Efficient Debugging of Environment Models

Marsha Chechik Mihaela Gheorghiu Arie Gurfinkel
Department of Computer Science
University of Toronto
40 St. George St.
Toronto, Canada M5S 2E4
{chechik,mg,arie}@cs.toronto.edu

ABSTRACT
One of the major threats to validity of model-checking systems is the ability of the analyst to correctly model the environment in which such systems are to be run. Undocumented and/or incorrect assumptions of the environment may lead to catastrophic errors when systems are deployed and often cause incorrect behaviour when existing software is being reused in other contexts.

In this paper, we introduce and formalize a notion of *environment guarantee* that captures those cases where general temporal logic properties hold just because of the assumptions on the environment. We also present a technique for the detection of environment guarantees. We applied this technique to an SMV model of TCAS II, finding a number of properties that suffer from environment guarantees.

Keywords
Model-checking, temporal logic, vacuity, reactive systems.

1 Introduction

Model-checking is used more and more as a "push-button" analysis procedure. With many translators from high-level modeling formalisms into model-checking languages, verification is becoming an effective method for debugging models as well as assuring that they are correct.

Model-checking is applied to closed systems, so analysts must create a faithful model of the environment in which their component operates. The ability to correctly capture such an environment is one of the major threats to validity of model-checking approaches.

Incorrect assumptions of the environment have been identified as a major threat to creating high quality systems [35, 29, 26]. They can lead to catastrophic errors when systems are deployed (*e.g.*, the Panama radiation accident [25], or the MAR knockout [39]) and can cause incorrect behaviour when existing software is being reused in other contexts (*e.g.*, the famous Therac-25 [31] example where code reuse led to loss of human life). Of course, when verification concentrates on a single component or an algorithm, creating a good model of the environment is even more difficult, and an ability of the verification process to flag the environment as faulty – even more pressing.

How do we know when the environment is at fault? Let's consider an example of a traffic light controller: cars arrive at the intersection, trip sensors, and wait for the green light. The traffic controller uses the sensors to maximize the flow of cars through the intersection. An essential property of such a system is that if a sensor is ever tripped, an appropriate light eventually turns green.This property is formalized in a temporal logic CTL [13] as

$$\varphi = AG\,(\mathsf{Sensor\_Tripped} \Rightarrow (AF\ \mathsf{Light} = \mathsf{green}))$$

If the model of the environment does not allow sensors to be tripped, then this property holds regardless of the correctness of the controller. Our experience in teaching a graduate-level course on automated verification over the past decade shows that students create models with faulty environments all the time: we see several of them each year.

Industrial researchers [3] also noted early on that properties like φ are often found in practice, referring to the problem as "antecedent failure". IBM researchers refined this notion, naming it *vacuity* [4]. Vacuity was defined for a class of formulas and always pointed to a real problem in "either the design or its specification or environment", which was observed for 20% of checked properties. The definition is *property-based*: a formula p is vacuous in a subformula q in a given model if q does not influence the value of p in this model. That is, in the traffic light example above with the faulty environment model, the property φ is vacuous in $\mathsf{Light} = \mathsf{green}$: it is satisfied independently of the color of the light because the antecedent of the implication is false. Vacuity reported by this approach always signaled errors because of the reliance on the assumption that in any property, the antecedent of the first implication is an environment-controlled event. Thus, this analysis would not have reported an error (*i.e.*, vacuity) in

$$AG\,(\neg\mathsf{Sensor\_Tripped} \vee AF\ (\mathsf{Light} = \mathsf{green}))$$

which is logically equivalent to φ, nor in a property such as

$$AG(\mathsf{Light} = \mathsf{red} \Rightarrow A[\mathsf{Light} = \mathsf{red}\ U\ \mathsf{Sensor\_Tripped}])$$

i.e., the light remains red until the sensor is tripped.

Other researchers, *e.g.*, [28, 2, 23], extended the notion of vacuity to checking all subformulas for general properties expressed in CTL and other common languages. However, these approaches remained *property-based*, and, without additional assumptions used by Beer *et al.*, can produce false positives, *i.e.*, cases of vacuity that are not indicative of an error in the system. For example, consider the property φ again. 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. This means that 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 additional assumption of the sensor being tripped. Another example is when Sensor_Tripped is under the control of the machine (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 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 [15] 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. Thus, engineers are usually reluctant to modify overengineered properties, and vacuity reports that point to such cases only distract from finding real problems.

