Efficient Detection of Vacuity in ACTL Formulas

Ilan Beer, Shoham Ben-David, Cindy Eisner and Yoav Rodeh

IBM Science and Technology
Haifa Research Laboratory, Matam, Haifa, Israel
shoham@vnet.ibm.com

Abstract. Propositional logic formulas containing implications can suffer from antecedent failure, in which the formula is true trivially because the pre-condition of the implication is not satisfiable. In other words, the post-condition of the implication does not affect the truth value of the formula. We call this a vacuous pass, and extend the definition of vacuity to cover other kinds of trivial passes in temporal logic. We define w-ACTL, a subset of CTL and show by construction that for every w-ACTL formula φ there is a formula $w(\varphi)$, such that: both φ and $w(\varphi)$ are true in some model M iff φ passes vacuously. A useful side-effect of $w(\varphi)$ is that if false, any counter-example is also a non-trivial witness of the original formula φ.

1 Introduction

Beatty and Bryant [BB94] have noted that antecedent failure is a problem in any application of formal verification, as it is an inherent problem in the use of logic, rather than of a particular approach (model checking or theorem proving). Antecedent failure means that a formula is trivially true because the pre-condition (antecedent) of the formula is not satisfiable in the model. We call this a vacuous pass, and extend the definition of vacuity to cover other kinds of trivially true formulas. If vacuity is not indicated to the user, the usefulness of formal verification is compromised, since a trivially true formula is not intentionally part of a specification (and therefore indicates a problem in the design or an error in the specification).

Several years of experience in practical formal verification of hardware at IBM [BB+96] have shown us that vacuity is a serious problem in the day-to-day use of formal verification. While it is possible to check vacuity using hand-written auxiliary formulas, the process is time-consuming and error prone, especially for long formulas containing many nested levels of pre-conditions.

In this paper we extend the notion of vacuity to cover many kinds of trivial passes in temporal logic. We then define a subset of ACTL [GL91] formulas, w-ACTL, for which it is possible to construct a single formula, $w(\varphi)$, which detects all vacuous passes of φ. In addition, $w(\varphi)$ has the useful side-effect that if no vacuity is detected, the model-checker produces a non-trivial witness to the original formula φ. We have implemented automatic generation of witness formulas as a feature of RuleBase [BB+96], the formal verification tool developed at the IBM Haifa Research Laboratory.

Detecting trivial passes of even relatively straightforward formulas is not in itself a trivial task. We use a typical Sugar[1] formula as an example:

[1] Sugar is a syntactic sugaring of CTL [CE81] formulas, and is the specification language used by the RuleBase formal verification tool. In [BB+96] we outlined its basic features.

$$AG(request \rightarrow next\_event(data)[4](last\_data)) \quad (1)$$

Formula 1 states that last_data should be asserted with the fourth data after a request. The translation into CTL is:

$$AG(request \rightarrow !E[!dataU(data \wedge EXE[!dataU(data \wedge EXE[!dataU(data \wedge EXE[!dataU(data \wedge !last\_data)])])])]) \quad (2)$$

A trivial pass of Formula 2 would be a pass in a model in which either request never occurs, or a request is never followed by four datas. A hand-written check of a trivial pass of Formula 2, which verifies that it is possible to receive a request followed by four datas, might look like the following:

$$EF(request \wedge (EF(data \wedge EXEF(data \wedge EXEFdata \wedge (EXEFdata))))) \quad (3)$$

Our goal was to automate the checking of trivial passes so as to free the user from coding Formula 3 or its equivalent.

A nice side-effect of our method is that the same witness formula w(φ) which detects trivial passes of the original formula when true, will provide a non-trivial witness (one instance of the truth of the formula) to φ when false. Witnesses are important because a formula may pass as the result of an error in the formula, rather than as a result of the design conforming to the formula that was intended by the user. Thus, examining a witness provides some confidence that the formal specification accurately reflects the intent of the user, one of the weak links in the practical application of formal verification to hardware design.

