A Circuit Approach to LTL Model Checking

Koen Claessen
koen@chalmers.se

Department of CSE, Chalmers University of Technology
Gothenburg, Sweden.

Niklas Een, Baruch Sterin
{een,sterin}@eecs.berkeley.edu

Department of EECS, University of California, Berkeley, USA.

***Abstract*—This paper presents a method for translating formulas written in assertion languages such as LTL into a monitor circuit suitable for model checking. Unlike the conventional approach, no automata is generated for the property, but instead the monitor is built directly from the property formula through a recursive traversal. This method was first introduced by Pnueli et. al. under the name of *Temporal Testers*. In this paper, we show the practicality of temporal testers through experimental evaluation, as well as offer a self-contained exposition for how to construct them in manner that meets the requirements of industrial model checking tools. These tools tend to operate on logic circuits with sequential elements, rather than transition relations, which means we only need to consider so called *positive testers* with no *future references*. This restriction both simplifies the presentation and allows for more efficient monitors to be generated. In the final part of the paper, we suggest several possible optimizations that can improve the quality of the monitors, and conclude with experimental data.**

I. At a glance

Consider the LTL formula:

$$\textit{for-all-paths: } \mathbf{G}\neg a \ \vee \ \mathbf{XF}\neg b$$

A witness to its negation satisfies:

$$\textit{there-exists-a-path: } \mathbf{F}a \ \wedge \ \mathbf{XG}b$$

If no such witness exists, the original formula holds. Construct the following equisatisfiable formula by introducing a variable for each subformula, including the full formula:

$$\begin{array}{ll} & z_0 \\ \wedge & \mathbf{G}(z_0 \ \leftrightarrow \ z_1 \wedge z_2) \\ \wedge & \mathbf{G}(z_1 \ \leftrightarrow \ \mathbf{F}a) \\ \wedge & \mathbf{G}(z_2 \ \leftrightarrow \ \mathbf{X}z_3) \\ \wedge & \mathbf{G}(z_3 \ \leftrightarrow \ \mathbf{G}b) \end{array} \qquad (1)$$

Since the specification is in *negated normal form* and all the operators are monotonic, bi-implications can be replaced by simple implications:

$$\begin{array}{ll} & z_0 \\ \wedge & \mathbf{G}(z_0 \ \rightarrow \ z_1 \wedge z_2) \\ \wedge & \mathbf{G}(z_1 \ \rightarrow \ \mathbf{F}a) \\ \wedge & \mathbf{G}(z_2 \ \rightarrow \ \mathbf{X}z_3) \\ \wedge & \mathbf{G}(z_3 \ \rightarrow \ \mathbf{G}b) \end{array} \qquad (2)$$

Two types of properties commonly supported by modern model checking tools are:

- **Safety.** A counterexample is a *finite* path to a *bad* state.
- **Liveness.** A counterexample is an *infinite* path where a set of signals $f_1, f_2, \ldots, f_k$ are each true infinitely often.

In the following sections, it is shown how the conjuncts in (2) can each be translated into a small monitor circuit together with a liveness property, yielding a new model that can be verified by existing model checking tools. Furthermore, it is shown how the safety fragment of a temporal formula can be checked more efficiently by producing a safety property for that part.

Example. The conjunct $\mathbf{G}(z_1 \ \rightarrow \ \mathbf{F}a)$ of (2) is translated into a circuit that outputs TRUE as long as $z_1 = 0$, then when $z_1 = 1$, it starts waiting for $a = 1$, outputting FALSE in the meanwhile. When $a = 1$ arrives, the circuit goes back to waiting for $z_1 = 1$, while again outputting TRUE. This output signal needs to hold infinitely often for a witness to the formula, and is thus added as a liveness property.

II. Introduction

A. Automata-theoretic approach

Vardi and Wolper [18] introduced the automata-theoretic approach to verification. Given a formula ϕ and a machine M, finding whether $M \models \phi$ is done by creating an automaton $A_{\neg\phi}$ that accepts the traces that violate ϕ, and then checking whether $M \times A_{\neg\phi}$ is empty. In this paper we discuss generating circuits representing finite automata for detecting finite traces and simple Büchi automata for liveness properties.

B. Related work

Vardi and Wolper [18] showed that every LTL formula can be translated to a Büchi automaton that accepts the same language. There are now many approaches to perform that translation. In this section we review the most common ones.

The first set of approaches use direct construction of a Büchi automaton. These methods tend to be complicated, and may generate exponentially large automata.

The second set [17] translates the LTL formula into an alternating automaton, which is then translated into a Büchi automaton. The main advantage is the simplicity of the resulting alternating automaton, whose size is linear in the size of the formula. The resulting Büchi automaton has an exponential number of states in the size of the formula, but the size of the symbolic description is linear. This approach is also compositional; the alternating automaton for a formula is obtained from the alternating automata for its subformulas.

ISBN 978-0-9835678-3-7/13.

An often overlooked problem with this approach is that a good understanding of this flow, and especially of alternation and its removal [5], is a non-trivial intellectual undertaking. In an industrial environment, a simpler approach, especially if it has few disadvantages, is to be preferred.

Kesten et. al. in [10], and later Pnueli and Zaks in [13], [14] explored the use of temporal testers for verification of LTL and PSL. A *temporal tester* for a formula ϕ is a transition system that has a variable x_ϕ such that $\mathbf{G}(x_\phi \leftrightarrow \phi)$ holds; a *positive temporal tester* is similar, except that $\mathbf{G}(x_\phi \rightarrow \phi)$ holds instead. Temporal testers for simple properties can be combined recursively for more complicated properties.

The approach presented in this paper is based on temporal testers. Given a conjunct $\mathbf{G}(z_i \rightarrow \phi)$, ϕ has to be true whenever z_i equals 1. This makes z_i and the monitor state machine for ϕ a positive temporal tester for ϕ.

Similarly, as noted in [13], translation through alternating automata also results in positive temporal testers. The symbolic description of the resulting Büchi automata (depending on the translation method) has a variable for each subformula, with the property that whenever the variable is true, so is the subformula.