To determine how often vacuity signifies environment guarantees, we analyzed 44 models written by graduate students at the University of Toronto as part of an automated verification course. Using our vacuity detector VaqUoT [23, 18], we checked 740 properties of these models, and found that 94 of those were reported as vacuous. 82 of those were examples of property overengineering such as "the light can be either green or red or yellow" (this was enforced by employing an enumerated type for light) or "when the sensor is tripped and the light is red, the light will eventually turn green", since the light was guaranteed to turn green regardless of its current color. Out of the remaining 12, 8 were caused by an environment that did not transition out of its initial state (a typically SMV modeling mistake), which we detected by inspecting the model. Another common cause was imposing an unjustifiable fairness condition on the environment which guaranteed the properties, *e.g.*, that the sensors be tripped infinitely often.

In contrast to the property-centric approach adopted by the work on vacuity detection, Shlyakhter *et al.* [40] devised a technique to look for *errors in models* directly. They defined the problem of overconstraining (declarative) models, and pointed out that overconstraining "... occurs most often in the definition of model-checked algorithms rather than in the specification of correctness properties". They have developed a technique for extracting and displaying the part of the model that was used for establishing satisfaction of a property. When most of the model was not necessary for proving a property, the authors were able to conclude that this is a result of an overconstraint, caused by subtle modeling errors.

In this paper, we also aim to provide a technique for model debugging, but in the case of operational models such as those specified by state-machines. In this domain, the model is "overconstrained" if a property that should hold in the system (*optative* [26]) is guaranteed solely by the environment. In other words, the machine can be replaced by another, *arbitrary*, machine with the same interface (*i.e.*, a fixed set of variables under the control of the machine and that of the environment), without affecting the satisfaction of the property. We say that such properties are *guaranteed by the environment*. Clearly, environment guarantees always indicate a problem: either the desired property should be labeled as *indicative* [26] and not be asked of the system, or there is an error in the model of the environment.

Needless to say, generating all systems with the desired interface to check for environment guarantees is infeasible. Instead, we show how to represent the environment as an open system and check properties on it directly, using a symbolic model-checking algorithm.

This paper makes the following contributions:

1. We argue that one way to discover faulty environment models is to look for properties which are guaranteed by the environment alone. Although this process does not find all possible environmental errors, errors reported by this analysis *always* point out to errors in the environment, and results of model-checking obtained in this situation are definitely faulty.
2. We formalize the meaning of a property being guaranteed by the environment via representing the environment as an open system (see Section 2).
3. We show how to model open systems and check them effectively, which leads to an automatic algorithm for discovering environment guarantees (see Section 3).
4. 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) [24, 8]. In our case study, we found that several essential properties of TCAS II, including some of those analyzed by Chan *et al.* in [8], hold as the result of environment guarantees: the model of the environment used in verifying this system has been simplified too much, rendering the analysis in [8] somewhat meaningless (see Section 4).

We compare our approach with related work in Section 5 and conclude in Section 6 with a summary of the paper and possible venues for future work.

2 Environment Guarantees

As we described in Section 1, when the environment guarantees a property, it is a clear indication of an error. In this section, we formalize the meaning of such environment guarantees.

We begin by reviewing the traditional model-checking process. A *system* $\mathcal{S}$, such as the traffic light controller, is modeled by 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 system is an assignment of values to all variables. Without loss of generality, we assume that all variables are boolean. For example, we simply define a predicate Light which is true iff the light is green ($\mathsf{Light} = \mathsf{green}$). Therefore, the set S of system states consists of all n-tuples of boolean values. A *rule* is an expression over the system variables that relates their values in a state at time t (current) with those at time $t+1$ (next). For a variable v, we use v and v' to denote its current and next state value, respectively. We assume that rules are described in the style of the SMV language [11]. Each rule is an assignment of the form $v' := \delta$, where v is a variable and δ is a boolean formula over unprimed variables only. Furthermore, there is at least one assignment for each variable. For example, a rule $\mathsf{Sensor}' := \neg\mathsf{Sensor} \vee \mathsf{Light}$ means that the sensor is on in the next state if it is currently off, or the light is on. It is convenient to think of each assignment $v' := \delta$ as a boolean formula, $(v' \wedge \delta) \vee (\neg v' \wedge \neg\delta)$. Multiple assignments per variable are used to indicate that the variable changes nondeterministically. For example, having both $v' := \text{true}$ and $v' := \text{false}$ means that v' is either true or false in the next state.

To $\mathcal{S}$ we associate a state-transition graph $\mathcal{K}_\mathcal{S}$, known as a *Kripke structure*. The Kripke structure $\mathcal{K}_\mathcal{S}$ 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 $I : V \times S \rightarrow \mathbf{2}$ is the interpretation of variables[1]. For each variable v_i and state s, $I(v_i, s)$ is the value of v at s. For convenience, we view states as variable valuations, and write $s(i)$ for $I(v_i, s)$. Let s and s' denote valuations of all unprimed and primed variables, respectively. The relation R is the set of all pairs (s, s') of states such that s and s' satisfy at least one of the rules of Δ for every variable in V.

