Int J Softw Tools Technol Transfer (2017) 19:409–426
DOI 10.1007/s10009-016-0444-z

FMICS 2015/2016

Require, test, and trace IT

Bernhard K. Aichernig[1] · **Klaus Hörmaier**[2] · **Florian Lorber**[1] · **Dejan Ničković**[3] · **Stefan Tiran**[1]

Published online: 29 November 2016

Abstract We propose a framework for requirement-driven test generation that combines contract-based interface theories with model-based testing. We design a specification language, requirement interfaces, for formalizing different views (aspects) of synchronous data-flow systems from informal requirements. Various views of a system, modeled as requirement interfaces, are naturally combined by conjunction. We develop an incremental test generation procedure with several advantages. The test generation is driven by a single requirement interface at a time. It follows that each test assesses a specific aspect or feature of the system, specified by its associated requirement interface. Since we do not explicitly compute the conjunction of all requirement interfaces of the system, we avoid state space explosion while generating tests. However, we incrementally complete a test for a specific feature with the constraints defined by other requirement interfaces. This allows catching violations of any other requirement during test execution, and not only of the one used to generate the test. This framework defines a natural association between informal requirements, their formal specifications, and the generated tests, thus facilitating traceability. Finally, we introduce a fault-based test-case generation technique, called model-based mutation testing, to requirement interfaces. It generates a test suite that covers a set of fault models, guaranteeing the detection of any corresponding faults in deterministic systems under test. We implemented a prototype test generation tool and demonstrate its applicability in two industrial use cases.

The research leading to these results has received funding from the ARTEMIS Joint Undertaking under grant agreements Nos. 269335 and 332830 and from the Austrian Research Promotion Agency (FFG) under grant agreements Nos. 829817 and 838498 for the implementation of the projects MBAT, Combined Model-based Analysis and Testing of Embedded Systems and CRYSTAL, Critical System Engineering Acceleration.

✉ Florian Lorber
florber@ist.tugraz.at

Bernhard K. Aichernig
aichernig@ist.tugraz.at

Klaus Hörmaier
Klaus.Hoermaier@infineon.com

Dejan Ničković
Dejan.Nickovic@ait.ac.at

Stefan Tiran
stiran@ist.tugraz.at; stefan.tiran.fl@ait.ac.at

[1] Graz University of Technology, Graz, Austria
[2] Infineon Technologies Austria AG, Villach, Austria
[3] AIT Austrian Institute of Technology, Seibersdorf, Austria

Keywords Model-based testing · Test-case generation · Requirements engineering · Traceability · Requirement interfaces · Formal specification · Synchronous systems · Consistency checking · Incremental test-case generation · Model-based mutation testing

1 Introduction

Modern software and hardware systems are becoming increasingly complex, resulting in new design challenges. For safety-critical applications, correctness evidence for designed systems must be presented to the regulatory bodies (see the automotive standard ISO 26262 [33]). It follows that verification and validation techniques must be used to provide evidence that the designed system meets its requirements. Testing remains the preferred practice in industry for gaining confidence in the design correctness. In classical testing, an engineer designs a test experiment, i.e., an input vector that is executed on the system under test (SUT) to check whether it satisfies its requirements. Due to the finite number of experiments, testing cannot prove the absence of errors. However, it is an effective technique for catching bugs. Testing remains a predominantly manual and ad-hoc activity that is prone to human errors. As a result, it is often a bottleneck in the complex system design.

Model-based testing (MBT) is a technology that enables systematic and automatic test-case generation (TCG) and execution, thus reducing system design time and cost. In MBT, the SUT is tested for conformance against its specification, a mathematical model of the SUT. In contrast to the specification, that is a formal object, the SUT is a physical implementation with often unknown internal structure, also called a "black-box". The SUT can be accessed by the tester only through its external interface. To reason about the conformance of the SUT to its specification, one needs to rely on the testing assumption [48], that the SUT can react at all times to all inputs and can be modeled in the same language as its specification.

The formal model of the SUT is derived from its informal requirements. The process of formulating, documenting, and maintaining system requirements is called requirement engineering. The requirements are typically written in a textual form, using possibly constrained English, and are gathered in a requirements document. The requirements document is structured into chapters describing various different views of the system, as, e.g., behavior, safety, timing. Intuitively, a system must correctly implement the conjunction of all its requirements. Sometimes, requirements can be inconsistent, resulting in a specification that does not admit any correct implementation.

In this paper, we propose a requirement-driven MBT-framework of synchronous data-flow reactive systems. In contrast to classical MBT, in which the requirements' document is usually formalized into one big monolithic specification, we exploit the structure of the requirements and adopt a multiple-viewpoint approach.

We first introduce requirement interfaces as the formalism for modeling system views as subsets of requirements. It is a state-transition formalism that supports compositional specification of synchronous data-flow systems by means of assume/guarantee rules that we call contracts. We associate subsets of contracts to requirement identifiers to facilitate their tracing to the informal requirements from which the specification is derived. These associations can, later on, be used to generate links between the work products [4], connecting several tools.

A requirement interface is intended to model a specific view of the SUT. We define the conjunction operation that enables combining different views of the SUT. Intuitively, a conjunction of two requirement interfaces is another requirement interface that requires contracts of both interfaces to hold. We assume that the overall specification of the SUT is given as a conjunction of requirement interfaces modeling its different views. Requirement interfaces are inspired by the synchronous interfaces [23], with the difference that we allow hidden variables in addition to the interface (input and output) variables and that the requirement identifiers are part of the formal model. The conjunction operator was first defined in [25] as shared refinement, while [13] establishes the link of the conjunction to multiple-viewpoint modeling and requirement engineering.

We then formally define consistency for requirement interfaces and develop a bounded consistency checking procedure. In addition, we show that falsifying consistency is compositional with respect to conjunction, i.e., the conjunction of an inconsistent interface with any other interface remains inconsistent. Next, we develop a requirement-driven TCG and execution procedure from requirement interfaces, with language inclusion as the conformance relation. We present a procedure for TCG from a specific SUT view, modeled as a requirement interface, and a test purpose. Here, the test purpose is a formal specification of the target state(s) that a test case should cover. Such a test case can be used directly to detect if the implementation by the SUT violates a given requirement, but cannot detect violation of other requirements in the conjunction. Next, we extend this procedure by completing such a partial test case with additional constraints from other view models that enable detection of violations of any other requirement.

Then, we develop a tracing procedure that exploits the natural mapping between informal requirements and our formal model. Thus, inconsistent contracts or failing test cases can be traced back to the violated requirements. We believe that such tracing information provides precious maintenance and debugging information to the engineers.

Finally, we show how to apply fault-based test generation to requirement interfaces. The used technique, called model-based mutation testing [2], is applied to automatically generate a set of test purposes. The corresponding test suite is able to provide fault coverage for a specified set of fault models, under the assumption of a deterministic SUT. The approach includes the following steps: first, we define a set of fault models for requirement interfaces. These are applied to all applicable parts of the contracts in the requirement interface, generating a set of faulty models, called mutants. We then check whether the mutated contract introduces any new behavior. This check is encoded as a test purpose, so we can simply pass it to the previously defined test generation. If the mutation introduces new behavior that deviates from the reference model, it will generate a test; otherwise, the test purpose will be unreachable, and the mutant is considered equivalent.

We illustrate the entire workflow of using requirement interfaces for consistency checking, testing, and tracing in Fig. 1, where the test purpose may be produced by

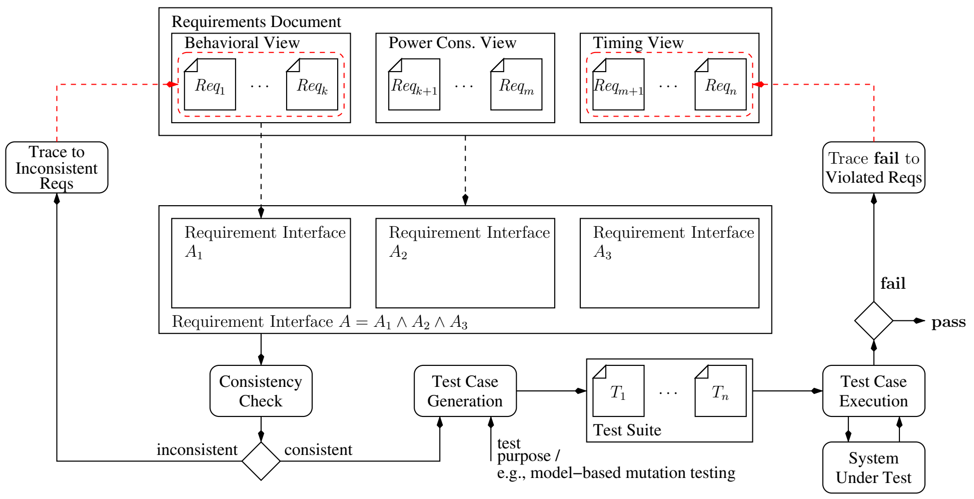

Fig. 1 Overview of using requirement interfaces for testing, analysis, and tracing

model-based mutation testing, or any arbitrary other technique.

Parts of this paper have already been published in the proceedings of the Formal Methods for Industrial Critical Systems 2015 workshop [5]. The current paper improves on that, by adding the theory for model-based mutation testing of requirement interfaces, proofs of all theorems, a second industrial case study and various improvements throughout the paper.

The rest of the paper is structured as follows: first, we introduce requirement interfaces in Sect. 2. Then, in Sect. 3, we present how to perform consistency checks and test-case generation from requirement interfaces, and how to trace the involved requirements from consistency violations or test cases. Next, Sect. 4 gives the theory for applying model-based mutation testing to requirement interfaces. Then, in Sect. 5, we present results of our test-case generation on our running example and on the two industrial case studies. Finally, we discuss related work (Sect. 6) and conclude our work (Sect. 7).

2 Requirement interfaces

We introduce requirement interfaces, a formalism for specification of synchronous data-flow systems. Their semantics is given in the form of labeled transition systems (LTS). We define consistent interfaces as the ones that admit at least one correct implementation. The refinement relation between interfaces is given as language inclusion. Finally, we define the conjunction of requirement interfaces as another interface that subsumes all behaviors of both interfaces.

2.1 Syntax

Let X be a set of typed variables. A valuation v over X is a function that assigns to each $x \in X$ a value $v(x)$ of the appropriate type. We denote by $V(X)$ the set of all valuations over X. We denote by $X' = \{x' \mid x \in X\}$ the set obtained by priming each variable in X. Given a valuation $v \in V(X)$ and a predicate φ on X, we denote by $v \models \varphi$ the fact that φ is satisfied under the variable valuation v. Given two valuations $v, v' \in V(X)$ and a predicate φ on $X \cup X'$, we denote by $(v, v') \models \varphi$ the fact that φ is satisfied by the valuation that assigns to $x \in X$ the value $v(x)$, and to $x' \in X'$ the value $v'(x')$.

Given a subset $Y \subseteq X$ of variables and a valuation $v \in V(X)$, we denote by $\pi(v)[Y]$, the projection of v to Y. We will commonly use the symbol w_Y to denote a valuation projected to the subset $Y \subseteq X$. Given the sets X, $Y_1 \subseteq X$, $Y_2 \subseteq X$, $w_1 \in V(Y_1)$, and $w_2 \in V(Y_2)$, we denote by $w = w_1 \cup w_2$ the valuation $w \in V(Y_1 \cup Y_2)$, such that $\pi(w)[Y_1] = w_1$ and $\pi(w)[Y_2] = w_2$.

Given a set X of variables, we denote by X_I, X_O, and X_H three disjoint partitions of X denoting sets of input, output, and hidden variables, such that $X = X_I \cup X_O \cup X_H$. We denote by $X_{\text{obs}} = X_I \cup X_O$ the set of observable variables and by $X_{\text{ctr}} = X_H \cup X_O$ the set of controllable variables.[1] A contract c on $X \cup X'$, denoted by $(\varphi \vdash \psi)$, is a pair consisting of an assumption predicate φ on $X'_I \cup X$ and a guarantee predicate ψ on $X'_{\text{ctr}} \cup X$. A contract $\hat{c} = (\hat{\varphi} \vdash \hat{\psi})$ is said to be an initial contract if $\hat{\varphi}$ and $\hat{\psi}$ are predicates on X'_I and X'_{ctr}, respectively, and an update contract otherwise. Given two valuations $v, v' \in V(X)$, and a contract $c = (\varphi \vdash \psi)$ over $X \cup X'$, we say that (v, v') satisfies c, denoted by $(v, v') \models c$, if $(v, \pi(v')[X_I]) \models \varphi \rightarrow (v, \pi(v')[X_{\text{ctr}}]) \models \psi$. In addition, we say that (v, v') satisfies the assumption of c, denoted by $(v, v') \models_A c$ if $(v, \pi(v')[X_I]) \models \varphi$. The valuation pair (v, v') satisfies the guarantee of c, denoted by $(v, v') \models_G c$, if $(v, \pi(v')[X_{\text{ctr}}]) \models \psi$).[2]

[1] We adopt SUT-centric conventions to naming the roles of variable.