C. Finite traces

Some formulas, such as $\mathbf{G}p$ can be shown to hold only by infinite traces, other formulas, like $\mathbf{F}p$, by a finite trace, i.e. the formula will hold on any infinite extension of that finite trace. While, other formulas, such as $\mathbf{G}p \vee \mathbf{F}q$, can sometimes be shown to hold by a finite trace, and in other cases require an infinite trace.

As verification tools are usually much more efficient in detecting finite traces, it is preferable to detect finite traces whenever possible. In *subsection VI-A* it is shown how this can be achieved. The finite traces detected by our method are the same as the *informative prefixes* defined by Kupferman and Vardi in [11]. This is shown in section X.

D. ω-regular specifications

Although this paper shows how to build monitor circuits for LTL and PLTL formulas, Pnueli and Zaks [14] showed how to to extend this method by adding regular events to implement support for ω-regular languages such as PSL or SVA.

III. Notation

By *circuit*, we mean a directed acyclic graph with two edge types, complemented and non-complemented, and the following node/gate types:

AND – A binary AND-gate.
PI – Primary input.
PO – Primary output.
FF – Flip-flop (unit delay).
TRUE – The constant true.

For the main discussion, temporal formulas are expressed in *Linear Temporal Logic* extended with *past operators* (PLTL). The temporal operators of PLTL are reviewed in *Figure 1*. The extension to include past operators is trivial, but it allows us to use a richer set of benchmarks. Detailed formal semantics of PLTL can be found in [3]. A PLTL formula is any expression using logical operators $\wedge$, $\vee$, and $\neg$, and the temporal operators reviewed in *Figure 1*. In our terminology, a *signal* (or *atomic proposition*) is the output of a gate in the design (possibly complemented) referred to by the specification.

ADJACENT STATE

- $\mathbf{X}\,a$ – "next": a holds in the next cycle
- $\mathbf{Y}\,a$ – "yesterday": a held in the previous cycle; FALSE in the first cycle
- $\mathbf{Z}\,a$ – "variant yesterday": same as $\mathbf{Y}$ but TRUE in the first cycle

SIMPLE OPERATORS

- $\mathbf{G}\,a$ – "globally": a holds forever
- $\mathbf{F}\,a$ – "future": a holds at least once in a future (or the current) cycle
- $\mathbf{H}\,a$ – "historically": a a held up to (and including) the current cycle (past dual of $\mathbf{G}$)
- $\mathbf{P}\,a$ – "past": a held at least once in a past (or the current) cycle (past dual of $\mathbf{F}$)

UNTIL OPERATORS

- $[a\ \mathbf{W}\ b]$ – "weak-until": a holds up to the cycle before b holds, or a holds forever
- $[a\ \mathbf{M}\ b]$ – "weak-since": a held since the cycle after b last held, or a held since the first cycle (past dual of $\mathbf{W}$)

$[a\ \mathbf{U}\ b] = [a\ \mathbf{W}\ b] \wedge \mathbf{F}b$ "until"
$[a\ \mathbf{R}\ b] = \neg[\neg a\ \mathbf{U}\ \neg b]$ "release"

$[a\ \mathbf{S}\ b] = [a\ \mathbf{M}\ b] \wedge \mathbf{P}b$ "since" (past dual of $\mathbf{U}$)
$[a\ \mathbf{T}\ b] = \neg[\neg a\ \mathbf{S}\ \neg b]$ "trigger" (past dual of $\mathbf{R}$)

Fig. 1. *Informal overview of the semantics of PLTL operators.*

A PLTL formula is in *negated normal form* (NNF) if negations are present only on the atomic propositions. A formula can be brought into NNF by using the identities $(\neg\mathbf{X}a = \mathbf{X}\neg a)$, $(\neg\mathbf{G}a = \mathbf{F}\neg a)$, $(\neg\mathbf{F}a = \mathbf{G}\neg a)$, $(\neg[a\ \mathbf{U}\ b] = [\neg a\ \mathbf{R}\ \neg b])$, and their past-operator duals. **Example:** $\neg\mathbf{G}(a \vee \mathbf{P}b) = \mathbf{F}(\neg a \wedge \mathbf{H}\neg b)$

IV. On introducing auxiliary variables

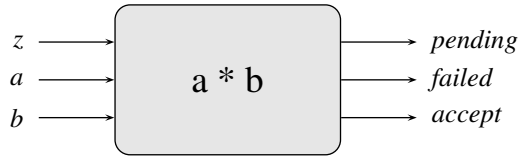
In *Section I*, it was shown how each subformula is given a *name* in the form of an auxiliary variable z_i. The construction is completely analogous to how the Tseitin transformation [15] is used in SAT to convert a propositional formula into an equisatisfiable CNF representation; but because we are dealing with a temporal formula, the $\mathbf{G}$ is needed to ensure that the auxiliary variable maintains its correspondence with the subformula it represents. Because there is an implicit existential quantifier around the LTL formula (*"there-exists-a-path"*), with some abuse of notation we can repeatedly use the identity $\phi(\psi) = \exists x.\mathbf{G}(x \leftrightarrow \psi) \wedge \phi(x)$, but leaving x to be implicitly quantified.

Why is it sound to turn bi-implications into simple implications, as was done from equation (1) to (2)? The operators we

allow in NNF, both logical and temporal, are all monotonically increasing in their inputs, meaning that if $op(x, y)$ is TRUE in a cycle, then so are $op(1, y)$ and $op(x, 1)$. Hence, any trace satisfying equation (2) can be "fixed" by identifying the $z_i \rightarrow$ *RHS* where z_i is 0 and *RHS* is 1 and simply flip the value of z_i. The modified trace will satisfy (1).

