Lecture 9
26th of March 2007

Keijo Heljanko
Keijo.Heljanko@tkk.fi
Semantics of Past Formulas (recap)

Recall from Lecture 8 that the semantics of past formulas are defined at each index $i$ in a word $\pi \in (2^{AP})^*$ such that $\pi = x_0x_1x_2 \ldots x_n$ as follows:

\[
\begin{align*}
\pi^i & \models p \iff p \text{ holds in } x_i \text{ for } p \in AP. \\
\pi^i & \models \neg \psi_1 \iff \pi^i \not\models \psi_1. \\
\pi^i & \models Y \psi_1 \iff i > 0 \text{ and } \pi^{i-1} \models \psi_1. \\
\pi^i & \models \psi_1 \lor \psi_2 \iff \pi^i \models \psi_1 \text{ or } \pi^i \models \psi_2. \\
\pi^i & \models \psi_1 S \psi_2 \iff \exists 0 \leq j \leq i \text{ such that } \pi^j \models \psi_2 \text{ and } \pi^n \models \psi_1 \text{ for all } j < n \leq i.
\end{align*}
\]
Alternative Semantic Definition

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

- $i = 0$:
  - $\pi^0 \not\models Y \psi_1$
  - $\pi^0 \models \psi_1 S \psi_2 \iff \pi^0 \models \psi_2$

- $i > 0$:
  - $\pi^i \models Y \psi_1 \iff \pi^{i-1} \models \psi_1$
  - $\pi^i \models \psi_1 S \psi_2 \iff \pi^i \models \psi_2 \lor (\psi_1 \land Y (\psi_1 S \psi_2))$
De Morgan Rules

The De Morgan rules are as follows:

\[
\neg(\neg \psi_1) \iff \psi_1 \\
\neg(\psi_1 \lor \psi_2) \iff (\neg \psi_1) \land (\neg \psi_2) \\
\neg(Y \psi_1) \iff Z(\neg \psi_1) \\
\neg(O \psi_1) \iff H(\neg \psi_1) \\
\neg(\psi_1 \mathcal{S} \psi_2) \iff (\neg \psi_1) T (\neg \psi_2)
\]

We also have the duals of the De Morgan rules above, e.g., \(\neg(Z \psi_1) \iff Y \neg \psi_1\).
Semantics in a Path

A formula $G(\varphi)$ ("always" $\varphi$), where $\varphi$ is a past formula is called a past safety formula. The semantics in a path $\pi = x_0x_1x_2 \ldots x_n$ is defined as follows:

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

or alternatively:

- $\pi \not\models G(\varphi)$ 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 $\sigma$ of $M$ is a sequence of states $\sigma = s_0s_1 \ldots s_n$ 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 $\pi$ of $M$ is a sequence of labels $\pi = x_0x_1 \ldots x_n$, such that $x_i = L(s_i)$ for some execution $\sigma = s_0s_1 \ldots s_n$ of $M$. 
Semantics in a Kripke Structure (cnt.)

- The formula $\varphi$ holds in $M$, denoted $M \models \varphi$ iff $\pi \models \varphi$ holds for every execution path $\pi$ of $M$.

- Or alternatively: the formula $\varphi$ does not hold in $M$, denoted $M \not\models \varphi$ iff there is an execution path $\pi = x_0x_1 \ldots x_n$ such that $\pi \models \neg \varphi$.

- Such a path $\varphi$ is called a *counterexample* to property $\varphi$, and the corresponding execution $\sigma$ is called the counterexample execution.
Examples

- \( G (\neg (cr_0 \land cr_1)) \): processes 0 and 1 are never at the same time in the critical section.
- \( G (starts \Rightarrow O(ignition)) \): if the car starts the ignition key has been turned in the past.
- \( G (alarm \Rightarrow O(crash)) \): an alarm is given only if the system has crashed in the past.
- \( G (alarm \Rightarrow (\neg resetS\crash)) \): an alarm is given only if the system has crashed in the past and no reset has been applied since.
- \( G (alarm \Rightarrow Y(crash)) \): 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 $\psi$. One bit to store the current value of $\psi$ and another bit to remember the value of $\psi$ at the previous time step, denoted by $\psi'$.
- 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 $G(\varphi)$ is violated.
Implementing the semantics (cnt.)

