Received: 31 December 2020 | Revised: 23 February 2021 | Accepted: 11 March 2021

IET Cyber-Systems and Robotics

DOI: 10.1049/csy2.12016

ZHEJIANG UNIVERSITY PRESS | IET The Institution of Engineering and Technology | WILEY

ORIGINAL RESEARCH PAPER

Model-based validation of diagnostic software with application in automotive systems

Jun Chen[1] | Ramesh S[2]

[1]Department of Electrical and Computer Engineering, Oakland University, Rochester, Michigan, USA

[2]General Motors R and D, Warren, Michigan, USA

Correspondence

Jun Chen, Department of Electrical and Computer Engineering, Oakland University, Rochester, MI 48309, USA.
Email: junchen@oakland.edu

Funding information

Faculty startup fund from School of Engineering and Computer Science at the Oakland University

Abstract

Software validation aims to ensure that a particular software product fulfils its intended purpose, and needs to be performed against both software requirement as well as its implementation (i.e. product). However, for diagnostic software (i.e. a diagnoser) performing online diagnosis against certain fault models and reports diagnosis decision, the underlying fault models are usually not explicitly specified, neither by formal language nor by descriptive language. The lack of formal representation of fault models leaves the intended purpose of the diagnostic software vague, making its validation difficult. To address this issue, the authors propose various model-based techniques that can generate concrete examples of the diagnoser's key properties. Such examples are represented in an intuitive and possibly visualised way, facilitating the designers/users to approve or disapprove the conformance of the diagnoser to the intended purpose. The proposed techniques work for validation of both the requirement and implementation that can be modelled as finite state machine, and are illustrated through applications on vehicle on-board diagnostic requirement.

1 | INTRODUCTION

To reduce the potential impact of system failures that can severely affect functionality, electronic controller unit (ECU) in modern vehicles must include on-board diagnosis (OBD) compliant diagnostic software, which provides a data link with the various controllers within the vehicle [1]. To meet the stringent emission regulations, vehicle system is becoming increasingly complex, making the diagnostic more necessary [2, 3]. According to [4], there are a total number of 3637 OBD-II diagnostic trouble codes (DTCs), including 1164 Body Code, 486 Chassis Code, 1688 Powertrain code, and 299 Network Code.

The development of the diagnostic software typically involves verification and validation. The goal of the verification is to ensure that the implementation conforms to the software requirement, while the validation tries to eliminate the discrepancies between the software requirement/implementation and the users' 'implicit' intention. Validation makes sure the final product fulfils its intended purpose, and is required for both software requirement and its implementation. See Figure 1 for illustration. Note that both 'software requirement' and 'software implementation' can be different from the user's intention which is 'implicit' due to the lack of formal representation.

The most popular methodology for software verification and validation is the model-based testing, where test cases can be generated by auto-generation tool [5–9]. For example, model-checking based techniques can be used in test case generation to synthesise counterexamples that violate the test objective [10–12]. Satisfiability modulo theories is another well-known technique in many verification tools that enable test case generation using constraint solving. For example, dReal [13] applies interval-based approach and can solve polynomial and exponential functions, while Yices [14] can solve (a subset of) non-linear equations. Acharya et al. [15] use the active learning tools for the verification and validation of software implementation, while Schordan et al. [16] evaluate different tools for software verification and validation. These formal method based approaches provide good coverage on the test case generation, while suffering computational complexity. To encounter the computation load, stochastic approach based on sampling are also considered in the literature. For example, Ref. [17–19] use probabilistic model-checking techniques in software

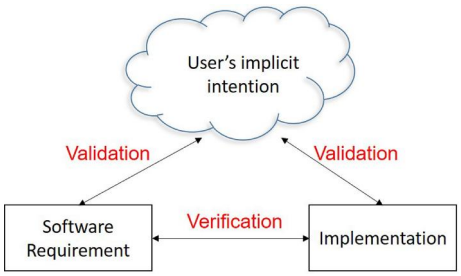

FIGURE 1 Software verification and validation process. The user's intention is deemed as 'implicit' as it is usually expressed as verbal/written language instead of formal representation

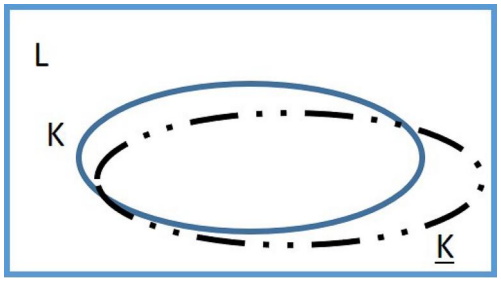

FIGURE 2 Illustration of diagnostic specification. L denotes the set of all system behaviour, K is the subset of behaviour that is deemed as 'no-fault' by the diagnoser, while K is the subset of behaviour that is deemed as 'no-fault' by the user

verification. Unlike model-checking, probabilistic model checking aims to verify whether a system that exhibits stochastic behaviour satisfies certain (quantitative) property [20–26]. Hence, these approaches provide a confidence level of the software correctness, instead of the logical good/bad result. The diagnosis problem in concurrent/modular systems are studied in [27–30].

All the above mentioned work are based on the assumption that the software requirements are available, which are deemed as the ground truth and used to verify the software. However, this is not the case for the validation of the diagnostic software, for which the goal is to make sure that the no-fault specification K (specifying the behaviour i.e. considered as no-fault by the diagnoser) of the given diagnoser is the same as the true system no-fault behaviour, denoted as $\bar{K}$. See Figure 2. However, in most cases $\bar{K}$ is not provided either as a formal representation or descriptive language, making the validation process difficult. One possible approach is to generate a formal model of K and leave it for the users to validate. However, this formal model may not be intuitive for the users to understand. Therefore, to facilitate the validation of diagnoser, we propose various model-based techniques that can generate concrete examples to illustrate the key properties of K in an intuitive and possibly visualised way, facilitating the designers/users to approve or disapprove the conformance of the diagnoser to the intended purpose. In particular, the following properties of K are investigated:

- *Boundary analysis*: The boundary analysis aims to generate a pair of sequences lying on each side of K, that is, one test case in K while the other in $L - K$. Such pair can be intuitive to understand, and can act as representative of the boundary of K. Then the users can verify whether the boundary, illustrated by the generated pairs, conforms to their intention or not. Specifically, we proposed algorithms to generate test case pairs in (1) boundary in time domain, (2) boundary in frequency domain, and (3) boundary in value domain.
- *Statistical analysis*: The statistical analysis tries to explore statistical properties of K, for example, the correlation between different diagnostic flags. Such correlation can be seen as key representative of the no-fault specification K, based on which the fulfilment of the users' intention can be decided.

Please note that in the paper various illustration examples are based on automotive diagnostic software. However, the proposed techniques work for diagnostic software in general.

