Applying verification methods to non-exhaustive verification of software/hardware systems

M.C.W. Geilen, D. R. Dams and J.P.M. Voeten

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. Since the systems that are being designed are often concurrent and real-time, the behaviour of these models can become rather complex. 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, which is an automata based approach in which the verification problem is reduced to standard checks on finite state automata, as used in the tool Spin [1] for example.**

In this paper we investigate the use of such automata based verification techniques in simulation of high-level system specifications in POOSL [2]. We show how certain properties expressed in the formalism linear temporal logic (LTL), can be automatically monitored during simulations of complex distributed systems.

Keywords— **formal verification, temporal logic, tableaux, simulation, object-oriented methods**

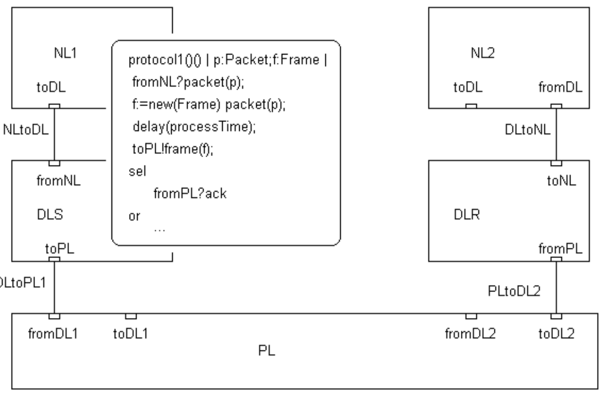
I. Introduction

Systems that need to be designed are becoming more an more complex. They are often real-time concurrent systems, consisting of a large number of components operating together. These systems need to be designed as fast as possible, containing as few errors as possible. Formal methods are very helpful to automate parts of the design process and to design with fewer errors. Systems, designed using formal methods have a well-defined behaviour, which allows the use of automated tools. Automatic verification techniques can be used to assess the correctness of the design. Such techniques include simulations and formal exhaustive verification techniques. Simulations are easy to use, easy to understand and provide much insight in the behaviour of the system. Their capability of finding errors however is limited. Exhaustive verification methods on the other hand are better at finding certain types of errors. They are however still hard to use and suffer from the state space explosion problem. This makes that they can only be applied to either very small or very abstract systems. In the latter case, it might be possible that the abstract model does no longer exhibit all errors present in the actual design. Both methods should therefore be used, simulation in early stages and on global design, exhaustive methods later on and concentrating on specific problem areas. This paper describes the possibility to verify certain temporal logic requirements during simulation of concurrent distributed system models. Section II describes an example in the language POOSL and the transition systems that are generated by POOSL models. Then, temporal logic is discussed as a way to formalise system requirements. A known method to build transition systems from temporal logic formulas, called tableau construction [3], [4] is shown. Section VI shows how these transition systems can be used during simulation, to verify if a run of the system satisfies its requirements.

Section of Information and Communication Systems, Faculty of Electrical Engineering, Eindhoven University of Technology, P.O.Box 513, 5600 MB Eindhoven, The Netherlands. E-mail: geilen@ics.ele.tue.nl

II. POOSL models

As an example of system-level formal models, we will look at models in the language POOSL (Parallel Object-Oriented Specification Language.) This language is part of a specification and design methodology for hardware/software systems, SHE (Software/Hardware Engineering, [2].) A POOSL model describes a system as a set of asynchronous concurrent processes, connected by channels. Figure 1 shows an example of a POOSL model of a datalink protocol. The inserted text box shows a part of the description of the behaviour of one of the processes, the datalink sender (DLS.)



```
protocol1()() | p:Packet;f:Frame |
 fromNL?packet(p);
 f:=new(Frame) packet(p);
 delay(processTime);
 toPL!frame(f);
sel
    fromPL?ack
or
    ...
```

Fig. 1
A POOSL model

The language POOSL has a formal semantics, which defines a discrete transition system, based on the POOSL model. This transition system can be finite or infinite. Figure 2 shows such a transition system for the Datalink Sender object of the POOSL specification. Transitions represent either internal transitions of the system or communications with other systems. Concurrency is modelled as interleaving of discrete transitions of the individual components. A transition in the global transition graph represents a local internal transition or, in case of a synchronous communication, a simultaneous transition of the communicating processes.

