Using Valued Booleans to Find Simpler Counterexamples in Random Testing of Cyber-Physical Systems

Citation for the original published paper (version of record):
Lindström Claessen, K., Smallbone, N., Lidén Eddeland, J. et al (2018). Using Valued Booleans to Find Simpler Counterexamples in Random Testing of Cyber-Physical Systems. IFAC-PapersOnLine, 51(7): 408-415. http://dx.doi.org/10.1016/j.ifacol.2018.06.333

(article starts on next page)

Available online at www.sciencedirect.com

ScienceDirect

IFAC PapersOnLine 51-7 (2018) 408–415

Using Valued Booleans to Find Simpler Counterexamples in Random Testing of Cyber-Physical Systems

Koen Claessen * **Nicholas Smallbone** * **Johan Eddeland** **,***
Zahra Ramezani ** **Knut Åkesson** **

* *Department of Computer Science and Engineering, Chalmers University of Technology, Gothenburg, Sweden (e-mail: {koen, nicsma}@chalmers.se*
** *Department of Electrical Engineering, Chalmers University of Technology, Gothenburg, Sweden (e-mail: {johedd, rzahra, knut}@chalmers.se*
*** *Volvo Car Corporation, Gothenburg, Sweden (e-mail: johan.eddeland@volvocars.com)*

Abstract: We propose a new logic of *valued Booleans* for writing properties which are not just true or false but compute how severely they are falsified. The logic is reminiscent of STL or MTL but gives the tester control over what severity means in the particular problem domain. We use this logic to simplify failing test inputs in the context of random testing of cyber-physical systems and show that it improves the quality of counterexamples found. The logic of valued Booleans might also be used as an alternative to the standard robust semantics of STL formulas in optimization-based approaches to falsification.

Keywords: Reachability analysis, verification and abstraction of hybrid systems; embedded computer control systems and applications; logical design, physical design, and implementation of embedded computer systems; supervision and testing; model-driven systems engineering.

1. INTRODUCTION

Automated systems typically consist of controllers that interact with a physical environment. These systems are becoming more complex and are, in many situations, also safety-critical. Therefore, rigorous methods are needed to establish that these systems behave according to given requirements. For finite-state systems, model-checking (Clarke et al., 2009) can be used to prove properties of the system. However, for systems that contain both discrete and continuous dynamics, i.e. hybrid systems, the problem of determining if a state is reachable is in general undecidable, as shown by Henzinger et al. (1995).

In software testing, as in cyber-physical systems testing, test suites are traditionally developed by hand. However, creating a comprehensive test suite is painstaking work and, unless the test suite is very large, bugs can easily slip through. A possible solution is to put the *computer* in charge of creating test cases. There are many methods for doing this (Anand et al., 2013). In this paper we consider *constrained random test case generation* as supported by the tool QuickCheck (Claessen and Hughes, 2000).

QuickCheck generates random test cases to attempt to falsify a *property* supplied by the tester. A property can be, for example, an invariant which must always hold during the execution of a system. Any test case that fails is reported as a *counterexample* to the property. The strength of QuickCheck is the sheer variety and amount of testing it does: random test cases often do things that no human tester would dream of trying, and thousands of test cases can easily be run. Because of this, QuickCheck is good at finding complex bugs which human testers miss (Arts et al., 2015; Hughes, 2016). Our hope is to translate this advantage to hybrid system testing.

Random testing may be good at triggering complex bugs, but the random counterexamples it finds usually contain a lot of irrelevant features and are therefore hard to understand for a human tester. Understanding *why* a test case fails is vital in any debugging process. After finding a failing test case, QuickCheck deploys a method called *shrinking* to reduce it to its bare essentials; after shrinking, all features present in the failing test case are needed to make the test case fail.

QuickCheck has until now mainly been used to test software. In this work we take the first steps towards adapting the QuickCheck approach to hybrid systems. In doing so, we encounter three main questions: (1) How do we randomly generate test cases suitable for use in a hybrid system? (2) How do we specify properties of a hybrid system in a way that is amenable to automated testing? (3) How do we simplify failing test cases in hybrid systems testing?

★ Research supported by Swedish Research Council (VR) project SyTeC VR 2016-06204, and Swedish Governmental Agency for Innovation Systems (VINNOVA) project TESTRON 2015-04893.

10.1016/j.ifacol.2018.06.333

The main focus of this paper is question (3). We found that simplifying a failing test case for a hybrid system typically resulted in a test case that still made a property fail, but would do so in a minimal, unconvincing way, which we refer to as a *glitch*. To solve this, we measure the severity of each failure, and never simplify a failing test case in a way that reduces its severity. To measure the severity of failing test cases, we define a new logic of *valued Booleans*, or VBools, which allows the tester to express severity information as part of a property.

1.1 Related work

Falsification of temporal logic properties is an emerging black-box approach to testing of hybrid systems. The tools S-TaLiRo (Annpureddy et al., 2011) and Breach (Donzé, 2010) use Metric Temporal Logic (MTL) and Signal Temporal Logic (STL). The expressiveness of MTL and STL is equivalent and the difference between them is that the predicates are explicitly stated in STL as opposed to in MTL. Because of this, we only introduce one of the two temporal logics in this paper, namely STL.

The main idea behind falsification, introduced by Fainekos and Pappas (2009), is to use a *robust semantics* (or *quantitative semantics*) of temporal logic to measure how far away a specification is from being broken. The robustness value of a temporal logic specification can thus be used as the objective function for an optimization problem where the goal is to falsify said specification. Previous work has proposed alternative definitions of robustness for temporal logic specifications (Akazaki and Hasuo, 2015) and the reasons for these changes (Eddeland et al., 2017).

