Applying Verification Methods to Non-Exhaustive Verification of Software/Hardware Systems

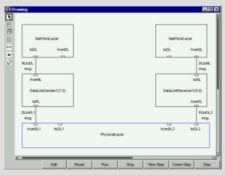
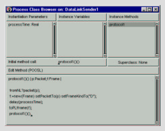
M.C.W. Geilen, D.R. Dams and J.P.M. Voeten
Information and Communication Systems Group
Eindhoven University of Technology, The Netherlands

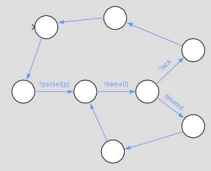
Abstract - In order to handle the increasing complexity of hardware / software designs, system level design methods are being used. These methods are directed to produce operational system models at a high level of abstraction. They can be used to assess early in the design phase if specific functional or performance requirements can be met. It is not always easy to check whether a model indeed satisfies the desired requirements. If the specification model has a well-defined semantics, and the requirements can be expressed exactly, it is possible to automate some of these checks. Several techniques exist to verify if a given model satisfies certain formally defined properties. A popular approach is model-checking, in which the verification problem is reduced to standard checks on finite state automata, as used in the tool Spin [Hol91] for example. We investigate the use of such automata based verification techniques in simulation of high-level system specifications in POOSL [PV97]. We show how certain properties expressed in the formalism Linear Temporal Logic (LTL), can be automatically monitored during simulations of complex distributed systems.

[Hol91] G. Holzman, Design and Validation of Computer Protocols, Prentice-Hall, Englewood Cliffs, New Jersey, 1991.
[PV97] P.H.A.v.d. Putten and J.P.M. Voeten, Specification of Reactive Hardware / Software Systems, Ph.D. thesis, Eindhoven University of Technology, Departement of Electrical Engineering, 1997.

POOSL Models

- In the SHE methodology [PV97], models are specified in the language POOSL
- The language POOSL has a formal semantics, which gives the meaning of a POOSL model in terms of a labelled transition system



- A POOSL model defines a (possibly infinite) labelled transition system
- One would like to check (automatically) if this labelled transition system has all the required properties

Temporal Logic Model Checking

- As a system executes by making discrete steps, a number of boolean properties can be observed in every state. Such a sequence of observations is called a trace $s=\sigma_0\sigma_1\sigma_2...$
  A non deterministic system S, has a set of possible traces it can produce during execution (L(S))
- Temporal Logic allows one to express properties of such traces. A property φ implicitly defines a set (L(φ)) of all traces, that satisfy the property

Syntax of LTL (Linear Temporal Logic)

$$true \mid p \mid \neg\varphi \mid \varphi_1 \vee \varphi_2 \mid \bigcirc\varphi \mid \varphi_1 \mathrm{U}\, \varphi_2$$

Semantics of LTL

- *true* holds for any trace
- *p* holds for a trace if *p* holds in its first state
- $\neg\varphi$ holds if property φ does not hold
- $\varphi_1 \vee \varphi_2$ holds if either φ_1 or φ_2 holds
- $\bigcirc\varphi$ holds if property φ holds from the second state onwards
- $\varphi_1 \mathrm{U}\, \varphi_2$ holds if φ_2 holds from some state number *n* onwards, and for all *k<n*, property φ_1 holds from state number *k* onwards.

- A system specification defines by means of its labelled transition system, a set of traces that represent its possible behaviours. Temporal Logic Verification then boils down to checking if L(S) is included in L(φ)