A tool has been implemented, which simulates the behaviour of POOSL models by traversing this transition system. Validation is currently done by manually observing this simulation.

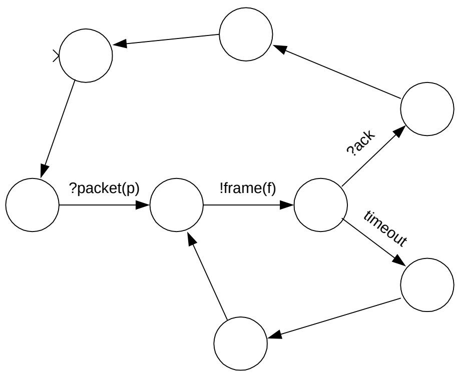


Fig. 2
TRANSITION SYSTEM DEFINED BY POOSL SEMANTICS

III. TEMPORAL LOGIC

A popular formalism to express properties of state-transition based concurrent systems is temporal logic. There exist several varieties of temporal logics. Linear temporal logic (LTL) expresses features of one individual (possibly infinitely long) execution of a system. An LTL formula is then supposed to hold for all possible executions that the system might produce. So-called branching time logics can in addition express features of choice points during system execution. In this paper we will consider LTL, because we want to verify an individual run that occurs during a simulation of the system. The execution of a system can be observed as an (infinite) discrete sequence of boolean values that can be evaluated in every state of the system. Such values can for example correspond to the fact that a message has just been received, or that a buffer is full. Without specifying how these observable features are defined, we will call them atomic propositions and denote them with the letters p, q, etc. We can now define a trace as the observation of the execution of a system. For simplicity, we will consider only infinite traces.

Definition 1: (trace)

A trace $s = \sigma_0\sigma_1\sigma_2\ldots$ *is an infinite sequence of sets of atomic propositions.*

$p \in \sigma_n$, precisely if the atomic proposition p is observed to be true in state n of the execution. If S is a system, we use $[S]$ to denote the set of traces that S can produce. If $s = \sigma_0\sigma_1\sigma_2\ldots$ then $s^n = \sigma_n\sigma_{n+1}\sigma_{n+2}\ldots$, the remainder of the execution from the n^{th} state onwards.

We will now give the syntax and the meaning of LTL formulas φ, by defining which traces satisfy them. The syntax of LTL is given by the following grammar:

$$\varphi ::= true \mid p \mid \neg\varphi \mid \varphi_1 \vee \varphi_2 \mid \bigcirc\varphi \mid \varphi_1 \mathbf{U} \varphi_2$$

The meaning of LTL formulas is:

- $true$ holds for any trace;
- the formula p refers to the atomic proposition p, and asserts that it is true in the first state of the trace;
- $\neg\varphi$ holds for a trace if the formula φ does not hold;
- $\varphi_1 \vee \varphi_2$ expresses that either formula φ_1 or formula φ_2 holds for the trace;
- $\bigcirc\varphi$ (pronounced as “next φ”) states that the formula φ should hold for the remainder of the execution without the first state;
- $\varphi_1\mathbf{U}\varphi_2$ (“φ_1 until φ_2”) is the most complex operator. It holds for a trace s, if there is some state σ_n, such that for the trace from the n^{th} state onwards (s^n) property φ_2 holds and for all $k < n$, s^k satisfies property φ_1.

The fact that a trace s satisfies a formula φ, will be denoted as $s \models \varphi$.

Other operators can be defined in terms of these operators:

$$\begin{aligned} false &\equiv \neg true \\ \varphi_1 \wedge \varphi_2 &\equiv \neg(\neg\varphi_1 \vee \neg\varphi_2) \\ \varphi_1 \Rightarrow \varphi_2 &\equiv \neg\varphi_1 \vee \varphi_2 \\ \Diamond\varphi &\equiv true\mathbf{U}\varphi \\ \Box\varphi &\equiv \neg\Diamond\neg\varphi \end{aligned}$$

The meaning of the formulas $false$, $\varphi_1 \wedge \varphi_2$ and $\varphi_1 \Rightarrow \varphi_2$ are obvious. $\Diamond\varphi$ holds for a trace s if φ will ‘eventually’ hold for some s^n. $\Box\varphi$ holds for s if φ ‘always’ holds, i.e. for every s^n.

