T–79.4301 Parallel and Distributed Systems (4 ECTS)

Lecture 9

20th of November 2008

Keijo Heljanko

Keijo.Heljanko@tkk.fi

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 ψ_1 is a past formula, then $\neg\psi_1$, and $\mathbf{Y}\,\psi_1$ ("yesterday") are past formulas,
- if ψ_1, ψ_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:

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

Alternative Semantic Definition

We can alternatively define the semantics of $\pi^i \models \mathbf{Y}\,\psi_1$ and $\pi^i \models \psi_1\,\mathbf{S}\,\psi_2$ recursively as follows:

- $i = 0$:
  - $\pi^0 \not\models \mathbf{Y}\,\psi_1$
  - $\pi^0 \models \psi_1\,\mathbf{S}\,\psi_2 \;\Leftrightarrow\; \pi^0 \models \psi_2$
- $i > 0$:
  - $\pi^i \models \mathbf{Y}\,\psi_1 \;\Leftrightarrow\; \pi^{i-1} \models \psi_1$
  - $\pi^i \models \psi_1\,\mathbf{S}\,\psi_2 \;\Leftrightarrow\; \pi^i \models \psi_2 \vee (\psi_1 \wedge \mathbf{Y}\,(\psi_1\,\mathbf{S}\,\psi_2))$

De Morgan Rules

The De Morgan rules are as follows:

$$
\begin{array}{lcl}
\neg(\neg \psi_1) & \Leftrightarrow & \psi_1 \\
\neg(\psi_1 \vee \psi_2) & \Leftrightarrow & (\neg \psi_1) \wedge (\neg \psi_2) \\
\neg(\mathbf{Y}\,\psi_1) & \Leftrightarrow & \mathbf{Z}\,(\neg \psi_1) \\
\neg(\mathbf{O}\,\psi_1) & \Leftrightarrow & \mathbf{H}\,(\neg \psi_1) \\
\neg(\psi_1\,\mathbf{S}\,\psi_2) & \Leftrightarrow & (\neg \psi_1)\,\mathbf{T}\,(\neg \psi_2)
\end{array}
$$

We also have the duals of the De Morgan rules above, e.g., $\neg(\mathbf{Z}\,\psi_1) \Leftrightarrow \mathbf{Y}\,\neg \psi_1$.

Semantics in a Path

A formula $\mathbf{G}\,(\varphi)$ (“always” φ), where φ is a past formula is called a *past safety formula*. The semantics in a path $\pi = x_0 x_1 x_2 \ldots x_n$ is defined as follows:

- $\pi \models \mathbf{G}\,(\varphi)$ iff for all indexes $0 \leq i \leq n$ it holds that $\pi^i \models \varphi$.

or alternatively:

- $\pi \not\models \mathbf{G}\,(\varphi)$ iff there is an index $0 \leq i \leq n$ such that $\pi^i \models \neg\varphi$.

Semantics in a Kripke Structure

- Recall the definition of a Kripke structure $M = (S, s^0, R, L)$ from Lecture 1.
- An execution σ of M is a sequence of states $\sigma = s_0 s_1 \ldots s_n$ such that $s_0 = s^0$ (starts from the initial state), and $(s_{i-1}, s_i) \in R$ for all $1 \leq i \leq n$ (follows the arcs of the Kripke structure).
- An execution path π of M is a sequence of labels $\pi = x_0 x_1 \ldots x_n$, such that $x_i = L(s_i)$ for some execution $\sigma = s_0 s_1 \ldots s_n$ of M.

Semantics in a Kripke Structure (cnt.)

- The formula φ holds in M, denoted $M \models \varphi$ iff $\pi \models \varphi$ holds for every execution path π of M.
- Or alternatively: the formula φ does not hold in M, denoted $M \not\models \varphi$ iff there is an execution path $\pi = x_0 x_1 \ldots x_n$ such that $\pi \models \neg\varphi$.
  - Such a path φ is called a *counterexample* to property φ, and the corresponding execution σ is called the counterexample execution.