V. MONITOR CIRCUITS

Assume the specification has been negated and expanded to an equisatisfiable formula as outlined in *Section I*. Each conjunct is either on the form "$\mathbf{G}\ (z \rightarrow *a)$" (for unary operators "*") or "$\mathbf{G}\ (z \rightarrow a * b)$" (for binary operators). For each operator, we describe a *monitor circuit*. In the next section, it is shown how the monitors are combined to formulate a model checking problem for the entire PLTL specification. Our monitors have the following set of inputs and outputs:



The meanings of these signals are as follows:

z: A fresh PI, also referred to as the *activator*, created to match the auxiliary variable of the expansion. When it non-deterministically goes high, the circuit starts monitoring inputs a and b to see if they adhere to the semantics of the operator.

a: Left input of the operator: either a signal from the design or the activator z_i of the i^{th} monitor, synthesized for the left subformula.

b: Right input of the operator.

pending: TRUE if the monitor has an outstanding requirement on one or both of its input signals to be TRUE either in this or in future cycles.

failed: TRUE if a violation has been detected, preventing any further extension of the current trace from being a valid witness.

accept: Must hold infinitely often for a trace to be a valid witness. Stated negatively: if this signal goes forever FALSE, then the trace is not valid.

The system of monitors can be thought of as follows: The top-monitor is activated by asserting $z_0 = 1$ in the first cycle. This monitor, in order to meet its *accept* condition and avoid its *failed* constraint, will force one or both of its subformulas to be activated, either now or later. The process propagates down through the formula tree. If we can find an infinite run with no monitor outputting *failed*, and with each monitor having an infinite number of *accepts*, then a witness to the temporal formula has been produced. Note that the non-deterministic activator variables are all existentially quantified, which means that we can defer to the underlying model checker to "guess" perfectly when they should be activated.

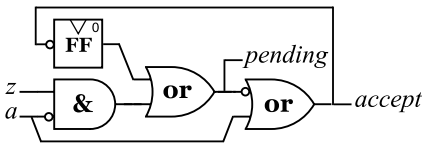


Fig. 2. *Monitor circuit for* $\mathbf{G}(z \rightarrow \mathbf{F}a)$.

Below, we illustrate some LTL operators as monitor circuits. $\mathbf{Y}f$ denotes the previous value of f (which translates directly into a zero-initialized FF whose next-state function is f), and *is_init* denotes a signal which is TRUE only in the first cycle.

If *accept* is left out, it is assumed to be constant TRUE. If *failed* or *pending* are left out, they are assumed to be constant FALSE.

$\mathbf{G}(z \rightarrow \mathbf{X}a)$
$pending = z$
$failed = \mathbf{Y}z \wedge \neg a$

$\mathbf{G}(z \rightarrow \mathbf{G}a)$
$pending = (\mathbf{Y}\ pending) \vee z \qquad [= \mathbf{P}z]$
$failed = pending \wedge \neg a \qquad [= \mathbf{P}z \wedge \neg a]$

$\mathbf{G}(z \rightarrow \mathbf{F}a)$
$pending = (z \vee (\mathbf{Y}\ pending)) \wedge \neg a$
$accept = \neg pending$

$\mathbf{G}(z \rightarrow [a\ \mathbf{W}\ b])$
$pending = (z \vee (\mathbf{Y}\ pending)) \wedge \neg b$
$failed = pending \wedge \neg a$

The 1-to-1 correspondence between this textual representation and a circuit diagram is illustrated for the $\mathbf{F}$ operator in *Figure 2*.

Past operators $\mathbf{P}a$ and $\mathbf{H}a$ are trivially implemented by a single flop remembering if a has held at least once, or always, in the past:

$$once\_a = a \vee (\mathbf{Y}\ once\_a)$$
$$always\_a = a \wedge \neg(\mathbf{Y}\ \neg always\_a)$$

VI. RUNNING THE MODEL CHECKER

Putting together all the steps of our approach:

1) The original specification ϕ is converted to an equivalent NNF formula ψ.
2) ψ is expanded to an equisatisfiable conjunction of "$\mathbf{G}(z_i \rightarrow \langle expr \rangle)$" formulas by introducing a variable z_i for each subformula.
3) For each such conjunct, a monitor circuit is created.
4) The initial activator z_0 is replaced by *is_init*.
5) All *failed* signals are OR-ed together and a flop is introduced to remember if any monitor has ever failed. Formally:

   init(*has_failed*) = 0
   next(*has_failed*) = *FAILED*
   FAILED = $failed_1 \vee \ldots \vee failed_n \vee has\_failed$

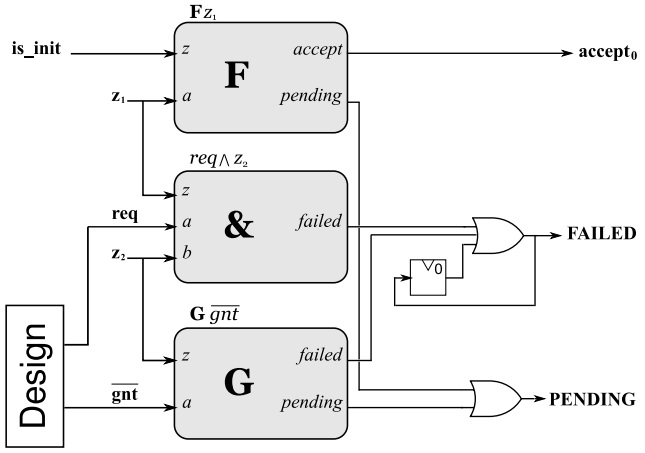


Fig. 3. *Monitor circuit for the LTL formula* $\mathbf{G}(\text{req} \rightarrow \mathbf{F}\,\text{gnt})$. After negation, the formula becomes $\mathbf{F}(req \wedge \mathbf{G}\neg gnt)$, which is translated into: $z_0 \wedge \mathbf{G}(z_0 \rightarrow \mathbf{F}z_1) \wedge \mathbf{G}(z_1 \rightarrow req \wedge z_2) \wedge \mathbf{G}(z_2 \rightarrow \mathbf{G}\neg gnt)$, where activators z_1 and z_2 are two new primary inputs introduced for the subformulas and "is_init", which is true only in the first cycle, replaces z_0 in the circuit.

