Deterministic Dynamic Monitors for Linear-Time Assertions

Roy Armoni, Dmitry Korchemny[1], Andreas Tiemeyer[1], Moshe Y. Vardi[⋆2], and Yael Zbar[1]

[1] Intel
[2] Rice University and Microsoft Research

Abstract. We describe a framework for dynamic verification of temporal assertions based on assertion compilation into deterministic automata. The novelty of our approach is that it allows efficient dynamic verification of general linear temporal formulas written in formal property specification languages such as LTL, ForSpec, PSL, and SVA, while the existing approaches are applicable to limited subsets only. We also show an advantage of the described framework over industrial simulators, which typically use transaction-based verification. Another advantage of our approach is its ability to use deterministic checkers directly for hardware emulation. Finally, we compare the deterministic compilation with the OBDD-based on-the-fly simulation of deterministic automata. We show that although the OBDD-based simulation method is much slower, the two methods may be efficiently combined for hybrid simulation, when the RTL signals in assertions are mixed with symbolic variables.

1 Introduction

One of the most significant results in the area of formal methods over the last two decades has been the development of algorithmic methods for verifying temporal specifications of *finite-state* programs, cf. [17]. By now, these methods have had important industrial impact, cf. [11, 19]. Nevertheless, even in design projects with significant application of formal methods, such methods rarely account for more than 10% of the verification effort [9]. It seems that, in the near future at least, traditional, simulation-based, dynamic verification techniques will continue to constitute the basic approach to design verification. This calls for exploration of techniques that combine the dynamic and formal approaches to verification, cf. [30].

An important aspect of verification consists of expressing assertions that capture the intended behavior of the design. Using the same formal verification languages based on linear temporal logic (LTL)[36] such as ForSpec [5], Sugar [7], PSL [18] or SVA[3]

⋆ Supported in part by NSF grants CCR-9988322, CCR-0124077, CCR-0311326, and ANI-0216467, by BSF grant 9800096, and by Texas ATP grant 003604-0058-2003. Part of this work was done while the author was visiting the Isaac Newton Institute for Mathematical Sciences, as part of a Special Programme on Logic and Algorithms.

[3] SVA is not, strictly speaking, a language based on LTL; nevertheless, the ideas described in this work apply to SVA as well.

[41] for both formal and dynamic verification leads to increased productivity in the verification effort.

Even if the same assertions are used in formal and dynamic verification, their treatment in the two contexts is different. In the formal context, temporal assertions are usually compiled into nondeterministic automata [39] (cf. [4], however), while in the dynamic context the compilation is into deterministic automata, also referred to as *monitors* or *checkers* [1, 12, 28]. This is because in formal verification we search through the transition graph of the product of the design and the automaton, while in dynamic verification we simulate the monitor along executions of the design. Indeed, the work in [1] reports on a compilation of a *fragment* of Sugar into deterministic monitors. In the same spirit, the PSL Reference Manual [2] defines a fragment called *Simple PSL*, which is suitable for compilation into deterministic monitors (cf. also [26]). Such an approach, however, poses two difficulties. First, it is possible that an assertion written for formal verification is not compilable for dynamic verification, making it difficult to use the same specification base for both formal and dynamic verification. Second, the existence of two compilers, one for formal verification and one for dynamic verification, raises the risk that the two compilers may actually differ in their semantics, as they might compile the same assertion into inequivalent automata. For example, in [29, 30], LTL semantics is defined over infinite traces for formal verification, but over finite traces for dynamic verification, creating a potential mismatch. The focus on assertion compilation for both formal and dynamic verification distinguishes our approach to dynamic temporal monitors from that of many published works in this area, cf. [6, 20, 35].

We describe in this paper a framework for dynamic verification, which uses assertions expressed in ForSpec. (We express the ideas in terms of LTL, but they are applicable also to other linear-temporal languages extended with regular events, such as PSL, without the Optional Branching Extension, and SVA). The key goals underlying the framework are: (1) the semantics of assertions in dynamic verification should be consistent with the semantics of assertions in formal verification; (2) all assertions in the underlying assertion language should be accepted, rather than just a fragment of the language; and (3) the compilation into deterministic monitors should be as close as possible to the compilation into nondeterministic automata.

To accomplish semantic consistency, we adopt a three-valued semantics for finite traces. When a formula is reported to hold in a finite trace, the formula holds in all extensions of the trace; this is a *pass*. When a formula is reported to fail in a finite traces, the formula fails in all extensions of the trace; this is a *fail*. When a formula is reported to be *ongoing*, it means that the finite trace does not provide enough information to decide the truth value of the formula. Our notion of success and failure over finite traces is that defined for ForSpec [3, 5], and elucidated further in [22]. We define it here in automata-theoretic terms (see below). This approach allows us to accomplish our second goal. *Every* assertion in the language can be compiled for dynamic verification; there is no need to define a “simple” fragment. (Of course, for some assertions this approach is not useful. For example, it is not useful to test the assertion `always eventually p` over finite traces. We generate a trivial monitor for this assertion.)

