Finding Environment Guarantees

Marsha Chechik, Mihaela Gheorghiu, Arie Gurfinkel

University of Toronto, Toronto, ON M5S 3G4, Canada
{chechik,mg,arie}@cs.toronto.edu

Abstract. When model checking a software component, a model of the environment in which that component is supposed to run is constructed. One of the major threats to the validity of this kind of analysis is the correctness of the environment model. In this paper, we identify and formalize a problem related to environment models — *environment guarantees*. It captures those cases where the correctness of the component under analysis is due solely to the model of its environment. Environment guarantees provides a model-based analog to a property-based notion of *vacuity* by identifying cases when the component is irrelevant to satisfaction of a property. The paper also presents a model checking technique for the detection of environment guarantees. We show the effectiveness of our technique by applying it to a previously published study of TCAS II, where it finds a number of environment guarantees.

1 Introduction

As software is controlling more and more critical aspects of our lives, its reliability is ever more important. Formal verification can help increase confidence in the software systems being built. Among the verification methods, model checking is gaining popularity due to its automated approach. In this approach, a model of the software component being analyzed is closed with a model of the environment in which the component is expected to run. Correctness properties of the component are then checked on the resulting model. One of the major threats to this kind of analysis is the correctness of the environment model. Creating a faithful model of the environment is error-prone, as often the environment consists of parts of the physical world whose behavior is only partially understood, or it is a complex system, e.g., an operating system, whose behavior is also hard to capture in a unified model. Moreover, the model of the environment is often simplified to enable effective model-checking, potentially leading to errors.

To illustrate the kinds of modeling errors we address, consider, for example, model checking a traffic light controller. In this system, cars arrive at an intersection, trip sensors, and wait for the green light. The controller, which is the component being analyzed, uses the sensors, that represent the environment, to maximize the flow of cars through the intersection. An essential property of the system is that if a sensor is ever tripped, an appropriate light eventually turns green. This property is formalized in CTL [10] (defined in Section 2) as $\varphi = AG\,(\mathsf{Sensor\_Tripped} \Rightarrow (AF\ \mathsf{Light} = \mathsf{green}))$. Suppose there is a bug in the environment model due to which Sensor_Tripped is always off. This property holds regardless of the correctness of the controller. Thus, although the desired property is satisfied, the component should not be deemed correct. Instead, we want the model checker to detect that the environment model may be wrong.

Industrial researchers noted that in practice properties with implication (such as φ) may hold for the wrong reasons, referring to the problem as "antecedent failure" [2].

IBM researchers generalized this notion to properties that are not necessarily implications, naming it *vacuity* [3]. The definition of vacuity is *property-based*: a formula α is vacuous in a subformula β in a given model if β does not influence the value of α in the model. That is, in the traffic light example above with the faulty environment model, the property φ is vacuous in Light=green: it is satisfied independently of the color of the light because the antecedent of the implication is false. [3] also defined a vacuity detection method for a restricted class of CTL formulas and noted that when found, vacuity always pointed to a problem in either the component, its environment, or in the property, which was observed for 20% of the properties checked. Other researchers [24, 1, 21] extended vacuity detection to general properties expressed in CTL and other common languages. All these approaches, however, remain property-based, and are not adequate to detect errors in the *model*. Vacuity information is not sufficient to decide when the environment model is faulty. Consider the property φ again — it is also vacuous in Light=green in a model where the activation of the sensors depends on a flag being set by the controller independently of the environment, and the controller never sets that flag. In this case, the vacuity is due to the *component*, and not to its *environment*, and is often not effective for finding problems with the model.

In contrast to the property-centric approach of vacuity detection, Shlyakhter *et al.* [30] devised a technique to debug models more directly. They identified the problem of "overconstraining" declarative models, and pointed out that overconstraining occurs most often in the definition of the models being checked rather than in the specification of their correctness properties. They have developed a technique for extracting and displaying the part of the model used for establishing satisfaction of a property. When most of the model was unnecessary to prove a property, the authors were able to conclude that was due to overconstraint, caused by subtle modeling errors. This technique, however, is restricted to declarative models, and does not exploit the view of the model that separates the component from its environment.

In our work, we also aim to provide a technique for model debugging, but in the case of operational models, such as those specified by state-machines, and we target the analysis toward debugging environment models. We consider a model to be "overconstrained" if a property that should hold of the software component in the given environment is guaranteed solely by the environment. In other words, the component can be replaced by another, *arbitrary*, component in the same environment, without affecting the satisfaction of the property. We say that such properties are *environment guarantees*. Environment guarantees always indicate a problem: either the desired property is not a property of the component, and should rather be reconsidered as a property of its environment, or there is an error in the model of the environment or in expressing the property. The naive approach to detect environment guarantees is to generate all possible components, compose each with the given environment, and check whether the property holds on all the composed models. This is clearly infeasible. Instead, we show how to model the environment as an open system and check properties on it directly, using a symbolic model-checking algorithm.

A similar approach, called *robust satisfaction* was proposed and studied in [23]. It is aiming to identify whether a property holds in *all* possible environments, and is the same as environment guarantees with the roles of the environment and the system

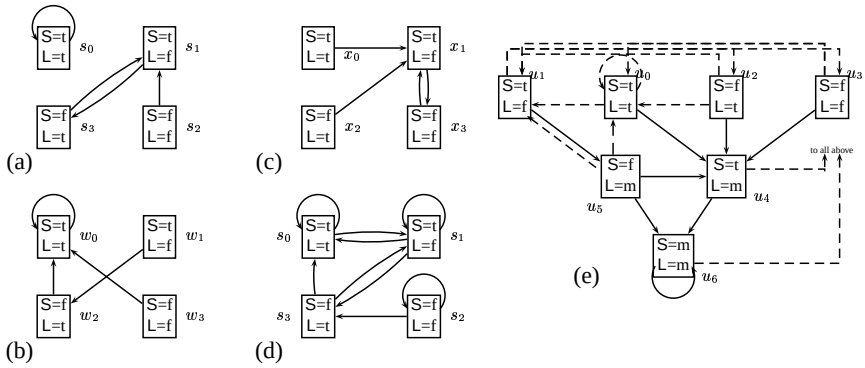

Fig. 1. An example Kripke structure (a). A four-valued Kripke structure of an open system (e), and its three completions (b), (c), and (d).

reversed. While the method in [23] is complete (it always finds errors when they are present), it is rather expensive (see Section 7 for a detailed discussion).

This paper makes the following contributions: (1) We argue that a way to discover faulty environment models is to detect cases where properties are guaranteed solely by the environment. Although this process does not find all possible environment modeling errors, the errors reported by this analysis *always* point to some error in understanding of the model-checking results. (2) We formalize the meaning of a property being guaranteed by the environment by modeling the environment as an open system. (3) We show how to model open systems and define a model-checking algorithm which can lead to a scalable technique for discovering environment guarantees. (4) We describe a simple implementation for checking environment guarantees for true universal properties. (5) We show that our technique finds real errors and is scalable for handling non-trivial systems by applying it to the well-known example of the Traffic Collision and Avoidance System (TCAS II) [26, 6]. In our case study, we found that several essential properties of TCAS II, including some analyzed by Chan *et al.* in [6], hold as the result of environment guarantees: the model of the environment used in verifying this system has been simplified too much.

