Vacuum Cleaning CTL Formulae[*]

Mitra Purandare and Fabio Somenzi

University of Colorado at Boulder
{Mitra.Purandare,Fabio}@Colorado.EDU

Abstract. Vacuity detection in model checking looks for properties that hold in a model, and can be strengthened without causing them to fail. Such properties often signal problems in the model, its environment, or the properties themselves. The seminal paper of Beer et al. [1] proposed an efficient algorithm applicable to a restricted set of properties. Subsequently, Kupferman and Vardi [15] extended vacuity detection to more expressive specification mechanisms. They advocated a more minute examination of temporal logic formulae than the one adopted in [1]. However, they did not address the issues of practicality and usefulness of this more scrupulous inspection. In this paper we discuss efficient algorithms for the detection of vacuous passes of temporal logic formulae, showing that a thorough vacuity check for CTL formulae can be carried out with very small overhead, and even, occasionally, in less time than plain model checking. We also demonstrate the usefulness of such a careful analysis with the help of case studies.

1 Introduction

The basic function of a model checker [9, 16] is to establish whether a certain property holds (or passes) in a given system; otherwise, to produce an error trace. By systematically exploring the state space of the system to be verified, a model checker relieves the user of the burden of generating test cases. However, the thoroughness of verification depends on the properties examined, and hence, ultimately, on the user.

In an effort to increase the efficacy of model checking as a debugging and verification approach, recent work has therefore considered how to assess the quality and comprehensiveness of a given set of properties. Two approaches have emerged: One consists of measuring the *coverage* of a set of properties [13, 14, 7, 6], defined in such a way that incomplete coverage exposes features of the model not adequately verified. As is commonly done in Automatic Test Pattern Generation [12], in this case one relates coverage to the fraction of alterations to the model that would be detected by the given set of properties.

The second approach to assess the quality of properties, which is the focus of this paper, is the detection of *vacuous passes* in temporal logic formulae [1, 15]. Following the definition of [1], a formula φ passes vacuously in a model K if it

---

passes in K, and there is a subformula φ' of φ that can be changed arbitrarily without affecting the outcome of model checking.

The vacuous pass of a formula often signals problems in any combination of the model, its environment, and the formula itself. Both in the approach based on measuring coverage, and in the one that checks for vacuous passes, the quality of a set of properties is related to "how snugly the properties fit the model." In coverage measurements one checks for modifications of the model that do not turn any passing properties into failing ones. In vacuity detection, by contrast, one looks for changes in the passing properties themselves that restrict the sets of states that satisfy them, without causing them to fail.

The authors of [1] identify a subset of ACTL called w-ACTL for which vacuity detection can be done efficiently. That is, vacuity detection for a w-ACTL formula φ amounts to model checking a *witness formula* that is obtained by replacing a subformula of φ with either true or false, and is therefore no more complex than φ. The definition of w-ACTL makes it possible to designate at most one operand of a binary operator as *interesting*. Vacuity detection is then restricted to replacement of the (unique) smallest interesting subformula of φ.

In [15] vacuity detection is extended to full CTL*. Furthermore, the restriction to interesting subformulae is lifted. The length of the witness formula for φ, however, is quadratic in the length of φ. Thus, an increase in the expressiveness of the logic and in the thoroughness of analysis is paid with an increase in the worst-case complexity of the algorithm.

The approach of [1] owes its efficiency to two factors. On the one hand, exactly one subformula is replaced for each formula. On the other hand, in the parse tree of φ, all the temporal operators are on the path connecting the root to the smallest interesting subformula, which is replaced by false. As a consequence, the witness formula for φ is very often trivial. This second factor does not show up in the worst-case complexity analysis, but is quite important in practice.

While the simplicity of the witness formulae for w-ACTL is an obvious advantage from the standpoint of speed, it is also the inherent limitation of the approach of [1], because drastic changes in the formula have only slight chances of detecting non-major flaws. Though in the initial stages of debugging, vacuity detection for w-ACTL has proved very useful, a more careful analysis may substantially improve the effectiveness of a model checker as a debugging tool.

In this paper we show that a thorough vacuity check as the one advocated in [15] can be implemented efficiently for Computational Tree Logic (CTL) formulae [9], so that the overhead relative to plain model checking is in practice very limited in spite of the worse complexity bound. Indeed, our algorithm may occasionally outperform plain model checking.

Instead of checking φ and the witness formulae generated by various replacements in a sequential fashion, we check φ and all its replacements in a single bottom-up pass over the parse tree of φ. At each node we exploit the relationships between the sets of states satisfying the various formulae. Depending on the number of negations along the path connecting a node to the root of the parse tree, the satisfying set of a witness subformula is either a lower bound

or an upper bound on the satisfying set of the corresponding subformula of φ. This allows us to speed up fixpoint computations by accelerating convergence, or simplifying the computation of preimages.

As demonstrated in [18, 3], starting a fixpoint computation from a good approximation may drastically reduce the time to convergence, especially with symbolic algorithms that may be greatly affected by the sizes of the Binary Decision Diagrams (BDDs [5]). In our approach to vacuity detection, the bounds on the fixpoints are not obtained by modifying the transition relation of the model. Hence, the effects are less dramatic, but sufficient to often allow several formulae to be model checked in about the same time as just one of them.

Other devices that help our algorithm limit the overhead are the detection of cases in which different replacements lead to equivalent formulae, or at least to identical computations; and the sharing of don't care and early termination conditions between φ and its witness formulae. The details of the algorithm are discussed in Sections 4 and 5.

A practical algorithm for thorough vacuity detection is only of limited import unless the analysis it performs is also useful: Presenting the user of a model checker with much information of scarce relevance is likely to decrease her productivity. Our experiments, however, indicate that a more minute examination of formulae than that based on replacement of just one subformula leads to the the discovery of more bugs and to the detection of weaknesses in formulae that would otherwise go unnoticed. The "signal to noise" ratio is also quite good, with most vacuous passes leading to improved verification. This is illustrated by the case studies described in Section 2, and the experiments summarized in Section 6.

