TECHNICAL REPORT

IST-MBT-2014-03

Require, Test and Trace IT

Bernhard K. Aichernig[1], Florian Lorber[1], and
Dejan Ničković[2], Stefan Tiran[12]

{aichernig, florber, stiran}@ist.tugraz.at, dejan.nickovic@ait.ac.at

March 3, 2014

[1] Institute for Software Technology
Graz University of Technology
Graz, Austria
.
[2] AIT Austrian Institute of Technology
Department of Safety and Security
Vienna, Austria

Require, Test and Trace IT

Abstract. We present a framework for requirement-driven test-case generation from specifications of synchronous reactive systems. We propose requirement interfaces as the formal specification model. Contract specifications of individual requirements are naturally combined by the conjunction operation. The conjunction of two requirement interfaces has the property that it subsumes all behaviors of the individual interfaces. We exploit this property to generate test cases incrementally. We use a small set of related requirements to drive test-case generation, thus avoiding the explosion of the state space to explore. In addition to test-case generation, we also provide a procedure for incremental consistency checking of the requirements.

Our requirement-driven approach has several advantages: (1) both consistency checking and test case generation are incremental and thus more efficient; (2) test cases are naturally related to requirements, hence enabling easier traceability; and (3) fail verdicts in a test case can be mapped to violated requirements. We implemented a prototype using the SMT-solver Z3.

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 [1]). It follows that verification and validation techniques must be used to provide evidence that the designed system meets its requirements. *Testing* remains the most common 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) 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 often remains the main bottleneck in the complex system design.

Model-based testing is a popular technology that enables systematic and automatic test case generation and execution, thus reducing system design time and cost. In model-based testing, the SUT is tested for conformance against its *specification*, a mathematical model of the SUT. In contrast to the specification, which is a formal object, the SUT is a physical implementation with 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* [26], 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 *views* (aspects) of the system. For instance, a requirement document can have different chapters for the behavioral, timing and safety views of the system. Intuitively, a system must correctly implement all requirements. Hence, a requirement document can be naturally seen as the *conjunction* of all its requirements. Sometimes, requirements can have conflicts, resulting in a specification that does not admit any correct implementation. Requirement validation consists in checking that the documented requirements are *consistent*.

In this paper, we propose a *requirement-driven* framework for model-based testing of *synchronous data-flow* reactive systems. In contrast to classical model-based testing, in which the formalization of the requirements document usually results in a single monolithic specification, we exploit the structure of the requirements and adopt a *multiple viewpoint* approach to modeling.

As a first contribution, we introduce *requirement interfaces* as the formalism for modeling system views as subsets of requirements. It is a state-transition formalism that supports the specification of open reactive systems communicating via shared variables. A requirement interface consists of a set of *contracts*. A contract is a rule that defines an *assumption* about the external environment of the interface, and a *guarantee* that the system must satisfy if the assumption is met. We associate subsets of contracts to requirement identifiers, which are extracted from requirement documents. This enables associating the formal model to the informal requirements from which the specification is derived.

A requirement interface is intended to model a specific view of the SUT. We define the *conjunction* of requirement interfaces as the operator that enables combining different views of the SUT. Intuitively, a conjunction of two requirement interfaces is another requirement interface that subsumes all the behaviors of both interfaces. We assume that the overall specification of the SUT is given as a conjunction of requirement interfaces modeling its different views.

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 – the conjunction of an inconsistent interface with any other interface remains inconsistent.

Next, we develop a requirement-driven test case generation and execution procedure from requirement interfaces, with *language inclusion* as the refinement (conformance) relation. We present a procedure for test case generation from a specific SUT view, derived from the requirement interface that models the view and a *test purpose*. The procedure is based on bounded reachability analysis of the view model. Such a test case can be used directly to detect if the implementation by the SUT violates the view model, but cannot detect violation of other modeled views. However, we show that falsification by testing is compositional with respect to conjunction – a test case that detects violation of the view implies the violation of the overall specification.

In addition, we present a procedure for extending a test case for a view to a test case for the full specification. The test case generated from a view model is extended by adding the constraints defined by the other view models in the conjunction. The extended test case drives the SUT to the test purpose defined for a single view, but is also able to detect violation of any other view.