For example, suppose the traffic light system only has variables Sensor and Light and rules

$$\begin{aligned}\mathsf{Sensor}' &:= \neg\mathsf{Sensor} \vee \mathsf{Light}\\ \mathsf{Light}' &:= \mathsf{Sensor}\end{aligned}$$

The Kripke structure corresponding to it 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 evaluate to true. Thus, both Sensor and Light have to be true in the successor state, creating the self-loop at s_0. The other transitions are computed similarly.

Properties of a system $\mathcal{S}$ are formulated in a temporal logic such as CTL [12], and are evaluated in the states of the associated Kripke structure $\mathcal{K}_\mathcal{S}$. For instance, a formula EF Light (the light eventually becomes green) holds in a state s if and only if there exists a path in $\mathcal{K}_\mathcal{S}$ starting at s and containing a state where Light is true. A formula AG $(\mathsf{Sensor} \vee \mathsf{Light})$ (always sensor is tripped or the light is green), holds in state s if and only if in every state, along every path starting at s, at least one of Sensor or Light is true. Each CTL formula evaluates to true or false in any state of a Kripke structure. For instance, on the structure of Figure 1(a), the former formula is true in all states, whereas the latter is true in states s_0, s_1, s_3 and false in state s_2. Often, some state of a Kripke structure is designated as *initial*. In this case, we say that a formula holds in a Kripke structure to mean that it holds in its initial state.

A system modeled as above is a composition of a machine and its environment, which in our example are the light controller and the sensor, respectively. Therefore, some of the system rules describe the behavior of the environment, and the others, that of the machine. A property of the system is *guaranteed by the environment* if its satisfaction depends solely on the rules of the environment. For instance, if the environment rule in our example were $\mathsf{Sensor}' := \text{true}$, then in any state where Sensor is true, the property AG $(\mathsf{Sensor} \vee \mathsf{Light})$ would be guaranteed by the environment. Here, it is obvious that the environment alone guarantees the property; in real-life systems, however, such as the one considered in Section 4, the intricate logic may hinder the easy detection of such environment guarantees. Therefore, we need a systematic and automated way of detecting these guarantees. To do so, we need to clarify a few points: (1) How to model the environment alone; (2) What it means for the environment model to guarantee a property; and, (3) How to check whether the environment guarantees a property. In the rest of this section, we elaborate on each of these points.

What is the environment? In specification of reactive systems, the boundary between the model and the environment is typically well defined (see, *e.g.*[37]). However, this boundary is often blurred during the verification: the environment and the model are simply specified using a collection of rules. In what follows, we use these rules to reconstruct this boundary.

We assume that the set of rules that describe the environment behaviour constitutes the model of the environment. Intuitively, every variable v that has a corresponding envi-

[1] Kripke structures used for model-checking also have initial states. We introduce them later in this section.

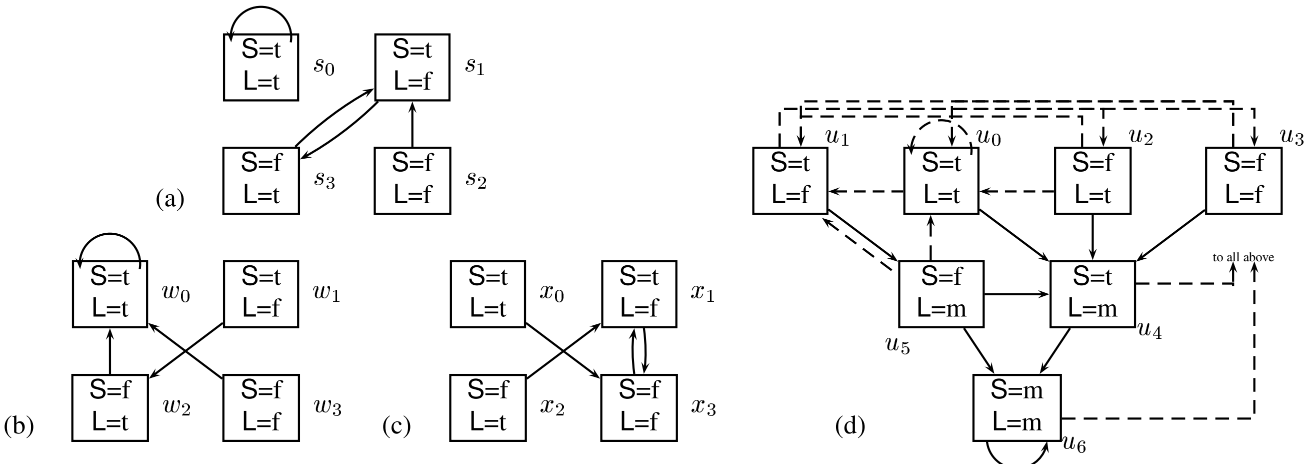


Figure 1: An example Kripke structure (a). A four-valued Kripke structure of an open system (d), and its two completions (b) and (c).