We describe two approaches to accomplish the third goal, which is to keep formal compilation and dynamic compilation as close as possible. The first approach attempts to eliminate the deterministic compilation completely. The ForSpec compiler compiles temporal assertions into alternating Büchi automata on infinite traces [38], where every state is responsible for checking one subformula. The fairness condition guarantees that all eventualities are satisfied rather than postponed forever. As suggested in [32], by considering this automaton as an automaton on finite words, we get an automaton that accepts finite traces on which the formula is guaranteed to succeed. The ForSpec compiler translates the alternating automaton to an implicit representation of a nondeterministic automaton. Instead of compiling this nondeterministic automaton into a deterministic monitor using the subset construction, we generate the states of the monitor on the fly during dynamic verification. Since the nondeterministic automaton is represented implicitly, states of the monitor are sets of truth assignments, represented using Ordered Binary Decision Diagrams (OBDDs) [13] over the variables of the implicit representation. While this approach offers the smallest "distance" between the compilations for formal and dynamic verification, it proves empirically to be, in general, too slow in comparison to our second approach.

The second approach addresses the performance issue by direct conversion of the alternating automaton into an explicitly represented nondeterministic automaton. This step involves an exponential blowup in the worst case, but our experience with LTL compilation has shown that this translation is quite amenable to optimization, cf. [21, 24, 37]. Once the nondeterministic automaton is represented explicitly, we do not explicitly generate a deterministic automaton, as, for example, in [29]. Rather, we synthesize an implicitly represented, equivalent deterministic monitor represented in a register transfer level (RTL). Every state of the nondeterministic automaton gives rise to a sequential element, and the combinational logic describes the subset construction. Thus, the size of the RTL description is *linear* in the size of the explicitly represented nondeterministic automaton, while an explicit determinization may incur an exponential blowup. The advantage of using RTL representation is that it matches the representation used in dynamic verification, as both design and assertion are now represented in RTL. We show that this approach yields dramatic improvement in performance. (Another advantage of this approach is that the monitors can also be used in hardware emulation, and not only in software simulation, of the design under verification.)

The two compilation methods may be combined to implement an efficient hybrid simulation engine: the deterministic part of the assertion can be compiled into a deterministic monitor, while the nondeterministic part can be simulate by means of OBDDs.

Example 1. Nondeterministic model[4]

```
rigid data[64]; // unknown constant
assert always /start & data = /datain ->
    eventually [2,5] /end & data = /dataout &
        /z[64] = /x[64] * /y[64];
```

[4] `rigid` declares property variables that are assigned a nondeterministic value at time 0, which then does not change. Names that are prefixed with "/" refer to design variables.

The above assertion is nondeterministic, since the variable `data` is symbolic and its value is unknown. On the other hand the product is deterministic and can be very efficiently computed by an RTL simulator. The assumption and the rest of the assertion are simulated as OBDD. Attempting to simulate everything as OBDD would result in a blowup, since the OBDD for the product is known to be exponential in the length of the arguments.

A key feature of our deterministic approach is its scalability with respect to simulation length. Since we generate a deterministic finite-state monitor, monitoring overhead is independent of simulation run length. A common approach in the semiconductor industry (e.g., Synopsys' VCS®) is transaction-based monitoring, which in effect constructs the monitor dynamically [16]. For example, in monitoring the property `always (p → eventually q)`, the simulator spawns a new thread waiting for q each time it observes $p \wedge \neg q$. Such a thread is called a *transaction*. With this approach the number of active transactions is potentially unbounded, resulting in degraded performance for long simulation runs. (The conventional methodology [10] advises users to prevent this problem by bounding the number of active transactions.)

The outline of the paper is as follows. In Section 2 we describe the theory underlying our approach. Section 3 describes experimental results for both OBDD-based and RTL-based monitors. Section 4 describes the framework. We conclude with a discussion in Section 5.

2 Underlying Theory

LTL formulas are constructed from a set AP of atomic proposition using the usual Boolean operators and the temporal operators X ("next time"), U ("until"), and V ("release"). Formally, given a set AP, an LTL formula in *positive normal form* (PNF) is defined as follows: (1) **true**, **false**, p, or $\neg p$, for $p \in AP$, (2) $\psi_1 \vee \psi_2$, $\psi_1 \wedge \psi_2$, $X\psi_1$, $\psi_1 U \psi_2$, or $\psi_1 V \psi_2$, where ψ_1 and ψ_2 are LTL formulas. The semantics of LTL is defined over infinite traces, which are elements of $(2^{AP})^\omega$; see details in [23]. For an LTL formula ψ over a set AP of atomic propositions, let $\|\psi\|$ denote the set of traces in that satisfy ψ.

For a given set X, let $\mathcal{B}^+(X)$ be the set of positive Boolean formulas over X, where we also allow the formulas **true** and **false**. For $Y \subseteq X$, we say that Y *satisfies* a formula $\theta \in \mathcal{B}^+(X)$ iff the truth assignment that assigns *true* to the members of Y and assigns *false* to the members of $X \setminus Y$ satisfies θ. The transition function $\rho : Q \times \Sigma \rightarrow 2^Q$ of a nondeterministic automaton with state space Q and alphabet Σ can be represented using $\mathcal{B}^+(Q)$. For example, a transition $\rho(q, \sigma) = \{q_1, q_2, q_3\}$ can be written as $\rho(q, \sigma) = q_1 \vee q_2 \vee q_3$. While transitions of nondeterministic automata correspond to disjunctions, transitions of *alternating automata* can be arbitrary formulas in $\mathcal{B}^+(Q)$. We can have, for instance, a transition $\delta(q, \sigma) = (q_1 \wedge q_2) \vee (q_3 \wedge q_4)$, meaning that the automaton accepts from state q a suffix w^l, starting by σ, of w, if it accepts w^{l+1} from both q_1 and q_2 or from both q_3 and q_4. For a formal definition of runs of alternating automata on finite or infinite words, see [38]. A word (either finite or infinite) is accepted by $\mathcal{A}$

iff there exists an accepting run on it. The language of $\mathcal{A}$, denoted $\mathcal{L}(\mathcal{A})$, is the set of words that $\mathcal{A}$ accepts.

