Università degli studi di Udine
Laurea Magistrale: Informatica
Lectures for April/May 2014
La verifica del software: temporal logic
Lecture 01 including Overview

Guest lecturer: Mark Reynolds,
The University of Western Australia

April 29, 2014

Description

This short series of 8 lectures introduces temporal logic as a means of formal verification of systems.

Topics

- temporal logic for formal verification overview
- satisfiability testing for LTL, CTL
- model checking for LTL, CTL
- taste of other topics: symbolic model checking, timed automata and metric temporal logic

Lecture 01

- Formal Verification
- Representing systems
- Representing Properties
- LTL
- CTL
- CTL*
- Reasoning tasks

Formal Verification:

Verification: making sure that the system meets its specification (compare to Validation, making sure the specification is what the customer wants).

Approaches to verification include testing, rigorous mathematical proof and formal verification of a model of the system against a formal specification.

Tasks:

The main task is model checking. This is to verify that a model of the system satisfies each formally expressed property capturing each part of the specification.

This is seen as an algorithm which takes as input a formal description of a system and a formal description of a property and outputs “yes” or “no”.

Other tasks we will mention include deciding satisfiability of a property, deciding validity of a property, deciding whether one property is a logical consequence of another, synthesis of a system to satisfy a property, and imperative programming of a system.

Associated Theoretical Questions:

We may mention the computability of a problem and its computational complexity and the computational complexity of an algorithmic solution to a problem.

Approaches:

Fully automatic model checking. Can be by exhaustive search of a model with finite states or using symmetry, symbolic evaluation, abstraction can be extended to some models where the system has an infinite number of states.

The search, especially with large numbers of states, or infinite states, can be “on the fly” where the system states are only explicitly constructed as needed, or only to a certain depth.

Interactive deductive proof is an alternative style of approach.

Modelling the System:

Typically the system is represented as a Finite State Machine.

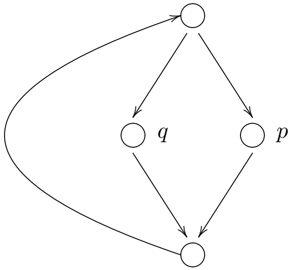
Software tools (e.g. Uppaal, NuSMV, SPIN, Spot, ...) for model checking often have their own formal system description languages.

Extensions may include timed automata, fair transition systems, ...

Model of a System:

Thus we suppose that we can model the system as a finite state machine: a set of states and a set of allowed transitions from some states to other states.

In order to allow abstraction of observable basic, or atomic properties, we also suppose that there are a set of atomic properties, and that some properties are true at some states.



Communicating Components:

Note that this model easily extends to the case of separate but communicating components.

If each component can be modelled as a FSM then the whole system can be taken to be the (Cartesian) product of the separate FSMs: also an FSM.

Each state of the combined system is a tuple of states from each component.

Properties:

Examples.
If a process stays in a waiting state then it will eventually be given a resource.
The program will eventually terminate.
The program will never return a null value.
The two processes will never both be in their critical states at the same time.
The program will never be in a logged-in state unless a correct PIN has been entered beforehand.
The system will always dispense a product after the correct money has been inserted.

Safety and Liveness:

Ideas from Lamport.

A safety property is one saying something of the form “something bad never happens”. Eg, the program never returns null values. Eg, the program never allows two processes to be in the mutual exclusion states.

A liveness property is one saying something of the form “something good eventually happens”. Eg, the program eventually terminates successfully. Eg, if a component is waiting for a resource then it will eventually be granted.

Deadlock:

Deadlock is a common property to want to check for (want to check that it is not possible).
It is simply a state which has no successor states.
It may seem easy to determine if such a state exists.
However, not so easy to tell if such a state is reachable from the initial state. Also not so clear when system FSM is generated as a product of component FSMs involving communication.
We assume in these lectures that all states do have successors. If you want to include deadlocks, then just introduce a deadlock sink state with an appropriate propositional label.