Note that simply negating the original formula will not provide a non-trivial witness. For instance, consider the CTL formula:

$$AG(p \rightarrow AX(q \rightarrow AXr)) \quad (4)$$

If we negate Formula 4, we get:

$$EF(p \wedge EX(q \wedge EX\neg r)) \quad (5)$$

Obviously, since Formula 5 is the negation of Formula 4, Formula 5 is false if Formula 4 is true. However, because Formula 5 is an existential formula, there is no trace which can show it is false, and the counter-example mechanisms of [HBK93] and of SMV [McM93, CG+95] will not generate a trace. We would like, however, to see a witness of Formula 4 which contains a sequence of states on which p occurs, followed by q in the next state, followed by r in the next.

Negating the single operand of the AG operator in Formula 4 as follows:

$$AG\neg(p \rightarrow AX(q \rightarrow AXr)) \quad (6)$$

will also not guarantee an interesting witness. For instance, a valid counter-example to Formula 6 is a path to a state in which p does not occur. Once again, this is a trivial positive example of the truth of the original Formula 4.

Both the ability to detect trivial passes and the ability to generate interesting witnesses are of great importance in the practical application of formal verification to hardware design. Our experience has shown that typically 20% of formulas pass vacuously during the first formal verification runs of a new hardware design, and that vacuous passes always point to a real problem in either the design or its specification or environment. Of the formulas which pass non-vacuously, examination of the witness traces discovers a problem for approximately 10% of the formulas. Of course, once the formula itself has been debugged by examination of a witness, there is no need to examine the witness on later runs. Thus, the model checking stays a fully automated process in the sense that a non-vacuous pass after a formula has been debugged requires no hand check by the user.

The remainder of this paper is organized as follows. In Section 2 we compare our work with related work. In Section 3 we give some background. Section 4 is the heart of the paper, and describes the theory and results. In Section 5 we conclude, and point to future directions for research.

2 Comparison with Related Work

Previous works, including [BB94] and [PP95], have noted the problem of trivial passes, and shown how to avoid them using hand-written checks. This work is, we believe, the first attempt to automatically detect trivial passes under symbolic model checking.

In this paper, we use the term interesting witness to mean a computation path showing one non-trivial example of the truth of the formula. In [HBK93], Hojati, Brayton and Kurshan describe counter-example generation for model checking using CTL and language containment using L-automata [Kur90]. They do not use the term witness, and do not produce a counter-example for a CTL formula containing an existential operator. In [CG+95], Clarke, Grumberg, McMillan and Zhao describe the counter-example and witness generation algorithm of SMV [McM93]. In their terminology, a witness is a computation path that shows that a formula with an existential path quantifier is true. For true formulas not containing an existential operator, no trace at all is generated.

Thus, neither [HBK93] nor [CG+95] produce witnesses for ACTL formulas. To the best of our knowledge, this work is the first to address the problem of generating a witness, in the sense of a positive non-trivial example for non-existential formulas, under symbolic model checking.

3 Preliminaries

CTL, or Computation Tree Logic [CE81], is a temporal logic useful for reasoning about the ongoing behavior of reactive systems, and is the logic used by the symbolic model checker SMV [McM93]. In CTL, temporal operators occur in pairs consisting of A or E, followed by F, G, U, or X, as follows:

1. Every atomic proposition is a CTL formula, and
2. If f and g are CTL formulas, then so are $\neg f, (f \wedge g), AXf, EXf, A[fUg], E[fUg]$

The remaining operators are viewed as abbreviations of the above, as follows: $f \vee g = \neg(\neg f \wedge \neg g), AFg = A[trueUg], EFg = E[trueUg], AGf = \neg E[trueU\neg f]$ and $EGf = \neg A[trueU\neg f]$.

The semantics of a CTL formula is defined with respect to a model M. A model is a quadruple (S, S_0, R, L), where S is a finite set of states, $S_0 \subseteq S$ is a set of initial states, $R \subseteq S \times S$ is the transition relation, and L is the valuation, a function mapping each state with a set of atomic propositions true in that state. We require that there is at least one transition from every state. A computation path of a model M is an infinite sequence of states $(s_0, s_1, s_2, \cdots)$ such that $R(s_i, s_{i+1})$ is true for every i.

