Require, Test and Trace IT

Bernhard K. Aichernig[1], Klaus Hörmaier[2], Florian Lorber[1(✉)], Dejan Ničković[3], and Stefan Tiran[1,3]

[1] Graz University of Technology, Graz, Austria
aichernig@ist.tugraz.at
[2] Infineon Technologies Austria AG, Villach, Austria
[3] AIT Austrian Institute of Technology, Vienna, Austria

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. Multiple 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. Finally, this framework defines a natural association between informal requirements, their formal specifications and the generated tests, thus facilitating traceability. We implemented a prototype test generation tool and we demonstrate its applicability on an industrial use case.

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

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 for example the automotive standard ISO 26262 [16]). 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

M. Núñez and M. Güdemann (Eds.): FMICS 2015, LNCS 9128, pp. 113–127, 2015.
DOI: 10.1007/978-3-319-19458-5_8

system-under-test (SUT) in order 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. In order to reason about the conformance of the SUT to its specification, one needs to use the *testing assumption* [24], stating 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*. 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 (behavioural, safety, timing, etc.) *views* of the system. 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* framework for MBT of *synchronous data-flow* reactive systems. In contrast to classical MBT, in which the requirements document is usually formalized into one monolithic specification, we exploit the structure of the requirements and adopt a *multiple viewpoint* approach.

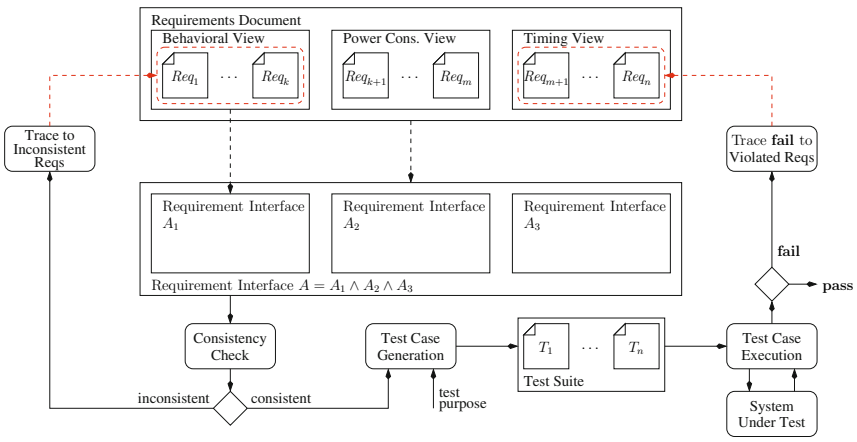

Fig. 1. Overview of using requirement interfaces for testing, analysis and tracing

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 [2], connecting severals 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.

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*. 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.

Finally, 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. We illustrate the entire workflow of using requirement interfaces for consistency checking, testing and tracing in Figure 1.

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 (φ, ψ), 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 $\hat{c} = (\hat{\varphi}, \hat{\psi})$ 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 $c = (\varphi, \psi)$ 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].

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* initial *and* update contracts*;*
- $\mathcal{R}$ *is a finite set of* requirement identifiers*;*
- $\rho : \mathcal{R} \rightarrow \mathcal{P}(C \cup \hat{C})$ *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 $\bigvee_{(\hat{\varphi}, \hat{\psi}) \in \hat{C}} \hat{\varphi}$ and $\bigvee_{(\varphi, \psi) \in C} \varphi$ are both valid. A requirement interface is *fully-observable* if $X_H = \emptyset$. A requirement interface is *deterministic* if for all $(\hat{\varphi}, \hat{\psi}) \in \hat{C}$, $\hat{\psi}$ has the form $\bigwedge_{x \in X_O} x' = c$, where c is a constant of the appropriate type, and for all $(\varphi, \psi) \in C$, ψ has the form $\bigwedge_{x \in X_{\text{ctr}}} x' = f(X)$, where f is a function over X that has the same type as x.

Example 1. We use the 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.

[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 $w_{\text{ctr}} \in V(X_{\text{ctr}})$.

