Module Checking

Maël Hörz

June 19, 2008

1 Introduction

In the paper "Module Checking" [2] the authors distinguish between self-contained systems, called closed systems, and systems that interact with an environment, called open systems. They argue that model checking can only verify closed systems and has to be extended to module checking to verify open systems. Furthermore they emphasize that closed systems are uncommon and often are more correctly modeled as open systems, hence module checking would be necessary. Unfortunately, checking if a module satisfies a specification given in CTL is much more expensive (EXPTIME) than the corresponding model checking (linear time). The good news is that for the commonly used fragment of CTL (universal possibly, and always possibly properties), model-checking tools work correctly or can be easily adapted to work correctly.

2 Closed Systems vs. Open Systems

Closed systems have total control, their behavior is completely determined by the state of the system. The behavior of open systems depends on the interaction with its environment.

When modeling systems non-determinism is used to hide complexity, abstract from details and model unknown behavior. In the context of open systems, we talk of internal non-determinism if it describes a choice/move done by the system, and of external non-determinism if it is a choice/move of the system's environment. So internal non-determinism is controllable but external non-determinism is uncontrollable. We can always choose in the implementation how to resolve internal non-determinism to a deterministic behavior that meets the specified requirements, but external non-determinism is uncontrollable and as such cannot be influenced.

2.1 Example

To show the difference between an open and a closed system, we will look at a drink-dispenser machine. Figure 1 depicts a machine that repeatedly boils water and non-deterministically chooses either tea or coffee. It models an open system, module *M,*

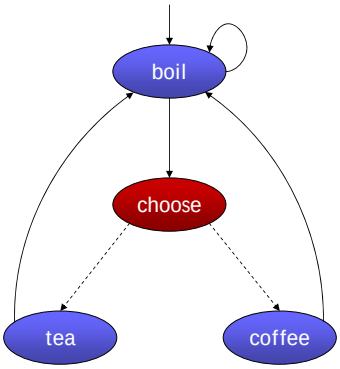

Figure 1: Drink dispenser machine

with the system states “boil”, “tea” and “coffee” and the environment state “choose”. In the state “boil” we know what its possible next states are, since it is a system state, which is controllable. However when M is in the environment state “choose”, there is no certainty w.r.t. to the environment and we cannot be sure that “coffee” and “tea” are possible next states. For example it could be that a user of the machine, i.e. the environment, does not like coffee.

If we regard the drink dispenser as a closed system, program P, then all states are considered to be system states. In this case we know for the “choose” state like for the “boil” state what its next possible states are.

Taking another perspective, this example could also be described as a game between a system player and an environment player. We are the system player and can only control our behavior but not that of the environment player.

3 Module checking

Model checking verifies closed systems, i.e. the system has to satisfy given requirements. Model checking of open systems, or for short module checking, verifies open systems, but here *for all environments* the system has to satisfy given requirements.

The general idea of model/module checking is:

- Express design as a formal model M (usually given as a finite transition system)
- Specify required behavior with a logic formula ψ
- Check that M satisfies ψ

Before going into more detail about module checking it is necessary to define transition systems to express the design and temporal logics to be able to specify requirements.

3.1 Transition systems

In closed systems the formal model is called a program. A program P is a transition system and defined as follows:
$P = (AP, W, R, w_0, L)$ where

- AP: set of atomic propositions
- W: set of states
- $R \subseteq W \times W$: transition relation (must be total)
- w_0: initial state
- $L : W \to 2^{AP}$: maps each state to a set of atomic propositions true in this state

Examples for atomic propositions are:

- The state of a variable are, e.g. x = 5 and y = 7 always holds in state s, then L(s) = {x = 5, y = 7}
- In state "tea" it holds that tea is served

In open systems the formal model is called a module. A module M is a transition system and defined as follows:
$M = (AP, W_s, W_e, R, w_0, L)$ where

- W_s: set of system states
- W_e: set of environment states
- $W = W_e \cup W_s$
- everything else as in definition of P

3.2 Temporal Logics[1]

Reactive systems do not terminate, that means we get infinite execution trees[2]. As there is no end state where we can check the requirements and we want to check properties while execution we need to express temporal aspects in the requirements.

3.2.1 LTL - Linear Temporal Logics

LTL formulas are composed of atomic propositions, boolean connectors and temporal operators.
Syntax:
$\psi ::= true|a|\psi_1 \wedge \psi_2|\neg\psi|X\psi|F\psi|G\psi|\psi_1 \cup \psi_2$, where $a \in AP$
An intuitive overview of LTL's semantics is given in Figure 2.

[1] Some of the presented information about temporal logics were found in [1].
[2] Execution trees are the unrolling of a transition system.