The valued Booleans presented in this paper are related to robust semantics and can be seen as an alternative to them. The main difference between the two approaches is that VBools allow the tester to control how robustness is measured for each property, while a robust semantics imposes the same robustness measure on all properties. As is shown in Sections 3 and 4, different robustness measures make sense for different applications.

VBools and robust semantics are not closely related to fuzzy logic (Driankov et al., 1993), although they may appear to be at first glance. Fuzzy logics also augment truth values with numbers, but the purpose of those numbers is to express the certainty of a value being true or false. With VBools (and in STL/MTL), it is always clear whether or not a given value is true or false; the numerical aspect only expresses *how* true or false the value is.

1.2 Contributions

The main contributions of this paper are:

i) Adaptation of random testing with QuickCheck to hybrid systems;
ii) definition of VBools as a way to simplify counterexamples when testing hybrid systems;
iii) a comparison with existing falsification tools, to illustrate the strengths and weaknesses of random testing and the importance of simplifying counterexamples.

The rest of this paper is organized as follows: in Section 2 an example is used to introduce random testing, falsification and shrinking. In Section 3 valued Booleans are introduced including a comparison to signal temporal logic. In Section 4 the use of valued Booleans for shrinking counterexamples is presented for two example models, as well as a discussion of how the approach compares to Breach.

2. EXAMPLE

This section illustrates the difficulties we encounter when using QuickCheck as-is to test a hybrid system model; we will see how to solve them in Section 3. It also shows how falsification works on the same model. For ease of understanding we have chosen a linear system as the example, but since the presented testing approach is black-box, it can be applied to any hybrid system.

2.1 The model

The model considered is a heater which is controlled by a PID controller. The variables and parameters of the model are described in Table 1. The only input to the model is the setpoint temperature $r(t)$.

Table 1. The signals and parameters of the heater example.

Signal	**Meaning**	
r(t)	setpoint temperature	
l(t)	pump level	
h(t)	heater temperature	
y(t)	room temperature	
Parameter	**Meaning**	**Value**
OT	Outside Temperature	-5
BT	Boiler Temperature	90
HC	Heater Coefficient	0.1
OC	Outside Coefficient	0.05
K_p	Proportional gain	0.012
K_i	Integral gain	$1.144689 \cdot 10^{-4}$
K_d	Derivative gain	0.005

The error fed to the PID controller is $r(t) - y(t)$. The continuous dynamics of the heater are given by equations (1) and (2), where the initial conditions are $h(0) = y(0) = OT$. The time is measured in minutes and the temperature is measured in °C. The heater is simulated for 300 minutes, *i.e.*, the start time is $t = 0$ and the end time is $t = 300$. Implementation-wise, the models are discretized with a fixed sampling time and then simulated.

$$\dot{h}(t) = \frac{-(l + HC) \cdot h(t) + BT \cdot l(t) + HC \cdot y(t)}{1 + HC} \tag{1}$$

$$\dot{y}(t) = \frac{-(HC + OC) \cdot y(t) + HC \cdot h(t) + OC \cdot OT}{1 + HC + OC} \tag{2}$$

We would like to check the following property.

Property 1. If the setpoint temperature has been constant (steady) for 50 minutes, then the difference between the setpoint temperature and the actual room temperature should be at most 1°C.

2.2 Testing and shrinking with QuickCheck

QuickCheck tests properties, like the one above, on random test inputs; the tester has control over the distribution of test inputs. To test any piece of software with QuickCheck, the tester supplies:

- A *test generator*, which describes how to generate random test inputs for the software;
- a *property*, which is a function that takes such a test input and returns true or false, true to indicate that the test passed and false to indicate that it failed.

QuickCheck uses the generator to produce a large number of random test inputs, and reports an error if the property returns false for any of those test inputs.

In order to use QuickCheck on a hybrid system model, we first discretise the model to be able to test it. QuickCheck generates a random input signal, i.e. a sequence of numeric values, runs the model and evaluates the property, resulting in a pass or fail. The tester determines what sort of input signals should be generated by choosing a test generator.

For the heating system, the test input is a discrete signal representing r. The QuickCheck property applies the heater model to r to get the output signal y, and returns the value of the following formula:

$$\Box(\mathit{steady}(r) \implies |y - r| \leq 1) \tag{3}$$

assuming that *steady* has been defined appropriately. The $\Box$ operator is used in STL to denote a property that should hold globally from that point on; for a more formal semantics see Section 3.2.

As described above, we must also supply a test generator. Rather than generating random noise, which would be extremely unlikely to keep r constant for 50 minutes, we generate a piecewise constant function, choosing at random the total simulation time and the number, lengths and values of the constant pieces.

When we run QuickCheck, it discovers that the property is false, and finds the counterexample shown in Figure 1.

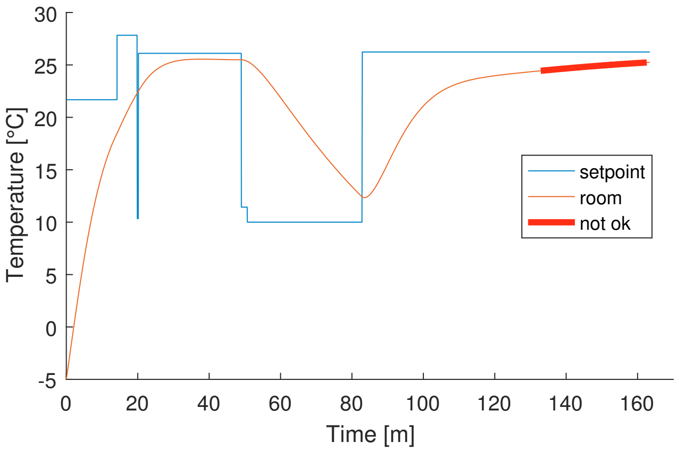

Fig. 1. Random testing performed on the heater example.