2   The Case for Thorough Vacuity Detection

`Fpmpy` is a floating-point multiplier included among the examples distributed with VIS [20]. It implements a simplified version of the IEEE 754 standard for floating point arithmetic. Several CTL formulae test properties primarily related to the handling of special cases like infinities, NaNs, and denormals (which are not implemented by the model). The multiplier takes three clock cycles to complete an operation. Hence, it is natural for properties checked on its model to have the form

$$\mathsf{AG}(p \to \mathsf{AX\,AX\,AX}\,q) \ ,$$

where p and q are propositional formulae. These formulae are in w-ACTL, and their witness formulae according to [1] have the form

$$\mathsf{AG}(p \to \mathsf{AX\,AX\,AX}\,\mathsf{false}) \ .$$

These witness formulae fail trivially because the model has fair paths—hence the formulae reduce to $\mathsf{AG}\,\neg p$—and p, albeit different from formula to formula, holds in the first cycle of a computation. Accordingly, vacuity detection for w-ACTL reports no problems. By contrast, when replacing each leaf of the parse

tree with either true or false depending on the number of negations along the path from the leaf to the root, many replacements result in vacuous passes. This is especially true of the following formula

$$\text{AG}(\text{START} \wedge \text{valid}(x) \wedge \text{valid}(y) \rightarrow \text{valid}(z)) \ , \qquad (1)$$

where START holds in the first clock cycle of a computation, and valid() tells whether the inputs (x and y) or the output (z) are not denormals. Out of 24 replacements, 20 produce vacuous passes. Examination of the passing witness formulae revealed that:

1. The environment of the model lacks an assignment to a primary input to the multiplier (start).
2. The MSB of the exponent could be incorrect due to overflow during its computation.
3. The multiplier maintains the invariant AG valid(z). Hence, (1) can be strengthened by replacing the antecedent of the implication with true.

Two features of the more extensive analysis based on replacing all leaves are instrumental in highlighting the bugs and weaknesses of fpmpy: the replacement of non-interesting formulae (to prove the antecedent redundant), and the replacement of individual atomic propositions (to expose the problem with the MSB of the exponent).

The MinMax parameterized circuit [19, 10] computes the average of the minimum and maximum of a stream of n-bit numbers. The following property

$$\text{AG}((\text{min} = 2^n - 1 \wedge \text{max} = 0) \rightarrow$$
$$\text{AX}((\text{min} = 2^n - 1 \wedge \text{max} = 0) \vee (\text{min} = \text{last} \wedge \text{last} = \text{max}))) \qquad (2)$$

states that from a reset state, in which min holds the largest possible n-bit integer, and max is 0, it is only possible to transit to another reset state, or to a state in which both min and max have the same value as the last input (last). This time, thorough vacuity detection uncovers no errors in the model, but it points out that the system satisfies the invariant

$$\text{AG}((\text{min}[n-1] = 1 \wedge \text{max}[n-1] = 0) \rightarrow (\text{min} = 2^n - 1 \wedge \text{max} = 0)) \ , \qquad (3)$$

where $n-1$ is the index of the most significant bit. The set of properties can be enhanced by adding (3) or by strengthening (2) to

$$\text{AG}((\text{min}[n-1] = 1 \wedge \text{max}[n-1] = 0) \rightarrow$$
$$\text{AX}((\text{min}[n-1] = 1 \wedge \text{max}[n-1] = 0) \vee (\text{min} = \text{last} \wedge \text{last} = \text{max}))) \ .$$

Our last example illustrates a possible drawback of exhaustively replacing all the leaves of the parse tree with either true or false. In a model of n dining philosophers [11], each philosopher may be in one of four states: thinking, left, right, and both, depending on which chopsticks she is holding. Mutual exclusion

requires that a philosopher may not hold the chopstick to her right if her right neighbor holds the one to his left. This may be written as

$$\mathsf{AG}\,\neg((p[i] = \mathtt{right} \vee p[i] = \mathtt{both}) \wedge$$
$$(p[(i + 1) \bmod n] = \mathtt{left} \vee p[(i + 1) \bmod n] = \mathtt{both}))\ . \qquad (4)$$

Suppose the state $p[i]$ of the i-th philosopher is encoded by two binary variables, $l[i]$ and $r[i]$, each one indicating possession of one chopstick. Then thorough vacuity detection will report vacuous passes, indicating that (4) can be rewritten as

$$\mathsf{AG}\,\neg(r[i] \wedge l[(i + 1) \bmod n])\ . \qquad (5)$$

Since the two formulations are equivalent, the quality of verification is not affected by this change, and, depending on the description style, converting (4) into (5) may require extensive modifications of the model. Therefore, report of vacuous passes in this case may be regarded as noise. However, this appears a reasonable price to pay for the advantages afforded by a careful examination the properties.