[2] We sometimes use the direct notation $(v, w'_I) \models_A c$ and $(v, w'_{\text{ctr}}) \models_G c$, where $w_I \in V(X_I)$ and $w_{\text{ctr}} \in V(X_{\text{ctr}})$.

Definition 1 A requirement interface A is a tuple $\langle X_I, X_O, X_H, \hat{C}, C, \mathcal{R}, \rho\rangle$, where

- X_I, X_O, and X_H are disjoint finite sets of input, output, and hidden variables, respectively, and $X = X_I \cup X_O \cup X_H$ denotes the set of all variables;
- $\hat{C}$ and C are finite non-empty sets of the initial and update contracts;
- $\mathcal{R}$ is a finite set of requirement identifiers;
- $\rho : \mathcal{R} \to \mathcal{P}(C \cup \hat{C})$ is a function mapping requirement identifiers to subsets of contracts, such that $\bigcup_{r\in\mathcal{R}} \rho(r) = C \cup \hat{C}$.

We say that a requirement interface is receptive if in any state, it has defined behaviors for all inputs, that is $\bigvee_{(\hat{\varphi}\vdash\hat{\psi})\in\hat{C}} \hat{\varphi}$ and $\bigvee_{(\varphi\vdash\psi)\in C} \varphi$ are both valid. A requirement interface is fully observable if $X_H = \emptyset$. A requirement interface is deterministic if for all $(\hat{\varphi} \vdash \hat{\psi}) \in \hat{C}$, $\hat{\psi}$ has the form $\bigwedge_{x\in X_O} x' = c$,[3] where c is a constant of the appropriate type, and for all $(\varphi \vdash \psi) \in C$, ψ has the form $\bigwedge_{x\in X_{\text{ctr}}} x' = f(X)$,[3] where f is a function over X that has the same type as x.

Example 1 We use an abstract N-bounded FIFO buffer example to illustrate all the concepts introduced in the paper. Let A^{beh} be the behavioral model of the buffer. The buffer has two Boolean input variables enq, deq, i.e., $X_I^{beh} = \{\mathsf{enq}, \mathsf{deq}\}$, two Boolean output variables E, F, i.e., $X_O^{beh} = \{\mathsf{E}, \mathsf{F}\}$, and a bounded integer internal variable $k \in [0 : N]$ for some $N \in \mathbb{N}$, i.e., $X_H^{beh} = \{k\}$. The textual requirements are listed below:

r_0: The buffer is empty and the inputs are ignored in the initial state.
r_1: enq triggers an enqueue operation when the buffer is not full.
r_2: deq triggers a dequeue operation when the buffer is not empty.
r_3: E signals that the buffer is empty.
r_4: F signals that the buffer is full.
r_5: Simultaneous enq and deq (or their simultaneous absence), an enq on the full buffer, or a deq on the empty buffer have no effect.

We formally define A^{beh} as $\hat{C}^{beh} = \{c_0\}$, $C^{beh} = \{c_i \mid i \in [1, 5]\}$, $\mathcal{R}^{beh} = \{r_i \mid i \in [0, 5]\}$ and $\rho^{beh}(r_i) = \{c_i\}$, where

c_0 : **true** $\vdash$ $(k' = 0) \wedge \mathsf{E}' \wedge \neg\mathsf{F}'$
c_1 : $\mathsf{enq}' \wedge \neg\mathsf{deq}' \wedge k < N \ \vdash \ k' = k + 1$
c_2 : $\neg\mathsf{enq}' \wedge \mathsf{deq}' \wedge k > 0 \ \vdash \ k' = k - 1$
c_3 : **true** $\vdash$ $k' = 0 \Leftrightarrow \mathsf{E}'$
c_4 : **true** $\vdash$ $k' = N \Leftrightarrow \mathsf{F}'$
c_5 : $(\mathsf{enq}' = \mathsf{deq}') \vee (\mathsf{enq}' \wedge \mathsf{F}) \vee (\mathsf{deq}' \wedge \mathsf{E}) \ \vdash \ k' = k.$

2.2 Semantics

Given a requirement interface A defined over X, let $V = V(X) \cup \{\hat{v}\}$ denote the set of states in A, where a state v is a valuation $v \in V(X)$ or the initial state $\hat{v} \notin V(X)$. The latter is not a valuation, as the initial contracts do not specify unprimed variables. There is a transition between two states v and v' if (v, v') satisfies all its contracts. The transitions are labeled by the (possibly empty) set of requirement identifiers corresponding to contracts for which (v, v') satisfies their assumptions. The semantics $[[A]]$ of A is the following LTS.

Definition 2 The semantics of the requirement interface A is the LTS $[[A]] = \langle V, \hat{v}, L, T\rangle$, where V is the set of states, $\hat{v}$ is the initial state, $L = \mathcal{P}(\mathcal{R})$ is the set of labels, and $T \subseteq V \times L \times V$ is the transition relation, such that:

- $(\hat{v}, R, v) \in T$ if $v \in V(X)$, $\bigwedge_{\hat{c}\in\hat{C}}(\hat{v}, v) \models \hat{c}$ and $R = \{r \mid (\hat{v}, v) \models_A \hat{c}$ for some $\hat{c} \in \hat{C}$ and $\hat{c} \in \rho(r)\}$;
- $(v, R, v') \in T$ if $v, v' \in V(X)$, $\bigwedge_{c\in C}(v, v') \models c$ and $R = \{r \mid (v, v') \models_A c$ for some $c \in C$ and $c \in \rho(r)\}$.

We say that $\tau = v_0 \xrightarrow{R_1} v_1 \xrightarrow{R_2} \cdots \xrightarrow{R_n} v_n$ is an execution of the requirements interface A if $v_0 = \hat{v}$ and for all $1 \le i \le n-1$, $(v_i, R_{i+1}, v_{i+1}) \in T$. In addition, we use the following notation: (1) $v \xrightarrow{R}$ iff $\exists v' \in V(X)$ s.t. $v \xrightarrow{R} v'$; (2) $v \to v'$ iff $\exists R \in L$ s.t. $v \xrightarrow{R} v'$; (3) $v \to$ iff $\exists v' \in V(X)$ s.t. $v \to v'$; (4) $v \overset{\epsilon}{\Rightarrow} v'$ iff $v = v'$; (5) $v \overset{w}{\Rightarrow} v'$ iff $\exists Y \subseteq X$ s.t. $\pi(v')[Y] = w$ and $v \to v'$; (6) $v \overset{w}{\Rightarrow}$ iff $\exists v', Y \subseteq X$ s.t. $\pi(v')[Y] = w$ and $v \to v'$; (7) $v \xRightarrow{w_1\cdot w_2\cdots w_n} v'$ iff $\exists v_1, \ldots, v_{n-1}, v_n$ s.t. $v \overset{w_1}{\Rightarrow} v_1 \overset{w_2}{\Rightarrow} \cdots v_n \overset{w_n}{\Rightarrow} v'$; and (8) $v \xRightarrow{w_1\cdot w_2\cdots w_n}$ iff $\exists v'$ s.t. $v \xRightarrow{w_1\cdot w_2\cdots w_n} v'$.

We say that a sequence $\sigma \in V(X_{\text{obs}})^*$ is a trace of A if $\hat{v} \overset{\sigma}{\Rightarrow}$. We denote by $\mathcal{L}(A)$ the set of all traces of A. Given a trace σ of A, let A after $\sigma = \{v \mid \hat{v} \overset{\sigma}{\Rightarrow} v\}$. Given a state $v \in V$, let $\text{succ}(v) = \{v' \mid v \to v'\}$ be the set of successors of v.

Example 2 In Fig. 2, we show the LTS $[[A^{beh}]]$ of A^{beh}. For instance, $\hat{v} \xrightarrow{r_0} v_3 \xrightarrow{r_{1,3,4}} v_5 \xrightarrow{r_{3,4,5}} v_6$ is an execution[4] in

[3] Here, we write X_O/X_{ctr} to denote the output/controllable variables involved in that contract.

[4] We use the notation $r_{1,2,3}$ to denote the set $\{r_1, r_2, r_3\}$.

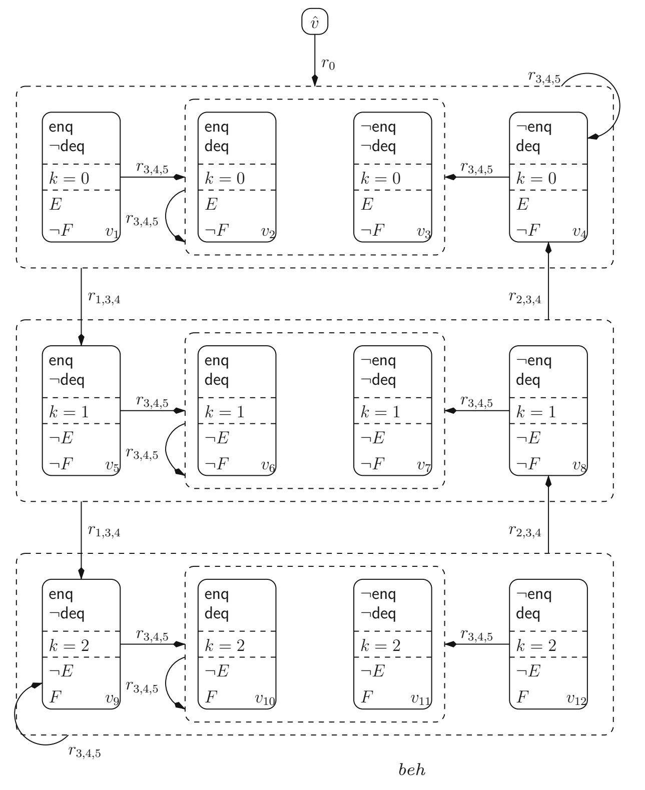
$\hat{v}$
r_0
$r_{3,4,5}$
enq ¬deq $k=0$ E ¬F v_1
enq deq $k=0$ E ¬F v_2
¬enq ¬deq $k=0$ E ¬F v_3
¬enq deq $k=0$ E ¬F v_4
$r_{1,3,4}$
$r_{2,3,4}$
enq ¬deq $k=1$ ¬E ¬F v_5
enq deq $k=1$ ¬E ¬F v_6
¬enq ¬deq $k=1$ ¬E ¬F v_7
¬enq deq $k=1$ ¬E ¬F v_8
enq ¬deq $k=2$ ¬E F v_9
enq deq $k=2$ ¬E F v_{10}
¬enq ¬deq $k=2$ ¬E F v_{11}
¬enq deq $k=2$ ¬E F v_{12}
beh

Fig. 2 Labeled transition graph $[[A^{beh}]]$ illustrating the semantics of the bounded FIFO specification A^{beh}, where $N = 2$

$[[A]]$, and the trace σ induced by the above execution is

$$\sigma = \begin{array}{l}(\neg\mathsf{enq}, \neg\mathsf{deq}, E, \neg F)\\ (\mathsf{enq}, \neg\mathsf{deq}, \neg E, \neg F)\\ (\mathsf{enq}, \mathsf{deq}, \neg E, \neg F).\end{array}$$

2.3 Consistency, refinement, and conjunction

A requirement interface consists of a set of contracts that can be conflicting. Such an interface does not allow any correct implementation. We say that a requirement interface is consistent if it allows at least one correct implementation.

Definition 3 Let A be a requirement interface, $[[A]]$ its associated LTS, $v \in V$ a state, and $\mathcal{C} = \hat{C}$ if v is initial, and C otherwise. We say that v is *consistent*, denoted by cons(v), if for all $w_I \in V(X_I)$, there exists v', such that $w_I = \pi(v')[X_I]$, $\bigwedge_{c\in\mathcal{C}}(v, v') \models c$ and cons(v'). We say that A is consistent if cons($\hat{v}$).

Example 3 A^{beh} is consistent, that is, every reachable state accepts every input valuation and generates an output valuation satisfying all contracts. Consider now replacing c_2 in A^{beh} with the contract $c_2' : \neg\mathsf{enq}' \wedge \mathsf{deq}' \wedge k \geq 0 \vdash k' = k - 1$, that incorrectly models r_2 and decreases the counter k upon deq even when the buffer is empty, setting it to the value minus one. This causes an inconsistency with the contracts c_3 and c_5 which state that if k equals zero, the buffer is empty, and that dequeue on an empty buffer has no effect on k.

We define the refinement relation between two requirement interfaces A^1 and A^2, denoted by $A^2 \preceq A^1$, as trace inclusion.

Definition 4 Let A^1 and A^2 be two requirement interfaces. We say that A^2 refines A^1, denoted by $A^2 \preceq A^1$, if (1) A^1 and A^2 have the same sets X_I, X_O, and X_H of variables; and (2) $\mathcal{L}(A^1) \subseteq \mathcal{L}(A^2)$.