This counterexample clearly shows that the property does not hold, but it is also rather complicated. For this specific example, only the last two changes in setpoint are responsible for the failure, but the tester can only know this after a manual and time-consuming analysis of the whole test case. This is a typical problem with random testing, and in general it is often hard for the tester to discover why the system failed to satisfy the property, given a randomly-generated counterexample.

QuickCheck therefore simplifies the failing test case before presenting it to the tester, a process called *shrinking*. Shrinking works by applying a small simplification to the counterexample, such as removing part of the test input, and seeing if it still fails. If not, the simplification is undone and a different one tried. Shrinking continues until none of the simplification steps that are tried works.

After shrinking, QuickCheck finds the counterexample shown in Figure 2. This test input is clearly much simpler, but the room temperature is out of specification for only one simulation step (in the next simulation step, the room temperature would return to within one degree of the setpoint). The property still fails, but only barely. Shrinking has thus reduced a serious failure into something a tester may well consider merely a glitch.

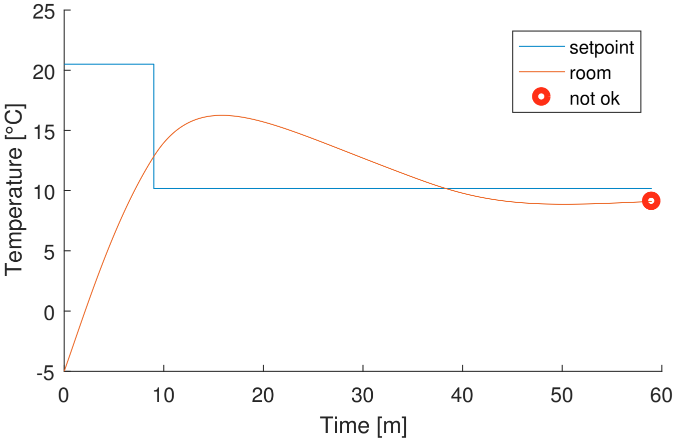

Fig. 2. Shrinking turns the counterexample into a glitch.

In software testing, random testing and shrinking are a powerful combination. Random test cases provoke bugs by exercising the software in unusual ways, but finds complicated counterexamples; shrinking reduces these counterexamples to their bare minimum, producing simple, easy-to-understand counterexamples.

In hybrid systems testing, shrinking produces simple counterexamples, but they are often mere glitches. This is because shrinking removes all irrelevant features of the test data, and a property is still falsified by test data that makes it only fail for a short interval. Fixing this problem is one of the main contributions of this paper.

2.3 Falsification with Breach

The standard falsification procedure used in Breach is illustrated in Figure 3. The *Generator* takes the input parametrization to generate an input to the system under test. The *Simulator* generates a simulation trace, which is used together with the requirement φ to evaluate the robustness function for the simulation. The robustness function ρ is evaluated to see whether the requirement is falsified or not. If it is not falsified, new parameters are sampled and the process is repeated. The *Parameter Optimizer* does this in one of two ways: in the first part of the process, the parameters are sampled in some structured way (global optimization), while in the second

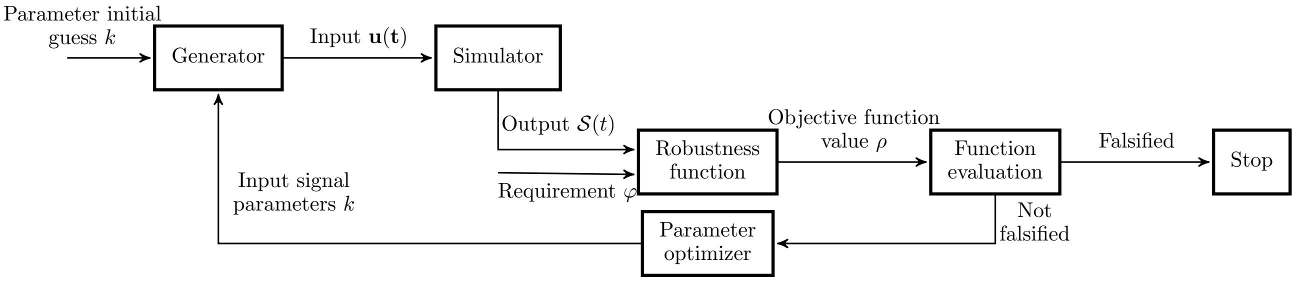

Fig. 3. A flowchart describing the optimization-based falsification procedure of Breach.

part, new parameters are found by means of optimization (local optimization). The algorithm is described in more detail by Donzé (2010).

For the heater example, we choose an input generator that generates piecewise constant signals, changing signal values at 9 different times. This means that the optimization problem has 19 parameters: 10 signal values, each in the range $[10, 25]$, and 9 time specifications in the range $[1, 80]$. Every time the signal changes, the new value will hold for between 1 and 80 minutes. Each time the system is simulated, the total simulation time is 300 minutes.

The system is modeled in Simulink, and the STL specification is

$$\varphi = \Box_{[0,250]}(steady \Rightarrow abs(y(t+50) - r(t+50)) < 1), \quad (4)$$

where $steady = \Box_{[0,49.99]}(|x(t + 0.01) - x(t)| < \epsilon)$ for some small $\epsilon > 0$.

Breach is able to falsify the system, with resulting outputs shown in Figure 4. The counterexample is reminiscent of the one in Figure 1 – the failure of the specification is clear, but the test case is quite complicated.