The notation $M, s \models f$ means that the formula f is true in state s of model M. The notation $M \models f$ is equivalent to $\forall t \in S_0\ M, t \models f$. The semantics of a CTL formula is defined as follows:

$M, s \models p \iff p \in L(s)$, where p is an atomic proposition
$M, s \models \neg f \iff M, s \not\models f$
$M, s \models f \wedge g \iff M, s \models f$ and $M, s \models g$
$M, s_i \models AX\ f \iff$ for all paths $(s_i, s_{i+1}, ...), M, s_{i+1} \models f$
$M, s_i \models EX\ f \iff$ for some path $(s_i, s_{i+1}, ...), M, s_{i+1} \models f$
$M, s_i \models A[fUg] \iff$ for all paths $(s_i, s_{i+1}, ...), \exists k \geq i$ such that $M, s_k \models g$, and $\forall j$ such that $i \leq j < k, M, s_j \models f$
$M, s_i \models E[fUg] \iff$ for some path $(s_i, s_{i+1}, ...), \exists k \geq i$ such that $M, s_k \models g$ and $\forall j$ such that $i \leq j < k, M, s_j \models f$

ACTL is a subset of CTL defined by Grumberg and Long in [GL91], and can be informally described as CTL without the "E" operators, in which the $\neg$ operator modifies only atomic propositions. ACTL includes an additional operator, "AV", where A[p V q] $\equiv \neg$ E[$\neg$ p U $\neg$ q]. A[p V q] can intuitively be understood as "p releases q", in the sense that q must hold up to and including the time that p holds, at which point q is "released". If p never occurs, then q must hold forever. Thus, the "AV" operator is a weak operator, in contrast to the "AU" operator, which is strong: A[p U q] requires that q eventually occur. Notice that because some of the "A" operators can be defined in terms of the "E" operators, and vice versa, a CTL formula containing the "E" operator may still be an ACTL formula if the "E" operator is negated. The formal description can be found in [GL91].

4 Detection of vacuity in w-ACTL formulas

In this section, we describe the detection of vacuity in w-ACTL formulas. First, we define vacuity, w-ACTL and interesting witnesses. Then we define witness formulas to be formulas which detect vacuity and provide interesting witnesses. We then give the main result of this paper, an algorithm for constructing witness formulas for w-ACTL formulas, and prove that it is correct. Finally, we show some examples.

4.1 Vacuity

In this section, we will define vacuity, first intuitively and then formally.

Propositional antecedent failure means that a formula trivially passes because some pre-condition is not satisfiable, where a pre-condition is the left-hand-side of an implication. Another way to think of the same thing is to say that the right-hand-side of the implication does not affect the validity of the formula. This gives an intuitive extension of vacuity to any operator: vacuity occurs when one of the operands does not affect the validity of the formula. We first define what we mean by a sub-formula not affecting the truth value of the formula, then define vacuous passes.

Definition 1 (Does Not Affect). *A sub-formula χ of formula φ does not affect the truth value of φ in model M if for every formula χ', the truth value of φ' in model M is the same as the truth value of φ in model M, where φ' is the formula obtained by replacing χ with χ' in φ.*

Definition 2 (Vacuous Passes). *Formula φ passes vacuously in model M if it passes, and contains a sub-formula χ such that χ does not affect the truth value of φ in M.*

As an example, consider the following formula:

$$AG(p \rightarrow AX(q \rightarrow AXr)) \tag{7}$$

Some trivial passes of Formula 7 are passes in which either p never occurs, and thus $AX(q \rightarrow AXr)$ does not affect the validity of Formula 7, or q never occurs at a next state of p, and thus AXr does not affect the validity of Formula 7. Thus, the idea of vacuity includes a notion of when q should occur, and not just that it should occur. For instance, if $M \models$ EF q, but also $M \models$ AG (p $\rightarrow$ AX $\neg$ q), a pass of Formula 7 is still trivial.