3   Preliminaries

The logic CTL [8] is defined over an alphabet A of atomic propositions: Any atomic proposition is a CTL property, and if φ and ψ are CTL properties, then so are $\varphi \wedge \psi$, $\varphi \vee \psi$, $\neg\varphi$, and $\mathsf{E}\,\varphi\,\mathsf{U}\,\psi$, $\mathsf{EG}\,\varphi$, and $\mathsf{EX}\,\varphi$. The semantics of CTL are defined over a Kripke structure $K = \langle S, T, S_0, A, L \rangle$, where S is the set of states, $T \subseteq S \times S$ is the transition relation, $S_0 \subseteq S$ is the set of initial states, A is the set of atomic propositions, and $L : S \to 2^A$ is the labeling function. The semantics of CTL are defined in Figure 1. If fairness constraints are specified, the path quantifiers are restricted to *fair paths*, that is, paths that intersect every fairness constraint infinitely often. A formula is said to hold in K if it is satisfied by every initial state of K. An ECTL formula is a CTL formula in which negation is only applied to the atomic propositions. An ACTL formula is the negation of an ECTL formula. A property that is neither ECTL nor ACTL is a *mixed property*.

$$
\begin{aligned}
&K, s_0 \models \varphi && \text{iff } \varphi \in L(s_0) \text{ for } \varphi \in A \\
&K, s_0 \models \neg\varphi && \text{iff } K, s_0 \not\models \varphi \\
&K, s_0 \models \varphi \vee \psi && \text{iff } K, s_0 \models \varphi \text{ or } K, s_0 \models \psi \\
&K, s_0 \models \varphi \wedge \psi && \text{iff } K, s_0 \models \varphi \text{ and } K, s_0 \models \psi \\
&K, s_0 \models \mathsf{EX}\,\varphi && \text{iff there exists a path } s_0, s_1, \ldots \text{ in } K \text{ such that } K, s_1 \models \varphi \\
&K, s_0 \models \mathsf{EG}\,\varphi && \text{iff there exists a path } s_0, s_1, \ldots \text{ in } K \text{ such that for } i \geq 0, \\
& && \quad K, s_i \models \varphi \\
&K, s_0 \models \mathsf{E}\,\varphi\,\mathsf{U}\,\psi && \text{iff there exists a path } s_0, s_1, \ldots \text{ in } K \text{ such that there exists} \\
& && \quad i \geq 0 \text{ for which } K, s_i \models \psi, \text{ and for } 0 \leq j < i, K, s_j \models \varphi.
\end{aligned}
$$

Fig. 1. Semantics of CTL

Boolean operators other than $\wedge$, $\vee$, and $\neg$, and the operators EF, AX, AG, AF, and AU can be defined as abbreviations, e.g., $\mathsf{EF}\,\varphi = \mathsf{E}(\varphi \vee \neg\varphi)\,\mathsf{U}\,\varphi$, $\mathsf{AX}\,\varphi = \neg\,\mathsf{EX}\,\neg\varphi$, $\mathsf{AG}\,\varphi = \neg\,\mathsf{EF}\,\neg\varphi$, $\mathsf{AF}\,\varphi = \neg\,\mathsf{EG}\,\neg\varphi$, and $\mathsf{A}\,\varphi\,\mathsf{U}\,\psi = \neg(\mathsf{E}\,\neg\psi\,\mathsf{U}\,\neg(\varphi \vee \psi)) \wedge \neg\,\mathsf{EG}\,\neg\psi$. Clearly, the abbreviations should be expanded before checking whether a formula is an ECTL or ACTL formula.

The model checking problem for CTL with fairness constraints can be translated into the computation of fixpoints of appropriate functionals [17]:

$$\mathsf{E}\,\varphi\,\mathsf{U}\,\psi = \mu Z \,.\, \psi \vee (\varphi \wedge \mathsf{EX}\,Z),$$
$$\mathsf{EG}\,\varphi = \nu Z \,.\, \varphi \wedge \mathsf{EX}\,Z,$$
$$\mathsf{E}_C\,\mathsf{G}\,\varphi = \nu Z \,.\, \varphi \wedge \mathsf{EX} \bigwedge_{c \in C} (\mathsf{E}\,Z\,\mathsf{U}(Z \wedge c)) \ ,$$

where C is a set of sets of states that must be traversed infinitely often by a fair path, and where with customary abuse of notation, we identify a formula and the set of states where it is satisfied. Also, we often do not distinguish between a set and its characteristic function. Thus, $p \wedge \neg\{s_0\}$ stands for the characteristic function of the set consisting of the states in the set whose characteristic function is p, except for s_0. When we want to mark the difference between a formula and its satisfying set, we let $[\![\varphi]\!]$ denote the satisfying set of φ.

Note that EU, EG, and EX are monotonic both in their arguments and in the transition relation.

In vacuity detection we replace an occurrence of subformula φ' in φ with another formula ψ; this is denoted by $\varphi[\varphi' \leftarrow \psi]$ and is called the *witness* of φ'. We write $\varphi[\varphi' \leftarrow \bot]$ for $\varphi[\varphi' \leftarrow \mathsf{false}]$ if φ' appears in φ under an even number of negations. Otherwise, $\varphi[\varphi' \leftarrow \bot]$ stands for $\varphi[\varphi' \leftarrow \mathsf{true}]$.

A formula φ passes vacuously in K if $K \models \varphi$, and there exists an occurrence of a subformula φ' of φ such that $K \models \varphi[\varphi' \leftarrow \bot]$. If this holds, then $K \models \varphi[\varphi' \leftarrow \psi]$ for any ψ. This follows from the monotonicity of the operators involved in model checking [15].

4   Combining Model Checking and Vacuity Detection

In this section we describe an efficient algorithm that combines the model checking of a CTL formula φ with thorough detection of vacuous passes. We assume that φ is given as a parse *tree* (as opposed to a parse *graph*). That is, each occurrence of a given subformula is considered separately.

We assume that φ only contains existential quantifiers. The only operators that label the internal nodes of the parse tree are therefore $\neg$, $\wedge$, EX, EU, and EG. This choice prevents us from putting formulae in negation normal form. Instead, each node of the parse tree is annotated with its *negation parity*, that is the number of nodes labeled $\neg$ on the path connecting the root of the tree to the parent of the node itself.[1]

---

[1] Our implementation also allows nodes of the parse tree labeled by $\vee$, $\rightarrow$ and $\oplus$. An implication node counts as a negation for its antecedent child, but not for its conse-

Let $\Pi = (N, E, \lambda)$ be the parse tree of a CTL formula φ with atomic propositions from $A \neq \emptyset$, where $N = \{i : 1 \leq i \leq n\}$ is the set of nodes; $E \subseteq N \times N$ is the set of edges; and $\lambda : N \to A \cup \{\neg, \wedge, \mathsf{EX}, \mathsf{EU}, \mathsf{EG}\}$ labels each node of the parse tree with either an atomic proposition or an operator. The root of the parse tree is node n.