The rest of this paper is organized as follows. After reviewing the relevant notation in Section 2, we formalize the meaning of a property being guaranteed by the environment in Section 3. In Section 4, we describe the modeling and reasoning about open systems. While this framework is general, we do not currently have an implementation for checking for environment guarantees for arbitrary temporal logic properties. In Section 5, we describe the algorithm we have implemented for checking true universal properties. We illustrate effectiveness of checking for environment guarantees by analyzing true universal properties of the TCAS II system in Section 6. We compare our approach with related work, specifically, with vacuity detection and module checking, in Section 7 and conclude the paper in Section 8.

2 Background

In this section, we review the model-checking process and fix the notation.

Definition 1. *A* model $\mathcal{M}$ *consists of a set* $V = \{v_1, \ldots, v_n\}$ *of* variables *and a set* Δ *of* rules *describing the temporal behavior of those variables. A* state *of the model* $\mathcal{M}$ *is a valuation of all variables in* V*. A* rule *is an expression over the model variables that relates their values in a state at time* t *(current) with those at time* $t + 1$ *(next).*

$\mathcal{K}_\mathcal{M}, s \models p$ iff $I(p, s)$ $\quad$ $\mathcal{K}_\mathcal{M}, s \models \neg\varphi$ iff $\mathcal{K}_\mathcal{M}, s \not\models \varphi$

$\mathcal{K}_\mathcal{M}, s \models \varphi \wedge \psi$ iff $\mathcal{K}_\mathcal{M}, s \models \varphi \ \wedge \ \mathcal{K}_\mathcal{M}, s \models \psi$ $\quad$ $\mathcal{K}_\mathcal{M}, s \models EX\varphi$ iff $\exists t \in S,\ (s,t) \in R \ \wedge \ \mathcal{K}_\mathcal{M}, t \models \varphi$

$\mathcal{K}_\mathcal{M}, s \models EG\varphi$ iff $\exists$ path $s = s_0, s_1, \ldots$ s.t. $\forall j \cdot \mathcal{K}_\mathcal{M}, s_j \models \varphi$

$\mathcal{K}_\mathcal{M}, s \models E[\varphi \ U \ \psi]$ iff $\exists$ path $s = s_0, s_1, \ldots,$ s.t. $\exists j \cdot \mathcal{K}_\mathcal{M}, s_j \models \psi \ \wedge \ \forall k \cdot k < j \Rightarrow \mathcal{K}_\mathcal{M}, s_k \models \varphi$

Fig. 2. Semantics of CTL.

Without loss of generality, we assume that the variables are boolean. Therefore, the set S of states consists of all n-tuples of boolean values. For instance, consider our previous example of a traffic light system. It can be modeled by a variable Light, which is true iff the light is green (Light=green), and a variable Sensor, which is true iff the sensor is tripped. For a variable v, we use v (unprimed) and v' (primed) to denote its current and next state value, respectively, and define $V' = \{v'_1, \ldots, v'_n\}$.

We assume that rules are described in the style of the SMV language [9]. Each rule is an assignment of the form $v'_i \leftarrow \delta$, where $v'_i \in V'$ and δ is a boolean formula over the variables in $V \cup V' \setminus \{v'_i\}$. We assume no circularity in the rules, *i.e.*, no variable depends on itself if we follow any chain of the rules, and there is at least one assignment for each variable. In our example, the rules may be Sensor$'$ $\leftarrow$ $\neg$Sensor $\vee$ Light and Light$'$ $\leftarrow$ Sensor, which indicate that the sensor is on in the next state if it is currently off, or the light is on, and the light is on in the next state if the sensor is currently on. It is convenient to think of each assignment as an equivalent boolean formula, i.e., $v' \leftarrow \delta$ iff $(v' \wedge \delta) \vee (\neg v' \wedge \neg\delta)$. Multiple assignments per variable are used to indicate that the variable changes non-deterministically, *e.g.*, when the model $\mathcal{M}$ contains rules $v' \leftarrow$ true and $v' \leftarrow$ false, it means that v' can be either true or false in the next state of $\mathcal{M}$.

Given a model $\mathcal{M}$, we associate with it a state-transition graph $\mathcal{K}_\mathcal{M}$, known as a *Kripke structure*: The *Kripke structure* $\mathcal{K}_\mathcal{M}$ is a tuple $\langle S, R, I\rangle$, where S is the set of states, $R \subseteq S \times S$ is the transition relation, and $I : V \times S \rightarrow \{\text{true}, \text{false}\}$ is the interpretation of variables. For each variable v_i and state s, $I(v_i, s)$, or $s(i)$ for short, is the value of v_i in s. Let s and s' denote valuations of all unprimed and primed variables, respectively. The relation R is the set of all pairs of states (s, s') such that s and s' satisfy at least one of the rules of Δ for every variable in V.

The Kripke structure for our example is shown in Figure 1(a). For brevity, we use S for Sensor and L for Light in the diagrams. For example, in state s_0, the right hand sides of the rules for both Sensor and Light are true. Thus, both Sensor and Light have to be true in the successor state, creating the self-loop at s_0.

Properties of a model $\mathcal{M}$ are formulated in a temporal logic CTL [10], the semantics of which is given in Figure 2 and evaluated in the states of the associated Kripke structure $\mathcal{K}_\mathcal{M}$. For example, $E[\varphi \ U \ \psi]$ is true in s if along some path from s, φ continuously holds until ψ becomes true. The following derived CTL formula is also commonly used: $EF \ \varphi \equiv E[\text{true} \ U \ \varphi]$. For instance, in the structure of Figure 1(a), the formula EF Light (the light eventually becomes green) is true in all states, whereas the formula AG (Sensor $\vee$ Light) (always the sensor is tripped or the light is green) is true in states s_0, s_1, s_3 and false in s_2. The subclass of CTL formulas containing only universal path quantifiers is called ACTL. Often, some state of a Kripke structure, say, s_0, is designated as *initial*. In this case, we say that a formula φ holds in a Kripke structure $\mathcal{K}_\mathcal{M}$ to mean that $\mathcal{K}_\mathcal{M}, s_0 \models \varphi$. A model $\mathcal{M}$ *satisfies* a CTL formula φ if $\mathcal{K}_\mathcal{M} \models \varphi$.

3 Environment Guarantees

In this section, we formalize the notion of environment guarantees. A model described in Section 2 is a composition of the software component being analyzed, called the

component from now on, with its environment. The boundary between them is often blurred during verification: they are simply specified using a collection of rules. In what follows, we make this boundary more explicit.