Examples

- $\mathbf{G}\,(\neg(cr_0 \wedge cr_1))$: processes 0 and 1 are never at the same time in the critical section.
- $\mathbf{G}\,(starts \Rightarrow \mathbf{O}\,(ignition))$: if the car starts the ignition key has been turned in the past.
- $\mathbf{G}\,(alarm \Rightarrow \mathbf{O}\,(crash))$: an alarm is given only if the system has crashed in the past.
- $\mathbf{G}\,(alarm \Rightarrow (\neg reset\,\mathbf{S}\,crash))$: an alarm is given only if the system has crashed in the past and no reset has been applied since.
- $\mathbf{G}\,(alarm \Rightarrow \mathbf{Y}\,(crash))$: if an alarm is given, the system crashed at the previous time step.

Implementing the semantics

- To find a safety violation, we need to observe the system state after each step it makes, and report an error at the first index i such that $\pi^i \models \neg\varphi$.
- We do this by using two boolean variables for each subformula ψ. One bit to store the current value of ψ and another bit to remember the value of ψ at the previous time step, denoted by ψ'.
- We can do the calculation of the new values for all the bits as shown in the following slides.
- If after running the system for i steps the top-level formula $\neg\varphi$ evaluates to true we need report that the past safety formula $\mathbf{G}\,(\varphi)$ is violated.

Implementing the semantics (cnt.)

- We will now evaluate the subformula value ψ in bottom-up order. Namely, the evaluation order must be such that both subformulas ψ_1 and ψ_2 of ψ have been evaluated at the current state s_i before ψ is evaluated.
- Each subformula ψ must also be evaluated exactly once at each s_i.
- The implementation is based on the alternative recursive semantic definition.
- To know the contents of the next two slides will not be part of the exam requirements.

The Translation at $i = 0$

Formula ψ	Update rules at $i = 0$
$\psi \in AP$	$\psi = evaluate(s_i, \psi)$
$\neg\psi_1$	$\psi = \neg\psi_1$
$\psi_1 \vee \psi_2$	$\psi = \psi_1 \vee \psi_2$
$\mathbf{Y}\,\psi_1$	$\psi = \bot$ (false)
$\psi_1\,\mathbf{S}\,\psi_2$	$\psi = \psi_2$

Where $evaluate(s_i, \psi)$ evaluates the atomic proposition ψ in the current state s_i.

The Translation at $i > 0$

Formula ψ	Update rules at $i > 0$
$\psi \in AP$	$\psi' = \psi;\ \psi = evaluate(s_i, \psi)$
$\neg\psi_1$	$\psi' = \psi;\ \psi = \neg\psi_1$
$\psi_1 \vee \psi_2$	$\psi' = \psi;\ \psi = \psi_1 \vee \psi_2$
$\mathbf{Y}\,\psi_1$	$\psi' = \psi;\ \psi = \psi_1'$
$\psi_1\,\mathbf{S}\,\psi_2$	$\psi' = \psi;\ \psi = \psi_2 \vee (\psi_1 \wedge \psi')$

Where ψ_1' (ψ') is the value of ψ_1 (ψ) at the previous time step, and $evaluate(s_i, \psi)$ evaluates the atomic proposition ψ in the current state s_i.

History-variables Implementation

- The implementation of the history variables method can be made extremely fast.
- The memory overhead is tiny, just two bits per subformula, out of which the ψ' variables are just temporaries needed to evaluate the new ψ variables.
- It can be used as a fast, low-overhead runtime verification observer for safety properties. The same observer can also be used in combination with a model checker to check safety properties.
- Unfortunately the procedure is not implemented in most model checkers, so it has to be usually implemented by hand.

Liveness

- Liveness properties are properties of systems that are characterised by the intuitive formulation: “eventually something good happens”.
- Another intuition is the following: For finite state systems all counterexamples demonstrating that a liveness property does not hold are of the form $s^0 \xrightarrow{p} s' \xrightarrow{l} s'$, where l is a non-empty execution of the system starting from state s' and ending in state s', and “nothing good” happens in l.
- Thus, intuitively, liveness properties specify what kinds of loops in the system behavior are allowed for correct implementations.

Liveness - Examples

- All executions of the system will pass through a state where *init_done* holds. (An eventuality property.)
- If a data request is sent to a server, the server will always eventually reply with the data. (A progress property: “always eventually” here means “after and arbitrary long but nevetheless a finite number of time steps”.)