Which is just another way of stating:

$$FAILED = \mathbf{P}\ (failed_1 \vee \ldots \vee failed_n)$$

6) The liveness checker is called on:

   infinitely_often($accept_1$, …, $accept_n$)

   under the constraint ¬*FAILED*. If the checker does not support constraints, it can be folded into the property:

   infinitely_often($accept_1 \wedge \neg FAILED$, …, $accept_n \wedge \neg FAILED$)

An example of a final monitor circuit is shown in *Figure 3*.

A. Safety fragment

Remember that we are working in the negative, and that disproving a safety property "*for-all-paths:* $\mathbf{G}\ p$" corresponds to finding a witness to "$\mathbf{F}\ \neg p$", i.e. a path to a bad state. Normally witnesses of temporal formulas are infinite traces, but in this case, *any* infinite extension of a finite prefix leading to the bad state is a valid witness. This is a *bounded witness* or *bad prefix* [11], and in our monitor formalism, it corresponds to having *no pending signals*. Therefore, a search for a witness to a temporal formula can be split into two parts: (i) the search for a finite, non-failing trace, where the last state has no pending signal; or (ii) the search for an infinite, non-failing trace where all *accepts* happen infinitely often. The key is that the first type of search can be carried out by a safety-checker, which is more efficient than the more general liveness-checker needed for the second type. The part of the property checkable by (i) is referred to as the *safety fragment*.

This observation can be used to improve our model checking process by:

1) Collecting pending signals:

   $$PENDING = pending_1 \vee \ldots \vee pending_n$$

2) Generating a safety check to be executed before the liveness check:

   reachable($\neg FAILED \wedge \neg PENDING$)

   If this call is UNSAT (no witness found), we run the liveness checker. The liveness property can then be constrained further by adding *PENDING* as a constraint.

B. Assumptions and Assertions

Generally, a specification is composed of two types of formulas, *assumptions*, modeling the behavior of the external environment, and *assertions*, describing the specific behavior of the design under verification. A counterexample for the specification must satisfy all the assumptions and violate at least one of the assertions. Unfortunately, if combined directly into a single LTL formula "*assumptions* → *assertions*", the constraints may force infinite counterexamples where finite ones are expected easier to find. Therefore, most verification tools check safety only under the requirement that assumptions have not yet been violated at the point where the assertion fails. As an example, consider a zero-initialized counter under the assumption "$\mathbf{G}$ (counter < 10)" and the assertion "$\mathbf{G}$ (counter $\neq 5$)". In five cycles, the counter will reach a bad state, but the system has no infinite runs that satisfy the assumption. A safety-checker would produce a counterexample, which is reasonable because the assumption fails after the bad state is reached. In contrast, a liveness tool would consider the property valid because there is no infinite counterexample.

To implement this relaxation in our framework, we ignore *accept* and *pending* for all monitors belonging to assumptions. This clearly changes the semantics of the property, but may be a reasonable compromise (and most probably what the user intended). This can be presented as an option to the user to be accepted or not.

VII. Optimizations

A. Monotonic signals

Suppose the user chose to use past operators to express weak until, as in the right-hand side of the following expression:

$$[a\ \mathbf{W}\ b] \ = \ \mathbf{G}(a \vee \mathbf{P}b)$$

Then, this will lead to the following translation:

$$\begin{array}{rl} & \mathbf{G}(z_0 \ \rightarrow \ \mathbf{G}z_1) \\ \wedge & \mathbf{G}(z_1 \ \rightarrow \ a \vee z_2) \\ \wedge & \mathbf{G}(z_2 \ \rightarrow \ \mathbf{P}b) \end{array}$$

Here we see a problem: as soon as the first monitor is activated ($z_0 = 1$), it will be forever pending. However, the native monitor of weak-until does not share this property. This can be resolved by observing that $\mathbf{P}b$ is a monotonic signal, and that once true, remains true, which motivates introducing a signal *done* for each monitor:

done: This signal should be TRUE only if the monitor has reached a state where *failed* can never happen and *accept*

will hold infinitely often. If this cannot be computed easily, *done* could conservatively be set to FALSE.

The *done* signal can be produced either explicitly by each monitor (extending the contract for what a monitor is), or derived by an analysis of the *failed* and *accept* signals. The pending condition for the $\mathbf{G}$ operator is updated to:

$$\begin{aligned}&\mathbf{G}(z \ \rightarrow \ \mathbf{G}a)\\ &\quad \mathit{pending} = \mathbf{P}z \ \wedge \ \neg a.\mathit{done}\\ &\quad \mathit{failed} = \mathit{pending} \wedge \neg a\end{aligned}$$

The default interpretation of "a.*done*" for a non-activator variable a, is FALSE. But all signals in the specification can be checked for monotonicity in the design by 1-induction, which is typically very fast. If an atomic signal a is monotonically increasing, "a.*done*" can be interpreted as just a. If not, "a.*done*" should still be treated as FALSE.

As an example of how the *done* signal can be explicitly produced, consider the operators:

$$\begin{aligned}&\mathbf{G}(z \ \rightarrow \ \mathbf{P}a)\\ &\quad \mathit{failed} = z \wedge \neg\mathbf{P}a\\ &\quad \mathit{done} = \mathbf{P}a\\ &\mathbf{G}(z \ \rightarrow \ a \vee b)\\ &\quad \mathit{failed} = z \wedge \neg(a \vee b)\\ &\quad \mathit{done} = a.\mathit{done} \ \vee \ b.\mathit{done}\end{aligned}$$

For reasonably sized LTL specifications, we can afford to do the following automated and more precise analysis using symbolic techniques, similar to the constraint analysis of [8]:

Done analysis. For each monitor M_i, let d_i denote "*accept*$\wedge\neg$*failed*" for the signals of that monitor, Let C denote a conjunction of constraints and invariants that known to hold for the system. This will include "$\neg$*FAILED*" as well as "$(\bigwedge_k s_k \ \rightarrow \ s'_k)$" for the monotonic signals $s_1, s_2, \ldots, s_k$ derived from the design. Now, for each monitor, check whether "$d_i \wedge C \rightarrow d'_i$" is true using SAT. If so, let the *done* signal for M_i be defined as d_i and continue the analysis. Optionally, $d_i \rightarrow d'_i$ can be added to C to strengthen future checks.