We use a requirement interface to model a view of a system. Multiple views are combined by conjunction. The conjunction of two requirement interfaces is another requirement interface that is either inconsistent due to a conflict between views, or is the greatest lower bound with respect to the refinement relation. The conjunction of A^1 and A^2, denoted by $A^1 \wedge A^2$, is defined if the two interfaces share the same sets X_I, X_O, and X_H of variables.

Definition 5 Let $A^1 = \langle X_I, X_H, X_O, \hat{C}^1, C^1, \mathcal{R}^1, \rho^1\rangle$ and $A^2 = \langle X_I, X_H, X_O, \hat{C}^2, C^2, \mathcal{R}^2, \rho^2\rangle$ be two Their conjunction $A = A^1 \wedge A^2$ is the requirement interface $\langle X_I, X_H, X_O, \hat{C}, C, \mathcal{R}, \rho\rangle$, where

- $\hat{C} = \hat{C}^1 \cup \hat{C}^2$ and $C = C^1 \cup C^2$;
- $\mathcal{R} = \mathcal{R}^1 \cup \mathcal{R}^2$; and
- $\rho(r) = \rho^1(r)$ if $r \in \rho^1$ and $\rho(r) = \rho^2(r)$ otherwise.

Remark For refinement and conjunction, we require the two interfaces to share the same alphabet. This additional condition is used to simplify definitions. It does not restrict the modeling—arbitrary interfaces can have their alphabets equalized without changing their properties by taking union of respective input, output, and hidden variables. Contracts in the transformed interfaces do not constrain newly introduced variables. For requirement interfaces A^1 and A^2, alphabet equalization is defined if $(X_I^1 \cup X_I^2) \cap (X_{\text{ctr}}^1 \cup X_{\text{ctr}}^2) = (X_O^1 \cup X_O^2) \cap (X_H^1 \cup X_H^2) = \emptyset$. Otherwise, $A_1 \not\preceq A_2$ and vice versa, and $A^1 \wedge A^2$ is not defined.

Example 4 We now consider a power consumption view of the bounded FIFO buffer. Its model A^{pc} has the Boolean input variables enq and deq and a bounded integer output variable pc. The following textual requirements specify A^{pc}:

r_a: The power consumption equals zero when no enq/deq is requested.
r_b: The power consumption is bounded to two units otherwise.

The interface A^{pc} consists of $\hat{C}^{pc} = C^{pc} = \{c_a, c_b\}$, $\mathcal{R}^{pc} = \{r_i \mid i \in \{a, b\}\}$, and $\rho(r_i) = \{c_i\}$, where:

$$\begin{aligned} c_a &: \neg\mathsf{enq} \wedge \neg\mathsf{deq} \vdash \mathsf{pc}' = 0 \\ c_b &: \mathsf{enq} \vee \mathsf{deq} \quad\ \vdash \mathsf{pc}' \leq 2. \end{aligned}$$

The conjunction $A^{buf} = A^{beh} \wedge A^{pc}$ is the requirement interface where $X_I^{buf} = \{\mathsf{enq}, \mathsf{deq}\}$, $X_O^{buf} = \{\mathsf{E}, \mathsf{F}, \mathsf{pc}\}$, $X_H^{buf} = \{k\}$, $\hat{C}^{buf} = \{c_0, c_a, c_b\}$, $C^{buf} = \{c_1, c_2, c_3, c_4, c_5, c_a, c_b\}$, $\mathcal{R}^{pc} = \{r_i \mid i \in \{a, b, 0, 1, 2, 3, 4, 5\}\}$, and $\rho(r_i) = \{c_i\}$.

We now show some properties of requirement interfaces. The conjunction of two requirement interfaces with the same alphabet is the intersection of their traces.

Theorem 1 *Let A^1 and A^2 be two consistent requirement interfaces defined over the same alphabet. Then, either $A^1 \wedge A^2$ is inconsistent, or $\mathcal{L}(A^1 \wedge A^2) = \mathcal{L}(A^1) \cap \mathcal{L}(A^2)$.*

A proof of the theorem can be found in the appendix.

The conjunction of two requirement interfaces with the same alphabet is either inconsistent, or it is the greatest lower bound with respect to refinement.

Theorem 2 *Let A^1 and A^2 be two consistent requirement interfaces defined over the same alphabet, such that $A^1 \wedge A^2$ is consistent. Then, $A^1 \wedge A^2 \preceq A^1$ and $A^1 \wedge A^2 \preceq A^2$, and for all consistent requirement interfaces A, if $A \preceq A^1$ and $A \preceq A^2$, then $A \preceq A^1 \wedge A^2$.*

A proof of the theorem can be found in the appendix.

The following theorem states that the conjunction of an inconsistent requirement interface with any other interface remains inconsistent. This result enables incremental detection of inconsistent specifications.

Theorem 3 *Let A be an inconsistent requirement interface. Then, for all consistent requirement interfaces A' with the same alphabet as A, $A \wedge A'$ is also inconsistent.*

The proof follows directly from the definition of conjunction, which constrains the guarantees of individual interfaces.

3 Consistency, testing, and tracing

In this section, we present our test-case generation and execution framework and instantiate it with bounded model checking techniques. For now, we assume that all variables range over finite domains. This restriction can be lifted by considering richer data domains in addition to theories that have decidable quantifier elimination, such as linear arithmetic over reals. Before executing the test-case generation, we can apply a consistency check on the requirement interface, to ensure the generation starts from an implementable specification.

3.1 Bounded consistency checking

To check k-bounded consistency of a requirement interface A, we unfold the transition relation of A in k steps, and encode the definition of consistency in a straight-forward manner. The transition relation of an interface is the conjunction of its contracts, where a contract is represented as an implication between its assumption and guarantee predicates. Let

$$\hat{\theta} = \bigwedge_{(\hat{\varphi} \vdash \hat{\psi}) \in \hat{C}} \hat{\varphi} \rightarrow \hat{\psi}$$

and

$$\theta = \bigwedge_{(\varphi \vdash \psi) \in C} \varphi \rightarrow \psi.$$

Then, the k-bounded consistency check for A corresponds to checking the satisfiability of the formula

$\forall X_I^0 . \exists X_{\text{ctr}}^0 \ldots \forall X_I^k . \exists X_{\text{ctr}}^k . \ \theta^0 \wedge \theta^1 \wedge \cdots \wedge \theta^k$ where

$\theta^0 = \hat{\theta}[X' \backslash X^0]$ and $\theta^i = \theta[X' \backslash X^i, X \backslash X^{i-1}]$, $1 \leq i \leq k$.

To implement a consistency check in our prototype, we transform it to a satisfiability problem and use the SMT solver Z3 to solve it.

The first step is to construct a symbolic representation of the initial contracts and the transition relation.

The transition relation is then unfolded for each step by renaming the occurrence of each variable, such that it is indexed by the corresponding step. In each step i, the undecorated variables are indexed with $i - 1$, while the decorated variables are indexed with i, thus keeping the relation between the valuations of each step. Given a set X of variables, we denote by X^i the copy of the set, in which every variable is indexed by i.

The conjunction of all instances up to a certain depth is an open formula, leaving all variables free. The consistency check is bounded by a certain depth.

3.2 Test-case generation

A test case is an experiment executed on the SUT by the tester. We assume that the SUT is a black-box that is only accessed via its observable interface. We assume that it can be modeled as an input-enabled, deterministic[5] requirement interface. Without loss of generality, we can represent the SUT as a total

[5] The restriction to deterministic implementations is for presentation purposes only, and the technique is general and can also be applied to non-deterministic systems.

sequential function SUT : $V(X_I) \times V(X_{\text{obs}})^* \to V(X_O)$. A test case T_A for a requirement interface A over X takes a history of actual input/output observations $\sigma \in \mathcal{L}(A)$ and returns either the next input value to be executed or a verdict. Hence, a test case can be represented as a partial function $T_A : \mathcal{L}(A) \to V(X_I) \cup \{\textbf{pass}, \textbf{fail}\}$.

We first consider the problem of generating a test case from A. The test-case generation procedure is driven by a test purpose. Here, a test purpose is a condition specifying the target set of states that a test execution should reach. Hence, it is a formula Π defined over X_{obs}.

Given a requirement interface A, let $\hat{\phi} = \bigvee_{(\hat{\varphi}\vdash\hat{\psi})\in\hat{C}} \hat{\varphi} \wedge \bigwedge_{(\hat{\varphi}\vdash\hat{\psi})\in\hat{C}} \hat{\varphi} \to \hat{\psi}$ and $\phi = \bigvee_{(\varphi\vdash\psi)\in C} \varphi \wedge \bigwedge_{(\varphi\vdash\psi)\in C} \varphi \to \psi$. The predicates $\hat{\phi}$ and ϕ encode the transition relation of A, with the additional requirement that at least one assumption must be satisfied, thus avoiding input vectors for which the test purpose can be trivially reached due to underspecification. A test case for A that can reach Π is defined iff there exists a trace $\sigma = \sigma' \cdot w_{\text{obs}}$ in $\mathcal{L}(A)$, such that $w_{\text{obs}} \models \Pi$. The test purpose Π can be reached in A in at most k steps if

$$\exists X^0, \ldots, X^k . \phi^0 \wedge \cdots \wedge \phi^k \wedge \bigvee_{i \leq k} \Pi[X_{\text{obs}} \backslash X^i_{\text{obs}}],$$

where $\phi^0 = \hat{\phi}[X' \backslash X^0]$ and $\phi^i = \phi[X' \backslash X^i, X \backslash X^{i-1}]$ represent the transition relation of A unfolded in the i-th step.

Given A and Π, assume that there exists a trace σ in $\mathcal{L}(A)$ that reaches Π. Let σ_I be a projection to inputs, $\pi(\sigma)[X_I] = w_I^0 \cdot w_I^1 \cdots w_I^n$. We first compute $\omega_{\sigma_I,A}$ (see Algorithm 1), a formula[6] characterizing the set of output sequences that A allows on input σ_I.

Algorithm 1 OutMonitor

```
Input: $\sigma_I = w_I^0 \cdot w_I^1 \cdots w_I^n, A$
Output: $\omega_{\sigma_I,A}$
1: $\omega^0_{\sigma_I,A} \leftarrow \hat{\theta}[X'_I \backslash w^0_I, X'_{\text{ctr}} \backslash X^0_{\text{ctr}}]$
2: for $i = 1$ to $n$ do
3:     $\omega^i_{\sigma_I,A} \leftarrow \theta[X_I \backslash w^{i-1}_I, X'_I \backslash w^i_I, X_{\text{ctr}} \backslash X^{i-1}_{\text{ctr}}, X'_{\text{ctr}} \backslash X^i_{\text{ctr}}]$
4: end for
5: $\omega^*_{\sigma_I,A} \leftarrow \omega^0_{\sigma_I,A} \wedge \cdots \wedge \omega^n_{\sigma_I,A}$
6: $\omega_{\sigma_I,A} \leftarrow \textbf{qe}(\exists X^0_H, X^1_H, \ldots, X^n_H . \omega^*_{\sigma_I,A})$
7: return $\omega_{\sigma_I,A}$
```

Let $\hat{\theta} = \bigwedge_{(\hat{\varphi}\vdash\hat{\psi})\in\hat{C}} \hat{\varphi} \to \hat{\psi}$ and $\theta = \bigwedge_{(\varphi\vdash\psi)} \varphi \to \psi$. For every step i, we represent by $\omega^i_{\sigma_I,A}$ the allowed behavior of A constrained by σ_I (Lines 1–4). The formula $\omega^*_{\sigma_I,A}$ (Line 5) describes the transition relation of A, unfolded to n steps, and constrained by σ_I. However, this formula refers to the hidden variables of A and cannot be directly used to characterize the set of output sequences allowed by A under σ_I. Since any implementation of hidden variables that preserve correctness of the outputs is acceptable, it suffices to existentially quantify over hidden variables in $\omega^*_{\sigma_I,A}$. After eliminating the existential quantifiers with strategy **qe**, we obtain a simplified formula $\omega_{\sigma_I,A}$ over output variables only (Line 6).

Algorithm 2 $T_{\sigma_I,A}$

```
Input: $\sigma_I = w_I^0 \cdots w_I^n, A, \sigma = w^0_{\text{obs}} \cdots w^k_{\text{obs}}$
Output: {pass, fail}
1: $\omega_{\sigma_I,A} \leftarrow$ OutMonitor($\sigma_I, A$)
2: for $i = 0$ to $k$ do
3:     $w^i_O \leftarrow \pi(w^i_{\text{obs}})[X_O]$
4: end for
5: $\omega^{0,k}_{\sigma_I,A} \leftarrow \omega_{\sigma_I,A}[X^0_O \backslash w^0_O, \ldots, X^k_O \backslash w^k_O]$
6: if $\omega^{0,k}_{\sigma_I,A} =$ true then
7:     return pass
8: else if $\omega^{0,k}_{\sigma_I,A} =$ false then
9:     return fail
10: end if
```

Let $T_{\sigma_I,A}$ be a test case, parameterized by the input sequence σ_I and the requirement interface A from which it was generated. It is a partial function, where $T_{\sigma_I,A}(\sigma)$ is defined if $|\sigma| \leq |\sigma_I|$ and for all $0 \leq i \leq |\sigma|$, $w^i_I = \pi(w^i_{\text{obs}})[X_I]$, where $\sigma_I = w^0_I \cdots w^n_I$ and $\sigma = w^0_{\text{obs}} \cdots w^k_{\text{obs}}$. Algorithm 2 gives a constructive definition of the test case $T_{\sigma_I,A}$. It starts by producing the output monitor for the given input sequence (Line 1). Then, it substitutes all output variables in the monitor, by the outputs observed from the SUT (Lines 2–5). If the monitor is satisfied by the outputs, it returns the verdict *pass*; otherwise, it returns *fail*.