- We will now evaluate the subformula value $\psi$ in bottom-up order. Namely, the evaluation order must be such that both subformulas $\psi_1$ and $\psi_2$ of $\psi$ have been evaluated at the current state $s_i$ before $\psi$ is evaluated.

- Each subformula $\psi$ 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 $i = 0$

<table>
<thead>
<tr>
<th>Formula $\psi$</th>
<th>Update rules at $i = 0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\psi \in AP$</td>
<td>$\psi = evaluate(s_i, \psi)$</td>
</tr>
<tr>
<td>$\neg \psi_1$</td>
<td>$\psi = \neg \psi_1$</td>
</tr>
<tr>
<td>$\psi_1 \lor \psi_2$</td>
<td>$\psi = \psi_1 \lor \psi_2$</td>
</tr>
<tr>
<td>$Y \psi_1$</td>
<td>$\psi = \bot$ (false)</td>
</tr>
<tr>
<td>$\psi_1 S \psi_2$</td>
<td>$\psi = \psi_2$</td>
</tr>
</tbody>
</table>

Where $evaluate(s_i, \psi)$ evaluates the atomic proposition $\psi$ in the current state $s_i$. 
The Translation at $i > 0$

<table>
<thead>
<tr>
<th>Formula $\psi$</th>
<th>Update rules at $i &gt; 0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\psi \in AP$</td>
<td>$\psi' = \psi$; $\psi = \text{evaluate}(s_i, \psi)$</td>
</tr>
<tr>
<td>$\neg \psi_1$</td>
<td>$\psi' = \psi$; $\psi = \neg \psi_1$</td>
</tr>
<tr>
<td>$\psi_1 \lor \psi_2$</td>
<td>$\psi' = \psi$; $\psi = \psi_1 \lor \psi_2$</td>
</tr>
<tr>
<td>$Y \psi_1$</td>
<td>$\psi' = \psi$; $\psi = \psi_1'$</td>
</tr>
<tr>
<td>$\psi_1 S \psi_2$</td>
<td>$\psi' = \psi$; $\psi = \psi_2 \lor (\psi_1 \land \psi')$</td>
</tr>
</tbody>
</table>

Where $\psi'_1$ ($\psi'$) is the value of $\psi_1$ ($\psi$) at the previous time step, and $\text{evaluate}(s_i, \psi)$ evaluates the atomic proposition $\psi$ 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 $\psi'$ variables are just temporaries needed to evaluate the new $\psi$ 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 $s'$, an “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 \textit{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 nevertheless 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:
  - $X \psi_1$ ("next"), the future time correspondent to $Y \psi_1$, and
  - $\psi_1 U \psi_2$ ("until"), the future time correspondent to $\psi_1 S \psi_2$.

- 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
  

- 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).
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 $\Sigma$ divided into inputs $\Sigma_I$ and outputs $\Sigma_O$.

- 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 $\tau$-moves. (Non-determinism allowed!)
  3. Send an output $b_i \in \Sigma_O$. 
Because of the assumptions above, any sequence \( a = a_0a_1 \ldots a_n \in \Sigma_I^* \) is a valid input test sequence for both \( G \) and \( P \).

Now feed the test sequence to \( P \). It produces the output sequence \( b = b_0b_1 \ldots b_n \in \Sigma_O^* \).

If \( a_0b_0a_1b_1 \ldots a_nb_n \notin \text{traces}(G) \) the test verdict is fail, otherwise pass.
Test Verdict Computation

- Intuitively, if $a_0b_0a_1b_1 \ldots a_nb_n \notin \text{traces}(G)$, then the concrete program $P$ can after some prefix $a_0b_0a_1b_1 \ldots a_l$ 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 $a_0b_0a_1b_1 \ldots a_nb_n \notin \text{traces}(G)$, 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 http://www.tcs.hut.fi/Studies/T-79.5304/