.

ronment rule $v' := \delta$ is *controllable* by the environment, *i.e.*, the environment chooses when and how to assign a value to v. On the other hand, the variables for which the environment has no rules but which it depends on, are *not controllable* by it. In our example, the only environment rule is $\mathsf{Sensor}' := \neg\mathsf{Sensor} \vee \mathsf{Light}$. Therefore, Sensor is controllable by the environment whereas Light is not.

Systems considered in model-checking are closed, *i.e.*, the system controls all of its variables. For example, the combination of the sensor and the light controller for our example, depicted in Figure 1(a), is a closed system. On the other hand, the rules of the environment alone do not specify changing of the non-controllable variables. Therefore, the environment is an open system.

When does the environment guarantee a property? We would like to say that this happens if the property is satisfied by the environment alone, *i.e.*, regardless of how its non-controllable variables are set. Since satisfaction of temporal properties is not defined for open systems, in what follows we "close" them, and then use the standard semantics of temporal logic over the resulting closed system.

In order to close an open system, we need to provide rules for all non-controllable variables. Moreover, we should not provide any additional rules for variables which such a system already controls. Formally, a closed system $\mathcal{S}'$ with rules Δ' is a *closure* of an open system $\mathcal{S}$ with rules Δ, if $\Delta \subseteq \Delta'$, where (a) rules are identified modulo logical equivalence, and (b) the rules in $\Delta' \setminus \Delta$ are for all and only the non-controllable variables of $\mathcal{S}$.

Note that an open system many have many closures. In our example, one closure was shown earlier, where rule $\mathsf{Light}' := \mathsf{Sensor}$ is added to the environment. We can construct another closure by adding rule $\mathsf{Light}' := \neg\mathsf{Sensor}'$ to the environment. We say that an open system $\mathcal{S}$ satisfies a temporal property φ at a state s if and only if *all of its* closures satisfy φ at s.

We say that an environment $\mathcal{E}$ guarantees a property φ at a state s if $\mathcal{E}$, as an open system, satisfies φ at s. For an environment, the rules added to close it represent the behavior of a possible machine in that environment. Intuitively, our notion of "environment guarantee" means that regardless of the machine the environment is combined with, the resulting closed system still satisfies the property.

How to check for environment guarantees? We cannot simply generate and model-check all closures of the environment, since there are exponentially many of them. To solve this problem, we need another representation of an open system that implicitly encodes all of its closures, and a model-checking algorithm over that representation that checks all closures at once. We present them in the next section.

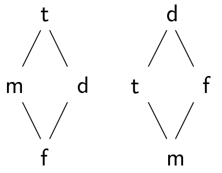
3 Modeling and Verifying Open Systems

In this section, we show how to represent open systems as state-transition graphs and reason about them directly using model-checking.

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 guarantees neither the property, nor 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 environment from Section 2 where the only rule is $\mathsf{Sensor}' := \neg\mathsf{Sensor} \vee \mathsf{Light}$. Suppose we want to check whether for any state where the sensor is off, in all its next states the sensor is on, or $\psi_1 = AG\ (\neg\mathsf{Sensor} \Rightarrow AX\ \mathsf{Sensor})$ in CTL. We check this in a state where both the sensor and the light are on. Note that the rule of the environment guarantees that this 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



(a) (b)

$\wedge$	t	f	m	d
t	t	f	m	d
f	f	f	f	f
m	m	f	m	f
d	d	f	f	d
$\neg$	f	t	m	d

(c)

Figure 2: Belnap logic **4**: (a) truth ordering, (b) information ordering, (c) truth table.

sensor is off, it remains off for one more time step, or $\psi_2 = AG$ ($\neg$Sensor $\Rightarrow$ AX $\neg$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) Light$'$:= true, in the other, with (2) Light$'$:= false. With (1), in any state, after one time step, 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 would like the property to evaluate on just the environment as "unknown", meaning that the environment by itself does not have enough knowledge to satisfy or refute the property.

Thus, classical Kripke structures are not appropriate for modeling open systems because reasoning about them is limited to two values. Instead, structures defined using multi-valued logics have been employed for this task [7, 16]. In our approach, we use the four-valued Belnap [6] logic pictured in Figure 2. We denote by **4** the set $\{\mathsf{t}, \mathsf{f}, \mathsf{m}, \mathsf{d}\}$; t and f stand for "known to be true" and "known to be false", respectively; m (maybe, or unknown) represents the fact that there is not enough evidence available to decide truth or falsity; and d represents too much information, that there is enough evidence to decide both truth and falsity. Their information content defines an *information ordering* (shown in Figure 2(b)) on these values, denoted by $\preceq$: $\mathsf{m} \preceq \mathsf{t}, \mathsf{f}$ and $\mathsf{t}, \mathsf{f} \preceq \mathsf{d}$. This ordering is also reflexive and transitive.