The outdegree of a node i obeys the obvious restrictions: if $\lambda(i) \in A$ it is 0; if $\lambda(i) \in \{\neg, \mathsf{EX}, \mathsf{EG}\}$ it is 1; otherwise, it is 2. Let $\nu : N \to \{0, 1\}$ map each node to its negation parity: $\nu(n) = 0$, and if $(i, j) \in E$, then $\nu(i) = \nu(j)$ if and only if $\lambda(i) \neq \neg$. If $\nu(i) = 0$, i is an *even-parity* node; otherwise, it is an *odd-parity* node. For $1 \leq i \leq n$ let Π_i denote the subtree of Π rooted at node i, and φ_i denote the CTL formula represented by Π_i.

To check whether subformula φ_j of φ affects the truth value of φ in K, we replace φ_j with $\bot$. If $\nu(i) = 0$, $[\![\varphi_i[\varphi_j \leftarrow \bot]]\!] \subseteq [\![\varphi_i]\!]$, whereas if $\nu(i) = 1$, $[\![\varphi_i]\!] \subseteq [\![\varphi_i[\varphi_j \leftarrow \bot]]\!]$. This observation is the basis for our algorithm.

Given a *replacement function* $\rho : N \to \{\emptyset, \bot\}$, let

$$\Psi(\Pi, \rho) = \{\varphi\} \cup \{\varphi[\varphi_i \leftarrow \bot] : \rho(i) = \bot\} \ .$$

A *vacuity detection experiment* for φ in K is defined by a triple (K, Π, ρ): It consists of answering, for each $\psi \in \Psi(\Pi, \rho)$ the model checking question $K \models \psi$. If $K \models \varphi$, each additional affirmative answer is a *vacuous pass*.

Let $\delta : N \to 2^N$ map node i to the set of nodes that are reachable from i and such that $j \in \delta(i)$ implies $\rho(j) = \bot$. That is, $\delta(i)$ is the set of descendants of i that have been marked for replacement. Our algorithm computes for each $i \in N$ a function $\sigma_i : \delta(i) \cup \{0\} \to (\delta(i) \cup \{0\}) \times 2^Q$ such that $\sigma_i(j) = (k, S)$ satisfies the following conditions.

1. $k \leq j$;
2. $\sigma_i(0) = (0, [\![\varphi_i]\!])$;
3. if $k = j \neq 0$, then $S = [\![\varphi_i[\varphi_j \leftarrow \bot]]\!]$;
4. if $k < j$, then $S = \emptyset$ and $[\![\varphi_i[\varphi_j \leftarrow \bot]]\!] = [\![\varphi_i[\varphi_k \leftarrow \bot]]\!]$.

The computation is performed by post-order traversal of the parse tree. At each node i, first we compute a "draft" of σ_i, that we call σ'_i, and then derive σ_i from it by a *reduction* process. The details of the computation of σ'_i depend on $\lambda(i)$ as follows, except for $\sigma'_i(i)$, which is always $(i, \bot)$ if $\rho(i) = \bot$, and undefined otherwise.

1. If $\lambda(i) \in A$, $\sigma'_i(0) = (0, [\![\varphi_i]\!])$.
2. If $\lambda(i) \in \{\neg, \mathsf{EX}\}$, let c be the child of i, and, for $j \in \delta(i) \cup \{0\} \setminus \{i\}$, let $\sigma_c(j) = (k, P)$. Then, if $k = j$, $\sigma'_i(j) = (j, \lambda(i)P)$; otherwise $\sigma'_i(j) = (k, \emptyset)$.
3. If $\lambda(i) = \wedge$, let l and r the two children of i and, for $j \in \delta(l) \cup \{0\}$, let $\sigma_l(j) = (k, P)$. Then, if $k = j$, $\sigma'_i(j) = (j, P \wedge [\![\varphi_r]\!])$; otherwise $\sigma'_i(j) = (k, \emptyset)$. Note that $i \notin \delta(l)$, $\delta(l) \subseteq \delta(i)$, and $\delta(l) \cap \delta(r) = \emptyset$. The case for $j \in \delta(r)$ is similar.

quent child. The exclusive-or requires special treatment, because it is not monotonic: We do not allow replacements of the descendants of a node labeled $\oplus$. Since these are implementation details, we shall not discuss them further.

4. If $\lambda(i) = \mathsf{EG}$ the computation proceeds as in the case of EX. However, the order in which the values of σ'_i are determined is relevant: $\sigma'_i(0)$ is computed last. If $\nu(i) = 1$,

$$U = \bigwedge \{P : \sigma'_i(j) = (j, P), j \in \delta(i) \setminus \{0\}\}$$

is used as an upper bound in computing $[\![\varphi_i]\!]$. Otherwise,

$$L = \bigvee \{P : \sigma'_i(j) = (j, P), j \in \delta(i) \setminus \{0\}\}$$

is used as lower bound. (The use of lower bounds in greatest fixpoint computations is discussed in Section 5.)
5. Finally, if $\lambda(i) = \mathsf{EU}$, the computation proceeds as in the case of EX. However, the order in which the values of σ'_i are determined is relevant: if $\nu(i) = 1$, $\sigma'_i(0)$ is computed first, and $[\![\varphi_i]\!]$ is used as lower bound in the other fixpoint computations; otherwise $\sigma'_i(0)$ is computed last, and

$$L = \bigvee \{P : \sigma'_i(j) = (j, P), j \in \delta(i) \setminus \{0\}\}$$

is used as a lower bound in computing $[\![\varphi_i]\!]$.