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[3] 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

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

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 and input 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 $1 \leq i \leq n-1$, $(v_i, R_{i+1}, v_{i+1}) \in T$. 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) $v \stackrel{\epsilon}{\Rightarrow} v'$ iff $v = v'$; (5) $v \stackrel{w}{\Rightarrow} v'$ iff $\exists Y \subseteq X$ s.t. $\pi(v')[Y] = w$ and $v \to v'$; (6) $v \stackrel{w}{\Rightarrow}$ 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. $v \stackrel{w_1}{\Rightarrow} v_1 \stackrel{w_2}{\Rightarrow} \cdots v_n \stackrel{w_n}{\Rightarrow} v'$; 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 $\hat{v} \stackrel{\sigma}{\Rightarrow}$. We denote by $\mathcal{L}(A)$ the set of all traces of A. Given a trace σ of A, let A after $\sigma = \{v \mid \hat{v} \stackrel{\sigma}{\Rightarrow} v\}$. Given a state $v \in V$, let $\text{succ}(v) = \{v' \mid v \to v'\}$ be the set of successors of v.

[3] For readability we use the concrete syntax $\varphi \vdash \psi$ to denote (φ, ψ) in our examples.

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 a state $v \in V$ 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 2. A^{beh} is consistent – 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, that 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 requirement interfaces. 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 3. 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 2 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 such that $X_I^{buf} = \{\mathsf{enq}, \mathsf{deq}\}$, $X_O^{buf} = \{\mathsf{E}, \mathsf{F}, \mathsf{pc}\}$, $X_H^{buf} = \{k\}$, $\hat{C}^{buf} = \{c_0, c_a, c_b\}$, $C^{buf} = \{c_1, c_2, c_3, c_4, c_5, c_a, c_b\}$, $\mathcal{R}^{pc} = \{r_i \mid i \in \{a, b, 0, 1, 2, 3, 4, 5\}\}$, and $\rho(r_i) = \{c_i\}$.

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 $\mathcal{L}(A^1 \wedge A^2) = \mathcal{L}(A^1) \cap \mathcal{L}(A^2)$.*

We now show some properties of requirement interfaces.

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 $A \preceq A^1 \wedge A^2$.*

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.*

For proofs we refer to our technical report [4].

3 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. Note that before executing the test-case generation, we can apply a consistency check on the requirement interface. For details, we refer to our technical report [4].

3.1 Test Case Generation

A *test case* is an experiment executed on the SUT I by the *tester*. We assume that I is a black-box that is only accessed via its observable interface. We assume that I can be modeled as an input-enabled, deterministic[4] requirement interface. Without loss of generality, we can represent I as a total sequential function $I : V(X_I) \times V(X_{\text{obs}})^* \to V(X_O)$. 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 $\hat{\phi} = \bigvee_{(\hat{\varphi},\hat{\psi})\in\hat{C}} \hat{\varphi} \wedge \bigwedge_{(\hat{\varphi},\hat{\psi})\in\hat{C}} \hat{\varphi} \to \hat{\psi}$ and $\phi = \bigvee_{(\varphi,\psi)\in C} \varphi \wedge \bigwedge_{(\varphi,\psi)\in C} \varphi \to \psi$. 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 under-specification. A test case for A that can reach Π is defined iff there exists a trace $\sigma = \sigma' \cdot w_{obs}$ in $\mathcal{L}(A)$ such that $w_{obs} \models \Pi$. The test purpose Π can be reached in A in at most k steps if

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

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 i steps.

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[5] 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: $\omega^0_{\sigma_I,A} \leftarrow \hat{\theta}[X'_I \backslash w_I^0, X'_{\text{ctr}} \backslash X^0_{\text{ctr}}]$
2: **for** $i = 1$ to n **do**
3: $\quad \omega^i_{\sigma_I,A} \leftarrow \theta[X_I \backslash w_I^{i-1}, X'_I \backslash w_I^i, X_{\text{ctr}} \backslash X^{i-1}_{\text{ctr}}, X'_{\text{ctr}} \backslash X^i_{\text{ctr}}]$
4: **end for**
5: $\omega^*_{\sigma_I,A} \leftarrow \omega^0_{\sigma_I,A} \wedge \ldots \wedge \omega^n_{\sigma_I,A}$
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 $\hat{\theta} = \bigwedge_{(\hat{\varphi},\hat{\psi})\in\hat{C}} \hat{\varphi} \to \hat{\psi}$ and $\theta = \bigwedge_{(\varphi,\psi)} \varphi \to \psi$. 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