We assume that the set V of model variables is partitioned into a set C of *component variables* and a set E of *environment variables*. We further assume that this partition can be determined *syntactically*, (*e.g.,* by the names or types of the variables, or their location, etc.), or by the model documentation. Environment variables represent the inputs to the software, coming from the environment. Component variables represent the outputs from the software to the environment. The variable partition induces a partition on the rules of a model into *component rules* and *environment rules*.

Definition 2. *A* partitioned *model $\mathcal{M}$ is a tuple $\langle (C, E), (\Delta_C, \Delta_E) \rangle$, where $V = C \cup E$ and $\Delta_C \cup \Delta_V = \Delta$, such that Δ_C consists of assignments to v' for each $v \in C$, and Δ_E consists of assignments to v' for each $v \in E$.*

Definition 3. *Given a partitioned model $\mathcal{M} = \langle (C, E), (\Delta_C, \Delta_E) \rangle$, the* environment *of $\mathcal{M}$ is a tuple $\mathcal{E} = \langle V, \Delta_E \rangle$.*

That is, the environment consists of its rules together with all variables in the model. In our traffic light example, Sensor is an environment variable, whereas Light is a component variable. Consequently, the rule $\mathsf{Sensor}' \leftarrow \neg\mathsf{Sensor} \vee \mathsf{Light}$ is the environment rule, and $\mathsf{Light}' \leftarrow \mathsf{Sensor}$ is the component rule. Models in which all variables have associated rules are called *closed*. For example, the combination of the sensor and the light controller for the model in Figure 1(a) is a closed system. A model that does not contain rules for all of its variables is called *open*. The *environment* is the open model obtained by removing the component rules from a closed model. In our example, it consists of a single rule $\mathsf{Sensor}' \leftarrow \neg\mathsf{Sensor} \vee \mathsf{Light}$, and variables Sensor and Light.

When does the environment guarantee a property? Intuitively, when satisfaction of some property of the model depends solely on the environment rules. For instance, if the environment rule in our example were $\mathsf{Sensor}' \leftarrow \text{true}$, then in any state where Sensor is true, the property $AG\ (\mathsf{Sensor} \vee \mathsf{Light})$ would be guaranteed by the environment. In this case, it is obvious that the environment alone guarantees the property; in real-life models, however, such as the one considered in our case study (see Section 6), the intricate logic may hinder the easy detection of such environment guarantees.

To define when the environment satisfies a property, we construct all possible "closures" of the environment with component rules, and then use the standard semantics of temporal logic over the resulting closed models. In our example, one closure was shown earlier, where the component rule $\mathsf{Light}' \leftarrow \mathsf{Sensor}$ is added to the rules of the environment. We can construct another closure by adding component rule $\mathsf{Light}' \leftarrow \neg\mathsf{Sensor}$ to the rules of the environment.

Definition 4. *A model $\mathcal{M} = (V, \Delta)$ is a* closure *of an environment $\mathcal{E} = (W, \Lambda)$ if $V = W$ and $\Lambda \subseteq \Delta$, where $\forall$ component variables in $\mathcal{M}$, $\exists$ a rule in $\Delta \setminus \Lambda$.*

Rules are identified modulo logical equivalence, so $v' \leftarrow \delta$ and $v' \leftarrow \neg\neg\delta$ are the same.

Definition 5. *An environment $\mathcal{E}$* guarantees a satisfaction *of temporal property φ in state s, written $\mathcal{E}, s \models \varphi$, if and only if* all of its closures *satisfy φ in s. An environment* guarantees a failure *of φ in state s, written $\mathcal{E}, s \models \neg\varphi$, if and only if φ fails in s in all closures of $\mathcal{E}$. Finally, an environment $\mathcal{E}$* guarantees *a property φ in state s iff $\mathcal{E}, s \models \varphi \ \vee\ \mathcal{E}, s \models \neg\varphi$.*

4 Environment Guarantees: Modeling and Algorithms

Given the environment rules, the rules closing them represent the behavior of a possible component in that environment. Intuitively, our notion of environment guarantee, given in Section 3, means that regardless of the component the environment is combined with, the resulting model still satisfies the property. This suggests the following naive approach to detecting environment guarantees: generate and model check all closures of the environment. Since there are exponentially many such closures, this approach is clearly infeasible. To solve this problem, in this section we use another representation of the environment that implicitly encodes all of its closures, and define a model-checking algorithm over this representation that checks all closures at once.

4.1 Logics for Open Systems

We aim to model open systems as state-transition graphs that can be model-checked directly. However, it is possible that an open system does not guarantee either the property, or its negation. That happens when the truth of the property depends on *how* the system is closed: in some closures the property is false; in the others, it is true.

Consider the model of the environment described in Section 3, with variables Sensor and Light and a single rule $\mathsf{Sensor}' \leftarrow \neg\mathsf{Sensor} \vee \mathsf{Light}$. Suppose we want to check a property that the sensor does not stay off for two consecutive states, *e.g.*, if the sensor is off in a given state, it will be on in all of its next states, formalized as $\psi_1 = AG\ (\neg\mathsf{Sensor} \Rightarrow AX\ \mathsf{Sensor})$. We check this property in a state where both the sensor and the light are on. Note that the rule of the environment guarantees that ψ_1 is true, independently of Light. Therefore, this is an environment guarantee and will evaluate to true on all of the closures. On the other hand, consider a slightly different property: in any state, if the sensor is off, it remains off for one more time step, or $\psi_2 = AG\ (\neg\mathsf{Sensor} \Rightarrow AX\ \neg\mathsf{Sensor})$. In this case, we can find two closures of the environment that disagree on the value of this property. In one of them, the environment is closed with rule (1) $\mathsf{Light}' \leftarrow \text{true}$, in the other – with (2) $\mathsf{Light}' \leftarrow \text{false}$. With (1), in any state, after at most two steps, the sensor becomes on and stays on forever. With (2), the sensor alternates between on and off. If checked in a state where both the sensor and the light are on, ψ_2 is true in the first closure, but false in the second. In this case, we want the property to evaluate to "unknown" on the model of the environment alone, meaning that the environment by itself does not have enough knowledge to satisfy or refute the property. By this discussion, classical Kripke structures are not appropriate for modeling open systems since thay limit reasoning to only two values. Instead, structures defined using multi-valued logics have been employed for this task [5, 13]. In our approach, we use the four-valued logic, known as Belnap [4] (see Figure 3), We use this logic instead of the 3-valued approach of [5] because it enables a more precise analysis by distinguishing between the partiality in the behaviour of the component and of the environment. We denote by $\mathbf{4}$ the set of values $\{\mathsf{t}, \mathsf{f}, \mathsf{m}, \mathsf{d}\}$: t and f stand for "known to be true" and "known to be false", respectively; m (maybe, or unknown) represents the lack of evidence to decide truth or falsity ("possibly true or false"); and d represents "necessarily true". Their information content defines an *information ordering* (see Figure 3(b)) $\preceq$: $\mathsf{m} \preceq \mathsf{t}, \mathsf{f}$ and $\mathsf{t}, \mathsf{f} \preceq \mathsf{d}$.