The reason for computing $\sigma'_i(0)$ last in case of greatest fixpoint, regardless of $\nu(i)$ is that the BDD for $[\![\varphi[\varphi_j \leftarrow \bot]]\!]$ are likely to be smaller than that for $[\![\varphi]\!]$ because the formula is simpler.

The reduction process that derives σ_i from σ'_i sets $\sigma_i(j) = (k, \emptyset)$ if $\sigma'_i(j) = (j, P)$, there is $k < j$ such that $\sigma'_i(k) = (k, P)$, and k is the least number that satisfies this condition. Furthermore, if $\sigma'_i(j) = (k, \emptyset)$, and $\sigma_i(k) = (k', \emptyset)$, then $\sigma_i(j) = (k', \emptyset)$.

The following result states that the solution to a vacuity detection experiment is contained in σ_n.

Theorem 1. *If* $\sigma_n(j) = (j, P)$ *then* $[\![\varphi[\varphi_j \leftarrow \bot]]\!] = P$*; otherwise, if* $\sigma_n(j) = (k, \emptyset)$*, then* $\sigma_n(k) = (k, P)$*,* $P \neq \emptyset$*, and* $[\![\varphi[\varphi_j \leftarrow \bot]]\!] = P$*.*

4.1   Early Termination and Don't Cares

It is sometimes possible to avoid computing either $[\![\varphi]\!]$ or $[\![\varphi[\varphi' \leftarrow \bot]]\!]$. Suppose a set of care states is given at the current node of the parse tree. At the root of the parse tree, this set of states is all the initial states. If we ignore vacuity detection, the computation can be terminated early as soon as it is known that all the care states satisfy the formula, or as soon as it is known that at least one of the care states does not satisfy it. Hence, if the root of the parse tree is a least fixpoint, then computation can be terminated as soon as it is known that the formula passes; if it is a greatest fixpoint, early termination occurs as soon as it is known that the formula fails.

Another source of care states is the satisfying set of the sibling of the current node when the parent is labeled $\wedge$. The computation of the second child of such

a node r can be stopped as soon as all the states satisfying the sibling are known to satisfy r as well, or when it is known that no state satisfying r satisfies its sibling.

These observations can be extended to vacuity detection. For instance, if the negation parity of the node is even, and $[\![\varphi[\varphi' \leftarrow \bot]]\!]$ contains all the care states, then the computation of $[\![\varphi]\!]$ can be skipped. Likewise, if the negation parity is odd and $[\![\varphi[\varphi' \leftarrow \bot]]\!]$ contains no care state, neither does $[\![\varphi]\!]$.

4.2   Complexity

In practice, sharing partial results between the evaluation of the given formula and the evaluation of its witness formula is beneficial. However, in the worst case, we still have a quadratic bound (cf.[15]). To make things more precise, we consider the number of node evaluations as our metric. If a formula and all the witness formulae obtained by replacing a subformula with $\bot$ are model checked independently, then the number of node evaluations is quadratic in the length of the formula.

When sharing takes place, if we assume that replacements are limited to the leaves, the number of evaluations of each node in the CTL parse tree is bounded by the number of leaves below the node plus 1 (for the original formula).

For a balanced binary tree the total number of node evaluations is $O(n \log n)$, but for generic 2-restricted trees (each node has at most two children), the number of evaluations is $O(n^2)$. To see this, consider a binary tree such that at least one child of each node is a leaf.

5   Updating Greatest Fixpoints Using Lower Bounds

In this section we show how knowledge of a lower bound to a greatest fixpoint can be used to speed up its computation. It is well-known how use a lower bound for the computation of least fixpoints as follows. Let $l = \mathsf{E}\,q\,\mathsf{U}\,p$ and $u \leq l$. Then,

$$l = \mathsf{E}\,q\,\mathsf{U}(p \vee u) \ . \tag{6}$$

We can similarly use an upper bound to compute a greatest fixpoint. Now suppose $g = \mathsf{E}\,q\,\mathsf{R}\,p$ and $u \leq g$. Then,

$$g = u \vee \mathsf{E}(q \vee \mathsf{EX}\,u)\,\mathsf{R}(\neg u \wedge p) \ . \tag{7}$$

In words, (7) says that a state in g is either a state in u, or a state on an infinite path entirely contained in $\neg u \wedge p$, or a state with a finite path in $\neg u \wedge p$ that leads to a state still in $\neg u \wedge p$ that satisfies q or has a successor in u.

The iterates of the fixpoint in (7) are all contained in $\neg u \wedge p$; hence, they can be regarded as frontiers. While using (6) may speed up convergence, (7) may decrease the cost of the image computations, but does not affect the number of iterations.

As in the case of least fixpoints, frontiers can be optimized using u as don't care. We can apply this approach also to the computation of greatest fixpoints under fairness constraints.

$$\mathsf{E}_C \, \mathsf{G} \, p = u \vee \mathsf{E}(\neg u \wedge p) \, \mathsf{U}(\neg u \wedge p \wedge \mathsf{EX} \, u) \vee \mathsf{E}_C \, \mathsf{G}(\neg u \wedge p) \ . \tag{8}$$

Equation (8) says that if the states in u are on fair paths contained in p, then a state on a fair path contained in p is either a state in u, or a state on a fair path entirely contained in $\neg u \wedge p$, or a state on a finite path reaching a state in $\neg u \wedge p$ that has a successor in u. In the absence of fairness constraints, one can show that (7) and (8) are equivalent.

This result can be combined with [2, Theorem 4]: In guided search, u must be a lower bound on the final fixpoint, not necessarily on the one that is being computed.