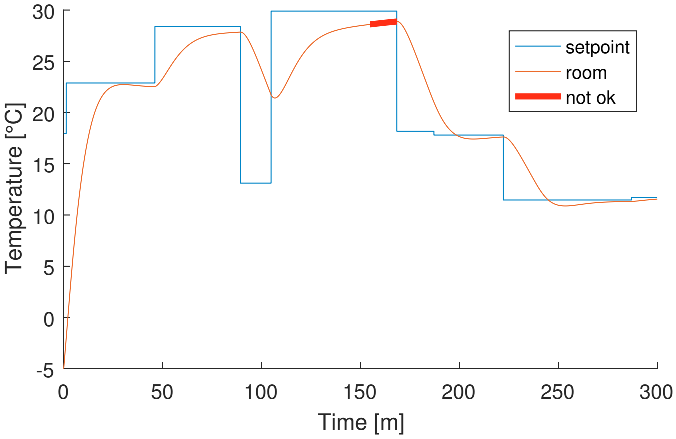

Fig. 4. Falsification of the heater property. The property is false near $t = 150$. The complexity of the test case is highly dependent on the input generator chosen.

3. APPROACH

In Section 2, we saw that shrinking turns counterexamples into glitches. This is a serious problem, as random testing without shrinking can produce very complicated counterexamples. Our approach to solve this is as follows:

- The tester will specify how to compute the *severity* of any counterexample to their property.
- During shrinking, QuickCheck will not perform any simplification step which reduces the severity of the counterexample.

The hope is that this approach will give us counterexamples as severe as Figure 1 but as simple as Figure 2.

For the heater model, the tester might consider the severity to be the integral of the part of the error above 1℃,

$$\int |y(t) - r(t)| - 1\, dt, \quad (5)$$

integrated over the time intervals in which the temperature has been constant for 50 minutes but the error is over 1℃.

If we define those time intervals formally, we will see that the severity formula almost exactly duplicates what was written in the property. This means the tester has to write everything down twice, once as a property and once as a severity formula. To avoid this problem, we now introduce *valued Booleans*, which allow the tester to express severity information as part of a property, rather than separately.

3.1 Valued Booleans

A valued Boolean, or *VBool*, is a Boolean value together with a *robustness* value, a non-negative real number that indicates how true or false the VBool is. VBool formulas look just like Boolean formulas, with the addition of extra annotations that describe how to compute the robustness. An important property of VBools is that the annotations do not affect the Boolean value but only the robustness.

Property 3 can be written using VBools as

$$\Box_{+}(steady(r) \Longrightarrow_{\mathrm{v}} |y - r| \leq_{\mathrm{v}} 1). \quad (6)$$

All we have done is to replace $\Box$ with $\Box_{+}$, $\Longrightarrow$ with $\Longrightarrow_{\mathrm{v}}$ and $\leq$ with $\leq_{\mathrm{v}}$. As mentioned above, this property has the same Boolean value as property 3. We shall see later that when this property fails, its robustness is precisely the value of the integral we saw above.

Formally, a VBool is a pair of a Boolean value and a robustness value, which is a non-negative number:

$$\mathbb{V} = \mathbb{B} \times \mathbb{R}_{\geq 0}$$

The robustness of a VBool represents *how much the test result would have to change* for the Boolean value to change. For a false VBool this roughly coincides with the severity of the failure, and for a true VBool it roughly coincides with how convincingly the test passed.

The comparison operator $\leq_{\mathrm{v}}$ corresponds to $\leq$, and takes the difference between its arguments as its robustness.

This is because, in order for the value of $x \leq y$ to change, one of the arguments has to change by at least $|x-y|$.

$$\leq_{\mathrm{v}} : \mathbb{R} \times \mathbb{R} \to \mathbb{V}$$

$$x \leq_{\mathrm{v}} y = \begin{cases} (\top, y-x) & \text{if} x \leq y \\ (\bot, x-y) & \text{otherwise} \end{cases}$$

The other comparison operators are defined in terms of $\leq_{\mathrm{v}}$. $\top$ and $\bot$ denote true and false, respectively.

We define conjunction $\wedge_+$ as follows. The Boolean part of $x \wedge_+ y$ is computed as $x \wedge y$. If both x and y are false, then we add their robustnesses. This is because in order for $x \wedge_+ y$ to become true, both x and y must become true.

$$(\bot, x) \wedge_+ (\bot, y) = (\bot, x+y)$$

Similar reasoning shows that if exactly one of x and y is false, we should take the robustness of the false argument:

$$(\bot, x) \wedge_+ (\top, y) = (\bot, x)$$
$$(\top, x) \wedge_+ (\bot, y) = (\bot, y)$$

When x and y are both true, only one of them has to become false for $x \wedge_+ y$ to become false. As it is easier to make $x \wedge_+ y$ false than it is to make x false, its robustness should be lower; the same argument applies to y. The following formula, inspired by parallel resistance, captures this idea and leads to satisfactory algebraic properties:

$$(\top, x) \wedge_+ (\top, y) = \left(\top, \frac{1}{\frac{1}{x} + \frac{1}{y}}\right)$$

(When computing reciprocals, we adopt the convention that $1/0 = \infty$ and $1/\infty = 0$.)

The other Boolean operators are defined as follows:

$$\top_{\mathrm{v}} = (\top, \infty)$$
$$\bot_{\mathrm{v}} = (\bot, \infty)$$
$$\neg_{\mathrm{v}}(b, x) = (\neg b, x)$$
$$x \vee_+ y = \neg_{\mathrm{v}}(\neg_{\mathrm{v}} x \wedge_+ \neg_{\mathrm{v}} y)$$

If we want to take the conjunction of two VBools whose robustnesses may be wildly different, it can be useful to first scale the two robustness values. We can do that using the # operator, which multiplies the robustness of a VBool by a constant:

$$(b, x) \mathbin{\#} k = (b, x \cdot k)$$

Implication is defined non-classically, in order to give a penalty to trivially true implications:

$$x \Longrightarrow_{\mathrm{v}} y = \neg_{\mathrm{v}}(x \mathbin{\#} K) \vee_+ y$$

Here K is an arbitrary constant, 1000 in our system.