The rest of this article is organised as follows: Section 2 presents problem formulation and needed notations on finite state machine (FSM), linear-time temporal logic (LTL), and model checking. Boundary analysis on time, frequency and value domains, and associated algorithms, are presented in Sections 3, 4 and 5, respectively. Section 6 includes statistical analysis and associated algorithm and the paper is concluded in Section 7.

2 | PROBLEM FORMULATION AND PRELIMINARY

In this section, we formulate the problem and briefly introduce related notations on FSM, LTL, and model checking.

2.1 | Problem formulation

The functionality of the diagnostic software (i.e. a diagnoser) can be illustrated by Figure 2, where L denotes the set of all possible system behaviours while K is the set of nominal behaviours, denoted as no-fault specification [31]. The diagnoser reports 'Fail' whenever the system executes a behaviour in $L - K$ while reports 'Pass' for K. Note that here K represents the set of behaviour that is considered as 'no-fault' by the diagnoser, while the set of true 'no-fault' system behaviour (denoted as L) may be different from K, which needs to be validated in the validation process.

Example 1 Figure 3 depicts the active fuel management (AFM) exhaust valve subsystem, where the software executing within the ECU uses various input data to decide a control signal for each of the valves, and the AFM exhaust valves provide valve position and actuator fault state feedback to the ECU for the purpose of executing OBD. An abstract description of the above system is given in Figure 4, where the controller performs control functionality over the plant, and the diagnoser monitors the behaviour of the controller,

plant, or both of them, by detecting the faulty sequence of the input/output of the plant (i.e. the control command I_C and the plant output I_P), and reports to ECU the Fail and/or Pass report based on certain no-fault specification K over the space of I_C and I_P.

The goal of validation of the diagnostic software is to make sure that the no-fault specification K (specifying the behaviour i.e. considered as no-fault by the diagnoser) of the given diagnoser is the same as the true system no-fault behaviour, denoted as $\bar{K}$, which is usually not provided in either the formal representation or in the descriptive language, making the validation process difficult. To address this issue, we propose various model-based techniques that can generate concrete examples to illustrate $\bar{K}$, facilitating the designers/users to approve or disapprove the conformance of the diagnoser to the intended purpose.

2.2 | Finite state machine

As illustrated by Figure 1, the validation is needed for both software requirement and its implementation, the first of which is the informal representation (by descriptive language) of the diagnoser while the second is executable (in a programming language) of the diagnoser. Both software requirement and its implementation may be translated into formal representation in the form of FSMs [32–35]. Note that the translation into FSM is outside the scope of this study, and interested readers can refer to the reference mentioned above. The techniques developed in this report are based on the FSM representation of diagnoser, and hence is applicable for validation process in both software development stages. Note that it generally requires infinite state representation to represent a software. Extension to such general case remains a future work.

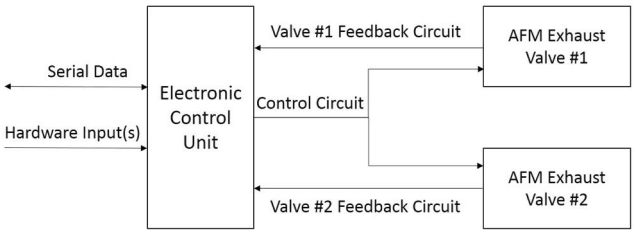

FIGURE 3 The AFM exhaust valve subsystem. AFM, active fuel management

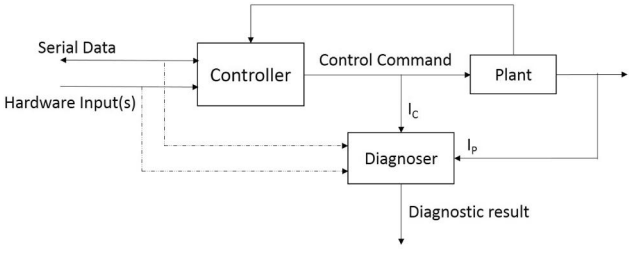

FIGURE 4 Abstracted system structure of active fuel management diagnosis

A diagnoser can be represented by a FSM D given as five tuples $D = (Q, q_0, U, \Sigma, E)$ where

- Q is the state space of D with initial state q_0;
- $U = \{I_C\} \cup \{I_P\}$ is the input to the diagnoser, where I_C is the control command to the plant while I_P is the output of the plant, as pictured by Figure 4;
- Σ is the set of internal and output variables with $\Sigma_F \subseteq \Sigma$ being the set of output Boolean variables corresponding to the diagnostic flag variables;
- E is the set of edges, such that for each $e = (q, q', g_e, a_e) \in E$,
  - q is the source state;
  - q' is the target state;
  - g_e is the guard condition which is a predicate over U and Σ, and
  - a_e is the action that assigns values to Σ.

The diagnoser D reports 'Fail' when it makes a transition that sets a flag variable $f \in \Sigma_F$ to TRUE. The following definition defines, for each diagnostic result $f \in \Sigma_F$, the set of passed test cases and the set of failed test cases, where a test case is defined to be a sequence of inputs $u_0u_1 \ldots u_n \in U^*$ that is accepted by FSM D.

Definition 1 For each $f \in \Sigma_F$, a test case u is a passed test case of f if there exists one execution of D on u that never sets f to True. Similarly, it is a failed test case of f if there exists one execution of D on u that sets f to True at the very last transition. The set of passed test cases of f is denoted as $T_P(f)$, and the set of failed test cases of f is denoted as $T_F(f)$.

Note that if we denote the set of test cases accepted by FSM D as L, that is, L is the set of input sequences accepted by the FSM, then by definition $\cap_{f \in \Sigma_F} T_P(f) = K$ and $\cup_{f \in \Sigma_F} T_F(f) = L - K$.

2.3 | Linear-time temporal logic

As will be seen later, the model-checking technique is used to generate cases that may reveal potential pitfalls in the diagnostic software. Specifically, LTL formula will be used for model-checking. In the following, we present a brief description of LTL; a more thorough introduction of LTL and model-checking can be found in [31, 36–38].

LTL is a formalism for describing properties of sequences of states. Such properties are expressed using *temporal operators* of the temporal logic which include:

- X ('next time'): it requires that a property holds in the next state of the state-trace;
- U ('until'): it is used to combine two properties. The combined property holds if there is a state on the state-trace

where the second property holds, and at every preceding state on the trace, the first property holds;
- F ('eventually' or 'in the future'): it requires that a property will hold at some future state on the state-trace;
- G ('always' or 'globally'): it requires that a property holds at every state on the trace; and
- B ('before'): it combines two properties. It requires that if there is a state on the state-trace where the second property holds, then there exists a preceding state on the trace where the first property holds.