Given an LTL formula ψ in PNF, one can build a nondeterministic Büchi automaton $\mathcal{A}_\psi$ such that $\mathcal{L}(\mathcal{A}_\psi) = \|\psi\|$ [40]. The size of $\mathcal{A}_\psi$ is exponential in $|\psi|$ in the worst case. It is shown in [38] that when alternating automata are used, the translation of ψ to $\mathcal{A}_\psi$ is inductive and it involves only a linear blow up in the number of states.

Theorem 1. [38] *Given an LTL formula ψ, we can construct an alternating Büchi automaton $\mathcal{A}_\psi = \langle 2^{AP}, cl(\psi), \delta, \{\psi\}, F\rangle$, such that $\mathcal{L}(\mathcal{A}_\psi) = \|\psi\|$.*

See [14] for an extension of this construction to LTL extended with a regular layer (which is essentially common to ForSpec, PSL, and SVA).

Using the (potentially exponential) translation described in [34] from alternating Büchi automata to nondeterministic Büchi automata, we get:

Corollary 1. [40] *Given an LTL formula ψ, we can construct a nondeterministic Büchi automaton $\mathcal{N}_\psi$ such that $\mathcal{L}(\mathcal{N}_\psi) = \|\psi\|$.*

Consider a language $L \subseteq \Sigma^\omega$ of infinite words over the alphabet Σ. A finite word $x \in \Sigma^*$ is a *bad prefix* for L iff for all $y \in \Sigma^\omega$, we have $x \cdot y \notin L$. Thus, a bad prefix is a finite word that cannot be extended to an infinite word in L. It is shown in [32] that a nondeterministic automaton that accepts all bad prefixes of ψ is, in general, of size that is doubly exponential in the length of ψ. Thus, instead of trying to detect *all* bad prefixes, we focus on detecting *informative* prefixes.

For an LTL formula ψ in PNF and a finite trace $\pi = \sigma_1 \cdot \sigma_2 \cdots \sigma_n$, with $\sigma_i \in 2^{AP}$, we say that π is an *informative prefix for* ψ iff there exists a mapping $L : \{1, \ldots, n+1\} \to 2^{cl(\neg\psi)}$ such that the following holds:

(1) $\neg\psi \in L(1)$.
(2) $L(n+1)$ is empty.
(3) For all $1 \leq i \leq n$ and $\varphi \in L(i)$, the following hold.
- If φ is a propositional assertion, it is satisfied by σ_i.
- If $\varphi = \varphi_1 \vee \varphi_2$ then $\varphi_1 \in L(i)$ or $\varphi_2 \in L(i)$.
- If $\varphi = \varphi_1 \wedge \varphi_2$ then $\varphi_1 \in L(i)$ and $\varphi_2 \in L(i)$.
- If $\varphi = X\varphi_1$, then $\varphi_1 \in L(i+1)$.
- If $\varphi = \varphi_1 U \varphi_2$, then $\varphi_2 \in L(i)$ or $[\varphi_1 \in L(i)$ and $\varphi_1 U \varphi_2 \in L(i+1)]$.
- If $\varphi = \varphi_1 V \varphi_2$, then $\varphi_2 \in L(i)$ and $[\varphi_1 \in L(i)$ or $\varphi_1 V \varphi_2 \in L(i+1)]$.

Note that the emptiness of $L(n+1)$ guarantees that all the requirements imposed by $\neg\psi$ are fulfilled along π. For example, while the finite computation $\{p\} \cdot \emptyset$ is informative for Gp (e.g., with a mapping L for which $L(1) = \{F\neg p\}$,$L(2) = \{F\neg p, \neg p\}$, and $L(3) = \emptyset$), it is not informative for $\psi = G(p \vee (Xq \wedge X\neg q))$. Indeed, as $\neg\psi = F(\neg p \wedge (X\neg q \vee Xq))$, an informative prefix for ψ must contain at least one state after the first state in which $\neg p$ holds.

An informative prefix for ψ is also a bad prefix for ψ. Thus, if π is an informative prefix for ψ, we say that ψ *fails* on π, and if π is an informative prefix for $\neg\psi$, we say that ψ *passes* on π. It is possible for π to be informative neither for ψ nor for $\neg\psi$. In this case ψ is *ongoing* on π. The vast majority of the assertions used in practice have `always` as the outmost temporal operator. Such assertions cannot pass, but only fail or remain ongoing. Dynamic verification engineers are typically interested in assertion failure, and do not distinguish between passed and ongoing assertions.

Given an LTL formula ψ and a finite trace π of length n, the problem of deciding whether π is an informative prefix for ψ can be solved in time $O(n \cdot |\psi|)$. [32]. However, this algorithm requires offline access to the finite trace π and, therefore is not appropriate for dynamic monitoring. For that we need a deterministic automaton that accepts the informative prefixes of ψ.