The usual boolean operations need to be extended to **4**. For example, to compute the conjunction of m with t, we reason as follows: m may be resolved to true, in which case the result is true, or to false, and then the result is false. Thus the result may be true or false, so it is m. Figure 2 presents a table for computing conjunction and negation of values of this logic. These operations can actually be computed on the *truth ordering* of these values that is also pictured in Figure 2(a), by using greatest lower bound for conjunction and symmetry for negation. We denote by **3** the subset $\{\mathsf{t}, \mathsf{f}, \mathsf{m}\}$ of **4**, and by **2** the boolean subset $\{\mathsf{t}, \mathsf{f}\}$.

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 our four-valued logic so that we satisfy these requirements.

The four-valued Kripke structure for our example open system is shown in Figure 1(d). This structure captures the interaction of the environment with all possible machines. Uncontrollable variables have values 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. A d transition indicates local environment guarantees. For example, the d transitions from u_2 and u_3 to u_4 in Figure 1(d) 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. Note that the light is unknown in u_4 since the environment cannot guarantee anything about it, as the light is not under its control. 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 the sensor to become on.

We now present the formal definition of four-valued Kripke structures. For an n-tuple s, we denote by $s(i)$ its ith component. A four-valued Kripke structure $\mathcal{M}$ over a set $V = \{v_1, \ldots, v_n\}$ of variables is a tuple $\langle S, I, T \rangle$, where S is the set of states, consisting of all possible n-tuples of values from **3**; $I : V \times S \to S$ is the interpretation of the variables that associates to every variable, in every state, a value from **3**, namely, for every $s \in S$ and $1 \leq i \leq n$, $I(v_i, s) = s(i)$; and $T : \mathbf{3}^n \times \mathbf{3}^n \to \mathbf{4}$ is a **4**-valued transition relation.

Given an open system described by a set of rules Δ, its associated four-valued Kripke is obtained as follows. Let s and s' be three-valued states. A transition (s, s') is d if for all variables v_i that are boolean in s', s and s' make the ith rule true; it is m if s and s' do not make all of the rules for any variable false, and s' is boolean. The structure in Figure 1(d) has been constructed using this algorithm.

Checking open systems. An interpretation of CTL formulas over multi-valued Kripke structures and a corresponding model-checking algorithm have already been defined [9], 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 in CTL as EX Light, is model-checked in state u_5 of the structure in Figure 1(d).

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 is given by the boolean formula

$$\bigvee_{s' \in \mathbf{2}^n} R(s, s') \wedge s'(\mathsf{Light}),$$

where $R(s, s')$ is true if and only if $(s, s') \in R$, and $s'(\mathsf{Light})$ denotes the value of Light in state s'. The same formula is used in the four-valued case, where the operations involved are over $\mathbf{4}$, and for our example we get:

$(\mathsf{m} \wedge \mathsf{t})$	// transition u_5u_0
$\vee(\mathsf{m} \wedge \mathsf{f})$	// transition u_5u_1
$\vee(\mathsf{d} \wedge \mathsf{m})$	// transition u_5u_6
$\vee\mathsf{f}$	// all missing transitions

which evaluates to m. This is expected, because it can be seen that 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 $p \vee EX\ (p \vee EX\ (\ldots))$, 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 our properties ψ_1 and ψ_2 on the structure of Figure 1 using this model-checking 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. In what follows, we formally establish correctness of our method.

Correctness.

Our method for detecting whether an environment guarantees a property is as follows. Given the environment rules,

1. construct the associated four-valued Kripke structure;
2. use the multi-valued model-checking algorithm to check the property on this structure; and,
3. if the algorithm answers t, the property is guaranteed by the environment.

To show that our method 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. We showed in Section 2 that each closed system is mapped to a classical Kripke structure. Thus, for every closure of the environment there is such a structure. For example, the structures for the two closures in our example are shown in Figure 1, parts (b) and (c). It can be shown that these structures are "compatible" with the four-valued structure. This compatibility is intuitively defined as follows. A three-valued state is compatible to all boolean states that are componentwise more informative than itself. 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 from the three-valued state. The transitions matching means that the destinations of those transitions are compatible. We can verify compatibility of the structures in Figure 1 parts (b) and (c) with the four-valued structure in Figure 1(d). For example, state u_5 is compatible with w_3. The d transition u_5u_4 is matched by w_3w_0, since u_4 and w_0 are compatible. The d transition u_5u_6 can also be matched by w_3w_0 since u_6 is also compatible with w_0. Conversely, transition w_3w_0 is matched by the m transition u_5u_0.