We develop tracing procedures that exploit natural mapping between informal requirements and formal models in our requirement interfaces. For inconsistent interfaces, we compute the minimal set of requirements that lead to inconcistency. Test cases generated from a view model are mapped to requirements from which the view model is derived. Finally, we compute the set of requirements that are violated upon getting a fail verdict when executing a test case. We believe that these tracing information provide precious maintainance and debugging information to the engineers.

Requirement interfaces are inspired by synchronous interfaces [9], although there is a number of differences. Requirement identifiers that allow relating formal models to the informal requirements are integral part of the formal model of the requirement interfaces. In contrast to synchronous interfaces, requirement interfaces allow modeling *hidden* in addition to interface (input/output) variables. Although the model allows under-specification of inputs, we complete the underlying semantics. Hence, we adopt language inclusion refinement relation instead of alternating simulation used in [9]. The conjunction operation, introduced in [14] as shared refinement, and used in [6] for requirement engineering, is studied in this paper in the context of model-based testing for the first time, to the best of our knowledge.

2 Requirement Interfaces

We introduce *requirement interfaces*, a formalism for specification of synchronous data-flow systems. Their semantics is given in 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. The 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 *initial* if $\hat{\varphi}$ and $\hat{\psi}$ are predicates on X'_I and X'_{ctr}, respectively. 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 (Requirement interfaces). *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:

[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_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 fomally 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

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

2.2 Semantics

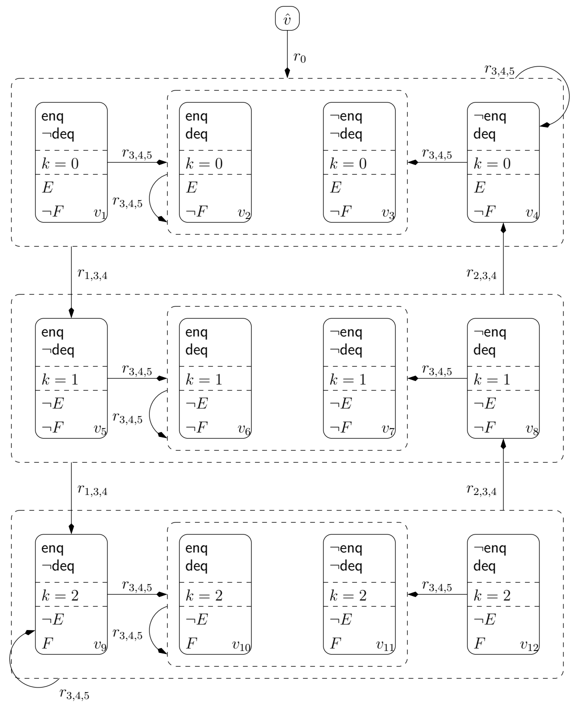
Given a requirement interface A defined over X, let $V = V(X) \cup \{\hat{v}\}$ denote the set of states in A, where a *state* v is a valuation $v \in V(X)$ or the *initial* state $\hat{v} \notin V(X)$. There is a transition between two states v and v' if (v, v') satisfies all its contracts. The transitions 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 (Semantics). *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) $v \rightarrow v'$ iff $\exists R \in L$ s.t. $v \xrightarrow{R} v'$; (3) $v \rightarrow$ iff $\exists v' \in V(X)$ s.t. $v \rightarrow v'$; (4) $v \overset{\epsilon}{\Rightarrow} v'$ iff $v = v'$; (5) $v \overset{w}{\Rightarrow} v'$ iff $\pi(v')[Y] = w$ and $v \rightarrow v'$; (6) $v \overset{w}{\Rightarrow}$ iff $\exists v'$ s.t. $\pi(v')[Y] = w$ and $v \rightarrow v'$; (7) $v \xRightarrow{w_1 \cdot w_2 \cdots w_n} v'$ iff $\exists v_1, \ldots, v_{n-1}, v_n$ s.t. $v \xRightarrow{w_1} v_1 \xRightarrow{w_2} \cdots v_n \xRightarrow{w_n} v'$; and (8) $v \xRightarrow{w_1 \cdot w_2 \cdots w_n}$ iff $\exists v'$ s.t. $v \xRightarrow{w_1 \cdot w_2 \cdots w_n} v'$.