The usual boolean operations are extended $\mathbf{4}$. For example, conjunction of m with t is m since m may be resolved to true, in which case the result is true, or to false,

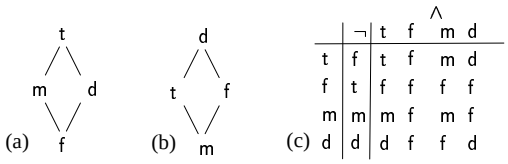

Fig. 3. Belnap logic $\mathbf{4}$: (a) truth ordering, (b) information ordering, (c) truth table.

and then the result is false. Figure 3(c) presents a table for computing conjunction and negation of values of this logic. These operations are computed on the *truth ordering* pictured in Figure 3(a), by using greatest lower bound for conjunction and symmetry for negation. We denote by $\mathbf{3}$ the subset $\{\mathsf{t}, \mathsf{f}, \mathsf{m}\}$ of $\mathbf{4}$, and by $\mathbf{2}$ – the subset $\{\mathsf{t}, \mathsf{f}\}$.

4.2 Representing an Open System as a State-Transition Graph

So far, we have established the requirements that a model of an open system should satisfy in order to help us in detecting environment guarantees: (1) it should support direct model-checking, (2) it should allow properties to evaluate to more values than just true or false, (3) it should represent all closures of the open system, and (4) its model-checking result should be equivalent to model-checking all those closures. In the rest of this section, we show that we can extend Kripke structures to the four-valued logic so that these requirements are satisfied.

The four-valued Kripke structure for our example open system is shown in Figure 1(e). It captures the interaction of the environment with all possible machines. Values of variables for which the environment does not have rules are unknown to the environment, captured by the logic value m. The transitions between states are four-valued. In the figure, solid and dashed lines are used to represent d and m transitions, respectively. Definite transitions, d, indicate local environment guarantees. For example, the d transitions from u_2 and u_3 to u_4 in Figure 1(e) indicate that in a state where the sensor is off, the environment guarantees that it will next become on, which can be inferred from the corresponding rule. The value of Light is unknown in u_4 since the environment cannot guarantee anything about it, as it does not have rules allowing changes to this variable. The m transitions capture what the machine can do, subject to environment restrictions. For example, the two m transitions from u_5 indicate that the machine has full control of the light. The absence of m transitions from u_5 to u_2 and u_3 means that the machine cannot violate the environment guarantee for sensor to be on.

Definition 6. *A* four-valued Kripke structure $\mathcal{M}$ *over a set of variables* $V = \{v_1, \ldots, v_n\}$ *is a tuple* $\langle S_\mathcal{M}, I_\mathcal{M}, R_\mathcal{M} \rangle$*, where* $S_\mathcal{M}$ *is the set of states, consisting of all possible* n*-tuples of values from* $\mathbf{3}$*;* $I_\mathcal{M} : V \times S_\mathcal{M} \to \mathbf{3}$ *is the interpretation of the variables that associates to every variable, in every state, a value from* $\mathbf{3}$*, i.e., for every* $s \in S_\mathcal{M}$ *and* $1 \leq i \leq n$*,* $I_\mathcal{M}(v_i, s) = s(i)$*; and* $R_\mathcal{M} : S_\mathcal{M} \times S_\mathcal{M} \to \mathbf{4}$ *is a* $\mathbf{4}$*-valued transition relation. For an* n*-tuple* s*, we denote by* $s(i)$ *its* i*th component.*

Given an open system described by a set of rules Δ, for every pair of three-valued states (s, s'), (1) transition (s, s') is m if s' is boolean (i.e., every variable in this state has value in $\mathbf{2}$); (2) transition (s, s') is d if for each variable v_i that is boolean in s', s and s' satisfy some rule $v_i' \leftarrow \delta \in \Delta$, and s and s' do not violate some rule $v_i' \leftarrow \delta \in \Delta$; (3) otherwise, the transition is false. The four-valued structure in Figure 1(e) has been constructed using this algorithm.

4.3 Checking for Environment Guarantees

Our method for detecting whether the environment guarantees a property is as follows. Given the environment rules, (1) construct the associated four-valued Kripke structure (using Definition 6). (2) use the multi-valued model-checking algorithm to check the property on this structure. (3) if the algorithm answers t or f, the property is guaranteed by the environment. An interpretation of CTL formulas over multi-valued Kripke structures and a corresponding model-checking algorithm have been defined [7], and apply to our four-valued Kripke structures without modification. We illustrate how the property "there is a next state where the light is on", written as $EX\ \mathsf{Light}$, is model-checked in state u_5 of the structure in Figure 1(e). In the classical case, the property is true if and only if there exists a next state where Light in true. Equivalently, the value of the property in a Kripke structure $\mathcal{K} = \langle S_\mathcal{K}, I_\mathcal{K}, R_\mathcal{K} \rangle$ is given by the boolean formula $\bigvee_{s' \in S_\mathcal{K}} R_\mathcal{K}(s, s') \wedge I_\mathcal{K}(s', \mathsf{Light})$, where $R_\mathcal{K}(s, s')$ is true if and only if $(s, s') \in R_\mathcal{K}$. The same formula is used in the four-valued case, where the operations involved are interpreted over $\mathbf{4}$, and for our example, we get:

$(\mathsf{m} \wedge \mathsf{t})$	// transition (u_5, u_0)	$\vee\ (\mathsf{d} \wedge \mathsf{m})$	// transition (u_5, u_6)
$\vee\ (\mathsf{m} \wedge \mathsf{f})$	// transition (u_5, u_1)	$\vee\ \mathsf{f}$	// all missing transitions

which evaluates to m. This is expected, because this property evaluates to true in the first closure of our example, and to false in the second. Model-checking of other CTL operators uses the evaluation of EX as a basic step. For example, for any formula p, "eventually p", or $EF\ p$, is expanded as $p \vee EX\ (p \vee EX\ (...))$, and this expansion is finite since the system is finite-state. A property "always p", or $AG\ p$, is equivalent to $\neg EF\ \neg p$. If we check properties ψ_1 and ψ_2 on the structure of Figure 1(e) using this algorithm, we obtain t and m, respectively. Our algorithm points to an environment guarantee if the property evaluates to either true or false, as it does for ψ_1.

4.4 Correctness

To show that the method presented in Section 4.3 is sound, we need to show that if the model-checking algorithm answers t, then all of the closures of the environment satisfy the property. In Section 3, we showed that each closed system is mapped to a classical Kripke structure. Thus, such a structure exists for every closure of the environment. For example, the structures for the two closures in our example are shown in Figure 1(b)-(c). We first define *state compatibility* by extending the information ordering to tuples of values component-wise.