4.2 w-ACTL

We now define w-ACTL, a subset of ACTL which, in our experience, is sufficient for expressing most of the formulas used by engineers to specify their designs. In addition, we will show that we can efficiently detect vacuity of w-ACTL formulas using CTL model checking. Informally, w-ACTL formulas are ACTL formulas in which for all binary operators ($\wedge$, $\vee$, AU, AV), at least one of the operands is a propositional formula. Formally, w-ACTL is the set of state formulas described by the following:

Definition 3 (w-ACTL).

1. *If p is an atomic proposition, then p and $\neg p$ are simple formulas.*
2. *If f and g are simple formulas, then $f \wedge g$ and $f \vee g$ are simple formulas.*
3. *If ψ is a simple formula, it is a state formula.*
4. *If ψ is a simple formula, and χ is a state formula, then $\psi \wedge \chi, \chi \wedge \psi, \psi \vee \chi, \chi \vee \psi$, are state formulas.*
5. *If ψ is a simple formula, and χ is a state formula, then $AF\ \chi$, $AG\ \chi$, $A[\chi\ U \psi]$, $A[\psi U \chi]$, $A[\chi\ V \psi]$, $A[\psi V \chi]$, and $AX\ \chi$ are state formulas.*

Note that simple formulas are conjunctions and disjunctions of atomic propositions and their negations. In the sequel, we will usually use φ to designate some w-ACTL formula, ψ to designate a simple w-ACTL formula, and χ to designate a possibly non-simple w-ACTL formula.

We call our method efficient because it can detect vacuity of many sub-formulas simultaneously. However, our algorithm requires us to choose one operand of every binary operator, for which vacuity will be detected. We call this operand the important operand, and choose it as follows.

Definition 4 (Important Operand).

1. *If φ is a simple formula, its operands are not important*[2].
2. *If φ is a non-simple formula of the form $\psi \vee \chi$, $\chi \vee \psi$, $\psi \wedge \chi$ or $\chi \wedge \psi$, where ψ is simple and χ is non-simple, then χ is the important operand.*
3. *If φ is a formula of the form $A[\psi\ U\ \chi]$, $A[\chi\ U\ \psi]$, $A[\psi\ V\ \chi]$ or $A[\chi\ V\ \psi]$, where ψ is simple and χ is non-simple, then χ is the important operand.*
4. *If φ is a formula of the form $A[\psi_1\ U\ \psi_2]$ or $A[\psi_1\ V\ \psi_2]$, where both ψ_1 and ψ_2 are simple, then ψ_1 is the important operand*[3].
5. *If φ is a non-simple formula of the form $AX\ \chi$, $AF\ \chi$ or $AG\ \chi$, where χ is either simple or non-simple, then χ is an important operand.*

The following lemma follows directly from Definition 4, because only one operand of every binary operator can be important:

Lemma 5. *For every w-ACTL formula φ, there is a smallest important sub-formula s_φ, such that s_φ contains no important sub-formulas, and s_φ is a sub-formula of every other important sub-formula of φ.*

We justify choice of the non-simple operand of $\vee$ and $\wedge$ as the important operand as follows. The choice is simply a reflection of how engineers tend to use CTL to code a specification, as well as how they tend to design their hardware. For instance, consider the following specification:

$$AG(request \rightarrow AX(req\_accepted \rightarrow AXAX(read\_busy \vee write\_busy))) \quad (8)$$

[2] Actually, if the $\vee$ operator is derived from the use of the $\rightarrow$ operator by the user, we consider the right-hand-side of the original formula to be important even if it is simple. Similarly, we consider the second operand of the next_event operator [BB+96] to be important if both are simple. Since these are implementation details, we ignore them in the rest of this paper.