Finally, the modal operators such as $\Box_+$ are defined in terms of the operators we have already seen. For reasons of space we present them slightly informally. In a discrete setting, the semantics of $\Box$ can be defined roughly as follows, where ϕ is a formula parametrised on the simulation step:

$$\Box\phi = \bigwedge \{\phi(n) \mid 1 \leq n \leq N\}$$

The semantics of $\Box_+$ is then

$$\Box_+\phi = \left(\bigwedge\nolimits_+ \{\phi(n) \mid 1 \leq n \leq N\}\right) \mathbin{\#'} \delta t$$

where δt is the simulation step size, and $\#'$ is the following temporal variant of #:

$$(\bot, x) \mathbin{\#'} k = (\bot, x \cdot k)$$
$$(\top, x) \mathbin{\#'} k = (\top, x/k)$$

When $\Box_+\phi$ is false, its robustness is $\int r(t)\,dt$, where $r(t)$ is the robustness of ϕ at time t, over all time intervals where ϕ is false. When $\Box_+\phi$ is true its robustness is $1/\int 1/r(t)\,dt$. For example, (6) is equivalent to the formula

$$\bigwedge\nolimits_+ \{\mathit{steady}(r,n) \Longrightarrow_{\mathrm{v}} |y[n] - r[n]| \leq_{\mathrm{v}} 1 \mid 0 \leq n \leq N\} \mathbin{\#} \delta t$$

The reader can check that the robustness of this formula when false is equal to the integral given in (5) [1].

A variant of $\wedge$ The semantics of $\wedge_+$, $\vee_+$ and $\Box_+$ is not always what we want. For example, take an aircraft collision avoidance system, where the distance d between the two aircraft must never be under 1000m. This property can be expressed as $\Box_+(1000 \leq_{\mathrm{v}} d)$, which computes a robustness of $\int 1000 - d(t)\,dt$ integrated over the faulty intervals, i.e. it considers the amount of time the planes are too close as well as their distance. The tester, however, may decide that nearer misses are always worse.

To allow this, we introduce a second conjunction operator $\wedge_{\max}$. This is defined so that the conjunction of two false values takes the maximum of the two robustnesses:

$$(\bot, x) \wedge_{\max} (\top, y) = (\bot, x)$$
$$(\top, x) \wedge_{\max} (\bot, y) = (\bot, y)$$
$$(\bot, x) \wedge_{\max} (\bot, y) = (\bot, x \max y)$$
$$(\top, x) \wedge_{\max} (\top, y) = (\top, x \min y)$$

We define $\vee_{\max}$ and $\Box_{\max}$ the same as we did above, but replacing $\wedge_+$ with $\wedge_{\max}$, and not scaling by δt in the case of $\Box_{\max}$. The robustness of $\Box_{\max}\phi$, when false, is then the maximum robustness of ϕ at a time instant when ϕ is false. For the collision avoidance system, we can now write the property $\Box_{\max}(1000 \leq_{\mathrm{v}} d)$, whose robustness is the maximum value of $1000 - d$ over all failing time intervals, exactly what we want.

In the rest of this paper, we will refer to a property which uses $\wedge_+/\vee_+/\Box_+$ as having "+" semantics, while a property that uses $\wedge_{\max}/\vee_{\max}/\Box_{\max}$ has "max" semantics. Despite this terminology, properties can freely mix and match both sets of operators.

Properties VBools satisfy some, but not all, of the usual properties of Boolean algebra. The most important property is that the Boolean part of the VBool behaves exactly as in Boolean algebra; using VBools adds robustness information but does not change the logical meaning of the property.

The connectives $\wedge_+$ and $\vee_+$ are associative and commutative and have an identity and zero element. These properties mean that the conjunction or disjunction of a set of formulas is a well-defined notion, even when the set may be empty. The connectives are deliberately not idempotent because e.g. $x \wedge_+ x$ has a robustness twice that

[1] When a continuous signal is represented we write it as $s(t)$, but when the closed-loop system is simulated the system will be discretized by the solver and approximated as a discrete time signal. In situations where we would like to emphasize that it is a discrete time signal we use the notation $s[k]$.

of x: a predicate that occurs several times in the property contributes more to the severity.

Implication $x \Longrightarrow_{\text{v}} y$ does not satisfy the classical definition of being $\neg x \vee y$, but it does satisfy the property that $(x \Longrightarrow_{\text{v}} (y \Longrightarrow_{\text{v}} z)$ is equivalent to $(x \wedge_{+} y) \Longrightarrow_{\text{v}} z$. This means that properties with several preconditions behave as expected.

3.2 Comparison with Signal Temporal Logic

We now compare VBools with Signal Temporal Logic. The syntax of STL formulas is defined as follows (Donzé and Maler, 2010):

$$\varphi ::= \mu | -\mu | \varphi \wedge \psi | \varphi \vee \psi | \Box_{[a,b]}\psi | \Diamond_{[a,b]}\psi | \varphi \mathcal{U}_{[a,b]}\psi, \tag{7}$$

where the predicate μ is $\mu \equiv \mu(x) > 0$, ψ and φ are STL formulae; $\Box_{[a,b]}$ denotes the *globally* operator between a and b; $\Diamond_{[a,b]}$ denotes the *finally* operator between a and b; $\boldsymbol{U}_{[a,b]}$ denotes the *until* operator between a and b.