Note! It should be emphasized at this point that the proposed analysis of *failed* and *accept* signals, as well as the analyses described in the next two subsections, are performed *only* on the combined monitor circuit, which is small, and *not* on the design, which may be large. The only exception is the 1-induction step, which *is* performed on the design and offers a highly selective way of bringing some particularly useful information about the design into the analysis of the monitors. This invariant information (monotonicity of signals) can be used for the *done* analysis above, as well as in the analyses described in the next two subsections.

B. Deadlock states, Acceptable states and Reachable states

Deriving constraints is useful both for strengthening the analyses described in this section, and for proving the property. We make two observations:

- States that for any sequence of inputs will eventually reach *FAILED* cannot be part of a witness.
- States that cannot, for any sequence of inputs, reach a given accept_i signal cannot be part of a witness.

The first type of states corresponds to *deadlock states*, and is characterized by the transitive strong preimage of *FAILED*.[1] This set can be computed symbolically for the combined monitor circuit using e.g. BDDs or SAT based cube-enumeration. The negation is then added as a constraint to the system.

Similarly, the second type of states can be derived by taking the transitive (weak) preimage of each accept_i and intersecting the results. This corresponds precisely to constraint extraction for safety properties as described in [6], interpreting each accept_i signal as a *bad* state.

Finally, the (forward) reachable states of the combined monitors can be computed, and this invariant added to the set of constraints. Although it is redundant in the sense that it will not restrict the search space for finding witnesses, it can make inductive proofs easier and strengthen the analyses presented in this section.

We hypothesize that deriving constraints and invariants will give similar benefits to determinizing the automaton [2] when used with inductive proof-methods, such as k-induction, interpolation and PDR/IC3.

C. Fewer auxiliary variables

Introducing an auxiliary variable for each subformula is not always necessary. It is most obvious for the logical operators, where a subformula with multiple ANDs and ORs can be turned trivially into a single monitor with only one activator z_i, introducing a single new PI.

Also we can save on PIs and get a smaller translation for the $\mathbf{G}$-operator. If we have:

$$\mathbf{G}(z_i \rightarrow \mathbf{G}z_j)$$

then, assuming z_j is a PI introduced for a subformula, we can simply remove it and replace each occurrence of z_j by $\mathbf{P}z_i$. In the same way, we can save up to two PIs for each $\wedge$-operator:

$$\mathbf{G}(z_i \rightarrow z_j \wedge z_k)$$

If z_j and z_k are PIs, they can be replaced by z_i.

These transformations can be understood by looking at the definition of *failed*, which for the $\mathbf{G}$-operator is "$\mathit{failed} = \mathbf{P}z_i \wedge \neg z_j$". Since the left-hand side is constrained to FALSE, we have: $\neg(\mathbf{P}z_i \wedge \neg z_j) \ = \ (\mathbf{P}z_i \rightarrow z_j)$, and since we only need to propagate activation downwards to subformulas, it is safe to substitute z_j for $\mathbf{P}z_i$.

Another way of achieving simplifications of the above sort is by performing *signal correspondence* [16], [4], [12], [9] under constraints. This analysis will detect equivalent nodes in the combined monitor circuit and simplify the netlist by transferring fanouts from all equivalent nodes into one representative node. The fact that these signals are equivalent must

[1] The strong preimage of S is the set of predecessor for which *all* next states are in S.

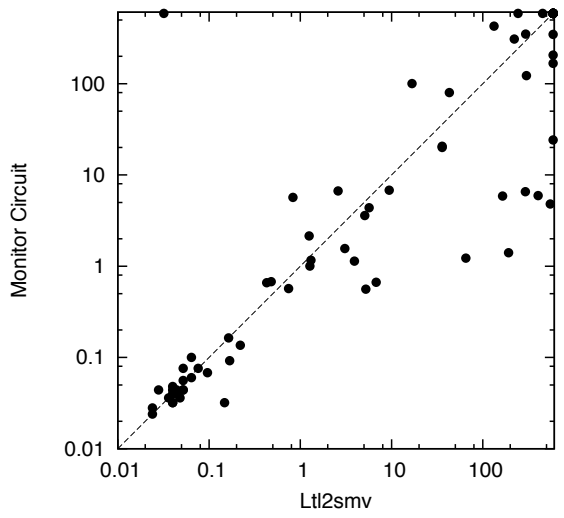


Fig. 4. *Scatter plot of results in log-scale.* This plots include all data points, even the short runs omitted from the table.

be maintained, but if on the left-hand side of the equivalence is a PI (as often happens) with no other fanouts, i.e. "PI $\leftrightarrow$ $\langle$*some node*$\rangle$", then that constraint can always be satisfied and hence dropped.

VIII. Experimental Results

For the experimental evaluation we used the same benchmarks as Biere et. al. in [3], which can be downloaded from [1]. The benchmark suite consists of 14 designs, 12 of which we could use (the *1394* and *csmacd* designs could not be handled correctly by our parser). Each design contains several PLTL properties. For each property, two monitors were generated, one using the method described in this paper, and one using the tool LTL2SMV [7], which builds a monitor from a Büchi automaton produced through the alternating automata approach. Both monitors were then combined with the design and given to a liveness checker [19]. Verification times are reported in *Figure 5* and plotted in *Figure 4*. The benchmarks were carried out on a dual 8-core Intel Xeon E5-2670 with 128 GB of memory, using a timeout of 600 seconds.

Analysis. LTL2SMV provides an alternating automata based approach without much optimizations. This was compared against an unoptimized implementation of the method presented in this paper. Verification runtimes suggests that the two methods are comparable with a small advantage to the new method. Because of its simplicity, this makes it an interesting option for industrial implementation as well as future research.

IX. Acknowledgments

The authors want to thank Shoham Ben-David, Robert Brayton and Alan Mishchenko for their invaluable feedback.

This work is partly supported by SRC contract 2265.001, NSA grant "Enhanced equivalence checking in cryptoanalytic applications", and NSF, grant# 1219154. We also thank industrial sponsors of BVSRC: Altera, Atrenta, Cadence, Calypto, IBM, Intel, Jasper, Mentor Graphics, Microsemi, Real Intent, Synopsys, Tabula, and Verific for their continued support.