For a LTL formula ϕ, we use the notation $D \vDash \phi$ (resp., $D \nvDash \phi$) to denote that ϕ holds (resp., does not hold) for every state trace π of FSM D. The relation $\vDash$ for state-trace π is defined inductively as follows:

1. $\pi \vDash \neg\phi$ if and only if $\pi \nvDash \phi$.
2. $\pi \vDash \phi_1 \wedge \phi_2$ if and only if $\pi \vDash \phi_1$ and $\pi \vDash \phi_2$.
3. $\pi \vDash X\phi$ if and only if $\pi^1 \vDash \phi$.
4. $\pi \vDash \phi_1 U \phi_2$ if and only if there exists a k such that $\pi^k \vDash \phi_2$ and for all $j \leq k-1$, $\pi^j \vDash \phi_1$.

As will be seen in the following sections, various modelling checking techniques are adopted in this work. For complexity analysis of the adopted technique, readers are referred to [36–39].

2.4 | An overview of statistical model checking

Probabilistic model checking aims to verify whether a system that exhibits stochastic behaviour satisfies certain (quantitative) property [20, 21]. One example of such quantitative property is expressed in probabilistic computation tree logic (PCTL), [40, 41], which is captured in the following definition:

Definition 2 The syntax of PCTL is given by:

$$\Phi \;:=\; \text{True} \mid a \mid \neg\Phi \mid \Phi \wedge \Phi \mid \Phi \vee \Phi \mid P_{\sim p}[\phi]$$
$$\phi \;:=\; X\Phi \mid \Phi U^{\leq k}\Phi \mid \Phi U \Phi$$

where a is an atomic proposition, and the operation $\sim$ is either $<, \leq, >$, or $\geq$, and p is a real value within unit interval, and k is a natural number.

The key operator in PCTL is $P_{\sim p}[\phi]$, which means that a path formula ϕ is true in a state with probability that is bounded by p (corresponding to the operator $\sim$). Given a system S and a path formula ϕ, there are two questions that the probabilistic model checking is trying to answer: (1) whether the probability that ϕ is true in S satisfies the condition $\sim p$, that is, $S \vDash P_{\sim p}[\phi]$; or (2) what is the exact probability that S satisfies ϕ, that is, what is the value of $P[S \vDash \phi]$.

Two types of approaches to address the probabilistic model checking problem have been developed, namely the *numerical* and *statistical*. The numerical approach iteratively computes or approximates the exact probability of the paths satisfying ϕ by exploring the whole state space of S. Also, the statistical approach simulates the system for finite number of times, and borrows techniques and theories from statistics to provide statistical inference of the answer to the probabilistic model checking problem. In this article, we focus on the second approach, namely the statistical model checking, and show how it can be used to help and validate the no-fault specification K given a diagnoser D.

One assumption of statistical model checking is that, ϕ is a bounded property, that is, a property that can be verified based on state trace of finite length. Let B_i be a Bernoulli random variable with distribution $Pr[B_i = 1] = \theta$ and $Pr[B_i = 0] = 1 - \theta$, and B_i is such that it equals one if the ith simulation of S satisfies ϕ and 0 otherwise. Therefore $\theta \approx P[S \vDash \phi]$. Algorithms to perform statistical model checking can be found in [17, 18], which we will apply to our problem in Section 6.

3 | VALIDATION IN TIME DOMAIN

Here, we investigate the boundary analysis, aiming to generate a pair of sequences lying on each side of K. Such a pair is intuitive to understand, and can act as the representative of the boundary of K, that is, one test case in K while the other in $L - K$. Then the users can verify whether the boundary, illustrated by the generated pair, conforms to their intention or not. Specifically, we will examine the boundary of K in time domain in this section, followed by the boundary analysis in frequency and value domain in the following sections.

We will first define the notion of 'similar' test cases, that is, two test cases that trigger the diagnoser to produce opposite diagnostic results while the two cases themselves satisfy certain similarity condition in the time domain. We start by the following example.

Example 2 Suppose D is intended to report 'Fail' whenever '2 "a" out of 3 measurements' occurs, and is wrongly implemented as a FSM shown in Figure 5. In this example, $u_F = aab$ will set f to True, while $u_P = bbaab$ will not, though it also contains '2 "a" out of 3 measurements'.

Note that in the above example, u_F and u_P are 'similar' and may help reveal possible design errors. In the following, we formalise this notion of the time shift similarity, such that two test cases are said to be similar if they share a large portion of the subsequence.

Definition 3 Two test cases $u_P = u_0^1, u_1^1, \ldots, u_K^1 \in T_P(f)$ and $u_F = u_0^2, u_1^2, \ldots, u_L^2 \in T_F(f)$ are said to be similar if $\exists k, l, n$, such that $u_{k+i}^1 = u_{l+i}^2$ for all $0 \leq i \leq n$, that is, two test cases share the same subsequence.

Remark 1 When n is small, similar test cases according to Definition 3 may be trivial. In the case where n is large enough, it potentially reveals the error that a fault can only be diagnosed at some specific timing, or a

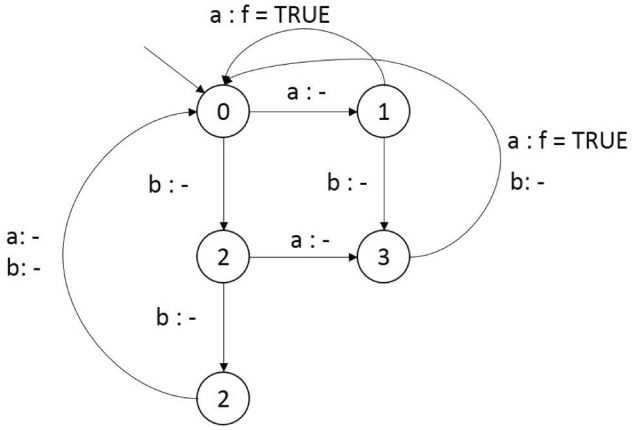

FIGURE 5 A diagnoser for '2 "a" out of 3 measurements', where $\Sigma = \{a, b\}$

fault may be hidden if it follows different prefix. To ensure that n is large enough, we choose $n = |u_F|$ in this section. Then the above definition reduces to 'u_P and u_F are similar if u_F is a suffix of u_P'.

Algorithm 1 takes a diagnoser in the form of FSM D as input, and computes a pair of test cases for $f \in \Sigma_F$ that is similar according to Definition 3 under the condition $n = |u_F|$.