Definition 7. *Let u be a 3-valued state and w be a boolean state over the same variables.* w is more informative *than u, e.g., $u \preceq w$ if, for all $1 \leq i \leq n$, $u(i) \preceq w(i)$.*

The compatibility relation between the boolean and four-valued structure is defined as follows: a three-valued state u is compatible to all boolean states w where $u \preceq w$; for any two compatible states, any d transition out of the three-valued state is matched by a transition out of the boolean state; conversely, any transition out of the boolean state is matched by a m transition out of the three-valued state. The matched transitions mean that the destinations of these transitions are compatible. We can verify compatibility of the structures in Figure 1(b)-(c) with the four-valued structure in Figure 1(e). For example, state u_5 is compatible with w_3. The d transition (u_5, u_4) is matched by (w_3, w_0), since u_4 and w_0 are compatible. The d transition (u_5, u_6) can also be matched

by (w_3, w_0) since u_6 is also compatible with w_0. Conversely, a transition (w_3, w_0) is matched by the m transition (u_5, u_0).

Any classical Kripke structure compatible with a four-valued Kripke structure is called its *completion*.

Definition 8. *A classical Kripke structure $\mathcal{K} = (S_\mathcal{K}, I_\mathcal{K}, R_\mathcal{K})$ is a* completion *of a four-valued Kripke structure $\mathcal{M} = (S_\mathcal{M}, I_\mathcal{M}, R_\mathcal{M})$ if for any $u \in S_\mathcal{M}$ and $w \in S_\mathcal{K}$, $u \preceq w$ implies: (1) for all $u' \in S_\mathcal{M}$ such that $R_\mathcal{M}(u, u') \succeq \mathsf{t}$, there exists $w' \in S_\mathcal{K}$ such that $u' \preceq w'$ and $(w, w') \in R_\mathcal{K}$, and (2) for all $w' \in S_\mathcal{K}$ such that $(w, w') \in R_\mathcal{K}$, there exists $u' \in S_\mathcal{M}$ such that $u' \preceq w'$ and $R_\mathcal{M}(u, u') \preceq \mathsf{t}$.*

The structures in Figure 1(b) and (c) are thus completions of that in Figure 1(e). In fact, all classical structures corresponding to the closures of an open system are completions of the four-valued structure associated with this system.

Theorem 1. *Let $\mathcal{E}$ be an open system and $\mathcal{M}_\mathcal{E}$ be its associated four-valued Kripke structure. Then, every closure of $\mathcal{E}$ corresponds to a classical Kripke structure that is a completion of $\mathcal{M}_\mathcal{E}$.*

Thus, we can conclude that if all completions of the four-valued Kripke structure satisfy a property, then all closures of the open system do so as well. To complete our soundness argument, we note that the multi-valued model-checking algorithm has the following property: *If on a given structure and a given property φ, the answer of the model-checking algorithm is t (f), then all completions of that structure satisfy (violate) φ, and thus φ is guaranteed by the environment.*

Theorem 2. *Let $\mathcal{E}$ be an open system and $\mathcal{M}_\mathcal{E}$ be its associated four-valued Kripke structure. For any CTL formula φ and any boolean state s, if the result of the multi-valued model-checking algorithm on $\mathcal{M}_\mathcal{E}$ is t or f, then $\mathcal{E}$ guarantees φ in state s.*

When model-checking yields t (f), we can further conclude that φ holds (fails to hold) in the composition of $\mathcal{E}$ with every component.

Our method is not complete, *i.e.*, if the answer of the model-checking algorithm is m, the environment may or may not guarantee the property.

5 Implementation

The multi-valued model-checking algorithm that reasons over four-valued Kripke structures has been implemented in a tool χChek [8] – a symbolic model-checker built on top of the state-of-the-art decision diagram library CUDD [31]. We can use χChek to check models of the environment directly or reduce the multi-valued model-checking problem to two classical ones, via a reduction described in [19], and thus use a classical model-checker such as NuSMV [9]. In either case, this approach is more efficient than checking all possible closures of the environment.

Unfortunately, while χChek can provide an effective reasoning over models once they have been constructed, building such models from text-based descriptions remains a challenge. Specifically, the case study in Section 6 involved a model specified in SMV [9], where the full generality of the SMV modeling language was used. In what follows, we discuss a simple implementation that can decide environment guarantees true ACTL formulas. An example ACTL property is $AF\ (\mathsf{Sensor} \vee \mathsf{Light})$. Intuitively, since any ACTL property refers to "all paths", if it holds on the model with the most paths, it will hold on any model having a subset of those paths. It was shown in [19] that

truth of ACTL properties can be decided by restricting the model-checking algorithm only to the m transitions. In the four-valued structures we use to model open systems (see Section 4.2), a destination of an m transition is always a boolean state. Thus, the reachable state space of a structure restricted to those transitions is completely boolean. Furthermore, this boolean structure corresponds to a composition of the environment with the component that changes its variables nondeterministically.

Let us consider the *most nondeterministic component* to be the one where all variables change nondeterministically, *i.e.*, for every $c \in C$, the rules for c are $c \leftarrow$ true and $c \leftarrow$ false. The closure of the environment with this component results in the model with the most paths. If this closure satisfies an ACTL property, the closure with any other component will satisfy the property as well. Thus, to check if an ACTL property is an environment guarantee, it is sufficient to check if it is satisfied by the closure of the given environment with the most nondeterministic component.

Consider the environment in our example, consisting of the rule $\mathsf{Sensor}' \leftarrow \neg\mathsf{Sensor} \vee \mathsf{Light}$. The most nondeterministic component in this case is described by rules $\mathsf{Light}' \leftarrow$ true, $\mathsf{Light}' \leftarrow$ false. The Kripke structure associated with their composition is shown in Figure 1(d). With s_0 as the initial state, the property $AF\ (\mathsf{Sensor} \vee \mathsf{Light})$ holds in this structure. Three other closures of the same environment, shown in Figures 1(a)-(c), satisfy the property as well.

Correctness of using the most nondeterministic component for checking environment guarantees also follows from the fact that the closure of the environment with such component simulates all other closures of that environment. For instance, the Kripke structure in Figure 1(d) is a simulation of models in Figure 1(a)-(c). By [18], simulation preserves true ACTL properties, giving us a correct algorithm. Because of the duality of CTL operators, the same result holds for false existential properties.

Theorem 3. *Let $\mathcal{E}$ be an environment, φ be a true ACTL property, and $\mathcal{M}$ be the closure of $\mathcal{E}$ with the most nondeterministic component.* $\mathcal{E}$ guarantees φ *iff* $\mathcal{M}$ *satisfies* φ.

The most nondeterministic environment is routinely used for checking correctness of true universal properties of the component, e.g., [16]. However, we believe we are the first to propose the use of this technique for finding environment guarantees. The composition between the environment and the most nondeterministic component is trivial to construct syntactically from a text-based description of a system. Specifically, we have implemented this method for the modeling language of NuSMV, to facilitate reasoning about the TCAS II system (see Section 6). The language of NuSMV is similar to ours, and its semantics is such that if for any variable a rule is not given, the variable is assumed to change nondeterministically. Thus, to implement the detection of environment guarantees for true ACTL properties, it suffices to remove the component rules from a model[1], and then check the properties on the remaining model using NuSMV [9]. The implementation is also highly efficient: increasing nondeterminism reduces the sizes of the decision diagrams used by NuSMV, and hence its running time.