[4] The restriction to deterministic implementations is for presentation purposes only, the technique is general and can also be applied to non-deterministic systems.
[5] The formula $\omega_{\sigma_I,A}$ can be seen as a monitor for A under input σ_I.

the set of output sequences allowed by A under σ_I. Since any implementation of hidden variables that preserves 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, $\sigma = w_{obs}^0 \cdots w_{obs}^k$
Output: $V(X_I^I) \cup \{\mathbf{pass}, \mathbf{fail}\}$
1: $\omega_{\sigma_I,A} \leftarrow \text{OutMonitor}(\sigma_I, A)$
2: **for** $i = 0$ to k **do**
3: $\quad w_O^i \leftarrow \pi(w_{obs}^i)[X_O]$
4: **end for**
5: $\omega^{0,k}_{\sigma_I,A} \leftarrow \omega_{\sigma_I,A}[X_O^0 \backslash w_O^0, \ldots, X_O^k \backslash w_O^k]$
6: **if** $\omega^{0,k}_{\sigma_I,A} = \mathbf{true}$ **then**
7: $\quad$ **return pass**
8: **else if** $\omega^{0,k}_{\sigma_I,A} = \mathbf{false}$ **then**
9: $\quad$ **return fail**
10: **else**
11: $\quad$ **return** w_I^{k+1}
12: **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 $0 \leq i \leq |\sigma|$, $w_I^i = \pi(w_{obs}^i)[X_I]$, where $\sigma_I = w_I^0 \cdots w_I^n$ and $\sigma = w_{obs}^0 \cdots w_{obs}^k$. Algorithm 2 gives a constructive definition of the test case $T_{\sigma_I,A}$.

Incremental test-case generation: So far, we considered test case generation for a flat requirement interface A. We now describe how test cases can be *incrementally* generated when the interface A consists of multiple views[6], 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 drives 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 drives 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.

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 4. Consider the requirement interface A_{beh} for the behavioral view of the 2-bounded buffer, and the test purpose F. Our test case generation procedure gives the input vector σ_I of size 3 such that $\sigma_I[0] = (\mathsf{enq}, \mathsf{deq})$, $\sigma_I[1] = (\mathsf{enq}, \neg\mathsf{deq})$ and $\sigma_I[2] = (\mathsf{enq}, \neg\mathsf{deq})$. The observable output constraints for σ_I (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 $T_{\sigma_I,beh}$. By using the incremental test case generation procedure,

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

we can extend $T_{\sigma_I,beh}$ to a test case $T_{\sigma_I,buf}$ that also takes into account 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.

3.2 Test Case Execution

Algorithm 3. TestExec

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

Let A be a requirement interface, I a SUT 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 I 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 SUT I 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 I be arbitrary requirement interface, test case generated from A and implementation, respectively. Then, we have that:*

1. *if $I \preceq A$, then $TestExec(I, T_{\sigma_I,A}) = \textbf{\textit{pass}}$; and*
2. *if $TestExec(I, T_{\sigma_I,A}) = \textbf{\textit{fail}}$, then $I \not\preceq A$.*

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, I an arbitrary implementation and T_{σ_I,A^1} an arbitrary test case generated from A^1. Then, if $TestExec(I, T_{\sigma_I,A^1}) = \textbf{\textit{fail}}$, then $I \not\preceq A^1 \wedge A^2$.*

3.3 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 $A = A^1 \wedge \ldots \wedge A^n$ 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 [17], in order to compute a minimal set of contracts $C' \subseteq C$ such that C' is inconsistent. Once we compute 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. Again, more details can be found in our technical report [4].

4 Implementation and Experimental Results

Implementation and experimental setup: We present a prototype that implements our test case generation framework introduced in Section 3. The prototype was integrated in our model-based testing toolchain MoMuT[7] and named MoMuT::REQs. The implementation uses Scala 2.10 and the SMT solver Z3. 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.

Demonstrating example: In order to experiment with our implementation, we model three variants of the buffer behavioral interface. All three variants model buffers of size 150, with different internal structure. *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).