[3] This is counter-intuitive. The reason that only ψ_1 is important is that ψ_2 is the only operand that can cause vacuity. For $A[\psi_1\ U\ \psi_2]$, ψ_2 can cause vacuity of ψ_1 if it is always true immediately. However, ψ_1 cannot cause vacuity of ψ_2 because even if ψ_1 is always true forever, the AU operator still requires something of ψ_2: that eventually it occurs. For the AV operator, ψ_2 can cause vacuity of ψ_1 if it is always true forever, because then nothing is required of ψ_1. However, ψ_1 cannot cause vacuity of ψ_2 if it is always true immediately, because in that case, the AV operator still requires something of ψ_2: that it occurs at the same time.

which expresses the requirement that if a request is accepted (which happens or not one cycle after it appears), then two cycles later either the read_busy signal is asserted, or the write_busy signal is asserted. Logically, this is equivalent to the formula:

$$AG(\neg request \vee AX(\neg req\_accepted \vee AXAX(read\_busy \vee write\_busy))) \quad (9)$$

A trivial pass of 8, in which it is detected that $M \models AG(\neg request)$ would probably detect a problem in the model, because otherwise the signal called request is meaningless. However, a trivial pass in which it is detected that $M \models AG(AX(\neg req\_accepted \vee AXAX(read\_busy \vee write\_busy)))$ is quite often useless to the engineer, as it is highly likely that she has designed her logic intentionally for this to be so, and prevents read_busy or write_busy from being asserted spuriously by not asserting req_accepted if there was not a request the previous cycle.

Thus, for the binary operators, we have chosen the non-simple operand to be the important operand. We now define important vacuous passes as follows:

Definition 6 (Important Vacuous Passes). *If formula φ passes in model M, and contains an important sub-formula χ such that χ does not affect the truth value of φ in M, we say that the vacuous pass is an important vacuous pass.*

In the remainder of this paper, we will use the term "passes vacuously" to refer to important vacuous passes as defined above.

4.3 Witnesses for w-ACTL formulas

In the previous section we defined vacuity, the main motivation of this paper. In this section, we define interesting witnesses, which is the second motivation. Informally, an interesting witness is a path showing one instance of the truth of the formula, on which every important sub-formula affects the truth of the formula. In the formal definition of an interesting witness we make use of the fact that every computation path can be viewed as a model.

Definition 7 (Interesting Witness). *An interesting witness of a passing formula φ in model M is a computation path C in M such that $C \models \varphi$ non-vacuously.*

In the following, we define a witness formula, and show how to construct one for any given w-ACTL formula. Because our generation of a witness makes use of the counter-example mechanism of SMV [CG+95], we first define a counter-example as follows.

Definition 8 (Counter-example). *A counter-example of a failing formula φ in model M is a computation path C in M such that $C \not\models \varphi$.*

It should be noted that according to Definition 8, there are w-ACTL formulas for which no counter-example exists in some models. For example, there is no counter-example for the formula $\varphi = AFAGp$ in a model $M = (S, S_0, R, L), S = \{s_0, s_1, s_2\}$, $S_0 = s_0, R = \{(s_0, s_0), (s_0, s_1), (s_1, s_2), (s_2, s_2)\}, L(s_0) = L(s_2) = \{p\}, L(s_1) = \{\}$.

This is because despite the fact that $M \not\models \varphi$, $C \models \varphi$ for any computation path C in model M. Despite this fact, Definition 8 captures the essence of what we mean by a counter-example, in a succinct and intuitive manner, for the vast majority of w-ACTL formulas and models encountered in the day-to-day verification of hardware.

We now define a witness formula, which is the main definition of this paper. A witness formula is a formula which performs a dual function: it both detects vacuity, and, if not vacuous, induces a positive example to the original formula. Formally,

Definition 9 (Witness Formula). *A formula w is a witness formula of formula φ, denoted by w(φ), if, for any model M,*

1. *($M \models \varphi$ and $M \models w(\varphi)$) $\Longleftrightarrow$ φ passes vacuously in M.*
2. *If $M \models \varphi$ and $M \not\models w(\varphi)$ then any counter-example of $w(\varphi)$ in M is also an interesting witness of φ in M.*

4.4 Construction of Witness Formulas

The main result of this paper is now presented. We show construction of a witness formula w(φ) for any w-ACTL formula φ, and then prove that w(φ) is indeed a witness formula of φ, according to Definition 9.

Algorithm 10 (Construction of Witness Formulas).