The semantics of STL are shown by considering the signal x at time t and the satisfaction relation $\models$, where $(x,t) \models \mu$ denotes that μ is true for signal value x at time t (Raman et al., 2014).

$$(x,t) \models \mu \quad \Leftrightarrow \quad \mu(x(t)) > 0 \tag{8}$$

$$(x,t) \models \neg\mu \quad \Leftrightarrow \quad \neg((x,t) \models \mu) \tag{9}$$

$$(x,t) \models \varphi \wedge \psi \quad \Leftrightarrow \quad (x,t) \models \varphi \wedge (x,t) \models \psi \tag{10}$$

$$(x,t) \models \varphi \vee \psi \quad \Leftrightarrow \quad (x,t) \models \varphi \vee (x,t) \models \psi \tag{11}$$

$$(x,t) \models \Box_{[a,b]}\varphi \quad \Leftrightarrow \quad \forall t' \in [t+a, t+b], (x,t') \models \varphi \tag{12}$$

$$(x,t) \models \Diamond_{[a,b]}\varphi \quad \Leftrightarrow \quad \exists t' \in [t+a, t+b], (x,t') \models \varphi \tag{13}$$

$$(x,t) \models \varphi\, \mathcal{U}_{[a,b]}\psi \quad \Leftrightarrow \quad \exists t' \in [t+a, t+b] \;\; (x,t') \models \psi \tag{14}$$
$$\wedge\, \forall t'' \in [t, t'], (x,t'') \models \varphi$$

A robust semantics of STL formulas is a real-valued function ρ of the signal x at time t.

$$\rho(\mu, x, t) \quad = \quad \mu(x(t)) \tag{15}$$

$$\rho(\neg\mu, x, t) \quad = \quad -\,\mu(x(t))) \tag{16}$$

$$\rho(\varphi \wedge \psi, x, t) \quad = \quad \min(\rho(\varphi, x, t), \rho(\psi, x, t)) \tag{17}$$

$$\rho(\varphi \vee \psi, x, t) \quad = \quad \max(\rho(\varphi, x, t), \rho(\psi, x, t)) \tag{18}$$

$$\rho(\Box_{[a,b]}\varphi, x, t) \quad = \quad \min_{t' \in [t+a, t+b]} \rho(\varphi, x, t') \tag{19}$$

$$\rho(\Diamond_{[a,b]}\varphi, x, t) \quad = \quad \max_{t' \in [t+a, t+b]} \rho(\varphi, x, t') \tag{20}$$

$$\rho(\varphi\, \mathcal{U}_{[a,b]}\psi, x, t) \quad = \quad \max_{t' \in [t+a, t+b]}(\min(\rho(\psi, x, t'), \tag{21}$$
$$\min_{t'' \in [t,t']} \rho(\varphi, x, t'')))$$

VBools and STL are in many ways quite similar. Indeed, the semantics of an STL formula is basically the same as the corresponding VBool formula in which we have used the "max" versions of every operator. [2] The reader might wonder why we have introduced VBools for shrinking, rather than using the existing robust semantics for STL.

The main reason is that STL imposes one particular definition of robustness, while VBools allow the tester to choose. The severity of a counterexample depends on the physical interpretation of the property being tested, so there is no single semantics that always fits best. We have seen that both the "+" and the "max" semantics are useful in different applications, depending on whether we want the integral of the severity or the maximum severity; in the next section we present some evidence that the semantics used makes a real difference to shrinking, and that shrinking based on VBools works better in some cases than shrinking based on the robust semantics for STL.

We do not suggest that "+" and "max" are the only reasonable semantics for VBools; our approach makes it relatively easy to add more variants as needed, as we encounter new sorts of properties with different physical interpretations and requirements.

We have a strong suspicion that using VBools in optimization based falsification, such as Breach and S-TaLiRo, may actually improve bug finding effectiveness. The reason is that in STL, a change in the robustness on one side of a $\wedge$ will only affect the result of that $\wedge$ if that side was the minimum already. In VBools, a change in any of the arguments of $\wedge_{+}$ will always affect its result. So, a black-box numerical optimization method gets more information from formulas built with $\wedge_{+}$ than in STL. However, we do not have enough experimental evidence yet to support this claim.

4. EVALUATION

In this section we: 1) compare random testing against falsification, chiefly using Breach; 2) compare VBools against STL in the context of shrinking; and 3) examine the quality of the counterexamples produced by VBool-aware shrinking.

The two models presented here are available in Simulink. Falsification with Breach, which is a MATLAB toolbox, is done directly in Simulink. However, to run QuickCheck for the models, we generate C code that is called through Haskell scripts. Because of this, the models in this section are run with a constant step time of 0.001 minutes (for the heater example) and 0.01 seconds (for the automotive transmission example). The complete code is available from `https://github.com/koengit/RealTesting`, subdirectory `simulink`.

For QuickCheck testing of the models, the test data generator works as follows. We generate piecewise linear input signals. The pieces can have any size, but are biased towards smaller sizes. Each piece is randomly chosen to be either a line or a constant. The line endpoints have a 10% chance of being extremal values, and a 90% chance of being chosen uniformly at random.

The shrinking steps QuickCheck tries on a piecewise linear function are:

- Removing a piece from the function;
- reducing a piece's duration;
- reducing the numerical values of a line endpoint;
- merging two adjacent pieces (x_1, y_1)–(x_2, y_2) and (x_2, y_2')–(x_3, y_3) into one piece (x_1, y_1)–(x_3, y_3);
- flattening a piece which is a line into a constant.

[2] Apart from the fact that the robustness 0 plays a special role in STL (it is neither true nor false), which in turn means that STL cannot express the difference between $<$ and $\leq$.

We intend the above design to be useful for random testing of hybrid systems in general, rather than a set-up specific to our examples.

4.1 The heater example

Figure 5 shows the result of shrinking a random counterexample for Section 2's heater property expressed using VBools with "+" semantics. The test case is now simple but has not become a glitch.

To achieve similar results in the falsification procedure, we would have to change our input generators to only generate signals that do not switch values many times. However, this also reduces signal expressivity, and thus might not be preferable when it is unknown which kind of faults or bugs exist in the system under test.

