specification S is relatively small).
- Trace containment is one of the most often used ways of checking properties of systems modelled with LTSs.