A formula implicitly defines a set of traces, that satisfy it. We will denote this set by $[\varphi]$. If φ is a requirement for system S, one would like to check if $[S] \subseteq [\varphi]$, i.e. if every trace generated by system S satisfies the property φ.

For example, suppose we have a system with the following observable properties: $msgArrived$ ($true$ if a new message has just arrived) and $bufferFull$ ($true$ if the buffer in which incoming messages will be stored is full). One could now express the requirement that a new message will never arrive when the message buffer is full, as follows

$$\varphi = \Box(msgArrived \Rightarrow \neg bufferFull).$$

This can be pronounced as, “it will always ($\Box$) be the case that if a message has just arrived ($msgArrived$), then ($\Rightarrow$) the buffer will not be full ($\neg bufferFull$.)”

IV. MODEL CHECKING

When we have a formal model of the system and have expressed (some of) its requirements in LTL formulas, we would like to show that this system does indeed satisfy these requirements. A popular technique for automatically showing that this is the case, is model checking.

A. Model checking techniques

Suppose we have a description of all possible behaviours of the system as a finite state ω-automaton A_S (finite state automata accepting infinite words rather than finite words, see section V.) We now want to check whether every possible behaviour of A_S, is a model of the desired property φ (i.e. satisfies the desired property φ), hence the name: model-checking. The automaton-based approach constructs for the property φ, an ω-automaton A_φ that accepts precisely the traces that satisfy φ. This automaton is called a tableau-automaton.

The model checking problem can be solved by solving the language inclusion problem. S satisfies φ if $L(A_S) \subseteq L(A_\varphi)$. This can be done in a time proportional to the product of the number of states of A_S and A_φ (by constructing the automaton for $\neg\varphi$ rather than φ, and checking the product automaton for emptiness.) The number of states of the automaton A_S however, is generally exponential in the size of the description of the system. The size of A_φ is exponential in the length of the formula. For complex systems, this check is often too hard to perform in reasonable amounts of memory and time.

Some techniques have been developed to reduce the complexity of the model-checking problem:

• *Abstraction*. The model can sometimes be made more abstract by removing processes or data that do not influence the property to be checked [5]. This is done preferably in such a way that the result can be proven to hold also in the original system, but is often based on the judgement of the designer.

• *Symbolic model checking*. This is a technique in which the state space is not explored explicitly. Sets of states, rather than individual states are represented symbolically, for example by BDDs as used in SMV[6] . If the state space contains much regularity, this representation can be compact.

• *Partial order reduction*. Sometimes the specific order of events in the system is unimportant. It is then not necessary to explore all possible orders. This is especially the case if the transition system originates from interleaving relatively independent concurrent processes. This is used for example in Spin [1].

Another technique to reduce the problem is non-exhaustive verification. The above-mentioned techniques reduce the verification problem while achieving the same result, deciding if the system satisfies the formula. But when even these techniques do not reduce the problem enough to be able to perform the check in a reasonable amount of time, one has to settle for a search through a part of the state space that is as large as possible. This is done for example in SPIN's supertrace algorithm[1]. If a trace is found, that does not satisfy φ, one knows that φ does not hold for the system. If such a trace is not found however, one cannot be sure that it does.

B. Model checking techniques and simulations

Certain model checking techniques can also be applied to simulation. Simulation is a popular technique for validation of a design. We will define simulation as a state space exploration that does not store the states that have been visited neither explicitly, nor implicitly. We will now compare some characteristics of exhaustive verification / model checking and simulation techniques.

B.1 Exhaustive verification

The following aspects apply to exhaustive verification methods.

• *State space explosion.* The number of states of a system grows exponentially with the size of the system. This makes that exhaustive exploration of the entire state-space (even symbolically or using other reduction techniques) is only feasible for relatively small systems.

• *Abstract models have to be used.* As a consequence of the first characteristic, very abstract models of the system under verification have to be used. This leads to the danger that this model might not capture all behaviour of the actual system and might not exhibit all errors contained in the concrete system.