As described earlier, given a PNF formula ψ, one can build an alternating Büchi automaton $\mathcal{A}_\psi = \langle 2^{AP}, 2^{cl(\psi)}, \delta, \psi, F\rangle$ such that $\mathcal{L}(\mathcal{A}_\psi) = \|\psi\|$. Essentially, each state of $\mathcal{L}(\mathcal{A}_\psi)$ corresponds to a subformula of ψ, and its transitions follow the semantics of LTL. We define the alternating Büchi automaton $\mathcal{A}_\psi^{true} = \langle 2^{AP}, 2^{cl(\psi)}, \delta, \psi, \emptyset\rangle$ by redefining the set of accepting states to be the empty set. So, while in $\mathcal{A}_\psi$ a copy of the automaton may accept by either reaching a state from which it proceeds to **true** or visiting states of the form $\varphi_1 V \varphi_2$ infinitely often, in $\mathcal{A}_\psi^{true}$ all copies must reach a state from which they proceed to **true**. Accordingly, $\mathcal{A}_\psi^{true}$ accepts exactly these computations that have a finite prefix that is informative for ψ. To see this, note that such computations can be accepted by a run of $\mathcal{A}_\psi$ in which all the copies eventually reach a state that is associated with propositional assertions that are satisfied. Now, let *fin*($\mathcal{A}_\psi^{true}$) be $\mathcal{A}_\psi^{true}$ when regarded as an automaton on finite words; we claim that we have constructed an automaton for informative prefixes [32] (see also [25, 27, 33]).

Theorem 2. *For every formula ψ, the automaton fin($\mathcal{A}_{\neg\psi}^{true}$) accepts exactly the prefixes that are informative for ψ.*

We note that searching for informative prefixes rather than for bad prefixes is the basic idea underlying finite failure of temporal formulas in [3, 22]. Thus, the construction of *fin*($\mathcal{A}_{\neg\psi}^{true}$) essentially extracts from ψ its "informative safety" part (a safety property is a property whose failure is always witnessed by a bad prefix).

The automaton *fin*($\mathcal{A}_{\neg\psi}^{true}$) is an alternating automaton, and it is compiled into a nondeterministic automaton. All that remains now is to apply a determinization construction and we get a monitor automaton for informative prefixes. We have thus accomplished the three goals we set out: (1) Informative prefixes provide an approximation to bad prefixes, using the *standard* semantics of LTL; (2) The construction applies to the *full* assertion language, with no need to define special fragments; (3) the compilation into deterministic monitors is very close to the compilation into nondeterministic automata, as it requires just a final determinization construction. Of course, automata determinization is exponential in the worst case, so the final step is far from trivial and is discussed further below. (Note, however, that directly constructing of deterministic monitors from LTL formulas, by induction on the structure of the formula, as in [12, 28], can yield monitors of nonelementary size [5], unless care is taken to minimize the monitor after each step of the induction.)

[5] That is, the blow-up may not be bounded by any finite tower of exponentials.

A nondeterministic automaton over finite words is a tuple $\mathcal{A} = \langle \Sigma, Q, \rho, Q_0, F \rangle$, where $\rho : Q \times \Sigma \rightarrow 2^Q$ is the transition function. To determinize, one applies the subset construction and obtain $\mathcal{A}^d = \langle \Sigma, 2^Q, \{Q_0\}, \rho^d, F^d \rangle$, where $F^d = \{P : P \cap F \neq \emptyset\}$, and $\rho^d(P, a) = \cup_{s \in P} \rho(s, a)$. Note that in our case the nondeterministic automaton is obtained via a potentially exponential translation from an alternating automaton [15]. Thus, the translation from an alternating automaton to a deterministic automaton is potentially doubly exponential [15].

One way to avoid this blow-up is to avoid constructing $\mathcal{A}^d$. Instead, we can simulate $\mathcal{A}^d$ *on the fly*. Given a finite trace $a_0, \ldots, a_{n-1}$, we construct a run $P_0, \ldots, P_n$ of $\mathcal{A}^d$ as follows: $P_0 = \{Q_0\}$, and $P_{i+1} = \rho^d(P_i, a_i)$. Note that each state of $\mathcal{A}^d$ in this run is of size linear in the size of $\mathcal{A}$. Thus, we have avoided the exponential blowup of the determinization construction, with the price of having to compute transitions on the fly. To implement this the nondeterministic automaton is represented implicitly. Recall that informative prefixes are accepted by the alternating automaton $\mathit{fin}(\mathcal{A}_\psi^{true}) = \langle 2^{AP}, S, \delta, \{s_0\}, \emptyset \rangle$, where $S = 2^{cl(\psi)}$ and $s_0 = \psi$. We can represent a nondeterministic automaton that is equivalent to $\mathit{fin}(\mathcal{A}_\psi^{true})$ as a *nondeterministic sequential circuit*, where we consider AP as the set of input variables and S as the set of state variables. (That is, the state set of the nondeterministic automaton is $Q = 2^S$.) Finally, the transition relation of the circuit is represented as a Boolean function f with domain $2^{AP} \times 2^S \times 2^S$: $f(a, P, P') = 1$ precisely when P' satisfies $\delta(p, a)$ for all $p \in P$. We can describe this function as the conjunction of constraints of the form $p \wedge a \rightarrow \mathit{next}(\delta(p, a))$, where *next* replaces each variable s by its primed version s', referring to the value of s in the next cycle. The initial and final state of the nondeterministic automaton can be expressed as Boolean functions over 2^S; the representation has size that is linear in the size of the input formula. Note that since a state of the nondeterministic automaton is an element of 2^S, a state of $\mathcal{A}^d$ is a subset of 2^S. Thus, a state of $\mathcal{A}^d$ can be represented via OBDD over the variables in S, and the next state function of ρ^d can be computed using an OBDD-based image operation.

