# T–79.4301 Parallel and Distributed Systems (4 ECTS)

*T–79.4301 Rinnakkaiset ja hajautetut järjestelmät (4 op)*

## *Lecture 8*

***2006.03.24***

Keijo Heljanko

Keijo.Heljanko@tkk.fi

HELSINKI UNIVERSITY OF TECHNOLOGY
Laboratory for Theoretical Computer Science

# Data Abstraction (recap of Lecture 7)

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.
- More information about abstraction and other methods used for software model checking can be obtained from the Autumn 2006 edition of the course: T–79.5305 Formal Methods (4 ECTS) P V.

# 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 characterised by the intuitive formulation: “nothing bad happens”.
- Another intuition is the following: If some execution $\sigma$ of the system breaks a safety property, then also all longer executions of the system which begin with $\sigma$ break the safety property.
- More formally, given an alphabet $\Sigma$, 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 synchronising 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.

# History-variables Method

- Another possibility for run-time verification is to use the so called history-variables method.
- This approach is based on using a temporal logic which is capable of only specifying safety properties of the system.
- For simplicity we use a state-based version of the logic and assume we are dealing with a Kripke structure (see Lecture 1) with a set of atomic propositions $AP$.
- The implementation needs to be able to evaluate for each reachable state $s$ whether an atomic proposition $p \in AP$ holds in $s$ or not.

# Past Formulas

The logic we use is a (proper) subset of the temporal logic PLTL (linear temporal logic with past) and will be defined using the following syntax:

- $p \in AP$ is a past formula,
- if $\psi_1$ is a past formula, then $\neg\psi_1$, and $\mathbf{Y}\,\psi_1$ ("yesterday") are past formulas,
- if $\psi_1, \psi_2$ are past formulas, then $\psi_1 \vee \psi_2$ and $\psi_1\,\mathbf{S}\,\psi_2$ ("since") are past formulas.

# Shorthands

We will define the following shorthands:

- $\top = p \vee \neg p$ (“true”) for an arbitrary $p \in AP$,
- $\bot = \neg\top$ (“false”),
- $\psi_1 \wedge \psi_2 = \neg((\neg\psi_1) \vee (\neg\psi_2))$,
- $\mathbf{Z}\,\psi_1 = \neg(\mathbf{Y}\,(\neg\psi_1))$ (“weak yesterday”),
- $\mathbf{O}\,\psi_1 = \top\,\mathbf{S}\,\psi_1$ (“once”),
- $\psi_1\,\mathbf{T}\,\psi_2 = \neg((\neg\psi_1)\,\mathbf{S}\,(\neg\psi_2))$ (“trigger”), and
- $\mathbf{H}\,\psi_1 = \bot\,\mathbf{T}\,\psi_1$ (“historically”).

# Semantics of Past Formulas

The semantics of past formulas is defined at each index $i$ in a word $\pi \in (2^{AP})^*$ such that $\pi = x_0 x_1 x_2 \ldots x_n$ as follows:

$$
\begin{aligned}
\pi^i \models p &\quad\Leftrightarrow\quad p \in x_i \text{ (i.e., } p \text{ holds in } x_i) \text{ for } p \in AP.\\
\pi^i \models \neg\psi_1 &\quad\Leftrightarrow\quad \pi^i \not\models \psi_1.\\
\pi^i \models \mathbf{Y}\,\psi_1 &\quad\Leftrightarrow\quad i > 0 \text{ and } \pi^{i-1} \models \psi_1.\\
\pi^i \models \psi_1 \vee \psi_2 &\quad\Leftrightarrow\quad \pi^i \models \psi_1 \text{ or } \pi^i \models \psi_2.\\
\pi^i \models \psi_1\,\mathbf{S}\,\psi_2 &\quad\Leftrightarrow\quad \exists\, 0 \le j \le i \text{ such that } \pi^j \models \psi_2 \text{ and}\\
&\qquad\qquad \pi^n \models \psi_1 \text{ for all } j < n \le i.
\end{aligned}
$$