Algorithm 1 Generating Test Cases in Time Domain Boundary

Input: Diagnoser $D = (Q, q_0, U, \Sigma, E)$, $f \in \Sigma_F$
Output: Test case pair (u_P, u_F)
1) Initialise $f^1 = f^2 =$ False;
2) Construct $D^i = (Q^i, q_0^i, U^i, \Sigma^i, E^i)$ for $i \in \{1, 2\}$,
where $Q^i = Q \cup \{q_0^i\}$, $U^i = U \cup \{e_1, e_2\}$, $\Sigma^i = \Sigma \cup \{f^i\}$,
E^i is the same as E except the following changes:
- Add $(q_0^i, q_0^i, !e_i, -)$ and $(q_0^i, q_0, e_i, -)$;
- Add $(q, q, e_j, -)$ for each $q \in Q$ where $j = \{1, 2\} - i$ (where $-$ denotes set difference;
- For each $e = (q, q', g_e, a_e) \in E$ such that $(f =$ True$) \in a_e$, change a_e into $a_e \cup \{f^i =$ True$\}$.

3) Construct standard synchronisation $T = D^1 \times D^2$;
4) Model check the LTL property $\phi =$!$(e_1 \wedge F(e_2 \wedge F(!f^1 \wedge f^2)))$.

Note that the new initial state q_0^i of D^i, $i \in \{1, 2\}$, is added as a waiting state, so that D^i enters D only after the input e_i arrives, and otherwise stays at the waiting state q_0^i (captured by first bullet of step 1). When the other augmented diagnoser D^j is entering its own copy of D, D^i simply stays in the current state by performing the newly added transition in the second bullet in brackets. The new Boolean flag f^i is added to track whether the failure flag f was ever set in the past (captured by the third bullet of Step 1). Note that the formula ϕ requires that it never happens that the event e occurs in one copy of D without triggering the diagnostic flag, but its later occurrence in another copy of D triggers the diagnostic flag, which aligns with Definition 3.

Remark 2 Note that there is no need to construct a symmetric LTL property at Step 4 since the construction of T is symmetric with respect to indexes 1 and 2.

The next lemma guarantees the correctness of the above algorithm. Note that all the proofs are given in the Appendix.

Lemma 1 *If $T \not\models \phi$ and the model checker returns a counterexample $u = e_1 u_1 \ldots u_m e_2 u_{m+1} \ldots u_l$, then $u_P = u_1 \ldots u_m u_{m+1} \ldots u_l$ and $u_F = u_{m+1} \ldots u_l$ are similar according to Definition 3 under the condition $n = |u_F|$.*

4 | VALIDATION IN FREQUENCY DOMAIN

When the diagnoser is operating in a frequency that is different from other software components, what is observed by the diagnoser may be dependent on the operating frequency, even if the sensor output from the plant and control command from the controller are the same. Because of this, the diagnostic results can be different. We define the frequency shift similarity for two timed test cases if one can be obtained by sampling the other one and lead to different diagnostic results. We start by the following example.

Example 3 Consider again the diagnoser in Example 2. Suppose the controller/plant is operating with a period of 20 ms, and generates sequence

$$u_F = ((0, a), (20, b), (40, b), (60, b), (80, a),) \\ ((100, a), (120, b)),$$

and the diagnoser is operating with a period of 25 ms, and observes.

$u_P = ((0, a), (25, b), (50, b), (75, b), (100, a), (125, b))$.

As can be checked, $u_F \in T_F(f)$ while $u_P \in T_P(f)$. It can also be the case that the controller/plant generates

$$u'_P = ((0, a), (20, b), (40, b), (60, a), (80, b),) \\ ((100, b), (120, a)),$$

while the diagnoser receives

$$u'_F = ((0,a),(25,b),(50,b),(75,a),(100,b),(125,a)),$$

such that $u'_P \in T_P(f)$ and $u'_F \in T_F(f)$. In both cases, the 'similar' pair would bring the operating frequency into designers' attentions.

The following two definitions formalise the notion of similarity illustrated in Example 3, the first of which introduces the definition of sampling.

Definition 4 Given two timed sequences $s = (t_1, \sigma_1)$, $(t_2, \sigma_2)\ldots(t_n, \sigma_n)$ and $s' = (t_1', \sigma'_1)\ (t_2', \sigma_2')\ldots(t_m, \sigma'_m)$, s' is a sampled version of s if $m \leq n$ and for all t'_i such that $t'_i < t_n$, there exists k such that $t_k \leq t_i' < t_{k+1}$ and $\sigma'_i = \sigma_k$; otherwise for t_i' such that $t'_i \geq t_n$, $\sigma'_i = \sigma_n$.

When the time advance (or equivalently, frequency) for a timed sequence s is fixed, that is, $t_{i+1} - t_i$ remains constant (e.g. δ) for alright i, then we write $s = (\delta|\sigma_1\sigma_2\ldots\sigma_n)$ for simplicity. In this case $t_i = i\delta$. The next definition considers both u_P and u_F are timed sequences with constant frequencies.

Definition 5 Two timed test cases $u_P = (\delta_P|u_0^1, u_1^1, \ldots, u_K^1) \in T_P(f)$ and $u_F = (\delta_F|u_0^2, u_1^2, \ldots, u_L^2) \in T_F(f)$ are said to be similar if u_P is a sampled version of u_F, or u_F is a sampled version of u_P.

When δ_P and δ_F are close enough, the pair of u_P and u_F serves as a concrete example to visualise the boundary of K in the frequency domain, which may assist the designers to re-examine the diagnoser's sampling rate. The following algorithm takes a diagnoser in form of FSM D and two positive integers $k_1, k_2 \in \mathbb{N}$ as inputs, and computes for $f \in \Sigma_F$ a pair of similar test cases with periods of $k_1\tau$ and $k_2\tau$, respectively, where τ is the base period which is known and fixed. Without loss of generality, we assume $k_1 < k_2$.

Algorithm 2 Generating Test Cases in the Frequency Domain Boundary

Input: Diagnoser $D = (Q, q_0, U, \Sigma, E)$, $k_1, k_2 \in \mathbb{N}$
Output: Test case pair (u_P, u_F)
1) Initialise $f^1 = f^2 =$ False
2) Construct $D^1 = (Q^1, q_0, U, \Sigma^1, E^1)$, where $\Sigma^1 = \Sigma \cup \{f^1\}$, E^1 is obtained as following:
   - For each $e = (q, q', g_e, a_e) \in E$, such that $(f =$ True$) \in a_e$, change a_e into $a_e \cup \{f^1 =$ True$\}$.
   - For each $e = (q, q', g_e, a_e) \in E$, replace it with k_1 transitions, $(q, q_{e,1}, g_e, a_e)$, $(q_{e,1}, q_{e,2},$ True, $-)$, $(q_{e,2}, q_{e,3},$ True, $-)$, …, $(q_{e,k_1-1}, q_{e,k_1},$ True, $-)$.
3) Construct $D^2 = (Q^2, q_0, U, \Sigma^2, E^2)$, where $\Sigma^2 = \Sigma \cup \{f^2\}$, E^2 is obtained as following:
   - For each $e = (q, q', g_e, a_e) \in E$, such that $(f =$ True$) \in a_e$, change a_e into $a_e \cup \{f^2 =$ True$\}$.
   - For each $e = (q, q', g_e, a_e) \in E$, replace it with k_2 transitions, $(q, q_{e,1}, g_e, a_e)$, $(q_{e,1}, q_{e,2},$ True, $-)$, $(q_{e,2}, q_{e,3},$ True, $-)$, …, $(q_{e,k_2-1}, q_{e,k_2},$ True, $-)$.
4) Construct standard synchronisation $T = D^1 \times D^2$;
5) Model check the LTL property: $\phi_1 = !F(!f^1 \wedge f^2)$ and $\phi_2 = !F(f^1 \wedge !f2)$.