Liveness - Examples (cnt.)

- Both process 0 and process 1 are scheduled infinitely often.
- If both process 0 and process 1 are scheduled infinitely often then the request of process 0 to enter the critical section will always eventually be followed by process 0 entering the critical section. (This is often called model checking under fairness. Namely, if the assumption about fair scheduling holds, then the systems satisfies the required progress property.)
- If process 0 is in the critical section, it will leave the critical section after an unbounded but finite number of time steps.

Liveness

- A practical way of specifying liveness properties is to use the temporal logic LTL (linear temporal logic), or its extension PLTL (linear temporal logic with past).
- In LTL we use operators like:
  - $\mathbf{X}\,\psi_1$ (“next”), the future time correspondent to $\mathbf{Y}\,\psi_1$, and
  - $\psi_1\,\mathbf{U}\,\psi_2$ (“until”), the future time correspondent to $\psi_1\,\mathbf{S}\,\psi_2$.
- The semantics of LTL is outside the scope of this course.

Liveness (cnt.)

- How to specify liveness properties in LTL and how to implement their model checking is covered in the course: T–79.5301 Reactive Systems
  `https://noppa.tkk.fi/noppa/kurssi/t-79.5301/etusivu`
- Spin has a full blown LTL model checker (as actually most model checkers do these days), so the tool support is available.

Model Based Testing

- Suppose you have verified safety properties of your system implementation G using model checking methods, and you want to implement it as a concrete program P.
- Can we use automated testing to increase our confidence that P satisfies all safety properties proved from the "golden design" model G?
- The answer is yes. The approach presented for doing so is called model based testing (MBT).

Simplified Testing Framework

To keep things simple we add a couple of restrictions needed to keep our intro to MBT short. We also keep the discussion a bit informal.

- Assume G is an LTS with alphabet Σ divided into inputs Σ_I and outputs Σ_O.
- Let both G and P behave in an input-internal-output loop for each test step i as follows:
  1. Wait for an input $a_i \in \Sigma_I$, all inputs are accepted and acted on.
  2. Do some finite sequence of internal τ-moves. (Non-determinism allowed!)
  3. Send an output $b_i \in \Sigma_O$.

Simplified Testing Framework

- Because of the assumptions above, any sequence $a = a_0 a_1 \ldots a_n \in \Sigma_I^*$ is a valid input test sequence for both G and P.
- Now feed the test sequence to P. It produces the output sequence $b = b_0 b_1 \ldots b_n \in \Sigma_O^*$.
- If $a_0 b_0 a_1 b_1 \ldots a_n b_n \notin traces(G)$ the test verdict is fail, otherwise pass.

Test Verdict Computation

- Intuitively, if $a_0b_0a_1b_1 \ldots a_nb_n \notin traces(G)$, then the concrete program P can after some prefix $a_0b_0a_1b_1 \ldots \ldots a_l$ with $l \leq n$ do b_l, and this cannot be matched by any execution of the golden design G.
- However, in this case P might also violate the safety properties proved for G, and therefore we'd better give a fail test verdict.

Test Verdict Computation (cnt.)

- To check whether $a_0b_0a_1b_1 \ldots a_nb_n \notin traces(G)$, we can see $a_0b_0a_1b_1 \ldots a_nb_n$ as an LTS A, and G as the specification LTS, and then check $A \leq_{tr} G$. If $A \leq_{tr} G$ we give test verdict pass, otherwise fail.
- As you may recall, checking $A \leq_{tr} G$ usually involves determinising G.
- Thus if G has $|G|$ states, the determinised version can have exponentially more states, namely $2^{|G|}$.
- By employing the so called on-the-fly determinisation technique, the memory needed to check $A \leq_{tr} G$ can be bounded by the number of states $|G|$.

Model Based Testing

- The first commercial model based testing tools have become available.
- For example, the testing tools by Conformiq (`http://www.conformiq.com/`) contain automated test generation and execution with MBT techniques.
- For more on model based testing, see the course: T–79.5304 Formal Conformance Testing
`https://noppa.tkk.fi/noppa/kurssi/t-79.5304/etusivu`