T–79.4301 Parallel and Distributed Systems (4 ECTS)

Lecture 8

13th of November 2008

Keijo Heljanko

Keijo.Heljanko@tkk.fi

Simulation

Closely related to bisimulation is of 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.

Commonly used Abstractions

We can for example use the following predicates:

- x is even or odd,
- x is zero or non-zero,
- x is `NULL`; and,
- x is positive, zero, or negative.

Predicate Abstraction

We can use abstractions which use boolean variables to record relationships between variables:

- $x\_equals\_z = (x == z)$, and
- $x\_is\_less\_than\_z = (x < z)$.

The last two talk about two variables, and thus can actually replace both x and z with a single boolean variable (the predicate) which is changed in the abstract program whenever either x or z is changed in the concrete program.

This is called predicate abstraction.

Abstraction (cnt.)

- Similar abstraction methods can also be used when manually modelling a system.
- The main goal is to be able to show that when an operation of the concrete program is performed, the abstract program can always simulate it by the abstract version of the operation.
- If this is systematically done, then the traces of the abstract program will always be a superset of the traces of the concrete program.

Abstraction (cnt.)

- Note that the data abstraction method described in the previous slides does not preserve bisimulation.
- However, because simulation is guaranteed, all traces are preserved, and thus the abstraction method can be used for checking trace containment.
- We have only scratched the surface of abstraction methods available.

Building Verification Models

When creating a verification model of a system the following aspects need to be considered:

- Which aspects of the design are important and need verification:
  - What are the correctness requirements of the design?
  - The requirements often tell a lot about which parts of the system to model in order to capture the relevant details. The modelled parts should be exactly those of the required system relevant to the property being verified.
  - Other detail should be discarded.

Building Verification Models (cnt.)

- Model checking is usually best for verifying control flow properties. Data manipulation not necessary for control flow should usually be checked by other means. Discard data in verification models when possible.
- We are looking for the smallest sufficient model to allow verification of the requirement at hand.

Smallest Sufficient Model

- State explosion is usually the main problem in any model checking effort.
- Abstraction is the most efficient way of alleviating the state explosion problem.
- Different properties of the system might in the end need different verification models. (This can lead to version control problems between different models!)
- One should try to avoid unnecessary redundancy in the model.

Avoiding Redundancy

Be careful with unnecessary data. Some examples:

- Temporary variables: get rid of values stored in temporaries not needed for the control flow.
- Generating dummy data that is never used: a sender in a protocol might generate data messages which are never read in the model, just passed along to be discarded in the end. Clearly such data should either be removed or be finally checked in the receiver.
- Sinks, sources, and filters: if model maintainability (readability) allows, processes that only generate, consume, or trivially filter messages can usually be gotten rid of by modifying the model slightly.

Verification Flow

1. Formalize the critical properties to be verified.
2. Construct the smallest sufficient model(s) for the verification task. Use abstraction: watch out for too much detail/concrete model - state explosion might occur.
3. Do the verification by choosing among different alternative options available in the model checker.
4. If a counterexample is found: Either refine the model to remove spurious counterexamples created by abstraction, or modify the concrete system design being modelled to meet its requirements if a real problem with the design shows up.

Verification Flow (cnt.)

- Phase 1. specifying the properties can be a major headache for many systems. If requirements exist, they are usually not formalized, so formalizing the requirements is usually a major undertaking.
- Experience has shown that actually most of the bugs in real designs are found in phase 2. of the verification flow even without starting the model checker. (Modelling is efficient design/code review.)
- Phase 3. can mean modelling changes if the capacity of the available model checker is exceeded.
- Phase 4. is in principle straightforward. However, quite often several iterations are needed.

Safety Properties

- Safety properties are properties of systems that are characterized by the intuitive formulation: "nothing bad happens".
- Another intuition is the following: If some execution σ of the system breaks a safety property, then also all longer executions of the system which begin with σ break the safety property.
- More formally, given an alphabet Σ, a safety property S is a language $S \subseteq \Sigma^*$ such that: for all words $\sigma \notin S$ it holds that $\sigma\sigma' \notin S$ for all $\sigma' \in \Sigma^*$.

Safety - Examples

Examples of safety properties are:

- It is always the case that two processes are never at the same time in the critical section.
- It is always the case that if the system reboots the reset button has been pressed in the past.
- It is always the case that if a message "`ack0`" arrives from the receiver, then a messages "`data0`" has been sent in the past from the sender.
- It is always the case that a multiply command is followed in three clock cycles by the multiply data.

Safety

- Now given an LTS L_S such that $traces(L_S) = S$, we can check whether for an implementation I it holds that $I \leq_{tr} L_S$ as shown earlier in this course.
- The main problem with this approach is that it requires one to compute the deterministic automaton $\overline{det(S)}$ accepting the complement language of S, which might be of exponential size in S.
- This worst-case blow-up happens very seldom, and thus the approach is actually quite useful in practice.

Run-time Verification

- Safety properties have also another nice feature: They can also be observed during the concrete system runtime.
- Thus synchronizing with a safety property observer $\overline{det(S)}$ can be easily simulated in the final implementation of the system to either shutdown a malfunctioning system or just to log (less crucial) violations of safety properties.
- This is called run-time verification and it can only be done for safety properties.