Example 1. Consider the structures of Fig. 2, adapted from [2]. Part (3) shows the original structure, while Parts (1) and (2) show structures obtained by the application of hints. Suppose we want to compute $\mathsf{EG} \, p$. Let $\mathsf{EX}_i \, Z$ the operator that computes the predecessors of the states in Z in the structure of Part (i) of Fig. 2. (The other temporal logic operators are similarly annotated.) From the graph of Part (1) we compute

$$\eta_1 = \nu Z \, . \, p \wedge \mathsf{EX}_1 \, Z = \{s_4\} \ .$$
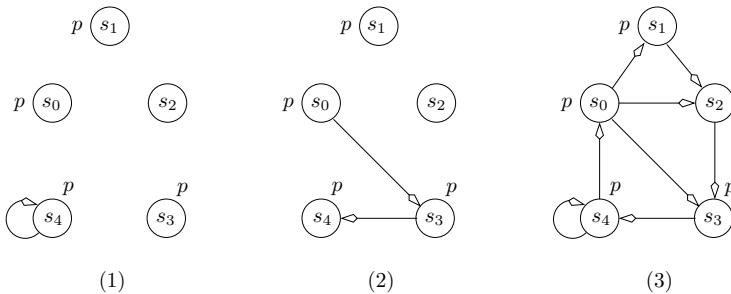
From the graph of Part (2) we compute:

$$\eta_2 = \{s_4\} \vee \mathsf{E}(\mathsf{EX}_2\{s_4\}) \, \mathsf{R}_2(p \wedge \neg\{s_4\}) = \{s_0, s_3, s_4\} \ .$$

The fixpoint computation requires three preimages:

$$\mathsf{EX}_2\{s_4\}, \mathsf{EX}_2\{s_0, s_1, s_3\}, \text{ and } \mathsf{EX}_2\{s_0, s_3\} \ .$$

By contrast, the non-incremental approach computes

$$\eta_2 = \nu Z \, . \{s_4\} \vee (\{s_0, s_1, s_3, s_4\} \wedge \mathsf{EX}_2 \, Z) \ ,$$

(1)   (2)   (3)

Fig. 2. Combining Theorem 4 of [2] with incremental greatest fixpoint computation

which requires the following preimages:

$$\mathsf{EX}_2\{s_0, s_1, s_3, s_4\} \text{ and } \mathsf{EX}_2\{s_0, s_3, s_4\} \ .$$

Finally, from the graph of Part (3) of Fig. 2, we compute:

$$\eta_3 = \{s_0, s_3, s_4\} \vee \mathsf{E}(\mathsf{EX}_3\{s_0, s_3, s_4\}) \, \mathsf{R}_3(p \wedge \neg\{s_0, s_3, s_4\}) = \{s_0, s_3, s_4\} \ ,$$

The fixpoint computation converges after one iteration with $Z = \emptyset$, having computed $\mathsf{EX}_3\{s_0, s_3, s_4\}$ and $\mathsf{EX}_3\{s_1\}$. By contrast, the non-incremental approach computes

$$\eta_3 = \nu Z \,.\{s_0, s_3, s_4\} \vee (\{s_0, s_1, s_3, s_4\} \wedge \mathsf{EX}_3 \, Z) \ ,$$

which in turn requires the computation of $\mathsf{EX}_3\{s_0, s_1, s_3, s_4\}$.   □

In general, we see that the incremental approach leads to more EXs (because $\mathsf{EX}\,u$ must be computed). On the other hand, each EX is applied to a smaller set.

6   Experiments

In this section we present preliminary results obtained with an implementation of the proposed algorithms in VIS 1.4 [4]. The CPU times were measured on an IBM IntelliStation running Linux with a 1.7 GHz Pentium 4 CPU and 1 GB of RAM. We checked a total of 588 formulae on 88 models.

Fig. 3 compares the run times for model checking without vacuity detection (Plain), checking all witnesses serially (Naive), and our algorithm (Vacuum). For reference, the time spent for reachability analysis is also shown (Reach). Only the experiments in which plain model checking took more than one second are shown in this plot. The replacements affected all the leaves of all formulae. (If no witnesses for the leaves pass, then no other witnesses will pass.) Though usually the witnesses are easier to model check than the given formula, this is not always the case. (See, for instance, vending.) However, in most cases, the time for a thorough vacuity detection is close to that for plain model checking. Our algorithm clearly dominates the naive approach.

Of the 588 formulae, 470 (80%) passed. A total of 2880 witnesses was generated, and 547 passed; the percentage of vacuous passes was therefore 19%. These vacuous passes were found in 35 different models (40%) and in a total of 100 formulae (17%). These results are detailed in Table 1. For each of the 88 designs, the table gives the number of properties that were checked, and how many passed; it also gives how many witnesses were generated and how many resulted in vacuous passes.