Incremental test-case generation So far, we considered test-case generation for a complete requirement interface A, without considering its internal structure. We now describe how test cases can be incrementally generated when the interface A consists of multiple views,[7] i.e., $A = A^1 \wedge A^2$. Let Π be a test purpose for the view modeled with A_1. We first check whether Π can be reached in A^1, which is a simpler check than doing it on the conjunction $A^1 \wedge A^2$. If Π can be reached, we fix the input sequence σ_I that steers A^1 to Π. Instead of creating the test case T_{σ_I,A^1}, we generate $T_{\sigma_I,A^1\wedge A^2}$, which keeps σ_I as the input sequence, but collects output guarantees of A^1 and A^2. Such a test case steers the SUT towards the test purpose in the view modeled by A^1, but is able to detect possible violations of both A^1 and A^2.

[6] The formula $\omega_{\sigma_I,A}$ can be seen as a monitor for A under input σ_I.

[7] We consider two views for the sake of simplicity.

We note that test-case generation for fully observable interfaces is simpler than the general case, because there is no need for the quantifier elimination, due to the absence of hidden variables in the model. A test case from a deterministic interface is even simpler as it is a direct mapping from the observable trace that reaches the test purpose—there is no need to collect constraints on the output, since the deterministic interface does not admit any freedom to the implementation on the choice of output valuations.

Example 5 Consider the requirement interface A_{beh} for the behavioral view of the two-bounded buffer, and the test purpose F. Our test-case generation procedure gives the input vector σ_I of size 3, such that

$$\sigma_I = \begin{array}{l}(\mathsf{enq}, \mathsf{deq})\\ (\mathsf{enq}, \neg\mathsf{deq})\\ (\mathsf{enq}, \neg\mathsf{deq}).\end{array}$$

The observable output constraints for σ_I (which are encoded in OutMonitor) are $\mathsf{E} \wedge \neg\mathsf{F}$ in Step 0, $\neg\mathsf{E} \wedge \neg\mathsf{F}$ in Step 1, and $\neg\mathsf{E} \wedge \mathsf{F}$ in Step 2. Together, the input vector σ_I and the associated output constraints form the test case $T_{\sigma_I,beh}$. Using the incremental test-case generation procedure, we can extend $T_{\sigma_I,beh}$ to a test case $T_{\sigma_I,buf}$ that also considers the power consumption view of the buffer, resulting in output constraints $\mathsf{E} \wedge \neg\mathsf{F} \wedge \mathsf{pc} \leq 2$ in Step 0, $\neg\mathsf{E} \wedge \neg\mathsf{F} \wedge \mathsf{pc} \leq 2$ in Step 1, and $\neg\mathsf{E} \wedge \mathsf{F} \wedge \mathsf{pc} \leq 2$ in Step 2.

Algorithm 3 TestExec

```
Input: SUT, $T_{\sigma_I,A}$
Output: {pass, fail}
1: in : $V(X_I) \cup \{\mathbf{pass}, \mathbf{fail}\}$
2: out : $V(X_O)$
3: $\sigma \leftarrow \epsilon$
4: in $\leftarrow T_{\sigma_I,A}(\sigma_I, A, \sigma)$
5: while in $\notin \{\mathbf{pass}, \mathbf{fail}\}$ do
6:     out $\leftarrow$ SUT(in, $\sigma$)
7:     $\sigma \leftarrow \sigma \cdot (\text{in} \cup \text{out})$
8:     in $\leftarrow T_{\sigma_I,A}(\sigma_I, A, \sigma)$
9: end while
10: return in
```

3.3 Test-case execution

Let A be a requirement interface, SUT a system under test with the same set of variables as A, and $T_{\sigma_I,A}$ a test case generated from A. Algorithm 3 defines the test-case execution procedure TestExec that takes as input the SUT and $T_{\sigma_I,A}$ and outputs a verdict **pass** or **fail**. TestExec gets the next test input *in* from the given test case $T_{\sigma_I,A}$ (Lines 4, 8), stimulates at every step the system under test with this input, and waits for an output *out* (Line 6). The new inputs/outputs observed are stored in σ (Line 7), which is given as input to $T_{\sigma_I,A}$. The test case monitors if the observed output is correct with respect to A. The procedure continues until a **pass** or **fail** verdict is reached (Line 5). Finally, the verdict is returned (Line 10).

Proposition 1 *Let A, $T_{\sigma_I,A}$, and* SUT *be arbitrary requirement interface, test case generated from A, and a system under test, respectively. Then, we have*

1. *if $I \preceq A$, then TestExec*(SUT, $T_{\sigma_I,A}$) = ***pass****;*
2. *if TestExec*(SUT, $T_{\sigma_I,A}$) = ***fail****, then* SUT $\npreceq A$.

Proposition 1 immediately holds for test cases generated incrementally from a requirement interface of the form $A = A^1 \wedge A^2$. In addition, we notice that a test case T_{σ_I,A^1} generated from a single view A^1 of A does not need to be extended to be useful, and can be used to incrementally show that a SUT does not conform to its specification. We state the property in the following corollary that follows directly from Proposition 1 and Theorem 2.

Corollary 1 *Let $A = A^1 \wedge A^2$ be an arbitrary requirement interface composed of A^1 and A^2,* SUT *an arbitrary system under test, and T_{σ_I,A^1} an arbitrary test case generated from A^1. Then, if TestExec*(SUT, T_{σ_I,A^1}) = ***fail****, then* SUT $\npreceq A^1 \wedge A^2$.

Algorithm 4 SUT: a 3-place-buffer implementation.

```
Input: enq, dec
Output: E, F, pc
1: wait for inputs
2: $k \leftarrow 0$; E $\leftarrow$ true; F $\leftarrow$ false
3: if ¬enq ∧ ¬dec then
4:     pc $\leftarrow 0$
5: else
6:     pc $\leftarrow 1$
7: end if
8: while true do
9:     wait for inputs
10:    if enq ∧ ¬dec ∧ $k < 3$ then
11:        $k \leftarrow k + 1$
12:    else if ¬enq ∧ dec ∧ $k > 0$ then
13:        $k \leftarrow k - 1$
14:    end if
15:    if ¬enq ∧ ¬dec then
16:        pc $\leftarrow 0$
17:    else
18:        pc $\leftarrow 1$
19:    end if
20:    if $k = 3$ then
21:        F $\leftarrow$ true; E $\leftarrow$ false
22:    else if $k = 0$ then
23:        F $\leftarrow$ false; E $\leftarrow$ true
24:    else
25:        F $\leftarrow$ false; E $\leftarrow$ false
26:    end if
27: end while
```

Example 6 Consider as an SUT the implementation of a 3-place-buffer, as illustrated in Algorithm 4. We assume that the power consumption is updated directly in a PC variable. Although SUT is correctly implementing a 3-place-buffer, it is a faulty implementation of a 2-place-buffer. In fact, when SUT already contains two items, the buffer is still not full, which is in contrast with requirement r_4 of a 2-place-buffer. Executing tests $T_{\sigma_I,beh}$ and $T_{\sigma_I,buf}$ from Example 5 will both result in a **fail** test verdict.

3.4 Traceability

Requirement identifiers as first-class elements in requirement interfaces facilitate traceability between informal requirements, views, and test cases. A test case generated from a view A^i of an interface $A = A^1 \wedge \cdots \wedge A^n$ is naturally mapped to the set $\mathcal{R}^i$ of requirements. In addition, requirement identifiers enable tracing violations caught during consistency checking and test-case execution back to the conflicting/violated requirements.

Tracing inconsistent interfaces to conflicting requirements When we detect an inconsistency in a requirement interface A defining a set of contracts C, we use QuickXPlain, a standard conflict set detection algorithm [36], to compute a minimal set of contracts $C' \subseteq C$, such that C' is inconsistent. Once we have computed C', we use the requirement mapping function ρ defined in A, to trace back the set $\mathcal{R}' \subseteq \mathcal{R}$ of conflicting requirements.

Tracing fail verdicts to violated requirements In fully observable interfaces, every trace induces at most one execution. In that case, a test case resulting in **fail** can be traced to a unique set of violated requirements. This is not the case in general for interfaces with hidden variables. A trace that violates such an interface may induce multiple executions resulting in **fail** with different valuations of hidden variables, and thus different sets of violated requirements. In this case, we report all sets to the user, but ignore internal valuations that would introduce an internal requirement violation before inducing the visible violation.

We propose a tracing procedure *TraceFailTC*, presented in Algorithm 5, that gives useful debugging data regarding violation of test cases in the general case. The algorithm takes as input a requirement interface A and a trace $\sigma \notin \mathcal{L}(A)$. The trace σ that is given as input to the algorithm is obtained from executing a test case for A that leads to a **fail** verdict. The algorithm runs a main loop that at each iteration computes a debugging pair that consists of an execution $\tau = \pi(\sigma)[X_{\text{obs}}]$ and a set failR $\subseteq \mathcal{R}$ of requirements.[8] The execution τ completes the faulty trace with valuations of hidden variables that are consistent with the violation of the requirement interface in the last step. The set failR contains all the requirements that are violated by the execution τ. We initialize the algorithm by setting an auxiliary variable C^* to the set of all update contracts C (Line 3). In every iteration of the main loop, we encode in ϕ^*_{obs} all the executions induced by σ that violate at least one contract in C^* (Lines 6 and 7). In the next step (Line 8), we check the satisfiability of the formula ϕ^*_{obs} (**sat**(ϕ^*_{obs})), a function that returns $b =$ **true**, and a sequence (model) of hidden variable valuations $w^0_H, \ldots, w^n_H$ if ϕ^*_{obs} is satisfiable, and ($b =$ **false**, $\sigma_H = \epsilon$) otherwise. In the former case, we combine σ and σ_H into an execution τ (Line 10). We collect in failR all requirements that are violated by τ and remove the corresponding contracts from C^* (Lines 11–16). The debugging pair (τ, failR) is added to debugSet (Line 16). The procedure terminates and returns debugSet when either C^* is empty or σ cannot violate any remaining contract in C^*, thus ensuring that every requirement that can be violated by σ is part of at least one debugging pair in debugSet.

Algorithm 5 *TraceFailTC*

```
Input: $\sigma = w^0_{\text{obs}} \cdots w^n_{\text{obs}}, A$
Output: debugSet
1: debugSet $\leftarrow \emptyset$
2: failR $\leftarrow \emptyset$
3: $C^* \leftarrow C$
4: $b \leftarrow$ true
5: while $b$ do
6:   $\phi^*_{\text{obs}} \leftarrow \phi^0 \wedge \ldots \wedge \phi^{n-1}$
       $\wedge(\bigvee_{(\varphi,\psi)\in C^*}(\varphi \wedge \neg\psi))\ [X \backslash X^{n-1}, X' \backslash X^n]$
7:   $\phi^*_{\text{obs}} \leftarrow \phi^*_{\text{obs}}[X^0_{\text{obs}} \backslash w^0_{\text{obs}}, \ldots, X^n_{\text{obs}} \backslash w^n_{\text{obs}}]$
8:   $((w^0_H, \ldots, w^n_H), b) \leftarrow \textbf{sat}(\phi^*_{\text{obs}})$
9:   if $b$ then
10:    $\tau \leftarrow (w^0_{\text{obs}} \cup w^0_H) \cdots (w^n_{\text{obs}} \cup w^n_H)$
11:    for all $c = (\varphi, \psi) \in C$ do
12:      if $(w^{n-1}_{\text{obs}} \cup w^{n-1}_H, w^n_{\text{obs}} \cup w^n_H) \models \varphi \wedge \neg\psi$ then
13:        failR $\leftarrow$ failR $\cup \{r \mid c \in \rho(r)\}$;
14:        $C^* \leftarrow C^* \backslash \{c\}$
15:      end if
16:    end for
17:    debugSet $\leftarrow$ debugSet $\cup \{(\tau, \text{failR})\}$
18:    failR $\leftarrow \emptyset$
19:  end if
20: end while
21: return debugSet
```

Example 7 Consider the execution trace

$$\sigma = \begin{matrix} (\text{enq}, \text{deq}, \text{E}, \neg\text{F}) \\ (\text{enq}, \neg\text{deq}, \neg\text{E}, \neg\text{F}) \\ (\text{enq}, \neg\text{deq}, \neg\text{E}, \neg\text{F}) \end{matrix}$$

that results in a **fail** verdict when executing the test $T_{\sigma_I,beh}$. The tracing procedure gives as debugging information the set debugSet $= \{(\tau_1, \{r_4\}), (\tau_2, \{r_1, r_3\})\}$, where τ_1 and τ_2 correspond to the following executions that can lead to vio-

[8] We assume that the trace does not violate initial contracts to simplify the presentation. The extension to the general case is straightforward.

lations of requirements r_4 and r_1, r_3, respectively.

$$\tau_1 = \begin{array}{l}(\mathsf{enq}, \mathsf{deq}, k = 0, \mathsf{E}, \neg\mathsf{F})\\ (\mathsf{enq}, \neg\mathsf{deq}, k = 1, \neg\mathsf{E}, \neg\mathsf{F})\\ (\mathsf{enq}, \neg\mathsf{deq}, k = 2, \neg\mathsf{E}, \neg\mathsf{F})\end{array}$$

$$\tau_2 = \begin{array}{l}(\mathsf{enq}, \mathsf{deq}, k = 0, \mathsf{E}, \neg\mathsf{F})\\ (\mathsf{enq}, \neg\mathsf{deq}, k = 1, \neg\mathsf{E}, \neg\mathsf{F})\\ (\mathsf{enq}, \neg\mathsf{deq}, k = 0, \neg\mathsf{E}, \neg\mathsf{F}).\end{array}$$

Requirements r_0 and r_5 cannot be violated in the last step of this test execution. We note that accessing the faulty 2-buffer implementation I from Algorithm 4, the debugging pair $(\tau_1, \{r_4\})$ would allow to exactly localize the error and trace it back to the violation of the requirement r_4.