Any classical Kripke structure compatible to a four-valued Kripke structure is called its *completion*. Formally, we define completions as follows. We first define state compatibility by extending the information ordering to tuples of values componentwise. For any $u \in \mathbf{3}^n$ and $w \in \mathbf{2}^n$, we say that w is more informative than u and write $u \preceq w$ if, for all $1 \leq i \leq n$, $u(i) \preceq w(i)$. A classical Kripke structure $\mathcal{K}$ is a *completion* of a four-valued Kripke structure $\mathcal{M}$ if for any $u \in \mathbf{3}^n$ and $w \in \mathbf{2}^n$, $u \preceq w$ implies:

1. for all $u' \in \mathbf{3}^n$ such that $T(u, u') \succeq \mathsf{t}$, there exists $w' \in \mathbf{2}^n$ such that $u' \preceq w'$ and $(w, w') \in R$, and
2. for all $w' \in \mathbf{2}^n$ such that $(w, w') \in R$, there exists $u' \in \mathbf{3}^n$ such that $u' \preceq w'$ and $T(u, u') \preceq \mathsf{t}$

The structures in Figure 1 parts (b) and (c) are thus completions of that in Figure 1(d). In fact, it can be shown that all classical structures corresponding to the closures of an open system are completions of the four-valued structure associated to the system. This is formalized by the following theorem.

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

From this it follows that if all completions of the four-valued Kripke structure satisfy a property, then all closures of the open system do so. To complete our soundness argument, it is enough to note that the multi-valued model-checking algorithm has the following property: if on a given structure and a given property its answer is t, then all completions of that structure satisfy the property. We are now ready to state formally the soundness of our method.

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

Note that 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.

Discussion.

The multi-valued model-checking algorithm, as described above, has been implemented in the tool χChek [10]. Thus we can use χChek to carry out our method by directly model-checking the four-valued models of environments. Alternatively, we can use the procedure described in [21] to reduce the multi-valued model-checking problem to two classical ones. This allows the use a classical model-checker like NuSMV [11]. It was also shown in [21] that truth of universal properties can be decided by restricting the model-checking algorithm only to the m transitions. Note that in the four-valued structures we use to model open systems all m transitions go into boolean states. Thus, the reachable state space of a structure restricted to those transitions is completely boolean. Moreover, this boolean structure corresponds to a composition of the environment with the machine that changes its variables nondeterministically. Such a composition is trivial to construct syntactically from the rules of the environment. We simply add, for each machine variable v, both of the rules $v' :=$ true and $v' :=$ false. This allows us to easily apply our technique to check environment guarantees of universal properties on existing SMV models; we use this insight in our case study described in the following section.

4 Case Study: Checking the TCAS II System

We illustrate our approach with the Traffic Collision Avoidance System, TCAS II [36]. TCAS II implements a protocol for conflict detection and resolution between an aircraft 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 [30] and formal verification [33, 32, 1].

An SMV model of TCAS II has been translated from RSML [30] by Chan *et al.* [1, 8] and is part of the current NuSMV distribution [11]. 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. On the basis of 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 will view Own Aircraft as the machine 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. We have checked these, as well as a few additional properties of this model, using our method. 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 [33]): this is essential for ensuring that Own Aircraft has predictable behavior and does not decide on different maneuvers under similar conditions. We checked that non-determinism is not attained using a macro `ND_Composite_RA` defined in the model to encode possible non-determinism (see row 1 of Table 1). We performed the check on the original model 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 (see 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 which we have noticed before, namely, 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 (see row 3 of Table 1). Using Dwyer *et al.*'s property patterns [17], we expanded `OA_Not_Evaluated_Meantime` as

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

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

Table 1: Results of checking properties of TCAS II.

and similarly `CRA_Not_Changed_Meantime` as

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

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 environmental assumption, rooted in the semantics of RSML: environment variables cannot change without being evaluated by the machine. Discovering such assumptions is important [19, 14] in order to make verification experts aware of those conditions under which their analysis is valid.

Finally, we checked one of the original properties of the model [8]. 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 machine. A vacuity check [28, 23, 18], however, indicates that the antecedent of the implication is vacuous, potentially misleading the user into thinking that something is wrong.

Chan notes in [8] 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 he does not provide an additional explanation. Our method helped us identifying the reason for these anomalies: the logic for computing the escape maneuver *does not* depend on the machine, whereas that for computing its strength *does*.

Note that all properties summarized in Table 1 are true universal properties. On these properties, we have observed that 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.

5 Related Work

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 [4]. 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 machine. 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 [28, 5, 2, 22, 23, 34], and several tools [38, 18] 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. So, the vacuity detection algorithms find problems where they do not exist. 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.