Design	LTL2SMV (in sec)	CIRCUIT (in sec)
abp4-p2false	6.8	0.7
abp4-p2true	65.3	1.2
abp4-pold	191.6	1.4
abp4-ptimo	5.2	0.6
abp4-ptimoneg	0.8	5.7
bc57-sensors-p0	35.9	20.0
bc57-sensors-p0neg	547.8	4.8
bc57-sensors-p1	0.0	–
bc57-sensors-p1neg	404.6	5.9
bc57-sensors-p2	–	24.2
bc57-sensors-p2neg	164.5	5.9
bc57-sensors-p3	293.1	6.6
brp-p1	43.0	79.9
brp-ptimonegnv	16.8	100.4
dme2-ptimo	5.1	3.6
dme2-ptimonegnv	5.7	4.4
pci-p1	133.0	427.5
pci-pFtimo	9.4	6.8
pci-ptimo	35.8	20.6
prod-cons-p0	1.3	2.1
prod-cons-p0neg	2.6	6.7
prod-cons-p1	0.7	0.6
prod-cons-p1negnv	1.3	1.0
prod-cons-p5	3.1	1.6
prod-cons-p5neg	0.4	0.7
prod-cons-pold1	3.9	1.1
prod-cons-pold3	1.3	1.2
prod-cons-pold4	0.5	0.7
production-cell-p0neg	–	167.2
production-cell-p1	295.1	349.6
production-cell-p1neg	300.6	122.5
production-cell-p2	243.3	–
production-cell-p2neg	–	205.6
production-cell-p3	221.0	308.7
production-cell-p3neg	452.8	–
production-cell-p4	–	347.1
Total solved:	32	33

Fig. 5. *Table of results.* A timeout of 10 minutes was used. Benchmarks that were solved by both approaches in less than 0.5 seconds were removed from the table to conserve space.

References

[1] http://www.tcs.hut.fi/Software/benchmarks/LMCS-2006/.

[2] Roy Armoni, Sergey Egorov, Ranan Fraer, Dmitry Korchemny, and Moshe Y. Vardi. **Efficient LTL compilation for SAT-based model checking**. In *ICCAD*, pages 877–884, 2005.

[3] A. Biere, K. Heljanko, T. Junttila, Latvala T, and V. Schuppan. **Linear Encodings of Bounded LTL Model Checking**. In *Logical Methods in Computer Science, Vol. 2 (5:5)*, pages 1–64, 2006.

[4] P. Bjesse and K. Claessen. **SAT-based Verification without State Space Traversal**. In *Proc. of FMCAD'00. LNCS, Vol. 1954, pp. 372-389.*

[5] Udi Boker, Orna Kupferman, and Adin Rosenberg. **Alternation Removal in Büchi Automata**. In Samson Abramsky, Cyril Gavoille, Claude Kirchner, Friedhelm Meyer auf der Heide, and Paul G. Spirakis, editors, *ICALP (2)*, volume 6199 of *Lecture Notes in Computer Science*, pages 76–87. Springer, 2010.

[6] Gianpiero Cabodi, Paolo Camurati, Luz Garcia, Marco Murciano, Sergio Nocco, and Stefano Quer. **Speeding up Model Checking by Exploiting Explicit and Hidden Verification Constraints**. In *Proc. of DATE*, 2009.

[7] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. **NuSMV Version 2:**

An OpenSource Tool for Symbolic Model Checking. In *Proc. International Conference on Computer-Aided Verification (CAV 2002)*, volume 2404 of *LNCS*, Copenhagen, Denmark, July 2002. Springer.

[8] Koen Claessen and Niklas Sörensson. **A Liveness Checking Algorithm that Counts**. In *Proc. of FMCAD*, pages 52–59, 2012.

[9] Berkeley Logic Synthesis Group. **ABC: A System for Sequential Synthesis and Verification**. *http://www.eecs.berkeley.edu/~alanmi/abc/*, v00127p.

[10] Yonit Kesten, Amir Pnueli, and Li on Raviv. **Algorithmic Verification of Linear Temporal Logic Specifications**. In Kim Guldstrand Larsen, Sven Skyum, and Glynn Winskel, editors, *ICALP*, volume 1443 of *Lecture Notes in Computer Science*, pages 1–16. Springer, 1998.

[11] Orna Kupferman and Moshe Y. Vardi. **Model Checking of Safety Properties**. In Nicolas Halbwachs and Doron Peled, editors, *CAV*, volume 1633 of *Lecture Notes in Computer Science*, pages 172–183. Springer, 1999.

[12] A. Mishchenko, M. L. Case, R. K. Brayton, and S. Jang. **Scalable and Scalably-verifiable Sequential Synthesis**. In *Proc. of ICCAD'08, pp. 234-241.*

[13] Amir Pnueli and Aleksandr Zaks. **PSL Model Checking and Run-Time Verification Via Testers**. In Jayadev Misra, Tobias Nipkow, and Emil Sekerinski, editors, *FM*, volume 4085 of *Lecture Notes in Computer Science*, pages 573–586. Springer, 2006.

[14] Amir Pnueli and Aleksandr Zaks. **On the Merits of Temporal Testers**. In Orna Grumberg and Helmut Veith, editors, *25 Years of Model Checking*, volume 5000 of *Lecture Notes in Computer Science*, pages 172–195. Springer, 2008.

[15] G. Tseitin. **On the complexity of derivation in propositional calculus**. *Studies in Constr. Math. and Math. Logic*, 1968.

[16] C. A. J. van Eijk. **Sequential equivalence checking based on structural similarities**. In *IEEE TCAD, 19(7), July 2000, pp. 814-819.*

[17] Moshe Y. Vardi. **Alternating Automata and Program Verification**. In Jan van Leeuwen, editor, *Computer Science Today*, volume 1000 of *Lecture Notes in Computer Science*, pages 471–485. Springer, 1995.

