T–79.4301 Parallel and Distributed Systems (4 ECTS)

T–79.4301 Rinnakkaiset ja hajautetut järjestelmät (4 op)

Lecture 5

2006.02.24

Keijo Heljanko

`Keijo.Heljanko@tkk.fi`

Home Exercise 1

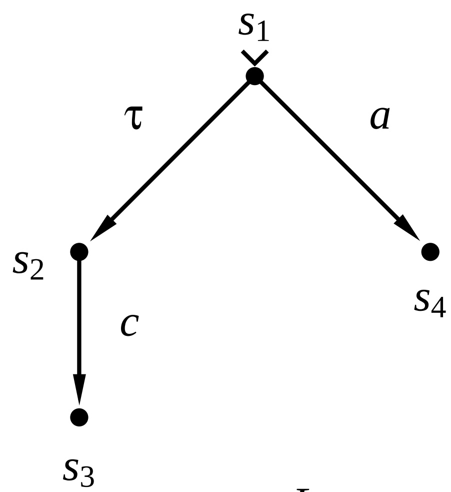
- The home exercise 1 is now available through the course homepage: `http://www.tcs.tkk.fi/Studies/T-79.4301/`
- The exercise is to be done individually, and the topic is modelling an elevator controller in Promela and verifying some safety properties of it with Spin
- The deadline is on Friday 17.3 at 12:15
- The deadline is strict!

Example: Parallel Composition

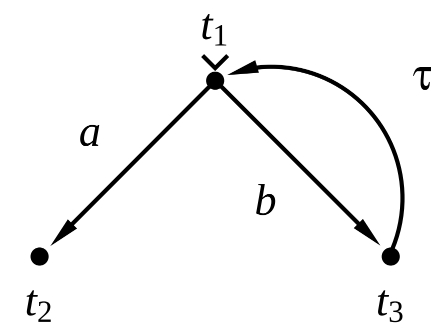
- Recall the definition of the parallel composition operator $||$ from the Lecture 4
- Compute the parallel composition $L = L_1 || L_2 || L_3$, where the LTSs L_1, L_2, and L_3 are given on the next slide

Example: Parallel Composition (cnt.)