6 Case Study: Checking the TCAS II System

We illustrate our approach with the Traffic Collision Avoidance System, TCAS II [32]. TCAS II implements a protocol for conflict detection and resolution between an aircraft

[1] If the language did not have this default semantics, we would have to also insert rules $v' :=$ true and $v' :=$ false for every component variable v, which is simple to do syntactically as well.

Properties	Results		Time (sec.)		BDD nodes	
	Full	Env.	Full	Env.	Full	Env.
0 reachability	—	—	1034.61	4.2	1349878	145246
1 AG ¬ND_Composite_RA	true	true	20.63	3.8	173041	37846
2 AG (New_Increase_Climb ⇒ AX ¬New_Increase_Descend)	true	true	20.81	3.84	175280	37991
3 AG (OA.in_Sense_Climb_Positive ∧ Composite_RA = Descend ∧ OA_Not_Evaluated_Meantime ⇒ CRA_Not_Changed_Meantime)	true	true	27.83	3.87	341216	38352
4 AG (Composite_RA_Evaluated_Event ⇒ ¬DMG_Inconsistent)	true	maybe	40.67	6.1	224611	39709

Table 1. Results of checking properties of TCAS II.

and neighboring aircraft so as to avoid collisions during flight. This is a safety-critical system required on every U.S. commercial aircraft transporting more than thirty passengers, and has also been deployed in other countries. TCAS II has also been used as a classical case study for requirements modeling [26] and formal verification [22, 27, 6].

An SMV model of TCAS II has been translated from RSML [26] by Chan *et al.* [6] and is part of the NuSMV distribution. It views TCAS as consisting of two main modules: Own Aircraft, which is the aircraft having TCAS II installed, and Other Aircraft, which is a neighboring aircraft that may or may not have TCAS installed. An instance of Own Aircraft may communicate with several instances of Other Aircraft. Own Aircraft maintains information about the state of the host aircraft, including its altitude, direction, horizontal and vertical speeds, and it also receives similar information from Other Aircraft. Based on this information, Own Aircraft assesses possible threats and, in case it finds any, computes an escape maneuver (*e.g.*, climb, or descend) and the strength of this maneuver (*i.e.*, the altitude rate at which it is to be carried out) and outputs both as advisory to the pilot. TCAS II escape maneuvers are limited to the vertical plane.

The SMV model we looked at contains one instance of the Own Aircraft state machine and one instance of an abstraction of Other Aircraft that behaves mostly nondeterministically. In this work, we view Own Aircraft as the component and Other Aircraft as its environment. Even with many features of TCAS II abstracted away, this SMV model is non-trivial for the NuSMV model-checker: computing the reachable states of this model takes 17 minutes on our machine (a Dell PC with an Intel Pentium 4 CPU at 2.8 GHz and 1 GB of RAM, running Red Hat Linux 7.3) yielding 1,349,878 BDD nodes. The model comes with several CTL formulas capturing essential properties of the system. All of these properties are in ACTL and happen to hold in the implementation provided. Therefore, our implementation described in Section 5 could be applied. We used it to check these properties, as well as a few additional ones. For the summary of results, please refer to Table 1.

Our analysis focuses on two SMV variables: `Composite_RA` which encodes the escape maneuver, or Resolution Advisory, and `Displayed_Model_Goal`, which encodes its strength. A desirable property of `Composite_RA` is that it should change deterministically (see [22]): this is essential for ensuring that Own Aircraft has predictable behavior and does not decide on different maneuvers under similar conditions. We checked that nondeterminism is not attained using a macro `ND_Composite_RA` defined in the model to encode possible nondeterminism (row 1 of Table 1). We performed the check on the original system and on the open model of the environment, and the property evaluated to true under both checks, which shows that it is in fact guaranteed by the environment. It seems that the modeler oversimplified the state machines, eliminating much of the logic that computes `Composite_RA`, which is essential in TCAS II. Table 1 summarizes

the performance of the check in terms of time and BDD node allocation, on the full model (Full) and the environment alone (Env.).

Next, we verified a property which we expect to hold in any aircraft controller system: no aircraft can immediately switch from increasing the rate of climbing to increasing the rate of descending, *i.e.*, an aircraft must stop climbing before descending. The model defines macros `New_Increase_Climb` and `New_Increase_Descend` to encode the respective resolution advisories, which we used to formulate the question (row 2 in Table 1). It also passed in both the original model and the environment only, hence being guaranteed by the environment. This confirms the modeling error we have noticed before, *i.e.*, that Other Aircraft controls the resolution advisories of Own Aircraft.

We also checked whether it is possible for the direction (up or down) of Other Aircraft to change without being noticed by Own Aircraft. More precisely, we checked whether it is possible for `Composite_RA` of Own Aircraft to remain constant if Other Aircraft changes from climb to descend (row 3 of Table 1). Using Dwyer *et al.*'s property patterns [14], we expanded `OA_Not_Evaluated_Meantime` as

$$A[(\neg \texttt{OA\_Evaluated\_Event} \vee AG\, \neg \texttt{OA.in\_Sense\_Descend\_Positive})\, U\, \texttt{OA.in\_Sense\_Descend\_Positive}\ ]$$

and similarly, `CRA_Not_Changed_Meantime` as

$$A[(\texttt{Composite\_RA} = \texttt{Descend} \vee AG\, \neg \texttt{OA.in\_Sense\_Descend\_Positive})\, U\, \texttt{OA.in\_Sense\_Descend\_Positive}\ ].$$

This property is also guaranteed by the environment. The antecedent of the implication fails both in the full model and in the environment This reveals an environment assumption, rooted in the semantics of RSML: environment variables cannot change without being evaluated by the component. Discovering such assumptions is important [16, 11] to make verification experts aware of conditions under which their analysis is valid.

Finally, we checked one of the original properties of the TCAS II system [6]. It states that when `Composite_RA` is evaluated, `Displayed_Model_Goal` is computed "consistently", *i.e.*, the cases by which its value is decided are mutually exclusive. A macro `DMG_Inconsistent`, defined in the model, captures the inconsistency conditions, resulting in the formula shown in row 4 in Table 1. Since the property holds in the full model, but not on the environment, the environment alone does not guarantee it, and the property does depend on the component. A vacuity check [24, 21, 15], however, indicates that the antecedent of the implication is vacuous, potentially misleading the user into thinking that something is wrong. [6] notes that in this model, `Composite_RA` sometimes disagrees with `Displayed_Model_Goal` (*e.g.*, the advisory indicates climb, but the strength of the maneuver is negative), but does not provide an additional explanation. Our method helped us in identifying the reason for these anomalies: the logic for computing the escape maneuver *does not* depend on the component, whereas that for computing its strength *does*.