For requirement interfaces with hidden variables, the underlying implementation is only partially observable. The best that the tracing procedure can do when the execution of a test leads to the **fail** verdict is to complete missing hidden variables with valuations that are consistent with the partial observations of input and output variables. It follows that the debugSet consists of “hints” on possible violated requirements and the causes of their violation. We note that Algorithm 5 attempts at finding the right compromise between minimizing the amount of data presented to the designer, while still providing useful information. In particular, it focuses on implementation errors that occur at the time of the failure, for both the hidden and the output variables. We note that in some faulty implementations, errors in updating hidden variables may not immediately result in observable faults. For instance, in the execution

$$\tau_3 = \begin{array}{l}(\mathsf{enq}, \mathsf{deq}, k = 1, \mathsf{E}, \neg\mathsf{F})\\ (\mathsf{enq}, \neg\mathsf{deq}, k = 1, \neg\mathsf{E}, \neg\mathsf{F})\\ (\mathsf{enq}, \neg\mathsf{deq}, k = 1, \neg\mathsf{E}, \neg\mathsf{F})\end{array}$$

the requirement r_0 is immediately violated in the initial step, but the implementation errors are only observed in the last step of the test execution. Algorithm 5 does not give such executions as possible causes that lead to a **fail** verdict. It is a design choice—we believe that choosing hidden variables without any restriction would result in executions that are too arbitrary and have little debugging value.

4 Model-based mutation testing

In this section, we apply a fault-based variant of model-based testing to requirement interfaces. In MBT, test cases are generated according to predefined coverage criteria, producing test suites that, e.g., cover all states in the specification model, or, in the case of a contract-based specification, enable all assumptions at least once.

Similar to that, we generate a test suite covering a set of faults. The faults are specified via a set of mutation operators that apply specific faults to all applicable parts of the model. When applied to requirement interfaces, we mutate one contract at a time. Then, we check for conformance between the original requirement interface and the mutated one. If the mutated requirement interface can produce controllable variable values that are forbidden by the original, the conformance is violated. In that case, we produce a test case leading exactly to that violation. If that test case is executed on a deterministic SUT and passes, we can guarantee that the corresponding fault was not implemented. Thus, by generating all tests for all fault models, we can prove the absence of all corresponding faults in the system.

Definition 6 We define a mutation operator μ as a function $\mu : C \rightarrow 2^C$, which takes a contract $c = (\varphi \vdash \psi) \in C$ and produces a set of mutated contracts $C^\mu \subseteq C$, where a specific kind of fault is applied to all valid parts of ψ. We only consider mutations in the guarantee, as the fault models should simulate situations where the system produces wrong outputs, after receiving valid inputs.

We currently consider the following fault models:

1. *Off-by-one* Mutate every integer constant or variable, both by adding and subtracting 1,
2. *Negation* Flip every boolean constant or variable,
3. *Change comparison operators* Replace equality operators by inequality operators, and vice versa; replace every operator in $\{<=, <, >, >=\}$ by every of the operators in $\{<=, <, ==, >, >=\}$.
4. *Change and/or* Replace every *and* operator by an *or* operator and vice versa,
5. *Change implication/bi-implication* Replace every implication by a bi-implication and vice versa;

Definition 7 A mutant $c_m = (\varphi \vdash \psi_m) \in C^\mu$ is an intentionally altered (mutated) version of the contract $c = (\varphi \vdash \psi)$. A mutant is called a first-order mutant, if it only contains one fault. This paper only considers first-order mutants.

If a mutation does not introduce new behavior to the requirement interface, it is considered an equivalent mutation. If it leads to an inconsistency, it is considered an unproductive mutation .

Given the contract $(\varphi \vdash \psi) \in C$, we denote by $\bar{C}$ the set of the other contracts in the requirement interface, i.e., $\bar{C} = C \setminus \{(\varphi \vdash \psi)\}$, by C^μ the set of mutants obtained by applying all mutation operators to $(\varphi \vdash \psi)$ and by $c_m = (\varphi \vdash \psi_m)$ one single mutant in C^μ.

c_m is a non-equivalent mutation, if there exist two valuations v, v', so that:

- v is reachable from $\hat{v}$
- $(v, v') \models \varphi$
- $\forall_{(\bar{\varphi} \vdash \bar{\psi}) \in \bar{C}} (v, v') \models (\bar{\varphi} \vdash \bar{\psi})$
- $(v, v') \models (\varphi \vdash \psi_m) \wedge \neg(\varphi \vdash \psi)$.

We considered a mutant k-equivalent to the original requirement interface, if it is equivalent up to a bound k.

The test purpose Π for detecting $(\varphi \vdash \psi_m)$ can be encoded by the formula

$$\Pi = \varphi \wedge \psi_m \wedge \neg\psi \wedge \bigwedge_{(\bar{\varphi} \vdash \bar{\psi}) \in \bar{C}} (\bar{\varphi} \rightarrow \bar{\psi}).$$

The reachability formula for such a test purpose differs from the one presented in Sect. 3.2 in two details: in the step of the test purpose, the transition relation does not hold, as we require the original contract to be violated. In addition, a test purpose is a relation over primed and unprimed variables.

A test purpose Π can be reached in step k, if

$$\exists X^0, \ldots, X^k . \phi^0 \wedge \cdots \wedge \phi^{k-1} \wedge \Pi[X \backslash X^{k-1}, X' \backslash X^k],$$

where, as in Sect. 3.2, $\phi^0 = \hat{\phi}[X' \backslash X^0]$ and $\phi^i = \phi[X' \backslash X^i, X \backslash X^{i-1}]$ represent the transition relation of A unfolded in the i-th step. If the test purpose is reachable, the mutation is not k-equivalent.

Remark 1 In this paper, we consider weak mutation testing [35]. This means that wrong behavior of internal variables is already considered a conformance violation. In contrast, strong mutation testing also requires that an internal fault propagates to an observable failure. The encoding of the reachability of the mutation as a test purpose, without altering the step relation, is only possible for weak mutation testing. Strong mutation testing would require the step relation to use the mutated contract in all steps, and then detect the failure in the last step. Due to considering weak mutation testing, we also weaken the definition of a test purpose compared with the definition in Sect. 3, allowing it to use internal variables.

Contrary to the previously defined test purposes, the test purposes in model-based mutation testing lead to negative counter examples, that is, counter examples steering towards an incorrect state. However, as defined in Sect. 3, we only extract the input vector σ_i, which is then combined with the correct requirement interface, to form a positive test case.

Often, different mutations of a contract will generate different negative counter examples, but those tests will then combine into the same positive test case. However, if the different mutations require different inputs to be enabled, they will also produce different positive test cases.

Example 8 Consider the requirement interface A_{beh} for the behavioral view of the 2-bounded buffer. Let $c_{2,m} : \neg\mathsf{enq}' \wedge \mathsf{deq}' \wedge k > 0 \vdash k' = k - 2$ be a mutant of c_2, where $k' = k - 1$ was mutated to $k' = k - 2$. The test purpose to detect this mutation is

$$\Pi = \varphi_{c_2} \wedge \psi_{c_{2,m}} \wedge \neg\psi_{c_2} \wedge \bigwedge_{i \in \{0,1,3,4,5\}} (\varphi_{c_i} \rightarrow \psi_{c_i}).$$

The test purpose is not valid in the initial state, as the assumption requires k to be greater than 0. Thus, a corresponding test case needs to execute at least one enqueue operation, before the mutated dequeue functionality can occur. The shortest vector $\bar{\sigma}$ leading to the test purpose is

$$\bar{\sigma} = \begin{array}{l} (\mathsf{enq}, \mathsf{deq}, k = 0, E, \neg F) \\ (\mathsf{enq}, \neg\mathsf{deq}, k = 1, \neg E, \neg F) \\ (\neg\mathsf{enq}, \mathsf{deq}, k = -1, \neg E, \neg F). \end{array}$$

We extract the input vector σ_I, so that

$$\sigma_I = \begin{array}{l} (\mathsf{enq}, \mathsf{deq}) \\ (\mathsf{enq}, \neg\mathsf{deq}) \\ (\neg\mathsf{enq}, \mathsf{deq}) \end{array}$$

Now, we can build the positive test case, by applying the correct step relation, thus gaining

$$\sigma_I = \begin{array}{l} (\mathsf{enq}, \mathsf{deq}, k = 0, E, \neg F) \\ (\mathsf{enq}, \neg\mathsf{deq}, k = 1, \neg E, \neg F) \\ (\neg\mathsf{enq}, \mathsf{deq}, k = 0, E, \neg F). \end{array}$$

As a second mutant, consider $c_{3,m}$: **true** $\vdash k' = 0 \Leftrightarrow \neg\mathsf{E}'$, where E' is mutated to $\neg E'$. In this case, the test purpose is not reachable, as the initial contract c_0 requires both $k' = 0$ and E', which causes an inconsistency with the mutated contract. This makes $c_{3,m}$ an unproductive mutation, as it does not generate a test case. However, consider another mutant of c_3, $c_{3,m'}$: **true** $\vdash k' = 0 \implies \mathsf{E}'$ that changes the bi-implication to an implication. In this case, the mutated contract can be enabled after an enqueue in the first step, when $k' = 1$, and thus, the left hand side of the implication is $false$, allowing E' to take any value. Thus, any vector of length two that starts with an enqueue operation, e.g., $\bar{\sigma}[0] = (\mathsf{enq}, \mathsf{deq}, k = 0, E, \neg F)$, $\bar{\sigma}[1] = (\mathsf{enq}, \neg\mathsf{deq}, k = 1, E, \neg F)$ detects the mutation. This example shows, that for different mutations on the same contract, the test generation results in different outcomes.

5 Implementation and experimental results

In this section, we present a prototype that implements our test-case generation framework introduced in Sects. 3 and 4. The prototype was added to the model-based testing tool family MoMuT[9] and goes by the name MoMuT::REQs. The implementation uses the programming language Scala 2.10 and Microsoft's SMT solver Z3 [40]. The tool implements both monolithic and incremental approaches to test-case generation. All experiments were run on a MacBook Pro with a 2.53 GHz Intel Core 2 Duo Processor and 4 GB RAM. We will demonstrate the tool on the buffer example and two industrial case studies.

5.1 Demonstrating example

To experiment with our algorithms, we model three variants of the buffer behavioral interface. All three variants model buffers of size 150, with different internal structures. *Buffer 1* models a simple buffer with a single counter variable k. *Buffer 2* models a buffer that is composed of two internal buffers of size 75 each, and *Buffer 3* models a buffer that is composed of three internal buffers of size 50 each. We also remodel a variant of the power consumption interface that created a dependency between the power used and the state of the internal buffers (idle/used).

All versions of the behavior interfaces can be combined with the power consumption view point, either using the incremental approach or doing the conjunction before generating test cases from the monolithic specification.

Incremental consistency checking To evaluate the consistency check, we introduce three faults to the behavioral and power consumption models of the buffer: *Fault 1* makes deq to decrease k when the buffer is empty; *Fault 2* mutates an assumption resulting in conflicting requirements for power consumption upon enq; and *Fault 3* makes enq to increase k when the buffer is full. The fault injection results in 9 single-faulty variants of interfaces.