As described in Section 3, the semantics of STL is essentially the same as VBools with "max" semantics. We can therefore test how well an STL-based shrinking method would have worked by changing our property to use "max" semantics. When we do so, we get a counterexample which is cut off early: it stops at the instant in time where the maximum error value is reached. The resulting counterexample is not quite a glitch, but the property is often only false for a minute or two. If the error had happened to be maximal at the point in time where the property first became false, the resulting counterexample would have been shrunk into a glitch. This suggests that STL's robust semantics is not always appropriate for shrinking.

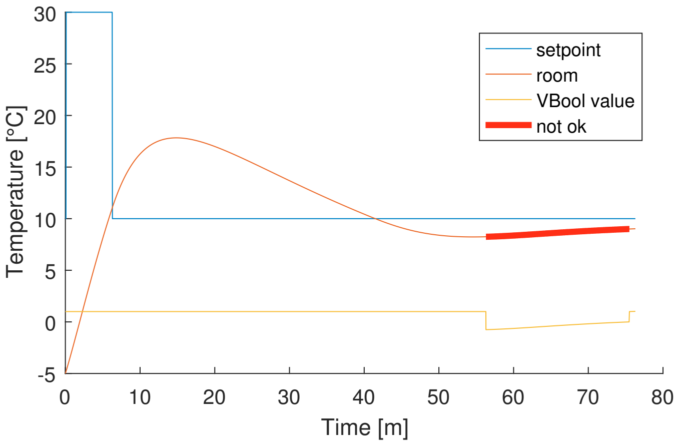

Fig. 5. Shrinking with valued Booleans. We have included the VBool value in the graph for clarity, even though it is not a temperature.

4.2 Automatic transmission example

The Automatic Transmission model was proposed as a benchmark by Hoxha et al. (2014) and is a version of a demo for the Simulink tool by Mathworks.

The model has one input, the throttle, which can vary in the interval $[0, 100]$. There are three outputs: the vehicle speed v in mph, the engine rotation speed ω in RPM, and the gear g. We attempt to falsify an augmented version $\varphi_{2'}$ of property ϕ_2^{AT} (Hoxha et al., 2014):

$$\varphi_{2'} = \Box((\omega < 4500) \vee (v < 120)). \qquad (22)$$

A counterexample of $\varphi_{2'}$ must make the speed greater than 120 mph and the engine rotation speed greater than 4500 RPM, *at the same time instant.* One way to falsify this is simply to run the vehicle at full throttle, and so both Breach and QuickCheck are able to falsify the property.

QuickCheck's counterexample is shown in Figure 6, in which the vehicle gets up to 160 mph and 5000 RPM. The counterexample before shrinking, which we omit for lack of space, was quite complicated. It also only got up to 140 mph and 4500 RPM – that is, in this case, shrinking not only preserved but *increased* the severity of the counterexample. The property used "+" semantics for robustness. If we use "max" semantics instead, mimicking STL, then shrinking does not manage to increase the severity of the counterexample, and furthermore, the test case is cut off the instant the vehicle reaches the desired speed and RPM.

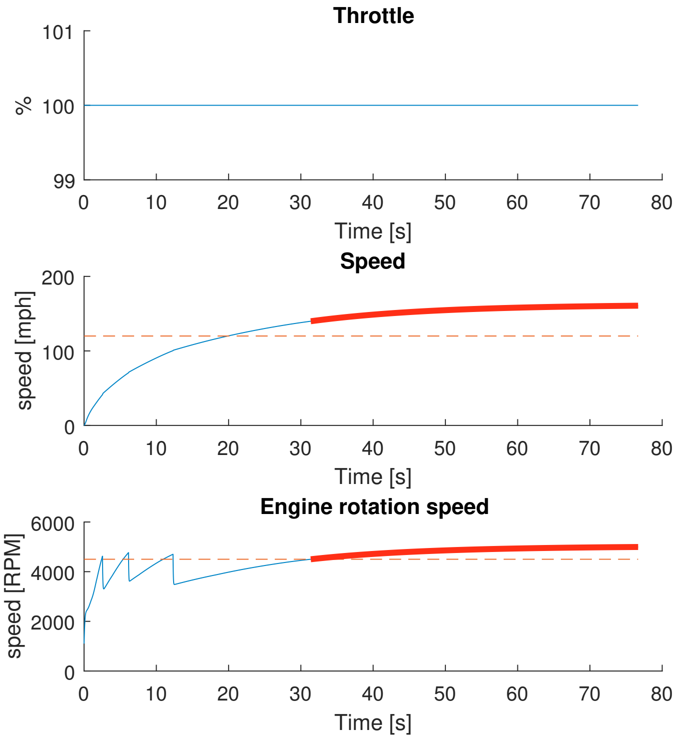

Fig. 6. The counterexample for $\varphi_{2'}$ from QuickCheck. The thick red lines indicate where the specification is broken, *i.e.*, where the speed is greater than 120 and the rotational speed is greater than 4500 at the same time.

QuickCheck is also able to falsify property φ_3^{AT}, which states that the transmission never switches from gear 2 to 1 to 2 within 2.5 seconds. The counterexample is shown in Figure 7. Finding the counterexample takes on average about 20 simulations, and the shrunk counterexample is indeed simpler than the original random one. Random testing seems to be good here, perhaps because complicated test cases are likely to falsify the property.