• *Hard to use and understand.* It requires a substantial amount of expertise to use formal verification methods. This expertise is necessary for example to model a system in such a way that its state space remains within reasonable bounds. Moreover, some expertise is required to express the desired properties in some form of formal logic and to select and apply specialised techniques for state space reduction.

• *Guaranteed to find errors.* Since an exhaustive verification will search the entire state-space for errors, it is guaranteed to find all errors in the model. This is however not necessarily true for the real-world system that is being verified. Since the model is an abstraction of the system, the system might show errors that the model does not (and possibly vice versa), or worse, the model might not adequately capture the real behaviour of the system. Furthermore, only those requirements are verified, that are captured by the specified formal requirements. It is often difficult to completely specify, which behaviour is correct or incorrect. Some requirements cannot be expressed by the formal logic at all. Finally, this guarantee is only useful if the verification algorithm terminates within a reasonable amount of time.

B.2 Simulation Techniques

To compare exhaustive verification techniques with simulations, we will now discuss some properties of simulations.

• *Non-exhaustive.* Simulations typically start exploring the state space from the initial state, without remembering which states have been visited before. It is therefore impossible to know when all states have been visited. It will rely on probability to explore new states instead of ones it has seen before. Therefore it will in general not be able to explore the entire state space.

• *Poor coverage.* The chance of finding an error by simulation depends heavily on the type of error [7]. Some errors occur in a large fraction of the entire state-space and can easily be found in a simulation. Other errors may depend on a specific order of events and manifest themselves only in small corners of the large state space. For certain errors, this may lead to poor coverage. Moreover, for some kinds of errors the chances of finding them by simulation are extremely small. It is furthermore hard to assess the coverage that has been achieved by the simulation.

• *Easy to use.* Since the size of the state space is not that important, systems are often modelled more straightforward than models intended for verification.

• *No storage of states.* Since visited states are not stored, it is possible to use larger or more detailed models. This makes that more adequate models can be used, better approximating the real-world behaviour of the system.

• *Explores only a single trace.* A single system simulation generates just one of the possibly infinite number of executions of the system. However, within one such infinite trace the same piece of behaviour often occurs more than once. This way in effect multiple execution paths are verified, instead of just one.

Simulations are helpful during the entire design phase. In the early phases, when there are still a lot of errors in the design, these errors are found with less effort than using exhaustive verification methods. Later on, it is helpful to study detailed system models by simulation and gain insights in the system to be designed. Exhaustive verification methods can then be used to tackle the hard problems, building dedicated abstract models focussing on these problems.

V. Tableau Construction

In this section we will describe the basic ideas behind the construction of the ω-automaton from an LTL formula φ [3], [4]. The formula specifies a requirement on the entire (infinite) trace of the system. After inspection of some finite prefix of this ω-trace, the state of the automaton represents the requirements that the remaining part of the trace must satisfy. The automaton will do this by splitting the formula in a requirement on the current system state and a requirement on the system's trace from the next state onwards. For example if a formula states that the system must never reach some control location l, this requirement can be split into the requirements that the system must not be in location l at the current state and it must never reach l in the future.

A. ω-automata

ω-automata are finite state automata, with the exception that they do not accept finite words, but infinite words ([**?**]). In order to achieve this, the acceptance conditions need to be defined differently. Traditional finite state automata have final states and accept a finite word, if the automaton resides in such a final state after consuming the input word. An ω-automaton will never finish consuming an infinite word. Therefore the acceptance condition requires that the automaton moves through some accepting state infinitely often.

We will not consider acceptance conditions in this paper. As we will see later, they will play no role when applied to simulations. ω-automata without acceptance conditions are sometimes called 'safety automata'. A safety automaton accepts any infinite word, for which it can always consume the next input symbol.

A (non-deterministic) ω-automaton A is a tuple (Q, Q_0, Σ, Δ). It consists of a finite set of states Q, a set of initial states $Q_0 \subseteq Q$, an alphabet Σ and a labelled transition relation Δ. The automaton accepts a word $w = \sigma_0\sigma_1\sigma_2\ldots$ ($\sigma_i \in \Sigma$) if there is a path $q_0q_1q_2\ldots$ through the automaton ($q_i \in Q$), starting from an initial state ($q_0 \in Q_0$) and such that there is always an edge from a state to the next $(q_i, \sigma_i, q_{i+1}) \in \Delta$.