Note that D^i is obtained by extending every transition of D to a sequence of k_i copies of original transition, such that only the first copy carries the action and all the other $k_i - 1$ copies assume inputs but do not change the output or internal variables. In D^1, the replicated transitions accept same inputs as the original transition, while in D^2 the replicated transitions accept any input. This guarantees that for every trace $s = \sigma_1\sigma_2\ldots$ that is accepted by D, there exist an accepted sequence $s_1 = \sigma_1\sigma_1\ldots\sigma_2\sigma_2\ldots$ of D^1 and an accepted sequence $s_2 = \sigma_1\Sigma\ldots\sigma_2\Sigma\ldots$ of D^2. The next lemma guarantees the correctness of the above algorithm.

Lemma 2 *If $T \not\models \phi_i$ and the model checker returns a counterexample $u = u_1\ldots u_m$ violating ϕ_i, then it can be checked that $u_P = (k_i\tau|\{u_{m*k_i+1}, m = 0, 1, \ldots\}) \in T_P(f)$ and $u_F = (k_2\tau|\{u_{m*k_j+1}, m = 0, 1, \ldots\}) \in T_F(f)$ satisfy the similarity definition above, where $j = \{1, 2\} - i$.*

5 | VALIDATION IN VALUE DOMAIN

This section focuses on the boundary of K in the value domain. We start by the following example.

Example 4 Suppose whenever the command in the form of $x > \rho$ is issued for three consecutive times, the plant output y is expected to be lower than a certain value τ, that is, it always holds that $x > \rho \wedge X(x > \rho) \wedge XX(x > \rho) \Rightarrow y < \tau \vee X(y < \tau) \vee XX(y < \tau)$. Assume both ϱ and τ are positive for simplicity. The correct diagnoser D is shown in Figure 6. Then $u_P = ((x = \rho + \delta, y = 2\tau), (x = \rho, y = 2\tau), (x = \rho + \delta, y = 2\tau))$ will not set f to True, while $u_F = ((x = \rho + \delta, y = 2\tau), (x = \varrho + \delta, y = 2\tau + \delta), (x = \rho + \delta, y = 2\tau))$ with $\delta > 0$ will. This concrete example can help the designer review whether the value of ρ is desired.

Furthermore, suppose that the diagnoser is erroneously designed as in Figure 7. For the edge from state '1' to state '2', instead of ρ, the lower bound for x is set to be $-\infty$. Then $u_P = ((x = 2\rho, y = 2\tau), (x = \rho - \delta, y = \tau - \delta), (x = 2\rho, y = 2\tau))$ will not set f to True, while $u_F = ((x = 2\rho, y = 2\tau), (x = \rho,$

$y = \tau$), $(x = 2\rho, y = 2\tau)$) may possibly set it, where δ is positive. This concrete example can, again, help the designer review the software for correctness.

The following definition formalises the notion of similarity illustrated in Example 4, by stating that two test cases are said to be similar if they trigger different diagnostic results, while their elements are bounded and deviated from each other.

Definition 6 Two test cases $u_P = u_0^1, u_1^1, \ldots, u_M^1 \in T_P(f)$ and $u_F = u_0^2, u_1^2, \ldots, u_M^2 \in T_F(f)$ are said to be similar if there exists $0 \le i_0 \le i_1 \le \cdots \le i_L \le K$ such that $u_{i_l}^1 = u_{i_l}^2 + \delta$ for all $0 \le l \le L$ and otherwise $u_i^1 = u_i^2$;

The following algorithm takes a diagnoser in the form of FSM D and a small number δ as inputs, computes a pair of similar test cases for $f \in \Sigma_F$.

Algorithm 3 Generating Test Cases in the Value Domain Boundary

Input: Diagnoser $D = (Q, q_0, U, \Sigma, E)$, $\delta > 0$
Output: Test case pair (u_P, u_F)
1) Initialise $f^1 = f^2 =$ False;
2) Construct $D^i = (Q, q_0, U \cup \{n\}, \Sigma^i, E^i)$ for $i \in \{1, 2\}$, where $\Sigma^i = \Sigma \cup \{f^i\}$; E^1 is same as E except the following changes:
   - For each $e = (q, q', g_e, a_e) \in E$, such that $(f =$ True$) \in a_e$, change a_e into $a_e \cup \{f^1 =$ True$\}$;
   - For each $e = (q, q', g_e, a_e) \in E$, change $g_e(u)$ into $g_e(u) \wedge (n == 0)$;
   - Duplicate every $e = (q, q', g_e(u), a_e) \in E$, with $g_e(u)$ changed into $g_e(u + \delta) \wedge (n == 1)$;
     And E^2 is same as E except the following changes:
   - For each $e = (q, q', g_e, a_e) \in E$, such that $(f =$ True$) \in a_e$, change a_e into $a_e \cup \{f^2 =$ True$\}$;
3) Construct standard synchronisation $T = D^1 \times D^2$;
4) Model check the LTL property: $\phi = !F(!f^1 \wedge f^2)$

The next lemma guarantees the correctness of the above algorithm.

Lemma 3 *If $T \not\models \phi$ and the model checker returns a counterexample $u = (n_1, u_1)\ldots(n_m, u_m)$, then $u_P = u_1 + n_1\delta, \ldots, u_m + n_m\delta$ and $u_F = u_1 \ldots u_m$ satisfy the similarity definition above.*

Remark 3 Note that both Algorithms 2 and 3 consider as input the distance of similar test cases, namely k_2/k_1 and δ, and generate the pair of similar test cases that satisfies the given distance. In the future, efforts would be made to find the minimal distance for which a similar test case pair exists, that is, the minimal k_2/k_1 or the minimal perturbation δ. Such minimal distance would more clearly depict the boundary of K so as to support the designers to validate the diagnostic software.