[18] Moshe Y. Vardi and Pierre Wolper. **An Automata-Theoretic Approach to Automatic Program Verification (Preliminary Report)**. In *LICS*, pages 332–344. IEEE Computer Society, 1986.

[19] Berkeley Verification and Synthesis Research Center (BVSRC). **ABC-ZZ: A C++ framework for verification and synthesis**. *https://bitbucket.org/niklaseen/abc-zz.*

X. Appendix – Informative Prefixes

This section shows that the monitors described in this paper accepts precisely the *informative prefixes*. For brevity, the exposition is limited to the temporal operators $\mathbf{X}$ and $\mathbf{U}$. A *prefix*, or a finite trace, refers to an assignment to the atomic propositions for the first n cycles.

Definition 1 (Kupferman and Vardi [11])**.** *The prefix $s_1, \ldots, s_n$ is **informative** for a formula ϕ if there exists a map L from $\{1, \ldots, n+1\}$ to the set of subformulas of ϕ such that:*

1) $\phi \in L(1)$
2) $L(n+1) = \emptyset$
3) *If an atomic proposition* $p \in L(i)$ *then* $s_i \models p$
4) *If* $a \vee b \in L(i)$ *then* $a \in L(i)$ *or* $b \in L(i)$
5) *If* $a \wedge b \in L(i)$ *then* $a \in L(i)$ *and* $b \in L(i)$
6) *If* $\mathbf{X}a \in L(i)$ *then* $a \in L(i+1)$
7) *if* $[a\ \mathbf{U}\ b] \in L(i)$ *then either* $b \in L(i)$ *or* $a \in L(i)$ *and* $[a\ \mathbf{U}\ b] \in L(i+1)$

To match precisely the formalism of [11], an additional trivial monitor is added for the atomic propositions. For practical purposes it can be optimized away, resulting in the construction described in this paper:

$$\mathbf{G}(z \rightarrow a)$$
$$\mathit{failed} = z \wedge \neg a$$

Proposition 1. *If $s_1, \ldots, s_n$ is an informative prefix for formula ϕ, then the trace $s_1, \ldots, s_n$ is accepted by the monitor circuit of ϕ.*

Proof: The prefix assigns values to the atomic propositions. In our monitor formalism, additional primary inputs are introduced for for the *activator* variables. To complete the trace, the activator for ψ, z_ψ, is set to TRUE in cycle i iff $\psi \in L(i)$. It must now be shown that the augmented trace is accepted by the monitor. More precisely:

(a) The activator of ϕ is TRUE in the initial cycle.
(b) All *failed* signals are FALSE everywhere on the trace.
(c) All *pending* signals are FALSE in cycle n.

We first observe that on this augmented trace, if the *pending* signal holds, the activator must hold as well in the same cycle. For the atomic propositions, boolean connectives and $\mathbf{X}$, the *pending* signal is defined to be the activator signal, so the observation trivially holds. Assume for contradiction that the observation does not hold for $\psi = [a\ \mathbf{U}\ b]$, and let i be the first cycle in which it is false, i.e. the *pending* signal holds but the activator does not. From the monitor constructions, the *pending* signal for $[a\ \mathbf{U}\ b]$ is defined to be:

$$\mathit{pending} = (z_\psi \vee \mathbf{Y}\mathit{pending}) \wedge \neg z_b)$$

which can be simplified by the assumption that $z_\psi = 0$:

$$\mathit{pending} = \mathbf{Y}\mathit{pending} \wedge \neg z_b$$

For *pending* to hold on cycle i, it must hold on cycle $i-1$, but since i is the first cycle in which *pending* can be set without the activator, the activator must be TRUE in cycle $i-1$, and therefore $[a\ \mathbf{U}\ b] \in L(i-1)$. The definition of *pending* also requires that z_b must not hold in cycle $i-1$ and therefore $b \notin L(i-1)$. But the definition of L then forces $[a\ \mathbf{U}\ b]$ to hold in cycle i which contradicts the assumption, proving the observation for $[a\ \mathbf{U}\ b]$. Now:

(a) Follows directly from $\phi \in L(1)$.

(b) We prove for each operator separately:

- For $\psi = p$, an atomic proposition. $\mathit{failed} := z_\psi \wedge \neg p$. However, by definition of L, if $p \in L(i)$ then $s_i \models p$. z_p is true iff $p \in L(i)$, so the combination of $s_i \models \neg p$, and $z_p =$ TRUE can never happen.
- For $\psi = a \wedge b$, $\mathit{failed} := z_\psi \wedge \neg(z_a \wedge z_b)$. However, by definition of L, if $a \wedge b \in L(i)$ then $a \in L(i)$ and $b \in L(i)$. The connectors are TRUE iff the corresponding formulas are in $L(i)$, so the combination of $a \wedge b \in L(i)$, $a \in L(i)$, and $b \in L(i)$, can never happen. The same applies to $\psi = a \vee b$.
- For $\psi = \mathbf{X}a$, $\mathit{failed} := Y z_\psi \wedge \neg z_a$. However, by definition of L, $\mathbf{X}a \in L(i-1)$ and $a \notin L(i)$ can never happen together.
- for $\psi = [a\ \mathbf{U}\ b]$, $\mathit{failed} := \mathit{pending} \wedge \neg z_a$. We proved that if *pending* holds then the activator z_ψ must hold as well, therefore $[a\ \mathbf{U}\ b] \in L(i)$. Substituting the definition of *pending* gives us $\mathit{failed} := (z_\psi \vee \mathbf{Y}\mathit{pending}) \wedge \neg z_b \wedge \neg z_a$. So for failed to be TRUE, we must also have $a, b \notin L(i)$, but cannot happen together with $[a\ \mathbf{U}\ b] \in L(i)$.