Figure 3 shows an example of an ω-automaton, accepting all infinite words consisting of the symbols a and b, such that there are never two b's next to each other. In the figure, circles represent the states of Q. The initial states (in this case only one) have a small arrowhead. The transition relation is represented by arrows from one state to the next, labelled with a symbol from the alphabet.

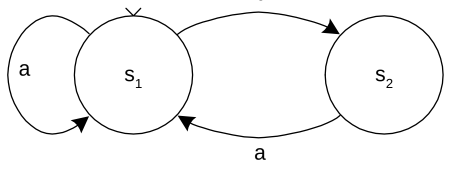


Fig. 3
An ω-automaton

B. Construction of A_φ

We want to construct an ω-automaton that accepts precisely all traces that satisfy a given formula φ. The alphabet of this automaton will consist of propositional formulas, expressing constraints upon the observable features of the system's current state.

The basic concept behind this construction is that the LTL formula can be rewritten in a normal form, which separates requirements on the current states and requirements on the remainder of the trace. Every formula φ can be written in the following form:

$$\varphi = p_{1,1} \wedge \ldots \wedge p_{1,n_1} \wedge \bigcirc(\varphi_{1,1} \wedge \ldots \wedge \varphi_{1,m_1}) \vee$$
$$p_{2,1} \wedge \ldots \wedge p_{2,n_2} \wedge \bigcirc(\varphi_{2,1} \wedge \ldots \wedge \varphi_{2,m_2}) \vee \ldots \vee$$
$$p_{k,1} \wedge \ldots \wedge p_{k,n_k} \wedge \bigcirc(\varphi_{k,1} \wedge \ldots \wedge \varphi_{k,m_k})$$

where all $p_{i,j}$ are atomic propositions and $\varphi_{i,j}$ are subexpressions of the formula φ.

This means that a trace can satisfy property φ in one of k different ways, namely by satisfying $p_{i,1} \wedge \ldots \wedge p_{i,n_i} \wedge \bigcirc(\varphi_{i,1} \wedge \ldots \wedge \varphi_{i,m_i})$ for some $1 \leq i \leq k$. This is possible if the current state of the system satisfies the constraint $p_{i,1} \wedge \ldots \wedge p_{i,n_i}$ and the remainder of the execution will satisfy $\varphi_{i,1} \wedge \ldots \wedge \varphi_{i,m_i}$. There will be edges in the automaton from the state representing the executions satisfying φ, to the state representing the executions that satisfy $\varphi_{i,1} \wedge \ldots \wedge \varphi_{i,m_i}$. There will be such an edge for every observation that is consistent with the constraint on the current state $p_{i,1} \wedge \ldots \wedge p_{i,n_i}$.

A formula can be converted to normal form, by using the following rewriting rules:

$$\begin{array}{rcl}
\neg(\varphi_1 \vee \varphi_2) & \rightarrow & \neg\varphi_1 \wedge \neg\varphi_2 \\
\neg(\varphi_1 \wedge \varphi_2) & \rightarrow & \neg\varphi_1 \vee \neg\varphi_2 \\
\neg\bigcirc\varphi & \rightarrow & \bigcirc\neg\varphi \\
\varphi_1 \wedge (\varphi_2 \vee \varphi_3) & \rightarrow & (\varphi_1 \wedge \varphi_2) \vee (\varphi_1 \wedge \varphi_3) \\
(\varphi_1 \vee \varphi_2) \wedge \varphi_3 & \rightarrow & (\varphi_1 \wedge \varphi_3) \vee (\varphi_2 \wedge \varphi_3) \\
\bigcirc(\varphi_1 \vee \varphi_2) & \rightarrow & \bigcirc\varphi_1 \vee \bigcirc\varphi_2 \\
\bigcirc\varphi_1 \wedge \bigcirc\varphi_2 & \rightarrow & \bigcirc(\varphi_1 \wedge \varphi_2) \\
\varphi_1 \mathbf{U} \varphi_2 & \rightarrow & \varphi_2 \vee (\varphi_1 \wedge \bigcirc(\varphi_1 \mathbf{U} \varphi_2)) \\
\neg(\varphi_1 \mathbf{U} \varphi_2) & \rightarrow & \neg\varphi_2 \wedge (\neg\varphi_1 \vee \bigcirc\neg(\varphi_1 \mathbf{U} \varphi_2))
\end{array}$$