It may seem surprising to consider using OBDD operations that are known to be quite "heavy" in formal-verification applications. There is a fundamental difference, however, between the symbolic operations used here and the symbolic operations used in the model checking. In the model checking, we conduct a breadth-first search over the state space of the design. In essence, we simulate $\mathcal{A}^d$ across *all* possible traces. Here, we simulate $\mathcal{A}^d$ over a *single* trace, so one would expect the OBDDs to be significantly smaller. In practice, however, we discovered that OBDDs operations did slow the simulator significantly, and symbolic simulation did not prove to be a viable approach.

A more successful approach is to construct the nondeterministic automaton *explicitly*. This translation can be exponential in the worst case, but a lot of research has been reported on optimizing this translation (cf. [21, 24, 37]) and an unacceptable blow-up is rarely seen in practice. Once the nondeterministic automaton $\mathcal{A}$ is represented explicitly, the deterministic automaton $\mathcal{A}^d$ can be represented *implicitly*, as a *deterministic sequential circuit*. That is, every proposition in AP is viewed as an input signal and every element of Q is viewed as a sequential element. The initial state of A^d is the one in which precisely the elements in Q_0 are assigned 1 and the accepting states are the one in which some element in F is assigned 1. Finally, with each element $q \in Q$ we associate a Boolean transition function $f_q : 2^Q \times 2^{AP} \to \{0,1\}$ defined as follows: $f_q(P,a) = 1$ if $q \in \rho^d(P,a)$.

By viewing A^d as a *deterministic* sequential circuit, we can synthesize an RTL representation of A^d from A with *no* blowup. The advantage of using RTL representation is that it matches the representation used in dynamic verification, as both design and assertion are now represented in RTL. We show in Section 4 that this approach yields significant improvement in performance over the symbolic-simulation approach.

However, as was mentioned in the introduction, the symbolic and the deterministic approaches may be successfully combined when some of the variables used in the assertion are symbolic.

Example 2. Consider the following safety property: `next a wuntil next b`. The property is first negated and then the negation is propagated:
`next !b until ((next !a) & (next !b))`. The negated property is then translated into an alternating automaton on finite words; see Fig. 1.

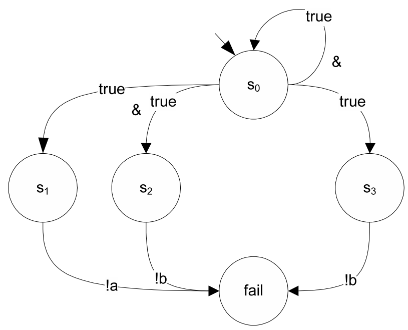

Fig. 1. Alternating automaton

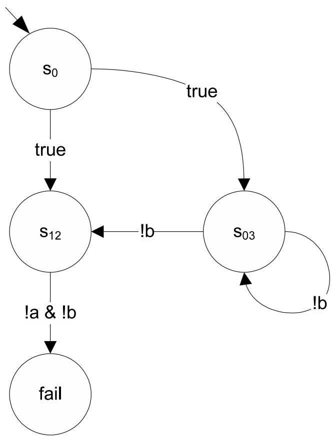

Fig. 2. Nondeterministic automaton

For the non-deterministic compilation the alternating automaton can be converted to an implicitly represented nondeterministic automaton:

$s_0 \to (s_1' \wedge s_2') \vee (s_0' \wedge s_3')$
$s_1 \wedge \neg a \to fail$
$s_2 \wedge \neg b \to fail$
$s_3 \wedge \neg b \to fail$

Alternatively, for the deterministic compilation we can convert the nondeterministic automaton to an explicit representation [4]; see Fig. 2. Finally, the explicitly represented nondeterministic automaton can be converted to an implicitly represented deterministic automaton, using an RTL; we use here Verilog. (In the following code `sysclk` is an auxiliary signal corresponding to the reference clock):

```
reg s0, s12, s03;
wire fail, sysclk;
assign fail = s12 && !a & !b;
initial begin
    s0 = 1'b1; s12 = 1'b0; s03 = 1'b0;
end
always @(posedge sysclk) begin
    s0   <= 1'b0;
    s12   <= s0 || !b && s03;
    s03 <= s0 || s03 && !b;
end
```

3 Dynamic-Verification Framework

ARCHITECTURE: This framework was developed for dynamic verification of ForSpec assertions. It can easily be adapted to other LTL-based property specification languages, such as PSL or SVA, by changing the compiler front-end.

The framework consists of two parts: a compiler and a run-time system. The compiler first checks whether the assertion is deterministic. If not, it generates an implicit representation of a nondeterministic automaton. For such assertions, the compiler produces a persistent OBDD representation of the nondeterministic automaton.

Otherwise, is generates an explicit representation of the nondeterministic automaton, cf. [4]. Although this stage theoretically involves an exponential blowup [38], it has always been feasible in practice. Then an RTL-represented deterministic monitor is generated from this automaton.

Explicitly compiled monitors are directly simulated by the run-time system. For implicitly compiled monitors the run-time system generates OBDD-represented states on the fly by means of OBDD operations. The run-time system runs in parallel to the DUT (Design Under Test) simulation environment and reads signal values from it. These signals are used to exercise the monitor. The run-time system queries the DUT model for checker signal changes and reports checker violation or success. The compilation flow overview is shown in Fig. 3; the simulation paradigm is shown in Fig. 4.

COMPILER OPTIMIZATIONS: The compiler applies several optimizations. Some optimizations result in a more efficient synthesis, while others are necessary to avoid the blowup, and to make the compilation feasible. In this section we sketch the following optimizations: reachability analysis, automata minimization, $next$ normalization, explicit conjunction, permanent state elimination, and state counting:

Reachability analysis for nondeterministic automata. To analyze a property failure, only states that are backward reachable from the accepting state need to be kept. Other states can be pruned away.