As we argued in Section 5 and observed in our experiments in Table 1, for true universal properties, verifying the environment alone is much more efficient than checking the full model. This suggests that at least for this class of properties, checking whether the environment guarantees the property can precede the verification process: if the property evaluates to true on an environment alone, it will yield this answer when the environment is composed with the system.

How representative our experience of finding a model with only true ACTL properties to verify? [6] indicate that the value of `Displayed_Model_Goal` is computed by a

case analysis consisting of seven cases. These cases are supposed to be mutually exclusive. The property $AG\ \neg$`DMG_Inconsistent` checks whether this is indeed the case. This property was initially found false[2]. Upon manual inspection of the counterexample produced by the model-checker, *the environment model was identified as the cause of the violation*, and it was fixed so that the property finally passed. Thus, we have evidence to believe that false universal properties exist "in the wild", and detecting environment guarantees for those is a worthwhile task which would eliminate the manual analysis of the counterexample. A yet more important class of properties to handle is CTL properties with mixed path quantifiers. For example, it is conceivable to demand that a reactive system can always be reset. One way to implement it is to have an initial state *Init* and require a property $AGEF$ *Init*, *i.e.*, from every state of the system, state *Init* is always reachable. Whether such a property holds or fails, a counterexample for it cannot be generated, and a special-purpose technique for detecting environment guarantees, such as the one proposed in Section 4 is required. We leave implementing a technique for checking environment guarantees of arbitrary CTL properties for future work.

7 Related Work and Discussion

The original definition of vacuity attempted to capture the conditions under which satisfaction of a property in the model does not indicate that the model behaves correctly. This definition was motivated by practical experience at the IBM Haifa Research Lab in applying model-checking to verifying hardware systems [3]. This definition was developed in a context of a rather restricted fragment of a temporal logic, in which a property is divided between a stimulus provided by the environment and an expected response of the component. In this context, this work provided an efficient algorithm for vacuity detection that identifies errors in practice. However, it does not work for more general properties and when the placed assumptions are not satisfied.

Over the years, the algorithm for vacuity has been generalized and extended to various temporal logics, e.g, see [24, 3, 1, 20, 21, 28], and several tools [29, 15] have been implemented. However, this work has concentrated on the technical definition of vacuity – *i.e.*, whether every subformula of a property is important for it satisfaction. Without additional assumptions used by Beer *et al.*, these techniques can produce false positives, *i.e.*, cases of vacuity that are not indicative of errors in the system. Instead of detecting trivial satisfaction, they indicate when a property can be simplified. Although this may be useful for model-checking, by itself it does not help in identifying problems.

For example, consider the property $\varphi = AG\,(\mathsf{Sensor\_Tripped} \Rightarrow (AF\ \mathsf{Light} = \mathsf{green}))$ from Section 1. It is vacuous in Sensor_Tripped in a model where the light changes color periodically, whether there is a car waiting at the intersection or not. Thus, a stronger property, $AG\ AF(\mathsf{Light} = \mathsf{green})$, holds in the model, but does not necessarily signal any errors. Suppose that φ was given by the requirements stakeholder for creating a more optimal traffic controller system, and the model is just one implementation of the controller that does not require the assumption of the sensor being tripped. Another example is when Sensor_Tripped is under the control of the component (and not the environment), so vacuity in it may not lead to a problem either: the controller might have some values of the sensor hard-coded into it, just to make sure that the rest of the

[2] Unfortunately, we were unable to obtain this erroneous model.

controller behaves correctly. We refer to such cases as *property overengineering* – requiring a property that is weaker than the one that actually holds in the model by making potentially unnecessary assumptions about the environment or the state of the system. Industrial experience [12] indicates that properties are hard to get right. This process is expensive, and the properties, once deemed correct and validated with all stakeholders, remain fixed throughout the duration of the project, and even between different releases of the system. So, engineers are often reluctant to modify overengineered properties, and vacuity reports that point to such cases only distract from finding real problems.

In this paper, we have shown that Kripke structures based on 4-valued Belnap logic can be used to approximate open systems. Godefroid [17] has also proposed to use multi-valued logic, in his case, 3-valued Kleene logic, to model open systems. He shows that under an assumption that the component can block the environment, his 3-valued model-checking technique is equivalent to module-checking. However, we believe that this assumption is highly unrealistic – a component can interact with the environment, but cannot deter its progress.

The setting of work on *robust satisfaction* [23] is similar to ours: given a system M, determining whether a property holds in all environments composed with M under synchronous parallelism (for environment guarantees, the roles of the system and the environment are reversed). This algorithm is complete, and the authors note that for checking satisfaction of ACTL, robust satisfaction has the same complexity as model-checking and can be decided using the same implementation as ours. For other properties, robust satisfaction is exponentially more expensive than model-checking. In contrast, our algorithm is partial, *i.e.*, in some cases it may fail to detect that a property is guaranteed by the environment, but is of the same complexity as model-checking.

Module checking [25] is similar to robust satisfaction, but is defined over asynchronous composition. It has the same complexity as model-checking for true ACTL properties as well as for reachability ($EF\varphi$) and universal reachability ($AGEF\varphi$). We plan to use this approach for detecting environment guarantees in asynchronous systems.

The work of [30] is the closest to ours in spirit: determining which part of the model is needed for checking the correctness property can alert the user to the presence of an overconstraint in their declarative models and help him/her locate its source. As in our case, the algorithm of [30] is efficient but not complete, and the authors report of several overconstraints that were not detectable by it.

8 Conclusion and Future Work

In this paper, we argued for the need to provide support for debugging environment models used in model-checking software systems. Specifically, we noted that when the environment single-handedly guarantees a (truth or falsity of) property which is expected of the component, then either the property should be reconsidered as one of the environment, or there is an error in the model of the environment. We have called this problem *environment guarantees* and argued that it can be found if the environment is modelled as an open system. We also discussed how to construct open models of the environment from rule-based descriptions of state-machine models, such as those created by SMV specifications, and implemented this technique for checking whether true ACTL properties are guaranteed by the environment. We reported our experience with a model of the TCAS II system which showed that environment guarantees present a

real threat, especially when the modeler attempts to create abstractions of their systems to overcome the state explosion problem of model-checking. We also argued that the problem is not limited to true ACTL properties, and while we have theoretical decision procedure for arbitrary CTL properties, implementing it and comparing its performance against an implementation of robust satisfaction [23] is left for future work.