For all states that are thus reachable from φ, the same procedure can be followed. The constraint they represent can be written in normal form, leading to new edges to new or existing states. One can show that the number of states that will be created this way is limited, and consequently that this construction will terminate.

One can show that the automaton obtained by this procedure accepts a trace precisely if it satisfies all safety aspects of the formula it was constructed for. This means that a trace s is accepted by A_φ iff every prefix of s can be extended to a full ω-trace that satisfies φ.

C. Example

As an example we will now show the construction of the tableau automaton for the following formula:

$$\Box(p \Rightarrow (p\mathbf{U}q))$$

it expresses that "as soon as some property p is true, it must remain true until q becomes true".

Using the abbreviations $\varphi = \Box(p \Rightarrow (p\mathbf{U}q))$ and $\psi = p\mathbf{U}q$, this formula can be written in normal form:

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

This leads to two edges from the initial state representing the requirement φ to itself and one edge leading to the requirement $\varphi \wedge \psi$. To complete the automaton, $\varphi \wedge \psi$ is again written in normal form

$$\varphi \wedge \psi \equiv q \wedge \bigcirc\varphi \vee p \wedge \bigcirc(\varphi \wedge \psi).$$

Figure 4 shows the corresponding automaton. Note that the property is not entirely a safety property, it also states that after p has become true, q should also eventually become true. This aspect of the formula is not covered by the safety automaton, since the automaton can remain in the state $\varphi \wedge \psi$ forever.

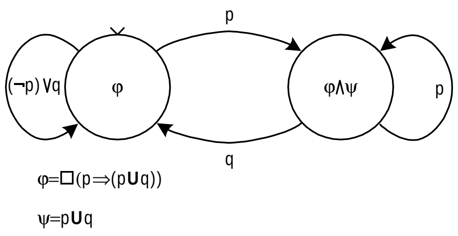


Fig. 4
A TABLEAU AUTOMATON OF $\Box(p \Rightarrow (p\mathbf{U}q))$

VI. Using the Tableau in Simulations

One will never finish the inspection of an infinite trace, since the system is potentially infinite-state and visited states are not stored. At any given time during a simulation, one has witnessed a *finite* prefix of an *infinite* execution of the system. This has some consequences for the types of properties that one can check. Violation of liveness properties can never be detected. A liveness property is a property that asserts that "something good will eventually happen." It cannot be detected because at any moment, the "good thing" may still happen in the future. Safety properties can never be established with certainty, but it can be detected when they are violated. A safety property is a property which states that "something bad will never happen." Since the verification is usually aimed at finding errors ([8]), the method is most useful for checking safety properties. Properties can have both safety and liveness aspects at the same time. $p\mathbf{U}q$ for example has the safety aspect, that it will never be the case that p no longer holds before q holds. On the other hand it expresses the liveness aspect, that q will eventually hold.

Summarising, by inspection of a finite prefix of a trace one can never detect that a liveness aspect is false and one can never detect that a safety aspect is true. One can however sometimes detect that a liveness aspect is true (for that particular trace only). And most importantly, one can detect that a safety property is violated (for that particulat trace, and thus the property does not hold for the system itself.) So simulation is mostly used to find unexpected unwanted behaviours, even though it will never prove that such behaviour cannot occur.

Using satisfiability analysis on the automaton before using it during simulation, states having a unsatisfiable constraint can be removed. This way, one can guarantee that as soon as the simulation has witnessed a prefix of the trace that can no longer lead to satisfaction of the constraint, this can be detected. It is done by maintaining during simulation, a set of states, that the tableau automaton can be in after acceptance of the prefix. This set will be empty as soon as the prefix can no longer satisfy the requirement.