Automata minimization. Automata minimization may be used to synthesize a more efficient code and sometimes to avoid an exponential blowup. The minimization may be held at different stages - at the stage of an alternating, non-deterministic and

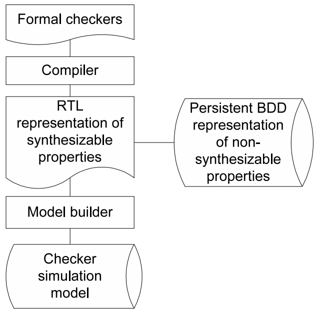

Fig. 3. Compilation flow

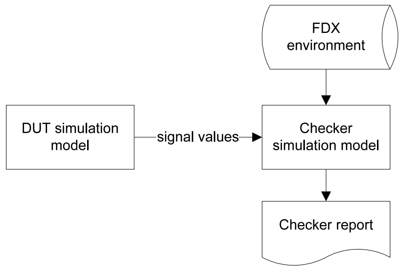

Fig. 4. Simulation flow

deterministic automaton. (It should be emphasized that the goal is not necessarily to minimize the *number* of the automaton states.)

$next$ *normalization.* In many cases it pays off to "pull out" $next$ operators. For example, consider the property `a until next[10] b`. It is preferable to rewrite it first as `next[10] past(a,10) until b`.

Explicit conjunction. If the complemented assertion in PNF has a form of a conjunction $\varphi_1 \wedge \varphi_2$, then it is preferable to construct separate monitors for φ_1 and $\varphi 2$ rather than a monitor for $\varphi_1 \wedge \varphi_2$. This is a rather common case, since when we start with an assumption φ and assertion ψ, the complemented assertion is $\phi \wedge \neg\psi$.

Permanent state elimination. The vast majority of the assertions in practice have the form `always` φ. The standard monitor construction generate an RTL variable that is always on. Such a variable can be eliminated.

State counting. There are cases when the assertion check is delayed by several cycles. i.e., the assertion has a form `next[`n`] A`, where n is a positive integer number, and `A` is an arbitrary assertion. Then instead of synthesizing a thread of n states for `next[`n`]`, an appropriate counter may be synthesized using $\log n$ variables.

4 Experimental Results

We ran two sets of experiments: the first one was to compare the OBDD-based and the RTL-based implementations of dynamic monitors for ForSpec assertions; the second one was to compare the RTL-based and the transaction-based monitoring for SVA assertions. (Our experiments were run on Intel®Xeon™CPU 3.06GHz with 4GB RAM, using the Linux operating system.)

RTL-BASED VERSUS BDD-BASED MONITORS: The first experiment was to compare the run time of the two implementations for simple assertions, where we scale up a time window. We ran the assertion `always a triggers always[0,n] b` for different values of n (we refer to the interval $[0, n]$ as the *time window*) for a simulation run of 25,000 clock cycles. The results are shown in Table 1. The time reported is the *additional* time required to complete the simulation, compared to an assertion-free simulation run (the latter time is less than one second). We used a large industrial design and the VCS simulation engine.

Window size	# BDD vars	#RTL variables	Time BDD (sec)	Time RTL (sec)
1	1	1	4.2	0.22
500	9	500	5.82	0.85
1,000	10	1,000	7.81	1.43
1,500	11	1,500	10.85	1.95
2,000	11	2,000	14.6	2.46

Table 1. Time Window Scaling

Note that while the number of OBDD variables grows logarithmically in n, the number of RTL variables grows linearly in n. The run time of both implementations of the monitors grows linearly with the size of the time, but the slope is quite steeper for the OBDD implementation.

The second experiment was to check the effect of the number of assertions on the run time. We execute n assertions of the form `always` a_i `triggers next` b_i, for different design signals a_i and b_i, for a simulation run of 25,000 clock cycles. The results are shown in Table 2. Again, we report on incremental running time.

n	# BDD vars	#RTL states	Time BDD (sec)	Time RTL (sec)
1	1	1	3.78	0.22
5	5	5	10.38	0.24
10	10	10	18.24	0.25
15	15	15	28.69	0.26
20	20	20	40.21	0.27
25	25	25	57.4	0.30

Table 2. Assertion Scaling

We can see that the number of assertions has a significant impact for the OBDD implementation, whereas the impact is marginal for the RTL implementation. Therefore the RTL implementation can handle large amount of assertions, while the OBDD implementation is capable of handling only a few tens of assertions. The RTL implementation proved to be efficient in handling thousands of real-life assertions; e.g., 10,000 assertions cause only a 10% overhead for the full-chip simulation of a CPU design.

The third experiment was to compare between the two implementations for real-life assertions. Only very few real-life monitors, covering a small number of assertions, can be run using pure OBDD implementation without a significant slow-down. The results are shown in Table 3.

Again, we can see that the RTL implementation runs in more than one order of magnitude faster than the OBDD implementation.

RTL-BASED VERSUS TRANSACTION-BASED MONITORING: As discussed in the introduction, many industrial assertion monitors use a transaction-based approach [16].