Fairness:

Typical examples of types of fairness constraints...

Every process should be executed infinitely often.

Every process that is enabled infinitely often should be executed infinitely often.

Every process that is eventually always enabled should be executed infinitely often.

Reachability:

Is a certain state (or set of states) reachable from the initial state?

Use a proposition to identify the interesting state(s) and see if that proposition is possibly sometime true, or never true.

Structures:

We assume a countable set $\mathcal{L}$ of propositional atoms, or atomic propositions.

A transition structure is a triple (S, R, g) with S a finite set of states, $R \subseteq S \times S$ a binary relation and for each $s \in S$, $g(s) \subseteq \mathcal{L}$.

R is the *transition relation* and *labelling* g tells us which atoms are true at each time.

R is assumed to be total: every state has at least one successor $\forall x \in S. \exists y \in S \text{ s.t.} (x, y) \in R$

Fullpaths:

Given a structure (S, R, g).
An ω-sequence of states $\langle s_0, s_1, s_2, ...\rangle$ from S is a fullpath (through (S, R, g)) iff for each i, $(s_i, s_{i+1}) \in R$.

If $\sigma = \langle s_0, s_1, s_2, ...\rangle$ is a fullpath then we write $\sigma_i = s_i$, $\sigma_{\geq j} = \langle s_j, s_{j+1}, s_{j+2}, ...\rangle$ (also a fullpath).

LTL Syntax:

Define some strings of symbols as (well formed) formulas, or wffs of LTL.

If $p \in \mathcal{L}$ then p is a wff.

If α and β are wff then so are $\neg\alpha$, $\alpha \wedge \beta$, $X\alpha$, and $\alpha U \beta$.

Read: Not, and(conjunction), tomorrow (or next), and until.

LTL Semantics:

Write $M, \sigma \models \alpha$ iff the formula α is true of the fullpath σ in the structure $M = (S, R, g)$ defined recursively by:

$$\begin{array}{lll}
M, \sigma \models p & \text{iff} & p \in g(\sigma_0), \text{ for } p \in \mathcal{L} \\
M, \sigma \models \neg\alpha & \text{iff} & M, \sigma \not\models \alpha \\
M, \sigma \models \alpha \land \beta & \text{iff} & M, \sigma \models \alpha \text{ and } M, \sigma \models \beta \\
M, \sigma \models X\alpha & \text{iff} & M, \sigma_{\geq 1} \models \alpha \\
M, \sigma \models \alpha U \beta & \text{iff} & \text{there is some } i \geq 0 \text{ such that } M, \sigma_{\geq i} \models \beta \\
 & & \text{and for each } j, \text{ if } 0 \leq j < i \text{ then } M, \sigma_{\geq j} \models \alpha
\end{array}$$

Abbreviations:

Classical: $\top \equiv p \vee \neg p$, $\bot \equiv \neg\top$, $\alpha \vee \beta \equiv \neg(\neg\alpha \wedge \neg\beta)$,
$\alpha \to \beta \equiv \neg\alpha \vee \beta$, $\alpha \leftrightarrow \beta \equiv (\alpha \to \beta) \wedge (\beta \to \alpha)$.
Read: truth, falsity, or(disjunction), implication, iff(equivalence).

Temporal: $F\alpha \equiv (\top U \alpha)$, $G\alpha \equiv \neg F(\neg\alpha)$.
Read: eventually, always.

Example Properties:

$$G\neg(p \wedge q)$$
$$G(p \rightarrow (Xq \vee XXq \vee XXXq))$$
$$G(p \rightarrow Fq)$$
$$G(Gp \rightarrow Fq)$$
$$GFp \wedge GFq$$
$$G(t \rightarrow Gt)$$
$$Fp \rightarrow ((\neg p)U(q \wedge \neg p))$$
$$FGp \vee GFq$$

And that's all for the first lecture.

See you tomorrow for CTL* and more ...