6 | STATISTICAL ANALYSIS OF DIAGNOSTIC FLAGS BY STATISTICAL MODEL CHECKING

In most cases, the diagnoser is monitoring against multiple faults, that is, $|\Sigma_F| > 1$. In this case, for each $f \in \Sigma_F$, there is one specification K_f and $K = \cap_{f \in \Sigma_F} K_f$. Furthermore, after the release of the software, more features (faults being diagnosed) can be added, that is, $|\Sigma|$ increases, and there is a chance that the newly added feature is already included, or there is a similar feature, in the existing version. In other words, there exist f, $f' \in \Sigma_F$ such that K_f and $K_{f'}$ have a large overlap. Such overlap may be due to redundancy in development and may be reduced to minimise the chance of bugs and to lower maintenance cost. However, on the other hand, the multiple diagnostic flags may be designed hierarchically so that one high-level diagnostic flag f depends on one or more low-level diagnostic flags f'. In this case the overlap between K_f and $K_{f'}$ are intentional that should not be considered as an error.

In either case, we aim to bring out the statistical correlation between two diagnostic flags, and pass such statistical correlation to designers for evaluation. For this purpose, we define the following notion of similarity for a pair of diagnostic flags (as opposed to pairs of test cases as in the previous sections). Two flags are said to be highly correlated, if one flag being set

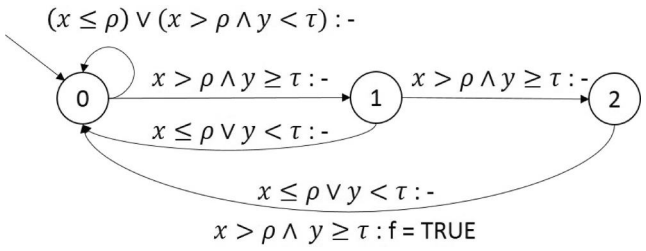

FIGURE 6 A diagnoser for $x > \rho \wedge X(x > \rho) \wedge XX(x > \rho) \Rightarrow y < \tau \vee X(y < \tau) \vee XX(y < \tau)$

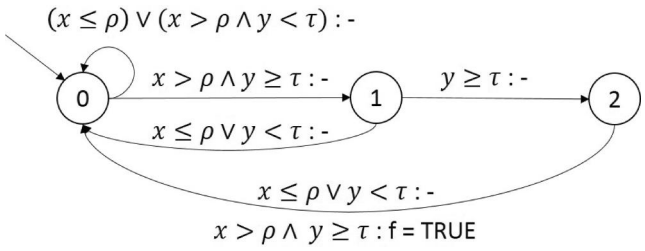

FIGURE 7 A erroneous diagnoser for $x > \rho \wedge X(x > \rho) \wedge XX(x > \rho) \Rightarrow y < \tau \vee X(y < \tau) \vee XX(y < \tau)$

implies the other was or will be set in the bounded timing window with high probability.

Definition 7 Let $f_1, f_2 \in \Sigma_F$ be two flags, and $p \in 0,1$ be a pre-defined threshold and $n \in \mathbb{N}$ be a pre-defined delay bound. Then f_1 and f_2 are said to be highly correlated if the following holds $G(f_i \Rightarrow Pr_{\geq p}[X^{<n} f_j])$ where $i \in \{1, 2\}$ and $j = \{1, 2\} - i$, that is, $G(f_1 \Rightarrow Pr_{\geq p}[X^{<n} f_2] \vee f_2 \Rightarrow Pr_{\geq p}[X^{<n} f_1])$[1]. In other words, if f_1 (resp., f_2) is True, then with probability higher than p that f_2 (resp., f_1) is True within n steps.

Example 5 Suppose D is intended to report fail f_1 whenever '2 "a" out of 3 measurements' occurs, and also reports fail f_2 whenever 'two consecutive "a" measurements' occurs. Then whenever f_2 is reported, f_1 is also reported, that is, $f_2 \Rightarrow Pr_{\geq p}[f_2]$ holds for any $0 < p < 1$. Therefore, f_1 and f_2 are highly correlated, and attention should be paid to examine whether such correlation is intended or due to unnecessary redundancy.

Note that the definition above is symmetric with respect to indexes 1 and 2. For simplicity, in the following discussion we only consider the following asymmetric version. Let $\phi \equiv X^{<n} f_2$, and the property to be model checked is given as $\Phi \equiv f_1 \Rightarrow Pr_{\geq p}[\phi]$. The idea is to simulate D with uniformly distributed random inputs, and for the ith simulation, define the random variable B_i such that $B_i = 1$ if $f_1 \wedge \phi$ holds, and $B_i = 0$ if $f_1 \wedge \neg\phi$ holds. Otherwise B_i is undefined. Let $\theta := Pr[B_i = 1 \mid f_1]$ (hereafter the conditioning f_1 stands for $f_1 =$ True) and then $Pr[B_i = 0 \mid f_1] = 1 - \theta$. Then two flags f_1 and f_2 are highly correlated, if and only if $\theta \geq p$. We then present the algorithm based on hypothesis testing, which works under the assumption that $|\theta - p|$ cannot be arbitrarily small. Then the checking of the correlation of f_1 and f_2 reduces to the testing of hypothesis $H_0 : \theta \geq p + \delta$ against the alternative hypothesis $H_1 : \theta \leq p - \delta$ based on observations of B_i, that is, whether a finite execution of D with $f_1 =$ True satisfies ϕ. Note that δ is a small positive number, and we further define $p_0 := p + \delta$ and $p_1 := p - \delta$.

The following algorithm tests hypothesis H_0 against H_1 based on the simulation of D, with errors upper-bounded by α and β. The algorithm is based on Wald's sequential probability ratio test (SPRT) [42, 43].

Algorithm 4 Checking Correlation of Diagnostic Flags

Input: Diagnoser D, diagnostic flags f_1 and f_2, integer $m > 1$, errors bounds $0 < \alpha < 1$ and $0 < \beta < 1$, small positive number δ
Output: TRUE or FALSE on correlation of f_1 and f_2

1) Set $p_0 = p + \delta$ and $p_1 = p - \delta$;
2) Simulate D with uniformly distributed random inputs, after m simulations, calculate the quantity

$$f_m = \prod_{i=1, f_1=True} \frac{Pr[B_i = b_i \mid f_1, \theta = p_1]}{Pr[B_i = b_i \mid f_1, \theta = p_0]} = \frac{p_1^{d_m}(1-p_1)^{s_m - d_m}}{p_0^{d_m}(1-p_0)^{s_m - d_m}}, \quad (1)$$

where $d_m = \sum_{i=1}^{m} b_i$ and $s_m = \sum_{i=1}^{m} I_{\{0,1\}}(b_i)$, that is s_m is the number of simulations where f_1 is true and d_m is the number of simulations where both f_1 and ϕ are true.
3) Hypothesis is accepted and f_1, f_2 are highly correlated if $f_m \leq \beta/(1-\alpha)$, and hypothesis H_1 is accepted and f_1, f_2 are not highly correlated if $f_m \geq (1-\beta)/\alpha$; Otherwise start another simulation D, and go to step 2.