Test number	#cycles	#assertions	Time BDD (sec)	Time RTL (sec)
1	4630	8	3.97	0.49
2	4630	9	5.2	0.52
3	4630	27	48.69	0.91
4	4630	25	9.41	0.55
5	1724	4	0.78	0.26
6	1724	8	1.38	0.17
7	1724	23	4.79	0.16

Table 3. Real-life Monitors

This approach is problematic in two aspects: 1) It is suitable mostly to assertions of the form r_1 `triggers` r_2, where r_1 and r_2 are regular events; it is difficult to adapt to general temporal assertions. 2) The performance of transaction-based monitors is very sensitive to the number of outstanding transactions. Consequently, the standard methodology recommends to limit the number of the outstanding transactions using time windows [10], which may be inefficient for formal-verification tools. This results in different assertions written for formal and dynamic verification.

We compared the performance of VCS, which implements transaction-based monitoring, with automata-based monitors. We measured only the run time overhead introduced by assertion monitoring for a simulation run of 100,000 cycles.

The following simple Verilog model has been used:

```
module test;
logic clk, a, b, c;
initial begin
    a = 1'b1; b = 1'b0; c = 1'b1; clk = 1'b0;
    #1000000 b = 1'b1;
    #1000000 $finish;
end
always #10 clk = !clk;
// assertion goes here
endmodule
```

with the following SVA assertions:

1. `@(posedge clk) a ##[0:$] b |=> c`
2. `@(posedge clk) a[*0:1000] ##1 b |=> c`
3. `@(posedge clk) a |=> c[*1000]`

The results are shown in Table 4.

Our results clearly show the advantage of the RTL-based approach when the number of active transactions is not bounded by a small constant. The RTL-based approach is insensitive to the number of active transactions since the assertion has been compiled into a finite-state monitor.

	Assertion 1	Assertion 2	Assertion 3
RTL-based monitors	1.46	3.58	3.58
VCS	525.13	26.71	7.70

Table 4. FDX vs. VCS

5 Discussion

We described here a framework for dynamic verification of temporal assertions. We showed how to obtain deterministic compilation for dynamic verification that is as close as possible to the nondeterministic compilation of temporal assertions for formal verification. Further research is called for, in view of our results. A general issue is how to optimize the construction of monitors. For example, it is known that deterministic automata can be minimized canonically [31]. We do not know, however, whether it would beneficial to minimize monitors using automata-minimization techniques . On one hand, minimization would reduce the number of state variables in the RTL representation of the monitor; on the other hand, the logic of the transition function may get significantly more complex. More generally, the results here and in [4] show the advantage of using deterministic compilation in various settings of functional design verification. We believe that this issue ought to be investigated also in other settings.

References

1. Y. Abarbanel, I. Beer, L. Gluhovsky, S. Keidar, and Y. Wolfstal. FoCs - automatic generation of simulation checkers from formal specifications. In *Computer Aided Verification, Proc. 12th International Conference*, volume 1855 of *Lecture Notes in Computer Science*, pages 538–542. Springer-Verlag, 2000.
2. K. Albin et al. Property specification language reference manual. Technical Report Version 1.1, Accellera, 2004.
3. R. Armoni, D. Bustan, O. Kupferman, and M.Y. Vardi. Resets vs. aborts in linear temporal logic. In *Proc. 9th International Conference on Tools and algorithms for the construction and analysis of systems*, number 2619 in Lecture Notes in Computer Science, pages 65 – 80. Springer-Verlag, 2003.
4. R. Armoni, S. Egorov, R. Fraer, D. Korchemny, and M.Y. Vardi. Efficient LTL compilation for SAT-based model checking. In *Proc. Int'l Conf. on Computer-Aided Design*, pages 877–884, 2005.
5. R. Armoni, L. Fix, A. Flaisher, R. Gerth, B. Ginsburg, T. Kanza, A. Landver, S. Mador-Haim, E. Singerman, A. Tiemeyer, M.Y. Vardi, and Y. Zbar. The ForSpec temporal logic: A new temporal property-specification logic. In *Proc. 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 2280 of *Lecture Notes in Computer Science*, pages 296–211, Grenoble, France, April 2002. Springer-Verlag.
6. H. Barringer, A. Goldberg, K. Havelund, and K. Sen. Program monitoring with LTL in EAGLE. In *Proc. 18th Int'l Parallel and Distributed Processing Symp.* IEEE Computer Society, 2004.
7. I. Beer, S. Ben-David, C. Eisner, D. Fisman, A. Gringauze, and Y. Rodeh. The temporal logic Sugar. In *Proc. 13th International Conference on Computer Aided Verification*, volume 2102