Our initial experience pointed out that properties guaranteed by the environment are vacuous in all of machine-controlled variables (consider, for example, property φ, defined in Section 1, when sensors are never tripped). Unfortunately, the converse is not true. For example, not every property that contains environment variables only, hence trivially vacuous in all machine variables, is guaranteed by the environment. Thus, there does not seem to be a *syntactic* restriction on answers given by vacuity detectors that necessarily leads to errors.

For a class of properties, namely, universal temporal properties which hold in the model, we can run the environment in parallel with the "most general machine" [19], *i.e.*, the machine that nondeterministically changes variables under its control. If such a combination still preserves the property, then it will hold no matter which system is being used. However, when a universal property fails in the system, it is not clear how to check whether this failure is due to the environment. Similarly, there is no algorithm for checking environment guarantees over a general class of temporal logic formulas.

In this paper, we have shown that Kripke structures based on 4-valued Belnap logic can be used to approximate open systems. Our algorithm is incomplete, *i.e.*, in some cases it may fail to detect that a property is guaranteed by the environment. An alternative approach, called Module Checking, was proposed by Kupferman and Vardi [27]. This approach is known to be complete; however, it is much more expensive than model-checking. This is not surprising, since a temporal property is guaranteed by an environment that has no controllable variables iff it is valid, and checking validity of CTL formulas is known to be exponentially more expensive than model-checking. Godefroid [20] 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 machine 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 machine can interact with the environment, but cannot deter its progress.

The work of [40] is the closest to ours in spirit: determining which part of the model is necessary to ensuring 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, their algorithm is efficient but not complete, and they report of several overconstraints that were not detectable by it.

6 Conclusion and Future Work

In this paper, we argued for the need to provide support for debugging environments used in checking software systems. Specifically, we noted that when the environment single-handedly guarantees properties which are expected of the entire system, *i.e.*, it provides *environment guarantees*, then either the property should be labeled as that of the environment (in the terminology of Parnas's 4-variable model, a property of NAT rather than REQ [37]), or there is an error in the model of the environment. We have argued that such cases can be found if we look at the environment as an open system, and described a method for checking open systems directly.

We also discussed how to construct the open models of the environment from rule-based descriptions of state-machine models, such as those created by SMV specifications. Our experience with a model of the TCAS II system showed that environmental 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 believe that our work opens a number of questions related to debugging models of the environment, and name just a few below:

1. We assumed that every variable is either controllable by the environment or by the system (but not both), *i.e.*, the environment and the system 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 need 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 systems, *i.e.*, via communication channels, that lead to environmental guarantees. We intend to study this problem in future work.
3. We also assumed that there is a clear separation between the system 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 system is composed with multiple environments, or determining what constitutes an environment is difficult. One potential direction to remedy these problems is to follow the approach of Shlyakhter et. al [40], 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 model 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.

REFERENCES

[1] R. J. Anderson, P. Beame, S. Burns, W. Chan, F. Modugno, D. Notkin, and J. Reese. "Model Checking Large Software Specifications". In *Proceedings of FSE'96*, October 1996.

[2] 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, Colorado, USA, July 2003. Springer-Verlag.

[3] D. Beatty and R. Bryant. "Formally Verifying a Microprocessor Using a Simulation Methodogoly". In *Proceedings of Design Automation Conference'94*, pages 596–602, 1994.

[4] I. Beer, S. Ben-David, C. Eisner, and Y. Rodeh. "Efficient Detection of Vacuity in ACTL Formulas". In

Proceedings of CAV'97, volume 1254 of *LNCS*, pages 279–290. Springer-Verlag, 1997.

[5] 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.

[6] N. Belnap. "A Useful Four-Valued Logic". In *Modern Uses of Multiple-Valued Logic*, pages 30–56. 1977.

[7] 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, Trento, Italy, 1999. Springer.

[8] W. Chan, R. Anderson, P. Beame, S. Burns, F. Modugno, and D. Notkin. "Model Checking Large Software Specifications". *IEEE Transactions on Software Engineering*, 24(7):498–520, July 1998.

[9] M. Chechik, B. Devereux, S. Easterbrook, and A. Gurfinkel. "Multi-Valued Symbolic Model-Checking". *ACM Transactions on Software Engineering and Methodology*, 12(4):1–38, October 2003.

[10] 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.

[11] A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri. "NuSMV: a new Symbolic Model Verifier". In N. Halbwachs and D. Peled, editors, *Proceedings of CAV'99*, volume 1633 of *LNCS*, pages 495–499, Trento, Italy, July 1999. Springer.

[12] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.

[13] E. M. Clarke, D. E. Long, and K. L. McMillan. "Compositional Model Checking". In *Proceedings of LICS'89*, pages 464–475, June 1989.

[14] J. M. Cobleigh, D. Giannakopoulou, and C. S. Pasareanu. "Learning Assumptions for Compositional Verification". In *Proceedings of TACAS'03*, volume 2619 of *LNCS*, pages 331–346, April 2003.