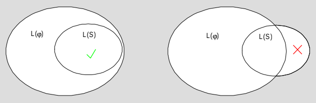


Tableau Construction

- It is possible to construct for any LTL formula, an ω-automaton, which accepts precisely all traces that satisfy this property [WVS83]
- Every state of the automaton corresponds to a formula, that all traces starting from that state will satisfy.
- The automaton is generated, by repeatedly separating constraints on the current state of the system and constraints on the rest of the trace, by rewriting the formula into disjunctive normal form:

$$p_{1,1} \wedge p_{1,2} \wedge \cdots \wedge p_{1,n1} \wedge \bigcirc(\varphi_{1,1} \wedge \varphi_{1,2} \wedge \cdots \wedge \varphi_{1,m1}) \vee$$
$$p_{2,1} \wedge p_{2,2} \wedge \cdots \wedge p_{2,n2} \wedge \bigcirc(\varphi_{2,1} \wedge \varphi_{2,2} \wedge \cdots \wedge \varphi_{2,m2}) \vee \ldots \vee$$
$$p_{k,1} \wedge p_{k,2} \wedge \cdots \wedge p_{k,nk} \wedge \bigcirc(\varphi_{k,1} \wedge \varphi_{k,2} \wedge \cdots \wedge \varphi_{k,mk})$$

Satisfying the 'until' formula $\varphi_1 \mathrm{U}\, \varphi_2$ can, for example, be done by satisfying φ_2 in the current state, or by satisfying φ_1 now and $\varphi_1 \mathrm{U}\, \varphi_2$ later:

$$\varphi_1 \mathrm{U}\, \varphi_2 \equiv \varphi_2 \vee (\varphi_1 \wedge \bigcirc\, \varphi_1 \mathrm{U}\, \varphi_2)$$
$$\neg(\varphi_1 \mathrm{U}\, \varphi_2) \equiv \neg\varphi_2 \wedge \neg\varphi_1 \vee \neg\varphi_2 \wedge \bigcirc\neg\varphi_1 \mathrm{U}\, \varphi_2$$

- For every conjunction in the normal form, an edge is added to the automaton, which is labelled with the constraints on the current state, and leading to a new state, representing the constraints behind the $\bigcirc$ -operator

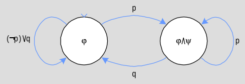
[WVS83] P. Wolper, M.Y. Vardi and A.P. Sistla, Reasoning about Infinite Computation Paths, Proc. 24th IEEE Symp. Foundations of Computer Science, Tuscan, 1983, pp. 185-194

Example

- As an example, an ω-automaton is constructed, accepting precisely all traces that satisfy the formula: $\Box(p \Rightarrow p \mathrm{U}\, q)$
- Determine the normal forms using the abbreviations: $\varphi = \Box(p \Rightarrow p \mathrm{U}\, q)$, $\psi = p \mathrm{U}\, q$

$$\varphi \equiv \neg p \wedge \bigcirc\varphi \;\vee\; q \wedge \bigcirc\varphi \;\vee\; p \wedge \bigcirc(\varphi \wedge \psi) \qquad \varphi \wedge \psi \equiv q \wedge \bigcirc\varphi \;\vee\; p \wedge \bigcirc(\varphi \wedge \psi)$$

- Leading to an automaton with two states. Note that the automaton will not verify the liveness property that every occurrence of a p-observation will eventually be followed by an occurrence of a q-observation, as explained below



Simulations

- Simulation involves systems with a potentially infinite state space and states are not stored. Therefore, infinite sequences are not dealt with explicitly, only finite prefixes of such sequences are inspected
- Properties of such systems can have *liveness* aspects, which informally state that: "something good will eventually happen" and *safety* aspects, stating that "something bad will never happen"
- Only safety aspects of LTL properties can be monitored during simulation, since for liveness properties, the "good thing" may always still happen in the future
- As soon as the automaton has witnessed a prefix of the execution of the system that can no longer lead to satisfaction of the property, it will detect this since it will have no more possible transitions

Conclusions

- For a property φ, an ω-automaton can be constructed that accepts precisely all executions, that satisfy this property
- These automata can be used during simulations to automatically verify if the execution can still satisfy all desired properties
- In contrast with exhaustive verification techniques, this technique can be applied to much larger / more adequate models, however at the cost of providing less coverage
- The technique can help in identifying the hard problems that require the extra effort of building adequate abstract models, especially for exhaustive formal verification

Future Work

- Creating a language in which to express LTL properties in terms of the language concept of POOSL, such as data objects, hierarchy, object-orientation etc.
- Implementation of the verification techniques in existing simulation tools for POOSL
- Extension of the techniques to timing properties