[3] We use the notation $\varphi \vdash \psi$ to denote the (φ, ψ) assumption/guarantee pair.



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

Fig. 1. Labelled transition graph $[[A^{beh}]]$ illustrating the semantics of the bounded FIFO specification A^{beh}, where $N = 2$.

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 $L(A)$ 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 $\text{succ}(v) = \{v' \mid v \rightarrow v'\}$ be the set of successors of v.

Example 2. Figure 1 shows 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 $[[A]]$ and the trace corresponding to the above execution is $(\neg\mathsf{enq}, \neg\mathsf{deq}, E, \neg F), (\mathsf{enq}, \neg\mathsf{deq}, \neg E, \neg F), (\mathsf{enq}, \mathsf{deq}, \neg E, \neg F)$.

2.3 Consistency, Refinement and Conjunction

A requirement interface consists of a set of contracts, which 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 (Consistency). *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*

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

exists v' such that $w_I = \pi(v')[X_I]$, $\bigwedge_{c \in C}(v, v') \models c$ and $cons(v')$. We say that A is consistent *if $cons(\hat{v})$.*

Example 3. A^{beh} is consistent, since 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

$$c_2' : \neg \mathsf{enq}' \wedge \mathsf{deq}' \wedge k \geq 0 \; \vdash k' = k - 1,$$

which 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 the *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) $L(A^1) \subseteq L(A^2)$.*

We use a requirement interface to model a view of a system. Multiple views are combined by *conjunction.* 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 (Conjunction). *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 4. We now consider the *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 $\mathsf{enq}/\mathsf{deq}$ 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:

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

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

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 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.*

3 Consistency Checking, Testing and Tracing

In this section, we present our consistency checking, test case generation and execution framework and instantiate it with bounded model checking techniques. In this section, 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.

3.1 Bounded Consistency Checking

In order to do k-bounded consistency checking 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 $\hat{\theta} = \bigwedge_{(\hat{\varphi},\hat{\psi})\in\hat{C}} \hat{\varphi} \rightarrow \hat{\psi}$ and $\theta = \bigwedge_{(\varphi,\psi)\in C} \varphi \rightarrow \psi$. Then, the k-bounded consistency check for A corresponds to checking the satisfiability of the formula

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

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

3.2 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 requirement interface. Without loss of generality, we can represent I as a total sequential function $I : V(X_I) \times V(X_{\text{obs}})^* \rightarrow V(X_O)$. A test case T_A for a requirement interface A over X takes a history of actual input/output observations $\sigma \in L(A)$ and returns either the next input value to be executed or a verdict. Hence, a test case can be represented as a *partial* function $T_A : L(A) \rightarrow V(X_I) \cup \{\textbf{pass}, \textbf{fail}\}$.

We first consider the problem of generating a test case from A. The test case generation procedure is driven by a a *test purpose*. Informally, a test purpose is the target set of states that the SUT is intended to reach during testing. We choose the *reachability* test purpose[5] which is a formula Π defined over X_{obs}.

Given a requirement interface A, let $\hat{\phi} = \bigvee_{(\hat{\varphi},\hat{\psi})\in\hat{C}} \hat{\varphi} \wedge \bigwedge_{(\hat{\varphi},\hat{\psi})\in\hat{C}} \hat{\varphi} \rightarrow \hat{\psi}$ and $\phi = \bigvee_{(\varphi,\psi)\in C} \varphi \wedge \bigwedge_{(\varphi,\psi)\in C} \varphi \rightarrow \psi$. 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 $L(A)$ such that $w_{obs} \models \Pi$. The test purpose Π can be reached in A in at most k steps if

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

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 $L(A)$ that reaches Π. Let σ_I be $\pi(\sigma)[X_I] = w_I^0 \cdot w_I^1 \cdots w_I^n$. 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.

[5] Other test purposes can easily replace the reachability test purpose in our framework.

