T–79.4301 Parallel and Distributed Systems (4 ECTS)

T–79.4301 Rinnakkaiset ja hajautetut järjestelmät (4 op)

Lecture 7

3rd of March 2008

Keijo Heljanko

Keijo.Heljanko@tkk.fi

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 $L' = (\Sigma, S', \{s^{0'}\}, \Delta')$, 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 $B(s^0, s^{0'})$.

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.

Simulation

Closely related to bisimulation is of course the simulation preorder $\leq_{sim}$:

Given LTSs $L = (\Sigma, S, \{s^0\}, \Delta)$ and $L' = (\Sigma, S', \{s^{0'}\}, \Delta')$, a relation $R \subseteq S \times S'$ is a simulation iff:

- For every state pair (s,t) such that $R(s,t)$:
  - 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 $R(s',t')$.

We say that L' simulates L, denoted $L \leq_{sim} L'$ iff there is some simulation relation R such that $R(s^0, s^{0'})$.

Simulation (cnt.)

- Recall that preorders, including simulation, are transitive: if $L \leq_{sim} L'$ and $L' \leq_{sim} L''$ then also $L \leq_{sim} L''$.
- Simulation implies trace preorder: $L \leq_{sim} L'$ implies $L \leq_{tr} L'$. (But not vice versa!)
- Note: Bisimulation is more than simulation both ways: It can be the case that $L \leq_{sim} L'$ and $L' \leq_{sim} L$ but the two LTSs are still not bisimilar: $L \not\sim L'$. (Hint: Simulation both ways at the initial states is not enough to guarantee simulation both ways in all the states.)

Simulation (cnt.)

- One way to show trace containment $I \leq_{tr} S$ is to instead show that $I \leq_{sim} S$.
- In other words, if we can show that the specification simulates the implementation, then also all the traces of the implementation are (good traces) allowed by the specification.

Simulation and Abstraction

- Another use for simulation is to use it to prove soundness of model abstractions.
- If an implementation I is abstracted to (a hopefully smaller/easier to verify) implementation I' such that every execution of I can be simulated by an execution of I' ($I \leq_{sim} I'$) then this implies $I \leq_{tr} I'$.
- Now if we can prove $I' \leq_{tr} S$, then also $I \leq_{tr} S$.

Data Abstraction

- Let us consider data abstraction using a running example.
- Assume the verified model contains an integer variable `x` with a large domain and this leads to state space explosion. Also assume that `x` has the initial value `0`.
- Assume that all operations on `x` are: `x++;`, `x--;` and comparisons: `(x == 0)`, `(x != 0)`.

Data Abstraction (cnt.)

- Now we can try to make the verification model more tractable by replacing all references to `x` with a reference to a Boolean variable `y` tracking the property whether `x` is an even number.
- `y` should be initially `true` as `x` was initially `0`.
- The basic idea is to now employ enough non-determinism in the abstract program in order to be able to simulate the concrete program with it.
- We need to define some new notation. Let `*` in the expression `(* ? foo : bar)` denote the non-deterministic choice between returning the value `foo` or the value `bar`.

Data Abstraction (cnt.)

The operations on `x` now become (in C syntax extended with the non-deterministic choice of a Boolean value `*`):

- `unsigned int x = 0;` becomes `bool y = true;`
- `x++;` becomes `y = !y;`
- `x--;` becomes `y = !y;`
- `(x == 0)` becomes
  `(y ? (* ? true : false) : false)`
- `(x != 0)` becomes
  `(y ? (* ? false : true) : true)`

Data Abstraction (cnt.)

- We get the abstract program by replacing each occurrence of the variable `x` in the concrete program by the syntactic replacement using the variable `y` as shown in the previous slide.
- Intuition on how the replacements were obtained:
  - Do case analysis on the potential values `v` of `x` the current value of `y` might map to, and combine the results with non-determinism:
    - Execute the concrete operation using the value `v` for `x` to obtain a new value `v′` for `x`.
    - Abstract the value of `v′` to the domain of `y` to obtain the new value of `y`.

Data Abstraction (cnt.)

- Consider now the case:
  `(x == 0)` becomes
  `(y ? (* ? true : false) : false)`
- Clearly if we know `x` is odd, the comparison `(x == 0)` will evaluate to false
- If we know `x` is even, in the original program `x` might either have the value `0` or not.
- In order to guarantee that the abstract version is able to *simulate* the behavior of the concrete one in both cases, we will have to do a non-deterministic choice on evaluating `(x == 0)` to either `true` or `false`.

Data Abstraction (cnt.)

- It is now easy to prove that after these syntactic replacements the abstract program P' containing `y` will be able to simulate any execution of the concrete program P containing `x`
- If we can now prove that $P' \leq_{tr} S$ for some specification S, then also $P \leq_{tr} S$.
- Note how non-determinism was required in order to perform the abstraction. Thus non-determinism is a valuable feature in a modelling language.