(c) For each operator that can possibly set *pending* to TRUE:

- For $\psi = \mathbf{X}a$, $\mathbf{X}a \in L(n)$ requires that $a \in L(n+1)$, and therefore the activator of $\mathbf{X}a$ cannot be set on cycle n. Therefore, neither can *pending*.
- For $\psi = [a\ \mathbf{U}\ b]$, $\mathit{pending} := (z_\psi \vee Y\mathit{pending}) \wedge \neg z_b$. to make the *pending* signal TRUE in cycle n, we must have $[a\ \mathbf{U}\ b] \in L(n)$, which requires that either $a \in L(n) \wedge [a\ \mathbf{U}\ b] \in L(n+1)$ or $b \in L(n)$. The former cannot happen because $L(n+1) = \emptyset$, and therefore z_b must hold, setting the *pending* signal to FALSE.

■

Proposition 2. *If the finite trace $s_1, \ldots, s_n$ is accepted by the monitor circuit of ϕ, then $s_1, \ldots, s_n$ is also an informative prefix for formula ϕ.*

Proof: We need to show the existence of a map L. To facilitate its definition, we change all the activator variables for formulas $[a\ \mathbf{U}\ b]$ in cycle i to TRUE if the *pending* signal holds in cycle $i-1$. This is allowed because activator variables are not atomic proposition, and are not used in the definition of an informative prefix.

If *pending* held at cycle i before the change, then the change does not affect *failed* or *pending*. If *pending* held at cycle $i-1$ and did not hold in cycle i before the change, it means that z_b must already have held in cycle i, so again, *failed* and *pending* are not affected and the trace is still accepted by the monitor.

We define for $i \in \{1, \ldots, n\}$, $L(i) := \{\psi \mid$ activator signal of ψ is set in cycle $i\}$ and $L(n+1) := \emptyset$. It remains to show that this L satisfies the requirements of definition 1. The construction of the monitor guarantees that the activator of ϕ holds in the initial cycle.

The only operators that can prevent $L(n+1)$ from being empty are $\mathbf{X}$ and $\mathbf{U}$. For $\psi = \mathbf{X}a$, the activator cannot hold in cycle n because it is equal to *pending*, therefore $\mathbf{X}a \notin L(n)$. If $[a\ \mathbf{U}\ b] \in L(n)$, and *pending* is FALSE, then z_b must hold on cycle i, and therefore $b \in L(n)$, consistent with $L(n+1) = \emptyset$.

The rest of the conditions also hold. For atomic propositions, the boolean connectives and $\mathbf{X}$, *failed* becomes TRUE when the activator holds and the condition of L is violated. It is only for $\psi = [a\ \mathbf{U}\ b]$ that this is not immediately obvious. If the activator of $[a\ \mathbf{U}\ b]$ holds, then either z_b holds, and *pending* becomes FALSE, or z_b is FALSE and *pending* becomes TRUE, forcing the activator to hold in the next cycle and forcing z_a to hold for *failed* not to become TRUE. This is exactly the condition of definition 1. ■

XI. APPENDIX – LIST OF MONITORS FOR PLTL

Below is a complete list of monitors for PLTL. Variable t is local to each monitor. If *accept* is left out, it is assumed to be constant TRUE. If *failed* or *pending* are left out, they are assumed to be constant FALSE.

$\mathbf{G}(z \rightarrow \mathbf{X}a)$
$\mathit{pending} = z$
$\mathit{failed} = \neg \mathit{is\_init} \wedge (\mathbf{Y}z \wedge \neg a)$

$\mathbf{G}(z \rightarrow \mathbf{Y}a)$
$\mathit{failed} = z \wedge \neg\mathbf{Y}(a)$

$\mathbf{G}(z \rightarrow \mathbf{Z}a)$
$\mathit{failed} = z \wedge \mathbf{Y}(\neg a)$

$\mathbf{G}(z \rightarrow \mathbf{F}a)$
$\mathit{pending} = (z \vee (\mathbf{Y}\ \mathit{pending})) \wedge \neg a$
$\mathit{accept} = \neg \mathit{pending}$

$\mathbf{G}(z \rightarrow \mathbf{G}a)$
$\mathit{pending} = (\mathbf{Y}\ \mathit{pending}) \vee z$
$\mathit{failed} = \mathit{pending} \wedge \neg a$

$\mathbf{G}(z \rightarrow \mathbf{P}a)$
$t = \mathbf{Y}(t) \vee a$
$\mathit{failed} = z \wedge \neg t$

$\mathbf{G}(z \rightarrow \mathbf{H}a)$
$t = \neg\mathbf{Y}(\neg t) \wedge a$
$\mathit{failed} = z \wedge \neg t$

$\mathbf{G}(z \rightarrow [a\ \mathbf{W}\ b])$
$\mathit{pending} = (z \vee (\mathbf{Y}\ \mathit{pending})) \wedge \neg b$
$\mathit{failed} = \mathit{pending} \wedge \neg a$

$\mathbf{G}(z \rightarrow [a\ \mathbf{U}\ b])$
$\mathit{pending} = (z \vee (\mathbf{Y}\ \mathit{pending})) \wedge \neg b$
$\mathit{failed} = \mathit{pending} \wedge \neg a$
$\mathit{accept} = \neg \mathit{pending}$

$\mathbf{G}(z \rightarrow [a\ \mathbf{R}\ b])$
$\mathit{pending} = (z \vee (\mathbf{Y}\ \mathit{pending})) \wedge \neg a$
$\mathit{failed} = (z \wedge \neg b) \vee ((\mathbf{Y}\mathit{pending}) \wedge \neg b)$

$\mathbf{G}(z \rightarrow [a\ \mathbf{S}\ b])$
$t = (\mathbf{Y}t \wedge a) \vee b$
$\mathit{failed} = z \wedge \neg t$

$\mathbf{G}(z \rightarrow [a\ \mathbf{T}\ b])$
$t = b \wedge (\neg\mathbf{Y}(\neg t) \vee a)$
$\mathit{failed} = z \wedge \neg t$

$\mathbf{G}(z \rightarrow a \wedge b)$
$\mathit{failed} = z \wedge \neg(a \wedge b)$

$\mathbf{G}(z \rightarrow a \vee b)$
$\mathit{failed} = z \wedge \neg(a \vee b)$

$\mathbf{G}(z \rightarrow$ FALSE$)$
$\mathit{failed} = z$

$\mathbf{G}(z \rightarrow$ TRUE$)$
(nothing)