VII. Related Work

The first construction of an ω-automaton from an LTL formula was done by Wolper, Vardi and Sistla [3]. It was done by constructing two automata, one to check state to state consistency (safety) and one to check that all eventualities (liveness requirements) are satisfied. The tableau automaton was then created by taking the product of these two automata. A more efficient on-the-fly construction is presented in [4].

The verification of temporal logic requirements in simulations is also addressed in [9]. In this paper a three valued logic is presented interpreted over finite traces, if the evaluation of the formula is not determined by the prefix, the outcome is defined as 'unknown.' Although the same separation of requirements on the current system state and the future execution is used, formulas are manipulated directly during simulation, rather than using tableau-automata.

There are also some connections to checking temporal preconditions in object-oriented databases, as in [10]. Here, a precondition for a database transaction can be expressed as a past-time temporal logic expression. It is similar to simulation because it is evaluated in a single forward run of the system. The difference is, that it only deals with finite traces, since the preconditions are about the past and the database was started at some point in history.

VIII. Future Work

Future work on his topic will include a definition of a full language, adapted to the object-oriented paradigm of POOSL and including a way to specify the atomic propositions. The method must be implemented in the existing simulation tools for the language POOSL. We will furthermore investigate the possibilities to verify quantitative timing requirements, using a similar construction in which some real-time logic based on LTL is translated to a timed ω-automaton.

IX. Conclusions

Model-checking techniques can be used to evaluate certain requirements expressed in LTL, during a system simulation. It is a kind of non-exhaustive verification, exploring a single execution path instead of all possible execution paths. LTL requirements are translated to ω-automata. These automata run together ('in lock-step') with the simulation and can detect when the execution can no longer satisfy the requirement that it represents.

Because the simulation does not search through the entire state space and does not have to store any states, it does not directly suffer from the state space explosion. Drawbacks are that it will not find all errors, it gives little information about coverage and the actual coverage figures are influenced by state space explosion. The method is easy to use and applicable to relatively large models. It can be used in all stages of the design and it can provide a path to exhaustive verification of the really hard problems.

REFERENCES

[1] G. Holzmann, *Design and Validation of Computer Protocols*, Prentice-Hall, Englewood Cliffs, New Jersey, 1991.

[2] P.H.A. V.D. Putten and J.P.M. Voeten, *Specification of Reactive Hardware / Software Systems*, Ph.D. thesis, Eindhoven University of Technology, Department of Electrical Engineering, 1997.

[3] P. Wolper, M.Y. Vardi, and A.P. Sistla, "Reasoning about infinite computation paths," in *Proceedings of 24th IEEE Symposium on Foundation of Computer Science, Tuscan*, 1983, pp. 185–194.

[4] R. Gerth, D. Peled, M.Y. Vardi, and P. Wolper, "Simple on-the-Fly automatic verification of linear temporal logic," in *Proc. IFIP/WG6.1 Symp. Protocol Specification Testing and Verification (PSTV95), Warsaw Poland*. June 1995, pp. 3–18, Chapman & Hall.

[5] D. R. Dams, *Abstract Interpretation and Partition Refinement for Model Checking*, Ph.D. thesis, Eindhoven University of Technology, P.O. Box 513, 5600MB Eindhoven, The Netherlands, july 1996.

[6] K.L. McMillan, *Symbolic Model Checking*, Kluwer Academic Publishers, Norwell, 1993.

[7] Colin H. West, "Protocol validation in complex systems," *Computer Communication Review*, vol. 19, no. 4, pp. 303–312, 1989.

[8] Thomas A. Henzinger, "Some myths about formal verification," *ACM Computing Surveys*, vol. 28, no. 4, pp. 119, December 1996.

[9] W. Canfield, E.A. Emerson, and A. Saha, "Checking formal specifications under simulation," in *Proceedings International Conference on Computer Design. VLSI in Computers and Processors*, Los Alamitos, CA, USA, 1997, pp. 455–460, IEEE Computer Society Press.

[10] S. Schwiderski, T. Hartmann, and G. Saake, "Monitoring temporal preconditions in a behaviour oriented object model," Tech. Rep. Informatik Berichte 93-07, Technische Universitaet Braunschweig, November 1993.