We compare the monolithic and incremental approach to test case generation, by generating 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. Table 1 summarizes the results. 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 33% to 68%. Results on the consistency check can be found in our technical report [4].

Industrial application: We present an automotive use case from the European ARTEMIS project[8], 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

[7] http://www.momut.org
[8] https://mbat-artemis.eu

Table 1. Run-time in seconds for incremental and monolithic test case generation

	# Contracts	# Variables	t_{inc}	t_{mon}	speed-up
Buffer 1	6	6	10	16.8	68 %
Buffer 2	15	12	36.7	48.8	33 %
Buffer 3	20	15	69	115.6	68 %

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. The average length of the test cases was 3.4, 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 the 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. In order to execute the test cases, Infineons 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 informal requirements were formalized with the following contracts with a one to one relationship between requirements and 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 [2] with test-case execution and a detailed mutation analysis evaluating the quality of the generated test cases.

We created 66 mutants (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 mutants, 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 10 test purposes generating 10 additional test cases. The combined 31 test cases finally reached a 100% mutation score. This means that all injected faults were detected. In order to achieve this high mutation score fully automatically, we will add support for fault-based test-case generation to our tool, like we recently did for UML [1] and timed automata [3].

5 Related Work

The main inspiration for this work was the introduction of the conjunction operation and the investigation of its properties [11] in the context of synchronous interface theories [9]. While the mathematical properties of the conjunction in different interface theories were further studied in [6,21,15], we are not aware of any similar work related to model-based testing.

Synchronous data-flow modeling [7] has been an active area of research in the past. The most important synchronous data-flow programming languages are Lustre [8] and SIGNAL [13]. 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. [20] and Papailiopoulou [19]. Compositional properties of specifications in the context of testing were studied before [25,18,22,5,10]. None of these workes 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 [14]. In that work, the authors encode test purposes as trap variables, and conjunct them in order 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 [12], yet for a weaker notion of consistency.

Our specifications using constraints share similarities with the Z specification language [23], 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.

Finally, the application of the TCG and consistency checking tool for requirement interfaces and its integration into a set of software engineering tools was presented in [2]. 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 TCG, consistency checking and tracing procedures.

6 Conclusions and Future Work

We presented a framework for requirement-driven modeling and testing of complex systems that naturally enables 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.

Our requirement-driven framework opens many future directions. We will extend our procedure to allow generation of adaptive test cases. We will investigate in the future other compositional operations in the context of testing synchronous systems such as the parallel composition and quotient. We also plan to study whether partitioning the requirements into views is feasible via (semi) automation, based on static analysis of input/output dependencies between requirements. We will consider additional coverage criteria and test purposes and will use our implementation to generate test cases for other industrial-size systems from our automotive, avionics and railways partners.

Acknowledgment. We are grateful to the anonymous reviewers for their valuable and detailled feedback. The research leading to these results has received funding from the ARTEMIS Joint Undertaking under grant agreements N° 269335 and N° 332830 and from the Austrian Research Promotion Agency (FFG) under grant agreements N° 829817 and N° 838498 for the implementation of the projects MBAT, Combined Model-based Analysis and Testing of Embedded Systems and CRYSTAL, Critical System Engineering Acceleration.

References

1. Aichernig, B.K., Auer, J., Jöbstl, E., Korošec, R., Krenn, W., Schlick, R., Schmidt, B.V.: Model-based mutation testing of an industrial measurement device. In: Seidl, M., Tillmann, N. (eds.) TAP 2014. LNCS, vol. 8570, pp. 1–19. Springer, Heidelberg (2014)
2. Bernhard, K.A., Hörmaier, K., Lorber, F., Ničković, D., Schlick, R., Simoneau, D., Tiran, S.: Integration of Requirements Engineering and Test-Case Generation via OSLC. In: QSIC, pp. 117–126 (2014)
3. Aichernig, B.K., Lorber, F., Ničković, D.: Time for mutants — model-based mutation testing with timed automata. In: Veanes, M., Viganò, L. (eds.) TAP 2013. LNCS, vol. 7942, pp. 20–38. Springer, Heidelberg (2013)
4. Aichernig, B.K., Lorber, F., Ničković, D., Tiran, S.: Require, test and trace it. Technical Report IST-MBT-2014-03, Graz University of Technology, Institute for Software Technology (2014), https://online.tugraz.at/tug_online/voe_main2.getVollText?pDocumentNr=637834&pCurrPk=77579
5. Aiguier, M., Boulanger, F., Kanso, B.: A formal abstract framework for modelling and testing complex software systems. Theor. Comput. Sci. 455, 66–97 (2012)
6. 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.) FMCO 2007. LNCS, vol. 5382, pp. 200–225. Springer, Heidelberg (2008)
7. Benveniste, A., Caspi, P., Le Guernic, P., Halbwachs, N.: Data-flow synchronous languages. In: de Bakker, J.W., de Roever, W.-P., Rozenberg, G. (eds.) REX 1993. LNCS, vol. 803, pp. 1–45. Springer, Heidelberg (1994)