1. *If φ is a simple important formula, w(φ) = FALSE.*
2. *If φ is non-simple, and has the form $\psi \wedge \chi$ or $\chi \wedge \psi$, where χ is important, w(φ) = $\psi \wedge w(\chi)$.*
3. *If φ is non-simple, and has the form $\psi \vee \chi$ or $\chi \vee \psi$, where χ is important, w(φ) = $\psi \vee w(\chi)$.*
4. *If φ has the form AF χ, w(φ) = AF w(χ).*
5. *If φ has the form AG χ, w(φ) = AG w(χ).*
6. *If φ has the form AX χ, w(φ) = AX w(χ)*
7. *If φ has the form A[χ U ψ], where χ is important, w(φ) = A[w(χ) U ψ][4].*
8. *If φ has the form A[ψ U χ], where χ is important, w(φ) = A[ψ U w(χ)].*
9. *If φ has the form A[χ V ψ], where χ is important, w(φ) = A[w(χ)Vψ].*
10. *If φ has the form A[ψ V χ], where χ is important, w(φ) = A[ψ V w(χ)].*

[4] Actually, we produce A[w$\prime$(χ) U ψ], where w$\prime$(χ) is similar to w(χ), except that we replace a simple important formula with AF FALSE rather than FALSE. The reason for this is practical rather than theoretical. In theory, a computation path is infinite and therefore, every witness is infinite. In practice, however, the algorithm of [CG+95] will sometimes give finite counter-examples, when a finite counter-example is enough to show that the formula is false. In every case but one, the finite counter-example given by [CG+95] is "interesting enough" for our purposes. The exception is the AU operator. As a witness to A[χ U ψ], we would like to see a trace on which ψ occurs, but [CG+95] may give us a counter-example to A[w(χ) U ψ] which ends before ψ has occurred. Therefore, we use AF FALSE to get an infinite counter-example, just as [CG+95] uses EG TRUE to get an infinite witness. Since this is an implementation detail, we ignore it in the rest of this paper.

Note that the witness construction algorithm replaces the smallest important sub-formula by FALSE.

In order to prove that Algorithm 10 produces a witness formula, the following two lemmas are needed.

Lemma 11. *Let φ be an ACTL formula, M be a model and C a computation path in M. If $M \models \varphi$ then $C \models \varphi$.*

The proof follows directly from [Lon93].

Lemma 12. *For every ACTL formula φ, and every χ, a sub-formula of φ which is not an operand of $\neg$: let χ' be any formula, and φ' be the formula obtained by replacing χ with χ' in φ. Then, for any model M, $M \models (\chi' \rightarrow \chi) \Longrightarrow M \models (\varphi' \rightarrow \varphi)$.*

The proof is by induction on the length of φ.

We are now ready to prove the correctness of our algorithm:

Theorem 13. *If φ is a w-ACTL formula, then w(φ) given by Algorithm 10 is a witness formula of φ.*

Proof. Let sub$(\chi, \chi')\varphi$ be the formula obtained by replacing χ with χ' in φ. Let s_φ be the smallest important sub-formula of φ. It is easy to see that by Algorithm 10,

$$w(\varphi) = sub(s_\varphi, FALSE)\varphi.$$

Furthermore, by Lemma 5, s_φ is a sub-formula of every important sub-formula χ of φ, so for every such χ,

$$w(\varphi) = sub(\chi, sub(s_\varphi, FALSE)\chi)\varphi.$$

In order for $w(\varphi)$ to be a witness formula we should prove that the two conditions of Definition 9 hold:

1. First we prove that ($M \models \varphi$ and $M \models w(\varphi)$) $\Longleftrightarrow$ φ passes vacuously.
   ($\Longleftarrow$)
   Let φ pass vacuously in M. By Definition 6, $M \models \varphi$, and $\exists\chi$, an important sub-formula of φ, such that $\forall\chi'$, $M \models sub(\chi, \chi')\varphi$. For this χ, let χ' be sub$(s_\varphi, FALSE)\chi$. Then by the definition of vacuity, $M \models sub(\chi, \chi')\varphi$, but as shown in the beginning of this proof, $w(\varphi) = sub(\chi, \chi')\varphi$. Thus $M \models w(\varphi)$.
   ($\Longrightarrow$)
   Let $M \models \varphi$ and $M \models w(\varphi)$. We must show that φ passes vacuously in M, that is, we must show that $\exists\chi$ important sub-formula of φ, such that $\forall\chi'$, $M \models sub(\chi, \chi')\varphi$. We choose χ to be s_φ.
   By Lemma 12, $\forall M$, $M \models (FALSE \rightarrow \chi') \Longrightarrow M \models (sub(s_\varphi, FALSE)\varphi \rightarrow sub(s_\varphi, \chi')\varphi)$.
   Since $\forall\chi', \forall M$, $M \models$ (FALSE $\rightarrow \chi'$), it follows that

   $$\forall\chi'\forall M,\ M \models (sub(s_\varphi, FALSE)\varphi \rightarrow sub(s_\varphi, \chi')\varphi)$$

   As shown in the beginning of this proof, $w(\varphi) = sub(s_\varphi, FALSE)\varphi$. Since we are given that $M \models w(\varphi)$, it follows that $\forall\chi'$, $M \models sub(s_\varphi, \chi')\varphi$. Thus φ passes vacuously in M.

2. Now, we prove that if $M \models \varphi$ and $M \not\models w(\varphi)$ then any counter-example of $w(\varphi)$ in M is also an interesting witness of φ in M.
By Definition 8, any counter-example produced for an ACTL formula, is a computation path in M. Let $CE_{w(\varphi)}$ be a computation path which is a counter-example for $w(\varphi)$ in M. By Lemma 11, $CE_{w(\varphi)} \models \varphi$. We have to show that $CE_{w(\varphi)} \models \varphi$ non-vacuously, that is, for every χ important sub-formula of φ, χ affects the truth value of φ in $CE_{w(\varphi)}$.
As shown at the beginning of this proof, for every χ important sub-formula of φ

$$w(\varphi) = sub(\chi, sub(s_\varphi, FALSE)\chi)\varphi.$$

Since $CE_{w(\varphi)} \models \varphi$ and $CE_{w(\varphi)} \not\models w(\varphi)$, we have found for any important sub-formula χ of φ, a formula $\chi' = sub(s_\varphi, FALSE)\chi$ which affects the value of φ in $CE_{w(\varphi)}$. Thus φ passes non-vacuously in $CE_{w(\varphi)}$. □

4.5 Examples

We now show the generation of the witness formula for Formula 1 from Section 1. We first convert Formula 2 (the CTL equivalent of Formula 1) into normal form:

$$AG(\neg request \vee A[dataV(\neg data \vee AXA[dataV(\neg data \vee AXA[dataV(\neg data \vee AXA[dataV(\neg data \vee last\_data)])])])]) \quad (10)$$

Since last_data is considered to be non-simple (because it is the second operand of a next_event operator, see Footnote 2) the witness formula is:

$$AG(\neg request \vee A[dataV(\neg data \vee AXA[dataV(\neg data \vee AXA[dataV(\neg data \vee AXA[dataV(\neg data \vee FALSE)])])])]) \quad (11)$$

It is easy to see that Formula 11 will pass iff either a request never occurs, or no request is ever followed by four datas. Also, it is clear that if Formula 11 fails, the counter-example will be an interesting witness of Formula 1, on which a request followed by four datas will occur.

Now examine the following formula, which expresses the fact that we require q to occur an infinite number of times:

$$AG\,AF\,q \quad (12)$$

The witness formula for Formula 12 is:

$$AG\,AF\,FALSE \quad (13)$$

If Formula 12 passes, it cannot pass vacuously unless there are no fair paths, and indeed Formula 13 will fail in all models unless there are no fair paths. If Formula 12 passes, the counter-example to Formula 13 will be a computation path, on which q will appear infinitely many times (because Formula 12 passed).