We compare monolithic consistency checking to the consistency checking of individual views. We first note that the consistency check is coupled with the algorithm for finding minimal inconsistent sets of contracts. We set the range of the integer values to $[-2 : 152]$ and we bound the search depth to 3. In the monolithic approach, we first conjunct all view models and then check for consistency. In the incremental approach, we first check the consistency of individual views, and then, if no inconsistency is found, conjunct them one by one, checking consistency of partial conjunctions. However, in the current example, as we knew which view was faulty, we always started with the faulty view. As the inconsistency was detectable by examining only one view, we did not need to conjunct the second view. Table 1 summarizes and compares the time it takes to find an inconsistency and to compute the minimal inconsistent set of requirements in the requirement interface of a single view and in the monolithic interface that is the conjunction of both views. It gives

[9] http://www.momut.org.

Table 1 Runtime in seconds for checking consistency of single and conjuncted interfaces

	Fault 1 (behavior)		Fault 2 (power)	
	Single	Monolithic	Single	Monolithic
Buffer 1	0.7	3.6	1.0	7.3
Buffer 2	5.3	13.4	1.0	26.7
Buffer 3	7.2	13.8	1.0	13

Table 2 Runtime in seconds for incremental and monolithic test-case generation

	# Contracts	# Variables	t_{inc}	t_{mon}	Speed-up
Buffer 1	6	6	10	16.8	1.68
Buffer 2	15	12	36.7	48.8	1.33
Buffer 3	20	15	69	115.6	1.68

a very nice expression on how separating the different views helps decreases the complexity, and thus the runtime, of the consistency checks. E.g., for *Fault 2* in the second buffer, it reduces the runtime of the consistency check from 26 to 1 s. *Fault 3* is omitted in the table, as neither approach was able to find an inconsistency. This is caused by the fact that the fault lies to deep in the system, and cannot be detected with the given search depth.

The bounded consistency checking is very sensitive to the search depth. Setting the bound to 5 increases the runtime from seconds to minutes—this is not surprising, since a search of depth n involves simplifying formulas with alternating quantifiers of depth n, which is a very hard problem for SMT solvers.

Test-case generation We compare the monolithic and incremental approach to test-case generation, by generating monolithic tests for the conjunction of the buffer interfaces and the power consumption interface, and incrementally, by generating tests only for the buffer interfaces, and completing them with the power consumption interface. The tests were generated according to manually defined test purposes that required the buffer to be full. Thus, the according test cases needed to perform 150 enqueue operations, and were of length 150. Table 2 summarizes the results, presenting the number of contracts and variables of the requirement interfaces, the runtime of the incremental test case generation, and the runtime of the monolithic approach. For the incremental approach, the runtime includes the test-case generation using only the behavioral view and the completion of the test case, according to the power consumption. The three examples diverge in complexity, expressed in the number of contracts and variables. Our results show that the incremental approach outperforms the monolithic one, resulting in speed-ups from 1.33 to 1.68.

Table 3 Results for model-based mutation testing on depth 150

	# Mutants (equiv.)	# Unique tests	[min] t_{mbmt}
Buffer 1	44 (0)	29	4.7
Buffer 2	138 (20)	52	44.0
Buffer 3	240 (46)	115	128.1

Model-based mutation testing We applied the model-based mutation testing technique on all three variants of the buffer. For these experiments, we did not consider the power consumption, which could be added incrementally after generation of the tests. We used all mutation operators defined in Sect. 4. Table 3 shows the results of the approach, giving the number of mutants, the number of k-equivalent mutants, the number of unique test cases that were produced, and the total time for applying the complete approach to all mutants. The bound k for the equivalence check was set to 150. The reported times include mutation, generation of according test purposes, test-case generation, conversion into positive test cases, and detecting the unique test cases. Buffer 2 and Buffer 3 are more complex and create more mutants, and thus have a longer runtime. Yet, they also generate more unique tests, and thus a more thorough test suite.

5.2 Safing engine

As a first industrial application, we present an automotive use case supplied by the European ARTEMIS project MBAT,[10] that partially motivated our work on requirement interfaces. The use case was initiated by our industrial partner Infineon and evolves around building a formal model for analysis and test-case generation for the safing engine of an airbag chip. The requirements document, developed by a customer of Infineon, is written in natural (English) language. We identified 39 requirements that represent the core of the system's functionality and iteratively formalized them in collaboration with the designers of Infineon. The resulting formal requirement interface is deterministic and consists of 36 contracts.

The formalization process revealed several under-specifications in the informal requirements that were causing some ambiguities. These ambiguities were resolved in collaboration with the designers. The consistency check revealed two inconsistencies between the requirements. Tracing the conflicts back to the informal requirements allowed their fixing in the customer requirements document.

We generated 21 test cases from the formalized requirements that were designed to ensure that every boolean internal and output variable is at least activated once and that every possible state of the underlying finite-state machine is reached at least once. Thus, the test suite provides state and signal coverage. The average length of the test cases was 3.4 and the maximal length was 6, but since the test cases are synchronous, each of the steps is able to trigger several inputs and outputs at once. The test cases were used to test the SIMULINK model of the system, developed by Infineon as part of their design process. The SIMULINK model of the safing engine consists of a state machine with seven states, ten smaller blocks transforming the input signals, and a MATLAB function calculating the final outputs according to the current state and the input signals. To execute the test cases, Infineon's engineers developed a test adapter that transforms abstract input values from the test cases to actual inputs passed to the SIMULINK model. We illustrate a part of the use case with three customer requirements that give the flavor of the underlying system's functionality:

r_1: There shall be seven operating states for the safing engine: RESET state, INITIAL state, DIAGNOSTIC state, TEST state, NORMAL state, SAFE state, and DESTRUCTION state.

r_2: The safing engine shall change per default from RESET state to INIT state.

r_3: On a reset signal, the safing engine shall enter RESET state and stay, while the reset signal is active.

These three informal requirements were formalized with the following contracts with a one-to-one relationship between these example requirements and the contracts:

c_1: $\mathsf{true} \vdash \mathsf{state'} = \mathsf{RESET} \vee \mathsf{state'} = \mathsf{INIT} \vee \mathsf{state'} = \mathsf{DIAG} \vee \mathsf{state'} = \mathsf{TEST} \vee \mathsf{state'} = \mathsf{NORM} \vee \mathsf{state'} = \mathsf{SAFE} \vee \mathsf{state'} = \mathsf{DESTR}$

c_2 : $\mathsf{state} = \mathsf{RESET} \vdash \mathsf{state'} = \mathsf{INIT}$

c_3 : $\mathsf{reset'} \vdash \mathsf{state'} = \mathsf{RESET}$.

This case study extends an earlier one [4] with test-case execution and a detailed mutation analysis evaluating the quality of the generated test cases. We created 66 faulty SIMULINK models (six turned out to be equivalent), by flipping every boolean signal (also internal ones) involved in the MATLAB function calculating the final output signals. Our 21 test cases were able to detect 31 of the 60 non-equivalent faulty models, giving a mutation score of 51.6%. These numbers show that state and signal coverage is not enough to find all faults and confirm the need to incorporate a more sophisticated test-case generation methodology. Therefore, we manually added TEN test purposes generating 10 additional test cases. The combined 31 test cases finally reached a 100% mutation score. This means that all injected faults in the SIMULINK models were detected.

Model-based mutation testing In addition, we applied two iterations of the model-based mutation testing approach, set-

[10] https://mbat-artemis.eu.

ting the bound k to 6. In the first iteration, we generated 362 mutants, applying all mutation operators. We generated 165 negative tests—197 mutants were k-equivalent. From the 165 negative tests, we extracted 28 unique positive test cases.

The mutation score achieved by these 28 test cases on the 60 faulty SIMULINK models was surprisingly low, with only 49.2%. A closer investigation of the requirement interface shows that many of the contracts work globally, without being bound to a specific state of the state machine. For mutants from these contracts, our approach only generates one test case, even though the mutants generate multiple faults, in several different states. Due to the decomposed structure, even though we only insert one fault, our mutants are not classic first-order mutants anymore.

There are two ways to deal with this problem. The first one would be the generation of multiple test cases per mutant that covers all possible faulty states. We already applied this technique previously, in a different context [3]. However, this technique might become very expensive, and impossible for systems with infinite-state space.

The second approach is based on refactoring of the contracts, splitting global contracts into multiple more fine-grained ones. E.g., contract c_3 could be refactored into several state-bound contracts like

$$\begin{aligned}
\mathsf{c}_{3,1}: &\quad \mathsf{reset'} \wedge \mathsf{state} = \mathsf{INIT} \vdash \mathsf{state'} = \mathsf{RESET}\\
\mathsf{c}_{3,2}: &\quad \mathsf{reset'} \wedge \mathsf{state} = \mathsf{DESTR} \vdash \mathsf{state'} = \mathsf{RESET}.
\end{aligned}$$

Applying this technique, we gained 17 new contracts. The second run of our test-case generation produced 525 mutants and 293 were detected as non-equivalent. This led to 61 unique test cases, which were able to detect 53 of the faulty mutants, resulting in a mutation score of 88%.

This shows that the quality of model-based mutation testing for requirement interfaces is severely depending on the modeling style. However, while the fine-grained contracts might slightly decrease the clarity of the requirement interfaces, they, in turn, increase the traceability and facilitate fault detection.

5.3 Automated speed limiter

The second industrial case study is a use case provided by the industrial partner Volvo in the ARTEMIS Project CRYSTAL. It revolves around an automated speed limiter (ASL), which adopts the current speed according to a desired speed limit. It contains an internal state machine with three states: OFF, LIMITING, and OVERRIDDEN. Upon activation, it either takes the current speed as limit, or a predefined value. The limit can then be increased and decreased manually, and a kickdown of the gas pedal overrides the speed limiter for some time threshold. Adjusting the speed, or setting it to the predefined value, ends the overridden mode. Finally, the speed limiter can be turned off again, both from overridden and active mode.

The part of the ASL that was analysed within the project was documented by 17 informal requirements. These were refined to 26 formal requirements, collected in one requirements interface. The interface contains two input variables, two output variables, and four internal variables. The example below shows three characteristic contracts which serve as an illustration of the functionality of the speed limiter, where set and state are output variables, in and kickdown are input variables, preset_value and timer are internal variables, and preset and plus are enum values. Contract c_1 switches the ASL on, assigning the preset value as current limit. c_2 adjusts the current limit, increasing it by one. In addition, c_3 activates the overridden mode, in case of a gas pedal kickdown. It also resets a clock variable for the automated timeout that would lead back to limiting mode.

$$\begin{aligned}
\mathsf{c}_1: &\quad \mathsf{state} = \mathsf{OFF} \wedge \mathsf{in'} = \mathsf{preset} \wedge \neg\mathsf{kickdown'} \vdash\\
&\quad \mathsf{state'} = \mathsf{LIMITING} \wedge \mathsf{set'} = \mathsf{preset\_value}\\
\mathsf{c}_2: &\quad \mathsf{state} = \mathsf{LIMITING} \wedge \mathsf{in'} = \mathsf{plus} \wedge \neg\mathsf{kickdown'} \vdash\\
&\quad \mathsf{state'} = \mathsf{LIMITING} \wedge \mathsf{set'} = \mathsf{set{+}1}\\
\mathsf{c}_3: &\quad \mathsf{state} = \mathsf{LIMITING} \wedge \mathsf{kickdown'} \vdash\\
&\quad \mathsf{state'} = \mathsf{OVERRIDDEN} \wedge \mathsf{timer'} = 0.
\end{aligned}$$

Applying the mutation-based test generation to this case study generates 291 mutants, using all mutation operators introduced in Sect. 4 and setting the bound k to 4. Fifty seven of the mutants are equivalent, leaving a total of 234 non-equivalent mutants. Ninety six of these mutants can be detected within one step, sixty-four mutants are detected after two steps, and seventy two after the third step. This reflects very clearly the state-based structure that consists of three states. An analysis of the test cases shows that 60 of the tests are unique. Given that the model is deterministic, each of the unique tests enables different contracts in the individual steps. A further analysis of these 60 unique test cases shows that 12 are of length one, 18 are of length two, and 30 are of length three.

To evaluate the quality of the test cases, we implemented a Java version of the ASL, and used the Major mutation framework [37] to generate a set of 64 faulty implementations, using all mutation operators supported by Major. By executing our generated tests on these faulty implementations, we could perform a classic mutation analysis: our test suite was able to detect 48 of the faults. Further investigation of the undetected faults revealed that 13 of the remaining Java mutants were equivalent, and could thus not be detected by any test case. Another two of the faults were introduced in the conditions of if statements. The conditions correspond to the assumptions of our interfaces, which we did not mutate during the test-case generation.

The last remaining fault was introduced in the timing behavior of the Java implementation, which was simulated via a tick method, indicating the passage of 1 s. In the requirement interface, it was modeled via a non-deterministically increasing variable. The fault caused the implementation to trigger the state change already after 9 s instead of 10 s. The test driver was not sensitive enough to detect that, as the test case only specified the behavior after 10 s, and did not specify what the correct behavior after 9 s would be.

