T–79.4301 Parallel and Distributed Systems (4 ECTS)

Lecture 7

6th of November 2008

Keijo Heljanko

Keijo.Heljanko@tkk.fi

Ample Sets

- The partial order reduction algorithm implemented in Spin exploits the independence between transitions and is called ample sets method. (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
  https://noppa.tkk.fi/noppa/kurssi/t-79.5301/etusivu

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 -DSAFETY
(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.

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 $a[0], a[1], \ldots, a[8589934591]$, 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 $0, 1, \ldots, 8589934591$.
- 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
  `https://noppa.tkk.fi/noppa/kurssi/t-79.5302/etusivu`

Abstraction with Traces

- Quite often we do not have resources to directly check that $I \leq_{tr} S$, because the parallel composition $I = L_1 || L_2 || \cdots || L_n$ is just too big to handle.
- We can often discard unnecessary detail from the implementation by creating some component L'_i such that $L_i \leq_{tr} L'_i$.
- Now if $L_i \leq_{tr} L'_i$ then it can be proved that also $I \leq_{tr} I'$, where I' is I with the component L_i replaced with L'_i.

Abstraction (cnt.)

- Now clearly, if $I' \leq_{tr} S$ then also $I \leq_{tr} S$.
- Thus when using trace containment as the way of checking properties, any component of the implementation can be replaced with another one provided that the new component "has more behavior" than the original.
- Hopefully the new component is smaller than the original one, leading to hopefully small I'.
- This is called abstraction: leaving out unnecessary detail by, e.g., replacing data dependent if-then-else constructs of the modelling language with purely non-deterministic choice.

Abstraction (cnt.)

Examples of abstraction in LTSs preserving traces:

- Some component L_i might be removed altogether by replacing it with the one-state component L_i', such that $traces(L_i') = \Sigma^*$.
- Sequences of τ-transitions can be compressed away in many cases, as long as their firing cannot be indirectly observed in the traces of the component.
- In the LTS domain in particular, it is always safe to add arcs to LTSs as doing so can only increase the set of traces of the component.

Abstraction (warning)

- Note: The set of allowed abstractions depends on the fact that we are using trace containment to check properties!
- A completely different set of allowed abstractions applies if we were, e.g., checking the implementation for deadlock freedom.
- Thus the set of modelling abstractions that are sound depends very closely on the properties that need to be verified from the model!
- Trace containment allows for more freedom in choosing the right abstraction than most other preorders.

Abstraction for Deadlock Detection

We will be less formal here, just a word of warning:

- Intuitively all changes of the implementation I are allowed which might make the modified version I' more "deadlock prone" than I.
- In particular, adding edges can sometimes make new deadlocks to be reachable. However, adding edges might also mean escaping from deadlocks of a component.
- Removing edges can also similarly either add or remove deadlocks.

Abstraction for Deadlock Detection

- A sound but non-optimal solution for preserving deadlocks is to use methods based on bisimulation equivalence (see next slide).
- Better solutions are also available but the details are beyond the scope of this course.

Bisimulation

- Bisimulation (also often called strong bisimulation) is one of the most widely used behavioral equivalences for LTSs.
- It is one of the strongest equivalences around: Replacing a component L_i in a parallel composition I with a bisimulation equivalent component L_i' (denoted $L_i' \sim L_i$) will result in a parallel composition I' such that $I' \sim I$, and will leave all interesting properties of I to be directly verified from I'.

Bisimulation (cnt.)

- In practice a component L_i can always be replaced by a component L_i' for which $L_i \sim L_i'$ holds. There is also a (reasonably) efficient algorithm to obtain such an L_i' with the minimum number of states.
- Properties preserved by bisimulation include traces, deadlocks, livelocks, and all properties expressible by all commonly used specification languages (for example the temporal logics LTL and CTL).
- Because bisimulation preserves so many properties, the changes to the component LTSs preserving bisimulation equivalence are significantly more limited than those preserving trace equivalence.

Bisimulation Definition

Given a pair of LTSs $L = (\Sigma, S, \{s^0\}, \Delta)$ and $L' = (\Sigma, S', \{s^{0\prime}\}, \Delta')$, a relation $B \subseteq S \times S'$ is a bisimulation iff:

- For every state pair (s,t) such that $B(s,t)$:
  - If $s \xrightarrow{x} s'$ for some $s' \in S, x \in \Sigma \cup \{\tau\}$ then there is some $t' \in S'$ such that $t \xrightarrow{x} t'$ and $B(s',t')$; and
  - If $t \xrightarrow{x} t'$ for some $t' \in S', x \in \Sigma \cup \{\tau\}$ then there is some $s' \in S$ such that $s \xrightarrow{x} s'$ and $B(s',t')$.

$L \sim L'$ iff there is some bisimulation B such that $B(s^0, s^{0\prime})$.

Bisimulation Notes

- If $L \sim L'$ then the two LTSs are bisimilar.
- There can be several relations $B_1, B_2, \ldots$ etc. such that L and L' are bisimilar.
- One can prove that the union of any two bisimulation relations is a bisimulation. The bisimulation $B_\sim$ is the largest relation which is still a bisimulation. In other words $B_i \subseteq B_\sim$ for all the other bisimulation relations B_i.

Bisimulation Notes (cnt.)

Bisimulation is

- reflexive: $L \sim L$,
- symmetric: if $L \sim L'$ then $L' \sim L$, and
- transitive: if $L \sim L'$ and $L' \sim L''$ then $L \sim L''$.

Thus bisimulation is an equivalence relation.

Bisimulation Algorithms

- There are reasonably efficient (low-order polynomial in LTS size) algorithms to check two structures for bisimulation equivalence. The algorithmic ideas used are similar to DFA minimization algorithms. (A straightforward implementation runs in time $O(|S| \cdot (|S| + |\Delta|))$).
- The same algorithm can be used for creating the LTS with the minimal number of states that is bisimilar to the LTS given as input.
- Quite often the bisimulation minimization algorithm is used as a preprocessing step before parallel composition.

Bisimulation and Other Equivalences

- There are literally hundreds of equivalences (and preorders) used and almost all of them are weaker than bisimulation and stronger than the trace equivalence.
- For example, the fact that the LTS consisting of a sequence of two τ-transitions is not strongly bisimilar to the LTS consisting of one τ-transition is already quite severe restriction speaking against strong bisimulation.

Bisimulation (recap)

To reformulate:

- Bisimulation makes very few LTSs equivalent which is bad for flexibility of use in abstraction. However, it preserves almost all interesting properties of the system at hand. In addition, the algorithms, especially minimization wrt. bisimulation, are cheap.
- Trace equivalence makes a large number of LTSs equivalent, which is good for the increased flexibility of abstraction. However, it loses several interesting properties of systems such as deadlocks and livelocks. Checking and minimizing (in the few cases it is possible) wrt. trace equivalence are expensive.