Of the 588 formulae, 411 (70%) are w-ACTL. For these we also ran the algorithm of [1]. We found that 34 formulae (6%) in 8 models (9%) caused vacuous passes.
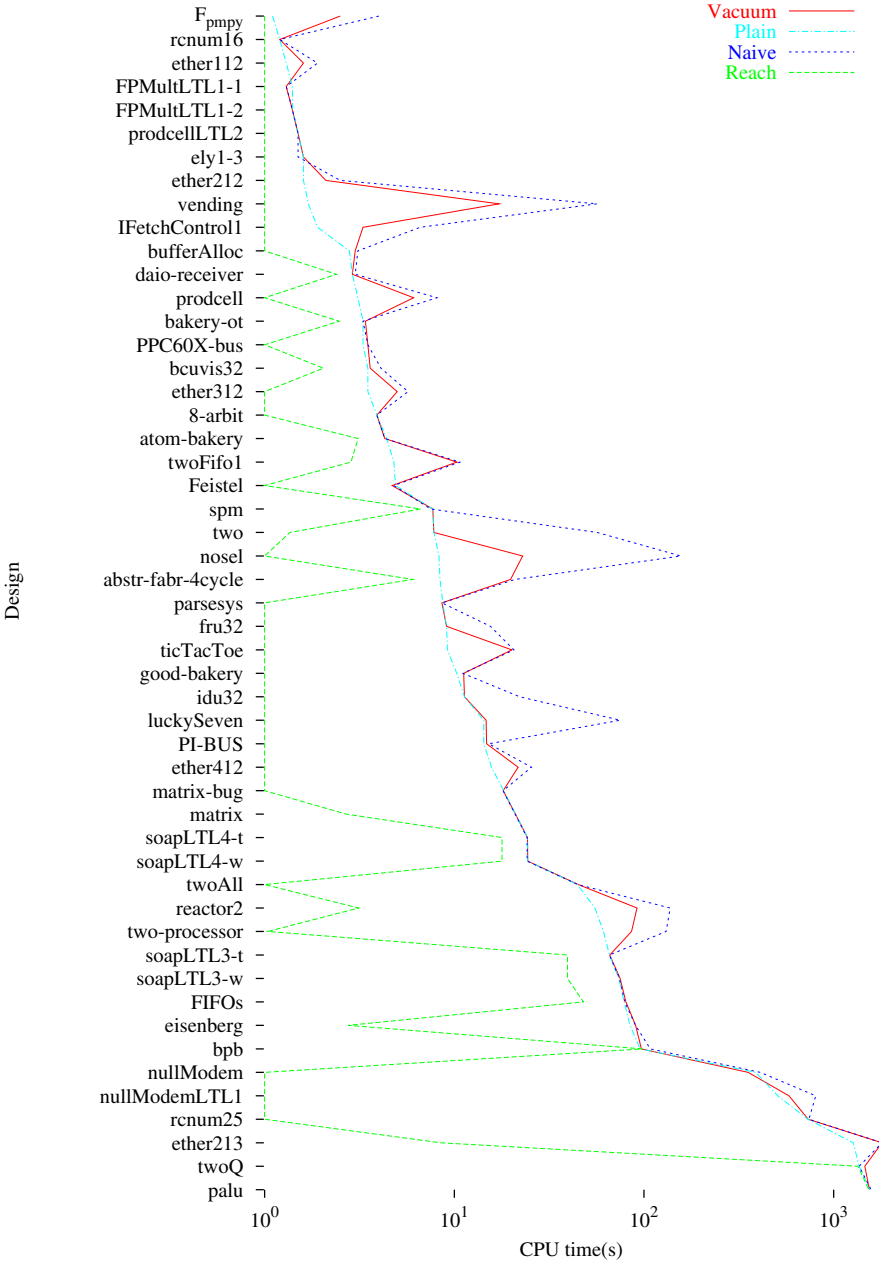
Fig. 3. CPU times

Table 1. Statistics for vacuity detection experiments

Design	Passed/ Total	Vacuous Passes/ Witnesses	Design	Passed/ Total	Vacuous Passes/ Witnesses
abp	3/3	5/19	nullModem	1/1	0/2
arbiter	4/4	0/12	nullModemLTL1	1/1	1/1
arbiter_bug	4/4	0/12	PI_BUS	19/25	6/43
eisenberg	9/9	0/18	PPC60X_bus	24/30	3/38
good_bakery	9/10	0/18	palu	1/2	0/1
bakery_ot	1/1	0/1	pf	0/2	0/0
atom_bakery	1/1	0/4	peterson	3/3	2/6
synch_bakery	4/6	0/8	philo	1/2	40/80
bpb	2/3	0/143	nosel	2/2	1/260
bufferAlloc	1/1	5/22	philo4	8/8	0/16
two_processor	5/13	0/11	drop4	4/8	0/8
coherence	8/10	9/32	ping_pong_new	3/5	1/6
counter	1/1	0/1	prodcell	9/10	0/19
crd	3/3	1/6	prodcellLTL2	1/1	1/1
ctlp3	1/1	0/2	rcnum16	1/1	0/15
daio_receiver	3/4	0/13	rcnum25	1/1	0/24
dcnew	3/7	1/6	reactor2	2/3	0/5
ely1-3	0/1	0/0	reqAck	2/2	3/13
ether112	6/6	2/20	reqAckRed	2/2	3/13
ether212	6/6	2/20	reset	0/1	0/0
ether312	6/6	2/20	rgraph	0/1	0/0
ether412	6/6	2/20	twoFifo1	3/4	6/19
FIFOs	1/1	0/1	twoQ	5/7	7/25
Fpmpy	5/8	24/99	short	2/3	1/4
FPMultLTL1-1	1/1	0/1	bcuvis32	10/10	43/66
FPMultLTL1-2	1/1	1/1	fru32	11/11	128/167
abstr_fabr_4cycle	8/8	0/62	idu32	21/21	69/126
Feistel	1/1	0/2	controlvis	17/17	28/103
two	1/2	0/2	pcuv	17/17	42/80
twoAll	5/5	0/35	spm	1/2	0/3
luckySeven	1/3	0/25	soapLTL3-t	0/1	0/0
gigamax	9/9	0/12	soapLTL3-w	0/1	0/0
gray	3/4	2/10	soapLTL4-t	1/1	0/1
ibuf	1/1	0/21	soapLTL4-w	1/1	1/1
IFetchControl1	15/22	17/89	solitaireVL	0/1	0/0
island	3/3	0/6	spinner4	3/6	0/18
jam	2/3	0/4	syncarb	33/33	0/300
lock	9/9	3/75	tcp	1/2	0/2
packstart	1/4	0/2	ticTacToe	29/42	0/107
parsepack	4/4	0/15	tlc	4/5	0/8
parsesys	4/4	0/8	4-arbit	3/3	0/6
matrix	30/32	0/45	8-arbit	2/3	0/4
matrix_bug	14/32	0/25	vending	12/15	53/132
minMax	4/5	32/200	sbc	3/3	0/9

7 Extensions to the Basic Algorithm

In this section we discuss possible extensions to our algorithm for vacuity detection. One of the main advantages of a through analysis of the formulae is the ability to identify weak properties. It is possible to extend this idea to include more cases than those created by replacements with $\bot$. Consider the following example. Once we know that $\psi = \mathsf{AG}(p \to \mathsf{AF}\, q)$, passes in a certain model, we already know that $\varphi = \mathsf{AG}(p \to \mathsf{EF}\, q)$ passes as well, because $\psi \to \varphi$. Also, the satisfying sets computed for φ are bounds for those of ψ. In our example, we have $[\![\neg\, \mathsf{EF}\, q]\!] \subseteq [\![\mathsf{EG}\, \neg q]\!]$. In general, if $\psi \to \varphi$, and φ is checked before ψ, we have two cases:

1. φ fails: then ψ also fails.
2. φ passes: we can use the satisfying sets computed for φ as bounds for those of ψ.