4.6 Discussion

If a vacuous pass is detected by a witness formula, there is no indication of which of the pre-conditions caused the vacuous pass. This can easily be determined by multiple formulas, each of which checks one pre-condition. It should be noted that these multiple formulas need be run only if both the original formula and the witness formula passed. The RuleBase [BB+96] formal verification tool makes this decision automatically.

It should be noted that determination of vacuity is intrinsically entwined with the model checking algorithm: because vacuity is dependent on the context of a pre-condition, it is natural to discover it by model checking another formula with identical pre-conditions. An attempt to discover vacuity by directly manipulating BDDs would end up mimicking many of the model checking steps. Thus, while the vacuity of AG (p $\rightarrow$ q) can be found by simply intersecting the BDD of the states for which $M, s \models p$ with the reachable states, the direct detection of the vacuity of AG (p $\rightarrow$ AX (q $\rightarrow$ AX r)) would need as well to intersect the BDD of the states in which $M, s \models q$ with the states which are reachable in one step from the states for which $M, s \models p$.

5 Conclusions and future directions

We have shown a method for efficient detection of vacuity in w-ACTL formulas, a subset of ACTL [GL91] formulas. In addition, we have shown that our algorithm has the capability, as a side-effect, of providing an interesting witness, one positive non-trivial example of the truth of the formula. As discussed above, the ability to detect vacuity and provide an interesting witness are extremely important in the practical application of model checking to industrial hardware designs.

Although w-ACTL formulas define, in our experience, almost all of the CTL formulas used by engineers to specify their designs, there are useful formulas not included in w-ACTL. Therefore, we would like to have a general method for generating witness formulas for any CTL formula, in particular formulas containing a combination of existential and universal operators.

Acknowledgements

We thank Danny Geist and Shmuel Ur for important remarks on early drafts of this paper. We thank an anonymous referee for very important observations, which improved the quality of the final version of this paper.

References

[BB94] D. Beatty and R. Bryant, "Formally verifying a microprocessor using a simulation methodology", Design Automation Conference '94, pp. 596-602.

[BB+96] I. Beer, S. Ben-David, C. Eisner, A. Landver, "RuleBase: an Industry-Oriented Formal Verification Tool", in Proc. 33^{rd} Design Automation Conference 1996, pp. 655-660.

[CE81] E.M. Clarke and E.A. Emerson, "Design and synthesis of synchronization skeletons using Branching Time Temporal Logic", in Proc. Workshop on Logics of Programs, Lecture Notes in Computer Science, Vol. 131 (Springer, Berlin, 1981) pp. 52-71.

[CE81b] E.M. Clark and E.A. Emerson, "Characterizing Properties of Parallel Programs as Fixed-point", in Seventh International Colloquium on Automata, Languages, and Programming, Volume 85 of LNCS, 1981.

[CG+95] E. Clarke, O. Grumberg, K. McMillan, X. Zhao, "Efficient Generation of Counterexamples and Witnesses in Symbolic Model Checking", Design Automation Conference 1995, pp. 427-432.

[GL91] O. Grumberg and D. Long, "Model checking and modular verification." In J.C.M. Baeten and J.F. Groote, editors, Proccedings of CONCUR '91: 2nd International Conference on Concurrency Theory, Volume 527 of LNCS, 1991.

[HBK93] R. Hojati, R.K. Brayton and R.P. Kurshan, "BDD-based debugging of designs using language containment and fair CTL." CAV '93, pp. 41-58.

[Kur90] R. Kurshan, "Analysis of Discrete Event Coordination," LNCS 1990.

[Lon93] D. Long, "Model Checking, Abstraction and Compositional Verification", Ph.D. Thesis, CMU, 1993.

[McM93] K.L. McMillan, "Symbolic Model Checking", Kluwer Academic Publishers, 1993.

[PP95] B. Plessier and C. Pixley, "Formal Verification of a Commercial Serial Bus Interface", International Phoenix Conference on Computers and Communications, 1995, pp. 378-382.

[SG90] G. Shurek, O. Grumberg, "The Computer-Aided Modular Framework - Motivation, Solutions and Evaluation Criteria", Workshop on Computer Aided Verification, 1990.