Lemma 4 *The above algorithm is guaranteed to terminate. It decides f_1, f_2 to be highly correlated with probability at most α when they are not highly correlated, while it decides f_1, f_2 to be not highly correlated with probability at most β when they are highly correlated.*

Since, the algorithm is a variant of [17], the correctness proof of above lemma is omitted.

7 | CONCLUSIONS

The fact that the validation is a dynamic process brings up a lot of challenges and areas to explore. Model-based and data-driven approaches complement each other and can provide a powerful tool for the diagnostic software validation. The methodologies described in this article pertain to diagnostic software validation. Boundary analysis and statistical techniques can provide useful insights about the diagnoser, and generate concrete samples to depict the key properties of the diagnoser under validation. The model-based approach can be applied to both requirement validation and implementation validation. Furthermore, the adoption of data-driven techniques (i.e. statistical model checking) allows the validation to be performed not only during software development but also after its release (i.e. post-market). Future work includes complexity/throughput analysis of the above proposed methodologies and non-uniform sampling to avoid simulations for which no diagnostic flag is set to TRUE, thus reducing the computational burden.

[1] The operator $X^{<n}$ denotes 'within the next n steps'.

ACKNOWLEDGEMENT

This work is supported in part by the faculty startup fund from School of Engineering and Computer Science at the Oakland University.

NOMENCLATURE

AFM	Active fuel management
ECU	Electronic controller unit
OBD	On-board diagnosis
FSM	Finite state machine
D	FSM representation of diagnoser
Q	State space of D
U	Input space of D
Σ	Input and output variable of D
Σ_f	Diagnostic flags
E	Edges of D
I_C	Control command
I_P	Plant output
L	All systems behaviour
K	No-fault specification
K_f	No-fault specification for f
$\bar{K}$	True no-fault specification
LTL	Linear-time temporal logic
Φ	LTL formula
$\models$	Satisfy
$\not\models$	Not satisfy
Π	State-trace
SPRT	Sequential probability ratio test

ORCID

Jun Chen https://orcid.org/0000-0002-0934-8519

REFERENCES

1. Farrugia, M., et al.: The usefulness of diesel vehicle onboard diagnostics (obd) information. In: Proceedings of 2016 17th International Conference on Mechatronics-Mechatronika (ME), 7-9 December 2016, IEEE, Prague, Czech Republic. pp. 1–5 (2016)
2. Kulkarni, P., Rajani, P., Varma, K.: Development of on board diagnostics (OBD) testing tool to scan emission control system. In: Proceedings of 2016 International Conference on Computing Communication Control and Automation (ICCUBEA), 12-13 August 2016, IEEE, Pune. pp. 1–4 (2016)
3. Jiang, Y., Yin, S.: Recursive total principle component regression based fault detection and its application to vehicular cyber-physical systems. IEEE Trans. Ind. Inf. 14(4), 1415–1423 (2017)
4. Wikipedia: On-board diagnostics. Available at: https://en.wikipedia.org/wiki/On-board_diagnostics. Accessed on June 15 2020
5. Peranandam, P., et al.: An integrated test generation tool for enhanced coverage of simulink/stateflow models. In: Proceedings of 2012 Design, Automation & Test in Europe Conference & Exhibition, pp. 308–311 IEEE, Dresden. (2012). https://doi.org/10.1109/DATE.2012.6176485
6. Oh, J., Harman, M., Yoo, S.: Transition coverage testing for simulink/stateflow models using messy genetic algorithms. In: Proceedings of the 13th Annual Conference on Genetic and Evolutionary Computation, pp. 1851–1858. Dublin, (2011)
7. Gotlieb, A., Botella, B., Rueher, M.: Automatic test data generation using constraint solving techniques. SIGSOFT Software Eng. Notes. 23(2), 53–62 (1998)
8. Binh, N.T., Tung, K.T. A novel fitness function of metaheuristic algorithms for test data generation for simulink models based on mutation analysis. J. Syst. Software. 120, 17–30 (2016)
9. Dang, T., Nahhal, T.: Coverage-guided test generation for continuous and hybrid systems. Formal Methods Syst. Des. 34(2), 183–213 (2009)
10. Gadkari, A.A., et al.: Automatic generation of test-cases using model checking for sl/sf models. In: Proceedings of the 4th Model-driven Engineering, Verification and Validation Workshop, March 2014. pp. 33–46. Nashville (2007). https://doi.org/10.1002/stvr.1489
11. Volk, M., Junges, S., Katoen, J.-P.: Fast dynamic fault tree analysis by model checking techniques. IEEE Trans. Ind. Inf. 14(1), 370–379 (2017)
12. Junges, S., et al.: Finite-state Controllers of POMDPs via Parameter Synthesis, In Uncertainty in Artificial Intelligence: Thirty-Fourth Conference, August 6-10, 2018, pp. 519–529. Monterey, (2018)
13. Gao, S., Avigad, J., & Clarke, E. M.: "δ-complete decision procedures for satisfiability over the reals." International Joint Conference on Automated Reasoning. Springer, Berlin (2012)
14. Dutertre, B.: "Yices 2.2." International Conference on Computer Aided Verification. Springer, Cham, 2014. http://yices.csl.sri.com/. Accessed on 31 December 2020
15. Acharya, S., et al.: Design, development and delivery of active learning tools in software verification & validation education. J. Educ. Learn. 7(1), 13–28 (2018)
16. Schordan, M., Beyer, D., Siegel, S.F.: Evaluating tools for software verification (track introduction). In: Proceedings of International Symposium on Leveraging Applications of Formal Methods, pp. 139–143. Limassol (2018)
17. Younes, H.L.: Error control for probabilistic model checking. In: Proceedings of Verification, Model Checking, and Abstract Interpretation. pp. 142–156. Berlin, Charleston (2006)
18. Grosu, R., Smolka, S.A.: Quantitative model checking. In: Preliminary Proceedings of Indian Society of Landscape Architects (ISoLA), pp. 165–174. India (2004)
19. Kwiatkowska, M., Parker, D., Wiltsche, C.: Prism-games: verification and strategy synthesis for stochastic multi-player games with multiple objectives. Int. J. Software Tool. Technol. Tran. 20(2), 195–210 (2018)
20. Legay, A., Delahaye, B., Bensalem, S.: Statistical model checking: an overview. In: Proceedings of International Conference on Runtime Verification. pp. 122–135. St. Julians, Malta (2010)
21. Kwiatkowska, M., Norman, G., Parker, D.: Advances and challenges of probabilistic model checking. In: Proceedings of 48th Annual Allerton Conference Communication, Control, and Computing, 29 Sept.-1 Oct. 2010, pp. 1691–1698. IEEE, Monticello (2010)
22. Calinescu, R., et al.: Efficient synthesis of robust models for stochastic systems. J. Syst. Software. 143, 140–158 (2018)
23. Hutschenreiter, L., Baier, C., Klein, J.: Parametric Markov Chains: PCTL Complexity and Fraction-free Gaussian Elimination. *EPTCS 256*, pp. 16–30. arXiv:1709.02093 (2017)
24. Hartmanns, A., et al.: The quantitative verification benchmark set. In: Proceedings of International Conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 344–350. Prague (2019)
25. Filipovikj, P., et al.: Report Analyzing Industrial Simulink Models by Statistical Model Checking. Technical Report. (2017). http://www.es.mdh.se/publications/4714. Accessed on June 15 2020
26. Vinod, A.P., Gleason, J.D., Oishi, M.M.: Sreachtools: a Matlab stochastic reachability toolbox. In: Proceedings of the 22nd ACM International Conference on Hybrid Systems: Computation and Control, April 2019. pp. 33–38. Montreal (2019)
27. Cabral, F.G., Moreira, M.V., Diene, O.: Online fault diagnosis of modular discrete-event systems. In: Proceedings of 2015 54th IEEE Conference on Decision and Control (CDC). 15-18 December 2015, pp. 4450–4455. IEEE, Osaka (2015)
28. Contant, O., Lafortune, S., Teneketzis, D.: Diagnosability of discrete event systems with modular structure. Discrete Event Dyn. Syst. 16(1), 9–37 (2006)