6 Related work

Synchronous languages were introduced in the 1980's, and mostly driven in France, where the most well-known three synchronous languages were developed: Lustre [22], Signal [27], and Esterel [17]. In 1991, the IEEE devoted a special issue to synchronous systems, featuring, e.g., a paper by Benveniste and Berry [11], discussing the major issues and approaches of synchronous specifications of real-time systems. A decade later, Benveniste et al. [14] gave an overview on the development of synchronous languages during that decade, especially mentioning the rising tool support and the industrial acceptance. They also mention globally asynchronous, locally synchronous systems [47] as an upcoming trend.

The main inspiration for this work was the introduction of the conjunction operation and the investigation of its properties [25] in the context of synchronous interface theories [23]. While the mathematical properties of the conjunction in different interface theories were further studied in [12,30,43], we are not aware of any similar work related to MBT.

Synchronous data-flow modeling [15] has been an active area of research in the past. The most important synchronous data-flow programming languages are Lustre [22] and SIGNAL [27]. These languages are implementation languages, while requirement interfaces enable specifying high-level properties of such programs. Testing of Lustre-like programs was studied by Raymond et al. [42] and Papailiopoulou [41]. The specification language SCADE [16] supports graphical representation of synchronous systems. Internally, SCADE models are stored in a textual representation very similar to Lustre. Experimental results for test-case generation were presented by Wakankar et al. [49] for experiments where the manually translated SCADE models to SAL-ATG models, and used the SAL-ATG for the test-case generation.

The tool STIMULUS [34] allows the synchronous specification and debugging of real-time systems, via predefined sentence templates, which can be simulated via a constraint solver. As in our approach, this enables a tight traceability between the natural language requirements, the formalized requirements, and the outcome of the verification tasks. They combine a user-friendly specification language with formal verification via BDDs. The tool is evaluated on an automotive case study.

Compositional properties of specifications in the context of testing were studied before [7,18,24,38,44]. None of these works consider synchronous data-flow specifications, and the compositional properties are investigated with respect to the parallel composition and hiding operations, but not conjunction. A different notion of conjunction is introduced for the test-case generation with SAL [28]. In that work, the authors encode test purposes as trap variables and conjunct them to drive the test-case generation process towards reaching all the test purposes with a single test case. Consistency checking of contracts has been studied in [26], yet for a weaker notion of consistency.

Our specifications using constraints share similarities with the Z specification language [45] that also follows a multiple-viewpoint approach to structuring a specification into pieces called schemas. However, a Z schema defines the dynamics of a system in terms of operations. In contrast, our requirement interfaces follow the style of synchronous languages.

Brillout et al. [20] performed mutation-based test-case generation on SIMULINK models. They implemented the approach in the tool COVER, based on the model-checker CBMC. He et al. [29] exploit similarity measures on mutants of SIMULINK models, to decrease the cost of mutation-based test-case generation. They provide experiments to show the advantages of model-based mutation testing compared with random testing, and compared with simpler mutation-based testing approaches.

There exist several tools for test-case generation for synchronous systems. The tool Lutess [19] is based on Lustre. It takes the specification of the environment (specified in Lustre), a test sequence generator, and an oracle and performed online testing on the system under test according to the environment and traces selected by the generator according to several different modes. Another tool based on Lustre is called Lurette [42]. Lurette only performs random testing, but is able to validate systems with numerical inputs and outputs. A third testing tool based on Lustre is called GATeL [39]. It generates tests according to test purposes, using constraint logic programming to search for suitable traces.

The tool Autofocus [32] facilitates test-case generation from time-synchronous communicating extended finite-state machines that build a distributed system. It is based on constraint logic programming, and supports functional, structural, and stochastic test specifications.

The application of the test-case generation and consistency checking tool for requirement interfaces and its integration into a set of software engineering tools was presented in [4]. That work focuses on the requirement-driven testing methodology, workflow, and tool integration, and gives no technical details about requirement interfaces. In contrast, this paper provides a sound mathematical theory for requirements interfaces and their associated incremental test-case generation, consistency checking, and tracing procedures.

Model-based mutation testing was initially used for predicate-calculus specifications [21] and later applied to formal Z specifications [46]. Amman et al. [8] used temporal formulae to check equivalence between models and mutants, and converted counter examples to test cases, in case of non-equivalence. Belli et al. [9,10] applied model-based mutation testing to event sequence graphs and pushdown automata. Hierons and Merayo [31] applied mutation-based test-case generation to probabilistic finite-state machines. The work presents mutation operators and describes how to create input sequences to kill a given mutated state machine.

Model-based mutation testing has already been applied to UML models [1,3], action systems [2], and timed automata [6].

7 Conclusions and future work

We presented a framework for requirement-driven modeling and testing of complex systems that naturally enable the multiple-view incremental modeling of synchronous data-flow systems. The formalism enables conformance testing of complex systems to their requirements and combining partial models via conjunction.

We also adapted the model-based mutation testing technique to requirement interfaces, and evaluated its applicability for two industrial case studies.

Our requirement-driven framework opens many future directions. We will extend our procedure to allow generation of adaptive test cases. In the context of model-based mutation testing, we will investigate strong mutation testing and mutation of assumptions. We will investigate in the future other compositional operations in the context of testing synchronous systems, such as the parallel composition and quotient. We intend on adding timed semantics to requirement interfaces, for a more thorough timing analysis. We will consider additional coverage criteria and test purposes, and will use our implementation to generate test cases for safety-critical domains, including automotive, avionics, and railways applications.

Acknowledgements Open access funding provided by Graz University of Technology.

Appendix-Proofs

Proof of Theorem 1

Proof Let A^1 and A^2 be two consistent requirement interfaces defined over the same alphabet. We first show that $A^1 \wedge A^2$ can be inconsistent. For this, we choose A^1 and A^2, such that $X_I^1 = X_I^2 = \{x\}$, $X_O^1 = X_O^2 = \{y\}$, $X_H^1 = X_H^2 = \emptyset$, $\hat{C}^1 = \{c^1\}$, $C^1 = \emptyset$, $\hat{C}^2 = \{c^2\}$, and $C^2 = \emptyset$, where $c^1 = \mathbf{true} \vdash y' = 0$ and $c^2 = \mathbf{true} \vdash y' = 1$. It is clear that both A^1 and A^2 are consistent—for any new value of x, A^1 (A^2) updates the value of y to 0 (1). However, $A^1 \wedge A^2$ is inconsistent, since no implementation can satisfy the guarantees of c^1 and c^2 simultaneously ($y' = 0 \wedge y' = 1$).

Assume that $A^1 \wedge A^2$ is consistent. We now prove that $\mathcal{L}(A^1 \wedge A^2) \subseteq \mathcal{L}(A^1) \cap \mathcal{L}(A^2)$. The proof is by induction on the size of σ.

Base case $\sigma = \epsilon$. We have that $A^1 \wedge A^2$ after $\epsilon = A^1$ after $\epsilon = A^2$ after $\epsilon = \{\hat{v}\}$.

Inductive step Let σ be an arbitrary trace of size n, such that $\sigma \in \mathcal{L}(A^1 \wedge A^2)$. By inductive hypothesis, $\sigma \in \mathcal{L}(A^1)$ and $\sigma \in \mathcal{L}(A^2)$. Consider an arbitrary w_{obs}, such that $\sigma \cdot w_{\text{obs}} \in \mathcal{L}(A^1 \wedge A^2)$. Let $V_{1\wedge2} = \{v \mid \hat{v} \overset{\sigma}{\Rightarrow} v\}$. By the definition of a semantics of requirement interfaces, it follows that $V'_{1\wedge2} = \{v' \mid v \xrightarrow{w_{\text{obs}}}_{1\wedge2} v' \text{ for some } v \in V_{1\wedge2}\}$ is non-empty. Let v' be an arbitrary state in $V'_{1\wedge2}$, hence we have that $v \rightarrow_{1\wedge2} v'$. Let $C_*^i = \{(\varphi \vdash \psi) \mid (\varphi \vdash \psi) \in C^i \text{ and } (v, \pi(v')[X_I]) \models \varphi\}$ for $i \in \{1, 2\}$ denote the (possibly empty) set of contracts in A^i for which the pair (v, v') satisfies its assumptions. By the definition of conjunction and semantics of requirement interfaces, we have that $(v, v') \models \bigwedge_{(\varphi\vdash\psi)\in C_*^1} \psi \wedge \bigwedge_{(\varphi\vdash\psi)\in C_*^2} \psi$. It follows that $(v, v') \models \bigwedge_{(\varphi\vdash\psi)\in C_*^1} \psi$, and $(v, v') \models \bigwedge_{(\varphi\vdash\psi)\in C_*^2} \psi$, hence we can conclude that $v \rightarrow_1 v'$ and $v \rightarrow_2 v'$, hence $\sigma \cdot w_{\text{obs}} \in \mathcal{L}(A^1)$ and $\sigma \cdot w_{\text{obs}} \in \mathcal{L}(A^2)$, which concludes the proof that $\mathcal{L}(A^1 \wedge A^2) \subseteq \mathcal{L}(A^1) \cap \mathcal{L}(A^2)$.

We now show that $\mathcal{L}(A^1 \wedge A^2) \supseteq \mathcal{L}(A^1) \cap \mathcal{L}(A^2)$. The proof is by induction on the size of σ.

Base case $\sigma = \epsilon$. We have that $A^1 \wedge A^2$ after $\epsilon = A^1$ after $\epsilon = A^2$ after $\epsilon = \{\hat{v}\}$.

Inductive step Let σ be an arbitrary trace of size n, such that $\sigma \in \mathcal{L}(A^1)$ and $\sigma \in \mathcal{L}(A^2)$. By inductive hypothesis, it follows that $\sigma \in \mathcal{L}(A^1 \wedge A^2)$. Let $\sigma' = \sigma \cdot v$. Consider an arbitrary w_{obs}, such that $\sigma \cdot w_{\text{obs}} \in \mathcal{L}(A^1)$ and $\sigma \cdot w_{\text{obs}} \in \mathcal{L}(A^2)$. It follows that $v \xrightarrow{w_{\text{obs}}}_1$ and $v \xrightarrow{w_{\text{obs}}}_2$. Let $C_*^i = \{(\varphi \vdash \psi) \mid (\varphi \vdash \psi) \in C^i \text{ and } (v, w_{\text{obs}}) \models \varphi\}$ for $i \in \{1, 2\}$ denote the (possibly empty) set of contracts in A^i for which the pair (v, w_{obs}) satisfies its assumptions. By the definition of conjunction and the semantics of requirement interfaces, we have that there exist v' and v'', such that $(v, v') \models \bigwedge_{(\varphi\vdash,\psi)\in C_*^1} \psi$, and $(v, v'') \models \bigwedge_{(\varphi\vdash\psi)\in C_*^2} \psi$. By the assumption that $A^1 \wedge A^2$ is consistent, we have that there exists v', such that $(v, v') \models \bigwedge_{(\varphi \vdash \psi) \in C^1_*} \psi \wedge \bigwedge_{(\varphi \vdash, \psi) \in C^2_*} \psi$ and that $\sigma \cdot w_{\text{obs}} \in \mathcal{L}(A^1 \wedge A^2)$, which concludes the proof that $\mathcal{L}(A^1 \wedge A^2) \supseteq \mathcal{L}(A^1) \cap \mathcal{L}(A^2)$. □

Proof of Theorem 2

Proof Assume that $A^1 \wedge A^2$ is consistent and consider an arbitrary consistent interface A that shares the same alphabet with A^1 and A^2. The proofs that $A^1 \wedge A^2 \preceq A^1$, $A^1 \wedge A^2 \preceq A^2$, and that if $A \preceq A^1$ and $A \preceq A^2$, then $A \preceq A^1 \wedge A^2$ follow directly from Theorem 1 and the definition of refinement. □

Proof of Proposition 1

- *Proof* We first prove the loop invariant that if SUT $\preceq A$, then in $\neq$ **fail** and $\sigma \in L(A)$. In Line 6, the next input in is by definition of the test case $T_{\sigma_I, A}$ the next valid input in σ_I. The extended trace in Line 7 is a trace of SUT. If SUT $\preceq A$, this extended trace is by definition of refinement also a trace of A. In this case, by definition of the test case $T_{\sigma_I, A}$ the next input in of Line 8 will be either the **pass** verdict or the next input of σ_I. Hence, the invariant holds. Consequently, when the loop terminates the **pass** verdict is returned. □
- *Proof* By negation, we obtain the proposition: if SUT $\preceq A$, then TestExec(SUT, $T_{\sigma_I, A}$) $\neq$ **fail**. This follows directly from the loop invariant established above. □