Our work opens a number of questions related to debugging models of the environment: (1) We assumed that every variable belongs to the environment or to the component (but not both), *i.e.*, the environment and the component do not have shared variables. Moreover, we considered only cases of synchronous parallelism between the two. To make our approach applicable to more general domains, we plan to address these limitations. (2) Clearly, there are environments that may not guarantee a property by themselves; however, there are additional constraints imposed on them by components, *i.e.*, via communication channels, that lead to environment guarantees. We intend to study this problem in future work. (3) We also assumed that there is a clear separation between the component and its environment, and thus the environment can be captured and its model constructed. This may not always be the case. For example, we may aim to verify a collection of components compositionally, so while checking one component, all remaining ones form its environment. This environment might be simply too big to analyze. There might also be cases when the component is composed with multiple environments, or when determining what constitutes an environment is difficult. One potential direction to remedy these problems is to follow the approach of Shlyakhter *et al.* [30], aimed at computing and highlighting the part of the overall system on which the property depends. The user can then see whether the highlighted part is the environment and decide whether this constitutes a problem. Of course, highlighting is useful even when the boundary between the component and the environment is well understood: it points the user to the part of the environment that is entirely responsible for satisfying the desired property, facilitating debugging.

Acknowledgment. We thank Shoham Ben-David for her comments on an earlier draft of this paper. Financial support has been provided by NSERC and IBM.

References

1. R. Armoni, L. Fix, A. Flaisher, O. Grumberg, N. Piterman, A. Tiemeyer, and M. Vardi. "Enhanced Vacuity Detection in Linear Temporal Logic ". In *Proceedings of CAV'03*, volume 2725 of *LNCS*, pages 368–380, July 2003.
2. D. Beatty and R. Bryant. "Formally Verifying a Microprocessor Using a Simulation Methodology". In *Proceedings of DAC'94*, pages 596–602, 1994.
3. I. Beer, S. Ben-David, C. Eisner, and Y. Rodeh. "Efficient Detection of Vacuity in Temporal Model Checking". *FMSD*, 18(2):141–163, March 2001.
4. N.D. Belnap. "A Useful Four-Valued Logic". In Dunn and Epstein, editors, *Modern Uses of Multiple-Valued Logic*, pages 30–56. 1977.
5. G. Bruns and P. Godefroid. "Model Checking Partial State Spaces with 3-Valued Temporal Logics". In *Proceedings of CAV'99*, volume 1633 of *LNCS*, pages 274–287, 1999.
6. W. Chan, R.J. Anderson, P. Beame, S. Burns, F. Modugno, and D. Notkin. "Model Checking Large Software Specifications". *IEEE TSE*, 24(7):498–520, July 1998.
7. M. Chechik, B. Devereux, S. Easterbrook, and A. Gurfinkel. "Multi-Valued Symbolic Model-Checking". *ACM TOSEM*, 12(4):1–38, October 2003.

8. M. Chechik, B. Devereux, and A. Gurfinkel. "χChek: A Multi-Valued Model-Checker". In *Proceedings of CAV'02*, volume 2404 of *LNCS*, pages 505–509, July 2002.
9. A. Cimatti, E.M. Clarke, , E. Giunchilia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. "NUSMV Version 2: An Open Source Tool for Symbolic Model Checking". In *Proceedings of CAV'02*, volume 2404 of *LNCS*, pages 359–364, 2002.
10. E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.
11. J. M. Cobleigh, D. Giannakopoulou, and C. S. Pasareanu. "Learning Assumptions for Compositional Verification". In *Proceedings of TACAS'03*, volume 2619 of *LNCS*, 2003.
12. F. Copty, A. Irron, O. Weissberg, N. Kropp, and G. Kamhi. "Efficient Debugging in a Formal Verification Environment". *STTT*, 4(3):335–348, May 2003.
13. L. de Alfaro, P. Godefroid, and R. Jagadeesan. "Three-Valued Abstractions of Games: Uncertainty, but with Precision". In *Proceedings of LICS'04*, pages 170–179, 2004.
14. M. Dwyer, G. Avrunin, and J. Corbett. "Patterns in Property Specifications for Finite-State Verification". In *Proceedings of ICSE'99*, May 1999.
15. M. Gheorghiu, A. Gurfinkel, and M. Chechik. "VaqUoT: A Tool for Vacuity Detection". In *Proceedings of Tool Track, FM'06*, August 2006.
16. D. Giannakopoulou, C. S. Pasareanu, and H. Barringer. "Assumption Generation for Software Component Verification". In *Proceedings of ASE'02*, pages 3–12, 2002.
17. P. Godefroid. "Reasoning about Abstract Open Systems with Generalized Module Checking". In *Proceedings of EMSOFT'03*, volume 2855 of *LNCS*, pages 223–240, October 2003.
18. O. Grumberg and D.E. Long. "Model Checking and Modular Verification". In *Proceedings of CONCUR'91*, 1991.
19. A. Gurfinkel and M. Chechik. "Multi-Valued Model-Checking via Classical Model-Checking". In *Proceedings of CONCUR'03*, volume 2761 of *LNCS*, pages 263–277, 2003.
20. A. Gurfinkel and M. Chechik. "Extending Extended Vacuity". In *Proceedings of FM-CAD'04*, volume 3312 of *LNCS*, pages 306–321, November 2004.
21. A. Gurfinkel and M. Chechik. "How Vacuous Is Vacuous?". In *Proceedings of TACAS'04*, volume 2988 of *LNCS*, pages 451–466, March 2004.
22. M. Heimdahl and N. Leveson. "Completeness and Consistency in Hierarchical State-Based Requirements". *IEEE TSE*, SE-22(6):363–377, June 1996.
23. O. Kupferman and M. Vardi. "Robust Satisfaction". In *Proceedings of CONCUR'99*, volume 1664 of *LNCS*, pages 383–398, 1999.
24. O. Kupferman and M. Vardi. "Vacuity Detection in Temporal Model Checking". *STTT*, 4(2):224–233, February 2003.
25. O. Kupferman, M.Y. Vardi, and P. Wolper. "Module Checking". *Information and Computation*, 164(2):322–344, January 2001.
26. N.G. Leveson, M.P.E. Heimdahl, H. Hildreth, and J.D. Reese. "Requirements Specification for Process-Control Systems". *IEEE TSE*, 20(9):684–707, September 1994.
27. J. Lygeros and N. Lynch. "On the Formal Verification of the TCAS Conflict Resolution Algorithms". In *Proceedings of Conf. on Decision and Control*, December 1997.
28. K. Namjoshi. "An Efficiently Checkable, Proof-Based Formulation of Vacuity in Model Checking". In *Proceedings of CAV'04*, volume 3114 of *LNCS*, pages 57–69, 2004.
29. M. Purandare and F. Somenzi. "Vacuum Cleaning CTL Formulae". In *Proceedings of CAV'02*, volume 2404 of *LNCS*, pages 485–499, July 2002.
30. I. Shlyakhter, R. Seater, D. Jackson, M. Sridharan, and M. Taghdiri. "Debugging Overconstrained Declarative Models Using Unsatisfiable Cores". In *Proceedings of ASE'03*, 2003.
31. F. Somenzi. "CUDD: CU Decision Diagram Package Release", 2001.
32. US Dept. of Transportation. "Introduction to TCAS II". FAA, March 1990.