We can produce strengthened formulae by extending the replacement function to specify a change of existential quantifier, or temporal operator.

Failing formulae are excluded by definition from vacuity checking. However, if $\mathsf{AF}\, p$ fails in a model, it is useful to know whether $\mathsf{EF}\, p$ fails as well. (It signals a bigger discrepancy between expectations and actual behavior of the model.) The mechanism that strengthen passing formulae for vacuity detection can be employed also to weaken them and help debug them if they fail.

We have presented and implemented our algorithm for CTL model checking. Extension to LTL and CTL* is possible. We briefly discuss the case of LTL. For each replacement formula, a different Büchi automaton is generated. The automaton is for the negation of the witness formula, and hence its language is larger than the language of the automaton for original formula. The result of the language emptiness check is therefore an upper bound on the result for the original formula.

References

[1] I. Beer, S. Ben-David, C. Eisner, and Y. Rodeh. Efficient detection of vacuity in ACTL formulas. In O. Grumberg, editor, *Ninth Conference on Computer Aided Verification (CAV'97)*, pages 279–290. Springer-Verlag, Berlin, 1997. LNCS 1254. 485, 486, 487, 495

[2] R. Bloem, K. Ravi, and F. Somenzi. Efficient decision procedures for model checking of linear time logic properties. In N. Halbwachs and D. Peled, editors, *Eleventh Conference on Computer Aided Verification (CAV'99)*, pages 222–235. Springer-Verlag, Berlin, 1999. LNCS 1633. 494

[3] R. Bloem, K. Ravi, and F. Somenzi. Symbolic guided search for CTL model checking. In *Proceedings of the Design Automation Conference*, pages 29–34, Los Angeles, CA, June 2000. 487

[4] R. K. Brayton et al. VIS: A system for verification and synthesis. In T. Henzinger and R. Alur, editors, *Eighth Conference on Computer Aided Verification (CAV'96)*, pages 428–432. Springer-Verlag, Rutgers University, 1996. LNCS 1102. 495

[5] R. E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986. 487

[6] H. Chockler, O. Kupferman, R. P. Kurshan, and M. Y. Vardi. A practical approach to coverage in model checking. In G. Berry, H. Comon, and A. Finkel, editors, *Thirteenth Conference on Computer Aided Verification (CAV'01)*, pages 66–78. Springer-Verlag, Berlin, July 2001. LNCS 2102. 485

[7] H. Chockler, O. Kupferman, and M. Y. Vardi. Coverage metrics for temporal logic model checking. In *Tools and algorithms for the construction and analysis of systems (TACAS)*, pages 528–542. Springer, 2001. LNCS 2031. 485

[8] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Proceedings Workshop on Logics of Programs*, pages 52–71, Berlin, 1981. Springer-Verlag. LNCS 131. 489

[9] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, Cambridge, MA, 1999. 485, 486

[10] O. Coudert, C. Berthet, and J. C. Madre. Verification of sequential machines using Boolean functional vectors. In L. Claesen, editor, *Proceedings IFIP International Workshop on Applied Formal Methods for Correct VLSI Design*, pages 111–128, Leuven, Belgium, November 1989. 488

[11] E. W. Dijkstra. Cooperating sequential processes. In Genuys, editor, *Programming Languages*, pages 43–112. Academic Press, 1968. 488

[12] H. Fujiwara. *Logic Testing and Design for Testability*. MIT Press, Cambridge, MA, 1985. 485

[13] Y. Hoskote, T. Kam, P.-H. Ho, and X. Zhao. Coverage estimation for symbolic model checking. In *Proceedings of the Design Automation Conference*, pages 300–305, New Orleans, LA, June 1999. 485

[14] S. Katz, O. Grumberg, and D. Geist. "Have I written enough properties?" — A method of comparison between specification and implementation. In *Correct Hardware Design and Verification Methods (CHARME'99)*, pages 280–297, Berlin, September 1999. Springer-Verlag. LNCS 1703. 485

[15] O. Kupferman and M. Y. Vardi. Vacuity detection in temporal model checking. In *Correct Hardware Design and Verification Methods (CHARME'99)*, pages 82–96, Berlin, September 1999. Springer-Verlag. LNCS 1703. 485, 486, 490, 493

[16] R. P. Kurshan. *Computer-Aided Verification of Coordinating Processes*. Princeton University Press, Princeton, NJ, 1994. 485

[17] K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, Boston, MA, 1994. 490

[18] K. Ravi and F. Somenzi. Hints to accelerate symbolic traversal. In *Correct Hardware Design and Verification Methods (CHARME'99)*, pages 250–264, Berlin, September 1999. Springer-Verlag. LNCS 1703. 487

[19] D. Verkest, L. Claesen, and H. De Man. Special benchmark session on formal system design. In L. Claesen, editor, *Proceedings IFIP International Workshop on Applied Formal Methods for Correct VLSI Design*, pages 75–76, Leuven, Belgium, November 1989. 488

[20] URL: http://vlsi.colorado.edu/~vis. 487