One property that we cannot falsify with QuickCheck is:

$$\varphi_6^{AT} = \neg(\Diamond_{[0,20]}(v > 120) \wedge \Box(\omega < 4000)), \qquad (23)$$

for which a counterexample means that engine speed ω is always less than 4000 RPM and the vehicle speed v goes above 120 within 20 seconds. The reason is that random

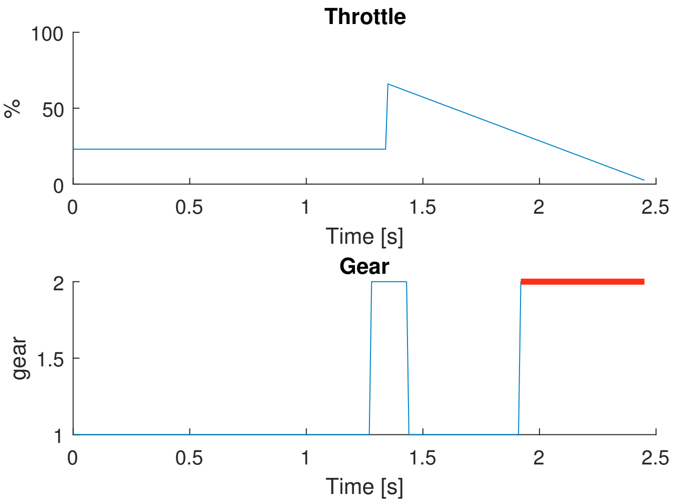

Fig. 7. The counterexample for φ_3^{AT} from QuickCheck. The thick red lines indicate where the gear switches to second within 2.5 seconds of switching to first.

test inputs are very unlikely to fulfil the prerequisite $\omega < 4000$. Breach actually manages to falsify this specification, since the robustness value guides the optimizer to generate inputs that satisfy the prerequisite $\omega < 4000$.

5. CONCLUSION

QuickCheck testing of hybrid systems is a promising approach. Random testing is able to falsify some properties such as φ_3^{AT} which are difficult for falsification. On the other hand, it performs poorly in some situations, such as when the property has a precondition that random inputs are unlikely to fulfil – in this case, an optimization-based technique such as falsification is likely more useful. Both techniques have a place in the tester's toolbox.

It is important to simplify random counterexamples before presenting them to the tester, but if not done carefully this leads to counterexamples with small "glitches", rather than clear violations of the specification. Robustness-aware shrinking, based on the VBool semantics presented in the paper, is able to simplify random counterexamples without making them less severe. The VBool semantics seems more suited to shrinking than the STL robust semantics, which does not take the length of the failure into account.

For future work, it would be interesting to use VBools in the optimization-based falsification procedure, to see what kind of counterexamples are produced when using different semantics for *e.g.* $\wedge$ and $\vee$. We believe that this will be of interest for many applications, since the different semantics can be used to tune the objective of the optimization problem.

Finally, shrinking seems to result in counterexamples that are simpler than those found by falsification. For future work, we will explore using VBool-aware shrinking to simplify counterexamples found by falsification.

REFERENCES

Akazaki, T. and Hasuo, I. (2015). Time robustness in MTL and expressivity in hybrid system falsification. In *International Conference on Computer Aided Verification*, 356–374. Springer.

Anand, S., Burke, E.K., Chen, T.Y., Clark, J., Cohen, M.B., Grieskamp, W., Harman, M., Harrold, M.J., and McMinn, P. (2013). An orchestrated survey of methodologies for automated software test case generation. *Journal of Systems and Software*, 86(8), 1978 – 2001.

Annpureddy, Y., Liu, C., Fainekos, G.E., and Sankaranarayanan, S. (2011). S-TaLiRo: A tool for temporal logic falsification for hybrid systems. In *TACAS*, volume 6605, 254–257. Springer.

Arts, T., Hughes, J., Norell, U., and Svensson, H. (2015). Testing AUTOSAR software with QuickCheck. In *Software Testing, Verification and Validation Workshops (ICSTW), 2015 IEEE Eighth International Conference on*, 1–4. IEEE.

Claessen, K. and Hughes, J. (2000). QuickCheck: A lightweight tool for random testing of Haskell programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, ICFP '00, 268–279. ACM, New York, NY, USA.

Clarke, E.M., Emerson, E.A., and Sifakis, J. (2009). Model checking: algorithmic verification and debugging. *Communications of the ACM*, 52(11), 74–84.

Donzé, A. (2010). Breach, a toolbox for verification and parameter synthesis of hybrid systems. In *CAV*, volume 10, 167–170. Springer.

Donzé, A. and Maler, O. (2010). Robust satisfaction of temporal logic over real-valued signals. In *FORMATS*, volume 6246, 92–106. Springer.

Driankov, D., Hellendoorn, H., and Reinfrank, M. (1993). *An Introduction to Fuzzy Control*. Springer-Verlag New York, Inc., New York, NY, USA.

Eddeland, J., Miremadi, S., Fabian, M., and Åkesson, K. (2017). Objective functions for falsification of signal temporal logic properties in cyber-physical systems. In *International Conference on Automation Science and Engineering*, 1326–1331.

Fainekos, G.E. and Pappas, G.J. (2009). Robustness of temporal logic specifications for continuous-time signals. *Theoretical Computer Science*, 410(42), 4262–4291.

Henzinger, T.A., Kopke, P.W., Puri, A., and Varaiya, P. (1995). What's decidable about hybrid automata? In *Proceedings of the twenty-seventh annual ACM symposium on Theory of computing*, 373–382. ACM.

Hoxha, B., Abbas, H., and Fainekos, G. (2014). Benchmarks for temporal logic requirements for automotive systems. *Proc. of Applied Verification for Continuous and Hybrid Systems*.

Hughes, J. (2016). Experiences with QuickCheck: testing the hard stuff and staying sane. In *A List of Successes That Can Change the World*, 169–186. Springer.

Raman, V., Donzé, A., Maasoumy, M., Murray, R.M., Sangiovanni-Vincentelli, A., and Seshia, S.A. (2014). Model predictive control with signal temporal logic specifications. In *Decision and Control (CDC), 2014 IEEE 53rd Annual Conference on*, 81–87. IEEE.