[15] F. Copty, A. Irron, O. Weissberg, N. Kropp, and G. Kamhi. "Efficient Debugging in a Formal Verification Environment". In *Proceedings of CHARME'01*, volume 2144 of *LNCS*, pages 275–292, UK, September 2001. Springer.

[16] 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.

[17] M. Dwyer, G. Avrunin, and J. Corbett. "Patterns in Property Specifications for Finite-State Verification". In *Proceedings of 21st International Conference on Software Engineering*, May 1999.

[18] M. Gheorghiu, A. Gurfinkel, and M. Chechik. "VaqUoT: A Tool for Vacuity Detection". CSRG Technical Report, University of Toronto, April 2005.

[19] D. Giannakopoulou, C. S. Pasareanu, and H. Barringer. "Assumption Generation for Software Component Verification". In *Proceedings of the 17th IEEE International Conference on Automated Software Engineering (ASE 2002)*, pages 3–12, Edinburgh, UK, 2002. IEEE Computer Society.

[20] P. Godefroid. "Reasoning about Abstract Open Systems with Generalized Module Checking". In *Proceedings of 3rd Conference on Embedded Software (EMSOFT'03)*, volume 2855 of *LNCS*, pages 223–240. Springer-Verlag, October 2003.

[21] A. Gurfinkel and M. Chechik. "Multi-Valued Model-Checking via Classical Model-Checking". In *Proceedings of 14th International Conference on Concurrency Theory (CONCUR'03)*, volume 2761 of *LNCS*, pages 263–277, September 2003.

[22] A. Gurfinkel and M. Chechik. "Extending Extended Vacuity". In *Proceedings of 5th International Conference on Formal Methods in Computer-Aided Design (FMCAD 2004)*, volume 3312 of *LNCS*, pages 306–321, Austin, Texas, November 2004. Springer.

[23] A. Gurfinkel and M. Chechik. "How Vacuous Is Vacuous?". In *Proceedings of TACAS'04*, volume 2988 of *LNCS*, pages 451–466, Barcelona, Spain, March 2004. Springer.

[24] M. P. Heimdahl. "Lessons from the Analysis of TCAS II". In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA'96)*, San Diego, CA, January 1996.

[25] International Atomic Energy Agency. "Investigation of an Accidental Exposure of Radiotherapy Patients in Panama". Report of a Team of Experts, May 2001.

[26] M. Jackson. "The Meaning of Requirements". *Annals of Software Engineering*, 3:5–21, 1997.

[27] O. Kupferman and M. Vardi. "Module Checking". In *Proceedings of CAV'96*, volume 1102 of *LNCS*, pages 75–86, 1996.

[28] O. Kupferman and M. Vardi. "Vacuity Detection in Temporal Model Checking". *International Journal on Software Tools for Technology Transfer (STTT)*, 4(2):224–233, February 2003.

[29] N. Leveson. *Safeware: System Safety and Computers*. Addison-Wesley, 1995.

[30] N. Leveson, M. Heimdahl, H. Hildreth, and J. Reese. "Requirements Specification for Process-Control Systems". *IEEE Transactions on Software Engineering*, 20(9):684–707, September 1994.

[31] N. Leveson and C. Turner. "An Investigation of the Therac-25 Accidents". *IEEE Computer*, 26(7):18–41, July 1993.

[32] J. Lygeros and N. Lynch. "On the Formal Verification of the TCAS Conflict Resolution Algorithms". In *Proceedings of the 36th IEEE Conference on Decision and Control*, San Diego, CA, December 1997.

[33] M.P.E.Heimdahl and N.G.Leveson. "Completeness and Consistency in Hierarchical State-Based Requirements". *IEEE Transactions on Software Engineering*, SE-22(6):363–377, June 1996.

[34] 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. Springer, 2004.

[35] P. Neumann. *Computer-Related Risks*. Addison-Wesley ACM Press, 1995.

[36] U. S. D. of Transportation. "Introduction to TCAS II". Federal Aviation Administration.

[37] D. L. Parnas and J. Madey. "Functional Documents for Computer Systems". *Science of Computer Programming*, 25:41–61, 1995.

[38] M. Purandare and F. Somenzi. "Vacuum Cleaning CTL Formulae". In *Proceedings of CAV'02*, volume 2404 of *LNCS*, pages 485–499. Springer, July 2002.

[39] R. Cook and M. O'Connor . "Thinking About Accidents and Systems". K. Thompson, H. Manasse, eds., Improving Medication Safety, to appear.

[40] I. Shlyakhter, R. Seater, D. Jackson, M. Sridharan, and M. Taghdiri. "Debugging Overconstrained Declarative Models Using Unsatisfiable Cores". In *Proceedings of ASE'03*, pages 94–105, October 2003.