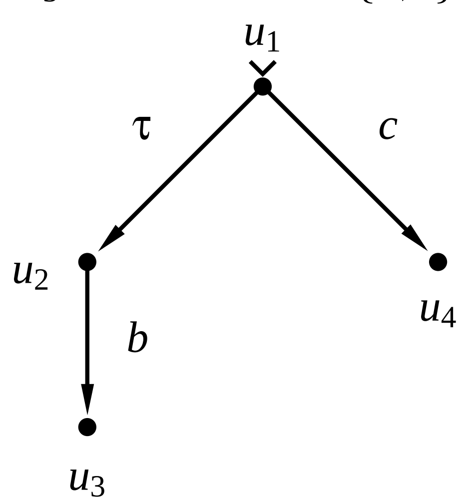
$L_1 :$ $\Sigma_1 = \{a, c\}$



$L_2 :$ $\Sigma_2 = \{a, b\}$

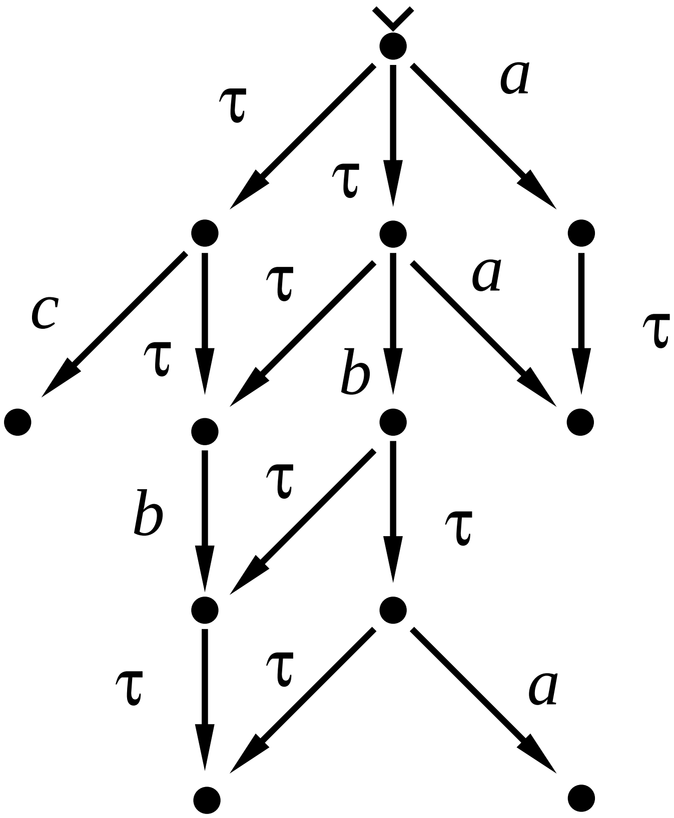


$L_3 :$ $\Sigma_3 = \{b, c\}$



Example: Result $L = L_1 || L_2 || L_3$

$L:$ $\Sigma = \{a, b, c\}$



Reachability Analysis

- Reachability analysis is a way to implement model checking
- We have now shown how parallel composition of LTSs is done directly based on the definition
- Most model checking algorithms are based on an algorithm which implements the generation of a graph containing all the reachable global states of the system
- Let's now give this algorithm in an abstract setting, independent of the used model of concurrency: Thus the algorithm works for, e.g., the parallel composition of LTSs or a Promela

Reachability Graph

- We want to generate a graph $G = (V, T, E, v^0)$, where
- V is the set of reachable global states of the system,
- T is the set of executable global transitions of the system,
- $E \subseteq V \times T \times V$ is the set of executable global state changes of the system (arcs/edges of the reachability graph), and
- $v^0 \in V$ is the initial global state of the system.

Reachability Graph: Subroutines

- We need the following subroutines:
  - `enabled(v)`: Given a global state `v` it returns the list of all global transitions `t` which are enabled in `v`
  - `v' = fire(v,t)`: Given a global state `v`, and a global transition `t` which is enabled at `v`, it returns the global state `v'` reached from `v` by firing `t`

Reachability Graph Algorithm (part 1)

```
graph RG;  /* Global - empty reachability graph */



reachability_graph(state v_0) {

    RG.init();              /* Initialize data structures  */
    RG.add_node(v_0);       /* Add initial state to the RG */
    RG.mark_initial(v_0);   /* Mark the initial state      */
    search(v_0);            /* Process initial state       */

    /* RG now contains the reachability graph */
}
```

Reachability Graph Algorithm (part 2)

```
search(state v) {
    state v';
    transition t;
    forall t in enabled(v) {
        /* Optionally add here: code to add t to T */
        v' = fire(v,t); /* firing t at v results in v' */
        if !RG.has_node(v') { /* v' already processed? */
            RG.add_node(v');  /* Add new state v' to V */
            search(v');       /* Process v'          */
        }
        RG.add_edge(v,t,v');  /* Add arc (v,t,v') to E */
    }
}
```

Implementation Issues

- Modern model checkers such as Spin can handle reachability graphs with the number of reachable states in tens of millions
- The most time and memory critical routines are `RG.has_node(v')` and `RG.add_node(v')`
- Usually the state storage inside model checker is very carefully engineered to minimize memory usage
- In more complex system models the routine `enabled(v)` can become the bottleneck
- In many cases the line `RG.add_edge(v,t,v')` can be removed if only state properties are of interest. Also, usually `enabled(v)` can be recomputed at will

Implementation Issues (cnt.)

- The algorithm presented is depth-first search (DFS), which is the default in Spin
- Also breadth-first search (BFS) is often implemented as it guarantees shortest paths to assertion failure states
- If the set of nodes is too large to fit in the memory, database techniques (B-trees etc.) can be used to implement `RG.has_node(v')` and `RG.add_node(v')`. However, this slows down search by several orders of magnitude.

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 $0, 1, \ldots, 88589934591$.
- 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.)