29. Zhou, C., Kumar, R., Sreenivas, R.S.: Decentralized modular diagnosis of concurrent discrete event systems. In: Proceedings of 2008 9th International Workshop on Discrete Event Systems. 28-30 May 2008, Gothenburg, Sweden. Piscataway, NJ, USA:IEEE. pp. 388–393.(2008)
30. Cabral, F.G., Moreira, M.V.: Synchronous diagnosis of discrete-event systems. IEEE Trans. Autom. Sci. Eng. 17(2), 921–932 (2019)
31. Chen, J., Kumar, R.: Fault detection of discrete-time stochastic systems subject to temporal logic correctness requirements. IEEE Trans. Autom. Sci. Eng. 12(4), 1369–1379 (2015)
32. Li, M., Kumar, R.: Recursive modeling of stateflow as input/output-extended automaton. IEEE Trans. Autom. Sci. Eng. 11(4), 1229–1239 (2013)
33. Zhou, C., Kumar, R.: Semantic translation of simulink diagrams to input/output extended finite automata. Discrete Event Dyn. Syst. 22(2), 223–247 (2012)
34. de Almeida, A.A., Ferreira, W.D.A., da Silva, A.C.: Automation tool to deploy simulink models into programmable system-on-chip. In: Proceedings of 2016 12th IEEE International Conference on Industry Applications (INDUSCON), 20-23 November 2016, pp. 1–7. IEEE, Curitiba (2016)
35. Li, M., et al.: Systems and Methods of Requirements Chaining and Applications Thereof. US Patent 10,585,779 (2020)
36. Baier, C., Katoen, J.-P.: Principles of Model Checking. MIT Press, Cambridge (2008)
37. Clarke, E.M. Jr, et al.: Model Checking. MIT Press, Cambridge, MA, USA (2018)
38. Abate, A., et al.: Approximate model checking of stochastic hybrid systems. Eur. J. Contr. 16(6), 624–641 (2010)
39. Madelaine, F.R., Martin, B.D.: On the complexity of the model checking problem. SIAM J. Comput. 47(3), 769–797 (2018)
40. Hansson, H., Jonsson, B.: A logic for reasoning about time and reliability. Formal Aspect Comput. 6(5), 512–535 (1994)
41. Li, Y., Li, Y., Ma, Z.: Computation tree logic model checking based on possibility measures. Fuzzy Set Syst. 262, 44–59 (2015)
42. Wald, A.: Sequential tests of statistical hypotheses. Ann. Math. Stat. 16(2), 117–186 (1945)
43. Siegmund, D.: Sequential Analysis: Tests and Confidence Intervals. Springer Science & Business Media, Berlin (2013)

How to cite this article: Chen J, S R. Model-based validation of diagnostic software with application in automotive systems. *IET Cyber-Syst. Robot.* 2021;3:140–149. https://doi.org/10.1049/csy2.12016

APPENDIX

Proof to Lemma 1 According to the construction of D^i, $i \in \{1, 2\}$ and $T = D^1 \times D^2$, D^i enters its D part only after the input e_i arrives, also the input e_i does not move D^j where $j \neq i$. Therefore, u would take D^1 to the same state that u_P would take D to; meanwhile, u would take D^2 to the same state that u_F would take D to. Moreover, $f^1 = \text{False} \wedge f^2 = \text{True}$ is true at a state of T upon u, and this concludes that $u_P \in T_P(f)$ and $u_F \in T_F(f)$. It is then trivial to see that u_F is a suffix of u_P. □

Proof to Lemma 2 Consider the case that $T \not\models \phi_1$ and the model checker returns a counterexample $u = u_1 \ldots u_m$ violating ϕ_1. With an abuse of notation, let $u = (\tau | u_1 \ldots u_m)$ be the timed sequence. It is trivial to show that $u_P = (k_1\tau | u_1, u_{k_1+1}, u_{2k_1+1}, \ldots) \in T_P(f)$ and $u_F = (k_2\tau | u_1, u_{k_2+1}, u_{2k_2+1}, \ldots) \in T_F(f)$. Furthermore, according to the construction of D^1, we know that u has the form

$$u = \Bigg(\tau | \underbrace{u_1, \ldots, u_1}_{k_1}, \underbrace{u_{k_1+1}, \ldots, u_{k_1+1}}_{k_1}, \ldots, \underbrace{u_{2k_1+1}, \ldots, u_{2k_1+1}}_{k_1}, \ldots \Bigg)$$

Therefore, for each $m = 0, 1, \ldots$, let $m' = \lfloor mk_2/k_1 \rfloor$. Then $m'k_1\tau \leq mk_2\tau < (m'+1)k_2\tau$, and $u_{mk_2+1} = u_{m'k_1+1}$. Hence u_F is a sampled version of u_P.

Similarly, one can prove the case that $T \not\models \phi_2$. □

Proof to Lemma 3 Since, the construction of D^2 does not change the transition structure of D, nor the guard condition on each edge, $u_F = u$ will take D to the same state as what u will take D^2 to. According to the construct of D^1, whenever the sequence goes through the duplicated transition with guard condition shifted by δ (i.e. $g_e(u + \delta)$), n will be set to 1, and otherwise n is set to 0. Therefore, $u_P = u_1 + n_1\delta, \ldots, u_m + n_m\delta$ will take D to the same state as what u will take D^1 to. Furthermore, $f^1 = \text{False} \wedge f^2 = \text{True}$ is held at a state of T upon u. This concludes the proof. □