References

1. Aichernig, B.K., Brandl, H., Jöbstl, E., Krenn, W.: Efficient mutation killers in action. In: Fourth IEEE international conference on software testing, verification and validation, ICST 2011, Berlin, Germany, March 21–25, pp. 120–129. IEEE Computer Society (2011). doi:10.1109/ICST.2011.57
2. Aichernig, B.K., Brandl, H., Jöbstl, E., Krenn, W.: UML in action: a two-layered interpretation for testing. ACM SIGSOFT Softw. Eng. Notes **36**(1), 1–8 (2011). doi:10.1145/1921532.1921559
3. Aichernig, B.K., Brandl, H., Jöbstl, E., Krenn, W., Schlick, R., Tiran, S.: Killing strategies for model-based mutation testing. Softw. Test. Verif. Reliab. **25**(8), 716–748 (2015). doi:10.1002/stvr.1522
4. Aichernig, B.K., Hörmaier, K., Lorber, F., Nickovic, D., Schlick, R., Simoneau, D., Tiran, S.: Integration of requirements engineering and test-case generation via OSLC. In: 2014 14th International Conference on Quality Software, Allen, TX, October 2–3, pp. 117–126. IEEE (2014). doi:10.1109/QSIC.2014.13
5. Aichernig, B.K., Hörmaier, K., Lorber, F., Nickovic, D., Tiran, S.: Require, test and trace IT. In: Formal Methods for Industrial Critical Systems—20th International Workshop, FMICS 2015, Oslo, Norway, June 22-23, 2015 Proceedings, pp. 113–127 (2015). doi:10.1007/978-3-319-19458-5_8
6. Aichernig, B. K., Lorber, F., Nickovic, D.: Time for mutants—model-based mutation testing with timed automata. In: Veanes, M., Viganò, L. (eds.) Tests and Proofs—7th International Conference, TAP 2013, Budapest, Hungary, June 16–20, 2013. Proceedings, Lecture Notes in Computer Science, vol. 7942, pp. 20–38. Springer, New York (2013). doi:10.1007/978-3-642-38916-0_2
7. Aiguier, M., Boulanger, F., Kanso, B.: A formal abstract framework for modelling and testing complex software systems. Theor. Comput. Sci. **455**, 66–97 (2012)
8. Ammann, P.E., Black, P.E., Majurski, W.: Using model checking to generate tests from specifications. In: In Proceedings of the Second IEEE International Conference on Formal Engineering Methods (ICFEM98), pp. 46–54. IEEE Computer Society (1998)
9. Belli, F., Beyazit, M., Takagi, T., Furukawa, Z.: Model-based mutation testing using pushdown automata. IEICE Trans. **95-D**(9), 2211–2218 (2012). http://search.ieice.org/bin/summary.php?id=e95-d_9_2211
10. Belli, F., Budnik, C.J., Wong, W.E.: Basic operations for generating behavioral mutants. In: MUTATION, pp. 9–. IEEE Computer Society (2006). doi:10.1109/MUTATION.2006.2
11. Benveniste, A., Berry, G.: The synchronous approach to reactive and real-time systems. In: De Micheli, G. , Ernst, R., Wolf, W. (eds.) Readings in Hardware/Software Co-design, pp. 147–159. Kluwer Academic Publishers, Norwell (2002). http://dl.acm.org/citation.cfm?id=567003.567015
12. Benveniste, A., Caillaud, B., Ferrari, A., Mangeruca, L., Passerone, R., Sofronis, C.: Multiple viewpoint contract-based specification and design. In: de Boer, F.S., Bonsangue, M.M. , Graf, S., de Roever, W.P. (eds.) Formal Methods for Components and Objects, 6th International Symposium, FMCO 2007, Amsterdam, The Netherlands, October 24–26, 2007, Revised Lectures, Lecture Notes in Computer Science, vol. 5382, pp. 200–225. Springer (2007). doi:10.1007/978-3-540-92188-2_9
13. Benveniste, A., Caillaud, B., Ničković, D., Passerone, R., Raclet, J.B., Reinkemeier, P., Sangiovanni-Vincentelli, A., Damm, W., Henzinger, T., Larsen, K.G.: Contracts for System Design. Rapport de recherche RR-8147, INRIA (2012)
14. Benveniste, A., Caspi, P., Edwards, S.A., Halbwachs, N., Le Guernic, P., De Simone, R.: The synchronous languages 12 years later. Proc. IEEE **91**(1), 64–83 (2003). doi:10.1109/JPROC.2002.805826
15. Benveniste, A., Caspi, P., Guernic, P.L., Halbwachs, N.: Data-flow synchronous languages. In: REX School/Symposium, LNCS, vol. 803, pp. 1–45. Springer, New York (1993)
16. Berry, G.: Scade: Synchronous design and validation of embedded control software. In: Ramesh, S., Sampath, P. (eds.) Next Generation Design and Verification Methodologies for Distributed Embedded Control Systems: Proceedings of the GM R&D Workshop, Bangalore, India, January 2007, pp. 19–33. Springer, The Netherlands (2007). doi:10.1007/978-1-4020-6254-4_2
17. Berry, G., Gonthier, G.: The esterel synchronous programming language: design, semantics, implementation. Sci. Comput. Program. **19**(2), 87–152 (1992)
18. van der Bijl, M., Rensink, A., Tretmans, J.: Compositional testing with ioco. In: Petrenko, A., Ulrich, A. (eds.) Formal Approaches to Software Testing, Third International Workshop on Formal Approaches to Testing of Software, FATES 2003, Montreal, Quebec, Canada, October 6th, 2003, Lecture Notes in Computer Science, vol. 2931, pp. 86–100. Springer, New York (2003). doi:10.1007/978-3-540-24617-6_7
19. du Bousquet, L., Ouabdesselam, F., Richier, J.L., Zuanon, N.: Lutess: a specification-driven testing environment for synchronous software. In: Proceedings of the 21st International Conference on Software Engineering, ICSE '99, pp. 267–276. ACM, New York (1999). doi:10.1145/302405.302634
20. Brillout, A., He, N., Mazzucchi, M., Kröning, D., Purandare, M., Rümmer, P., Weissenbacher, G.: Mutation-based test case generation for Simulink models. In: Revised Selected Papers of the 8th International Symposium on Formal Methods for Components and

Objects (FMCO 2009), Lecture Notes in Computer Science, vol. 6286, pp. 208–227. Springer, New York (2010)
21. Budd, T.A., Gopal, A.S.: Program testing by specification mutation. Comput. Lang. **10**(1), 63–73 (1985). doi:10.1016/0096-0551(85)90011-6
22. Caspi, P., Pilaud, D., Halbwachs, N., Plaice, J.: Lustre: A declarative language for programming synchronous systems. In: Conference Record of the Fourteenth Annual ACM Symposium on Principles of Programming Languages, Munich, Germany, January 21-23, 1987, pp. 178–188. ACM Press (1987). doi:10.1145/41625.41641
23. Chakrabarti, A., de Alfaro, L., Henzinger, T.A., Mang, F.Y.C.: Synchronous and bidirectional component interfaces. In: Brinksma, E., Larsen, K.G. (eds.) Computer Aided Verification, 14th International Conference, CAV 2002, Copenhagen, Denmark, July 27-31, 2002, Proceedings, Lecture Notes in Computer Science, vol. 2404, pp. 414–427. Springer, New York (2002). doi:10.1007/3-540-45657-0_34
24. Daca, P., Henzinger, T.A., Krenn, W., Ničković, D.: Compositional specifications for ioco testing: Technical report. Technical report, IST Austria (2014). http://repository.ist.ac.at/152/
25. Doyen, L., Henzinger, T.A., Jobstmann, B., Petrov, T.: Interface theories with component reuse. In: de Alfaro, L., Palsberg, J. (eds.) Proceedings of the 8th ACM & IEEE International conference on Embedded software, EMSOFT 2008, Atlanta, October 19-24, pp. 79–88. ACM (2008). doi:10.1145/1450058.1450070
26. Ellen, C., Sieverding, S., Hungar, H.: Detecting consistencies and inconsistencies of pattern-based functional requirements. In: Lang, F., Flammini, F. (eds.) Formal Methods for Industrial Critical Systems—19th International Conference, FMICS 2014, Florence, September 11–12, 2014. Proceedings, Lecture Notes in Computer Science, vol. 8718, pp. 155–169. Springer, New York (2014). doi:10.1007/978-3-319-10702-8_11
27. Gautier, T., Guernic, P.L.: SIGNAL: A declarative language for synchronous programming of real-time systems. In: Kahn, G. (ed.) Functional Programming Languages and Computer Architecture, Portland, Oregon, USA, September 14–16, 1987, Proceedings, Lecture Notes in Computer Science, vol. 274, pp. 257–277. Springer, New York (1987). doi:10.1007/3-540-18317-5_15
28. Hamon, G., De Moura, L., Rushby, J.: Automated test generation with sal. CSL Technical Note (2005)
29. He, N., Rümmer, P., Kröning, D.: Test-case generation for embedded simulink via formal concept analysis. In: Proceedings of the 48th Design Automation Conference, DAC '11, pp. 224–229. ACM, New York (2011). doi:10.1145/2024724.2024777
30. Henzinger, T.A., Ničković, D.: Independent implementability of viewpoints. In: Monterey Workshop, LNCS, vol. 7539, pp. 380–395. Springer, New York (2012)
31. Hierons, R., Merayo, M.: Mutation testing from probabilistic finite state machines. In: Testing: Academic and Industrial Conference Practice and Research Techniques—MUTATION, 2007. TAICPART-MUTATION 2007, pp. 141–150 (2007). doi:10.1109/TAIC.PART.2007.20
32. Huber, F., Schätz, B., Schmidt, A., Spies, K.: AutoFocus: A tool for distributed systems specification. In: Proceedings of the 4th International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems (FTRTFT 1996), Lecture Notes in Computer Science, vol. 1135, pp. 467–470. Springer, New York (1996)
33. ISO: ISO/DIS 26262-1 - Road vehicles—Functional safety—Part 1 Glossary. Tech. rep., International Organization for Standardization/Technical Committee 22 (ISO/TC 22), Geneva, Switzerland (2009)
34. Jeannet, B., Gaucher, F.: Debugging embedded systems requirements with stimulus: an automotive case-study. In: 8th European Congress on Embedded Real Time Software and Systems (ERTS 2016) (2016)
35. Jia, Y., Harman, M.: An analysis and survey of the development of mutation testing. Softw. Eng. IEEE Trans. **37**(5), 649–678 (2011). doi:10.1109/TSE.2010.62
36. Junker, U.: Quickxplain: Preferred explanations and relaxations for over-constrained problems. In: AAAI, pp. 167–172. AAAI Press, California (2004)
37. Just, R., Schweiggert, F., Kapfhammer, G.M.: MAJOR: An efficient and extensible tool for mutation analysis in a Java compiler. In: Proceedings of the International Conference on Automated Software Engineering (ASE), pp. 612–615 (2011)
38. Krichen, M., Tripakis, S.: Conformance testing for real-time systems. Formal Methods Syst. Des. **34**(3), 238–304 (2009)
39. Marre, B., Arnould, A.: Test sequences generation from LUSTRE descriptions: GATEL. In: Automated Software Engineering, 2000. Proceedings ASE 2000. The Fifteenth IEEE International Conference on, pp. 229–237 (2000). doi:10.1109/ASE.2000.873667
40. Moura, L., Björner, N.: Z3: an efficient smt solver. In: Tools and algorithms for the construction and analysis of systems. LNCS, vol. 4963, pp. 337–340. Springer, New York (2008). doi:10.1007/978-3-540-78800-3_24
41. Papailiopoulou, V.: Automatic test generation for lustre/scade programs. In: ASE, pp. 517–520. IEEE Computer Society, Washington, DC (2008). doi:10.1109/ASE.2008.96
42. Raymond, P., Nicollin, X., Halbwachs, N., Weber, D.: Automatic testing of reactive systems. In: RTSS, pp. 200–209. IEEE Computer Society (1998)
43. Reineke, J., Tripakis, S.: Basic problems in multi-view modeling. Tech. Rep. UCB/EECS-2014-4. EECS Department, University of California, Berkeley (2014)
44. Sampaio, A., Nogueira, S., Mota, A.: Compositional verification of input-output conformance via CSP refinement checking. In: Breitman, K., Cavalcanti, A. (eds.) Formal Methods and Software Engineering, 11th International Conference on Formal Engineering Methods, ICFEM 2009, Rio de Janeiro, Brazil, December 9-12, 2009. Proceedings, Lecture Notes in Computer Science, vol. 5885, pp. 20–48. Springer, New York (2009). doi:10.1007/978-3-642-10373-5_2
45. Spivey, J.M.: Z Notation—a reference manual, 2nd edn. Prentice Hall, Prentice Hall International Series in Computer Science (1992)
46. Stocks, P.A.: Applying formal methods to software testing. Ph.D. thesis, Department of computer science, University of Queensland (1993)
47. Teehan, P., Greenstreet, M., Lemieux, G.: A survey and taxonomy of GALS design styles. IEEE Des. Test Comput. **24**(5), 418–428 (2007). doi:10.1109/MDT.2007.151
48. Tretmans, J.: Test generation with inputs, outputs and repetitive quiescence. Softw. Concepts Tools **17**(3), 103–120 (1996)
49. Wakankar, A., Bhattacharjee, A.K., Dhodapkar, S.D., Pandya, P.K., Arya, K.: Automatic test case generation in model based software design to achieve higher reliability. In: Reliability, Safety and Hazard (ICRESH), 2010 2nd International Conference on, pp. 493–499 (2010). doi:10.1109/ICRESH.2010.5779600