8. Caspi, P., Pilaud, D., Halbwachs, N., Plaice, J.: Lustre: A declarative language for programming synchronous systems. In: POPL, pp. 178–188. ACM Press (1987)
9. Chakrabarti, A., de Alfaro, L., Henzinger, T.A., Mang, F.Y.C.: Synchronous and bidirectional component interfaces. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, pp. 414–427. Springer, Heidelberg (2002)
10. 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/
11. Doyen, L., Henzinger, T.A., Jobstmann, B., Petrov, T.: Interface theories with component reuse. In: EMSOFT, pp. 79–88. ACM (2008)
12. Ellen, C., Sieverding, S., Hungar, H.: Detecting consistencies and inconsistencies of pattern-based functional requirements. In: Lang, F., Flammini, F. (eds.) FMICS 2014. LNCS, vol. 8718, pp. 155–169. Springer, Heidelberg (2014)
13. Gautier, T., Le Guernic, P.: Signal: A declarative language for synchronous programming of real-time systems. In: Kahn, G. (ed.) FPCA 1987. LNCS, vol. 274, pp. 257–277. Springer, Heidelberg (1987)
14. Hamon, G., De Moura, L., Rushby, J.: Automated test generation with sal. CSL Technical Note (2005)
15. Henzinger, T.A., Ničković, D.: Independent implementability of viewpoints. In: Calinescu, R., Garlan, D. (eds.) Monterey Workshop 2012. LNCS, vol. 7539, pp. 380–395. Springer, Heidelberg (2012)
16. ISO. ISO/DIS 26262-1 - Road vehicles - Functional safety - Part 1 Glossary. Technical report, International Organization for Standardization / Technical Committee 22 (ISO/TC 22), Geneva, Switzerland (July 2009)
17. Junker, U.: Quickxplain: Preferred explanations and relaxations for over-constrained problems. In: AAAI, pp. 167–172. AAAI Press (2004)
18. Krichen, M., Tripakis, S.: Conformance testing for real-time systems. Formal Methods in System Design 34(3), 238–304 (2009)
19. Papailiopoulou, V.: Automatic test generation for lustre/scade programs. In: ASE, pp. 517–520. IEEE Computer Society, Washington, DC (2008)
20. Raymond, P., Nicollin, X., Halbwachs, N., Weber, D.: Automatic testing of reactive systems. In: RTSS, pp. 200–209. IEEE Computer Society (1998)
21. Reineke, J., Tripakis, S.: Basic problems in multi-view modeling. Technical Report UCB/EECS-2014-4, EECS Department, University of California, Berkeley (January 2014)
22. Sampaio, A., Nogueira, S., Mota, A.: Compositional verification of input-output conformance via csp refinement checking. In: Breitman, K., Cavalcanti, A. (eds.) ICFEM 2009. LNCS, vol. 5885, pp. 20–48. Springer, Heidelberg (2009)
23. Michael Spivey, J.: Z Notation - a reference manual, 2nd edn. Prentice Hall International Series in Computer Science. Prentice Hall (1992)
24. Tretmans, J.: Test generation with inputs, outputs and repetitive quiescence. Software - Concepts and Tools 17(3), 103–120 (1996)
25. van der Bijl, M., Rensink, A., Tretmans, J.: Compositional testing with IOCO. In: Petrenko, A., Ulrich, A. (eds.) FATES 2003. LNCS, vol. 2931, pp. 86–100. Springer, Heidelberg (2004)