of *Lecture Notes in Computer Science*, pages 363–367, Paris, France, July 2001. Springer-Verlag.
8. L. Bening and H. Foster. *Principles of verifiable RTL design – a functional coding style supporting verification processes*. Springer, 2001.
9. B. Bentley. High level validation of next-generation microprocessors. In *Proc. IEEE Int'l Workshop on High Level Design Validation and Test*, pages 31–35, 2002.
10. J. Bergeron, E. Cerny, A. Hunter, and A. Nightingale. *Verification Methodology Manual for SystemVerilog*. Springer, 2005.
11. P. Biesse, T. Leonard, and A. Mokkedem. Finding bugs in an alpha microprocessor using satisfiability solvers. In *Computer Aided Verification, Proc. 13th International Conference*, volume 2102 of *Lecture Notes in Computer Science*, pages 454–464. Springer-Verlag, 2001.
12. M. Boule and Z. Zilic. Incorporating efficient assertion checkers into hardware emulation. In *Proc. 23rd Int'l Conf. on Computer Design*, pages 221–228. IEEE Computer Society, 2005.
13. R.E. Bryant. Graph-based algorithms for boolean-function manipulation. *IEEE Trans. on Computers*, C-35(8), 1986.
14. D. Bustan, A. Flaisher, O. Grumberg, O. Kupferman, and M.Y. Vardi. Regular vacuity. In *Proc. 13th Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, volume 3725 of *Lecture Notes in Computer Science*, pages 191–206. Springer-Verlag, 2005.
15. A.K. Chandra, D.C. Kozen, and L.J. Stockmeyer. Alternation. *Journal of the Association for Computing Machinery*, 28(1):114–133, January 1981.
16. K.H. Chang, W.T. Tu, Y.J. Yeh, and Kuo S.Y. A simulation-based temporal assertion checker for PSL. In *Proc. 46th IEEE Int'l Midwest Stmp. on Circuits and Systems*, pages 1528–1531. IEEE Computer Society, 2003.
17. E.M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.
18. B. Cohen. *Using PSL/Sugar with Verilog and VHDL, Guide to Property Specification Language for ABV*. Addison-Wesley, 2003.
19. F. Copty, L. Fix, R. Fraer, E. Giunchiglia, G. Kamhi, A. Tacchella, and M.Y. Vardi. Benefits of bounded model checking at an industrial setting. In *Computer Aided Verification, Proc. 13th International Conference*, volume 2102 of *Lecture Notes in Computer Science*, pages 436–453. Springer-Verlag, 2001.
20. M. d'Amorim and G. Rosu. Efficient monitoring of omega-languages. In *Proc. 17th Int'l Conf. on Computer Aided Verification*, volume 3576 of *Lecture Notes in Computer Science*, pages 364–378. Springer, 2005.
21. N. Daniele, F. Guinchiglia, and M.Y. Vardi. Improved automata generation for linear temporal logic. In *Computer Aided Verification, Proc. 11th International Conference*, volume 1633 of *Lecture Notes in Computer Science*, pages 249–260. Springer-Verlag, 1999.
22. C. Eisner, D. Fisman, J. Havlicek, Y. Lustig, A. McIsaac, and D. Van Campenhout. Reasoning with temporal logic on truncated paths. In *Proc. 15th Int'l Conf. on Computer Aided Verification*, volume 2725 of *Lecture Notes in Computer Science*, pages 27–39. Springer, 2003.
23. E.A. Emerson. Temporal and modal logic. In J. Van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, chapter 16, pages 997–1072. Elsevier, MIT Press, 1990.
24. K. Etessami and G.J. Holzmann. Optimizing Büchi automata. In *Proc. 11th Int'l Conf. on Concurrency Theory*, Lecture Notes in Computer Science 1877, pages 153–167. Springer-Verlag, 2000.
25. B. Finkbeiner and H. Sipma. Checking finite traces using alternating automata. *Electr. Notes Theor. Comput. Sci.*, 55(2), 2001.

26. D. Fisman. The Subset of Linear Violation. *The Weizmann Institute of Science, IBM Haifa Research Lab*, 2005.
27. M. Geilen. On the construction of monitors for temporal logic properties. *Electr. Notes Theor. Comput. Sci.*, 55(2), 2001.
28. S.V. Gheorghita and R. Grigore. Constructing checkers from PSL properties. In *Proc. 15th Int'l Conf. on Control Systems and Computer Science , Vol. 2*, pages 757–762, 2005.
29. D. Giannakopoulou and K. Havelund. Automata-based verification of temporal properties on running programs. In *Proc. 16th International Conference on Automated Software Engineering*, pages 412–416. IEEE Computer Society, 2001.
30. D.W. Hoffmann, J. Ruf, T. Kropf, and W. Rosenstiel. Simulation meets verification: Checking temporal properties in SystemC. In *Proc. 26th EUROMICRO 2000 Conference*, page 1435. IEEE Computer Society, 2000.
31. J.E. Hopcroft and J.D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 1979.
32. O. Kupferman and M.Y. Vardi. Model checking of safety properties. *Formal methods in System Design*, 19(3):291–314, November 2001.
33. T. Latvala. Efficient model checking of safety properties. In *Proc. 10th SPIN Workshop on Model Checking of Software*, volume 2648 of *Lecture Notes in Computer Science*, pages 74–88, 2003.
34. S. Miyano and T. Hayashi. Alternating finite automata on ω-words. *Theoretical Computer Science*, 32:321–330, 1984.
35. D. Pidan, S. Keider-Barner, M. Moulin, and D. Fisman. Optimized algorithms for dynamic verification. Technical Report FP6-IST-507219, PROSYD, 2005.
36. A. Pnueli. The temporal logic of programs. In *Proc. 18th IEEE Symp. on Foundation of Computer Science*, pages 46–57, 1977.
37. F. Somenzi and R. Bloem. Efficient Büchi automata from LTL formulae. In *Computer Aided Verification, Proc. 12th International Conference*, volume 1855 of *Lecture Notes in Computer Science*, pages 248–263. Springer-Verlag, 2000.
38. M.Y. Vardi. An automata-theoretic approach to linear temporal logic. In F. Moller and G. Birtwistle, editors, *Logics for Concurrency: Structure versus Automata*, volume 1043 of *Lecture Notes in Computer Science*, pages 238–266. Springer-Verlag, Berlin, 1996.
39. M.Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Proc. 1st Symp. on Logic in Computer Science*, pages 332–344, Cambridge, June 1986.
40. M.Y. Vardi and P. Wolper. Reasoning about infinite computations. *Information and Computation*, 115(1):1–37, November 1994.
41. S. Vijayaraghavan and M. Ramanathan. *A Practical Guide for SystemVerilog Assertions*. Springer, 2005.