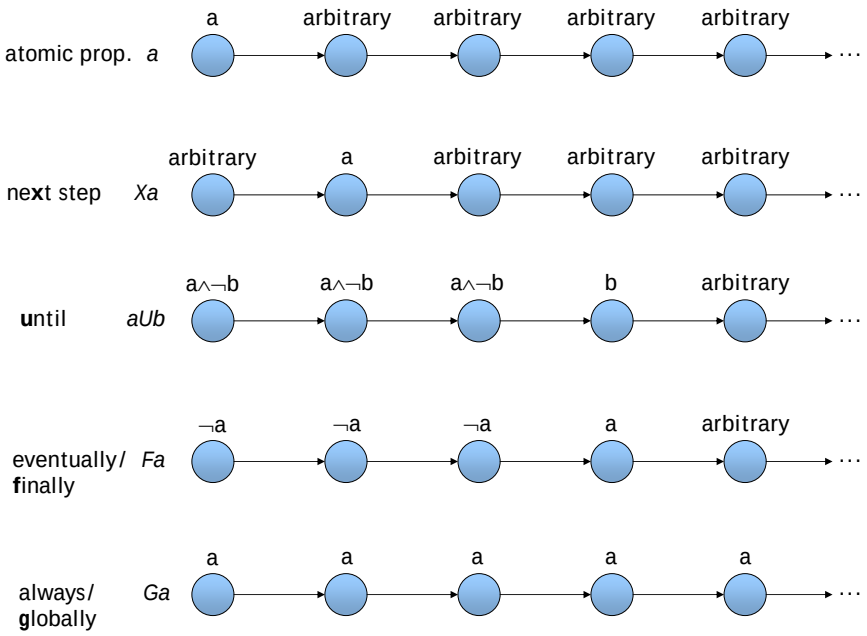

Figure 2: Intuitive semantics of LTL

3.2.2 CTL - Computational Tree Logic

CTL is built of state and path formulas.
State-formulas:
$\Phi = true|a|\psi_1 \wedge \psi_2|\neg\psi|\exists\psi|\forall\psi$, where ψ, ψ_1, ψ_2 are path formulas.
Path formulas:
$\psi = X\Phi|F\Phi|G\Phi|\Phi_1 \cup \Phi_2$, where Φ, Φ_1, Φ_2 are state formulas.
Temporal operators X,F,G,U have analog semantics to their LTL-semantics.
New: Path-quantifiers:

- Let ψ be a path-formula
- Universal path quantifier $\forall$: ψ has to hold on all paths
- Existential path quantifier $\exists$: ψ has to hold on at least one path

Finally CTL* is like CTL, but temporal operators can be freely mixed.

3.3 Exec-Function

Execution tree is the tree obtained from unrolling a program/module. In module checking we have a set of execution trees, *exec(M)* instead of just one execution tree. By pruning from the execution tree of M sub-trees, which have as root node a successor

of an environment node, we obtain *exec(M)*. Figure 3 exemplifies the computation of *exec(M)*.

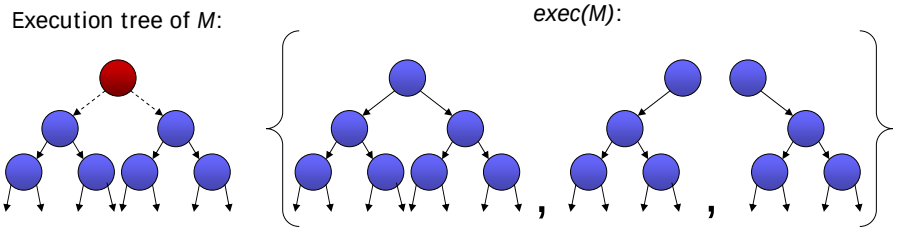

Figure 3: Prune sub-trees to obtain *exec(M)*

Intuitively each tree in *exec(M)* corresponds to a different behavior of the environment. Formally, given a module M, CTL* formula Ψ, model checking is verifying that all trees in *exec(M)* satisfy Ψ. I.e. apply model checking to all trees in *exec(M)*.

3.4 Complexity

Module checking problem for ∀CTL is in linear time and for LTL, ∀CTL* it is PSPACE-complete. Program complexity of module checking for LTL, ∀CTL, ∀CTL* is NLOGSPACE-complete.

Proof-idea: For LTL, ∀CTL, ∀CTL* model checking and module checking coincide. This is because we have no existential quantification but only universal quantification, therefore we cannot distinguish between environment and system, i.e. closed and open systems are not distinguishable either.

Module checking problem for CTL is EXPTIME-complete. Module checking problem for CTL* 2EXPTIME-complete.

Proof-idea: Model checking problem for CTL is in linear-time. Module checking of model M runs model-checking on all trees in exec(M). And as exec(M) is a subset of the power set of M we get an exponential blow-up.

3.4.1 Good news

Is it really that bad? Fortunately for the commonly-used fragment of CTL (universal possibly, and always possibly properties) model checking tools can be easily adjusted. For these commonly-used CTL fragments model checking problem is in linear time and program-complexity of module is PTIME-complete.

References

[1] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. 2006.

[2] Orna Kupfermann, Moshe Y. Vardi, and Pierre Wolper. Module checking. 1998.