[6] The formula $\omega_{\sigma_I,A}$ can be seen as a monitor for A under 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:     $\omega^i_{\sigma_I,A} \leftarrow \theta[X_I \backslash w_I^{i\text{-}1}, X'_I \backslash w_I^i, X_{\text{ctr}} \backslash X^{i\text{-}1}_{\text{ctr}}, X'_{\text{ctr}} \backslash X^i_{\text{ctr}}]$
4: end for
5: $\omega^*_{\sigma_I,A} \leftarrow \omega^0_{\sigma_I,A} \wedge \ldots \wedge \omega^n_{\sigma_I,A}$
6: $\omega_{\sigma_I,A} \leftarrow \mathbf{qe}(\exists X_H^0, X_H^1, \ldots, X_H^n . \omega^*_{\sigma_I,A})$
7: return $\omega_{\sigma_I,A}$
```

Let $\hat{\theta} = \bigwedge_{(\hat{\varphi},\hat{\psi}) \in \hat{C}} \hat{\varphi} \rightarrow \hat{\psi}$ and $\theta = \bigwedge_{(\varphi,\psi)} \varphi \rightarrow \psi$. 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 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: $\sigma_I = w_I^0 \cdots w_I^n$, $A$, $\sigma = w^0_{obs} \cdots w^k_{obs}$
Output: $V(X_I^I) \cup \{\mathbf{pass}, \mathbf{fail}\}$
1: $\omega_{\sigma_I,A} \leftarrow$ OutMonitor$(\sigma_I, A)$
2: for $i = 0$ to $k$ do
3:     $w_O^i \leftarrow \pi(w^i_{obs})[X_O]$
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:     return pass
8: else if $\omega^{0,k}_{\sigma_I,A} = \mathbf{false}$ then
9:     return fail
10: else
11:     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 $0 \leq i \leq |\sigma|$, $w_I^i = \pi(w^i_{obs})[X_I]$, where $\sigma_I = w_I^0 \cdots w_I^n$ and $\sigma = w^0_{obs} \cdots w^k_{obs}$. Algorithm 2 gives a constructive definition of the test case $T_{\sigma_I,A}$.

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[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 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.

Test case generation for fully observable and deterministic interfaces: 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

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

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.

3.3 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 which 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$(I, T_{\sigma_I,A}) =$ **pass**; and*
2. *if TestExec$(I, T_{\sigma_I,A}) =$ **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, which follows directly from Proposition 1 and Theorem 2.

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

3.4 Traceability

Making requirement identifiers first class elements in requirement interfaces facilitates traceability between informal requirements, views and test cases. Consider a specification of the form $A = A^1 \wedge \ldots \wedge A^n$. A test case generated from an interface A^i is naturally mapped to its associated set of requirements $\mathcal{R}^i$.

Algorithm 4 Tracing possibly violated requirements.

```
Input: Trace $T =< w_I^0, w_O^0, ..., w_I^d, w_O^d >, A = \langle X_I, X_O, X_H, \hat{C}, C, \mathcal{R}, \rho \rangle$
Output: $r \subseteq \mathcal{R}$
 1: $r \leftarrow \{r_i \mid \exists (\hat{\varphi}, \hat{\psi}) \in \hat{C} \,.\, \hat{\varphi}[X'_I \backslash w_I^0] \wedge (\hat{\varphi}, \hat{\psi}) \in \rho(r_i)\}$; $t \leftarrow \hat{\theta}[X'_I \backslash w_I^0, X'_H \backslash X_H^0, X'_O \backslash w_O^0]$
 2: for $k \in \{1..d\}$ do
 3:     $t \leftarrow t \wedge \phi[X_I \backslash w_I^{k\text{-}1}, X'_I \backslash w_I^k, X_H \backslash X_H^{k\text{-}1}, X'_H \backslash X_H^k, X_O \backslash w_O^{k\text{-}1}, X'_O \backslash w_O^k]$
 4: end for
 5: for $k \in \{1..d\}$ do
 6:     for all $(\varphi, \psi) \in C$ do
 7:         if $\exists X_H^0, \ldots, X_H^d \,.\, (t \;\wedge\; \varphi[X_I \backslash w_I^{k\text{-}1}, X'_I \backslash w_I^k, X_H \backslash X_H^{k\text{-}1}, X_O \backslash w_O^{k\text{-}1}])$ then
 8:             $r \leftarrow r \cup \{r_i \mid (\varphi, \psi) \in \rho(r_i)\}$
 9:         end if
10:     end for
11: end for
12: return $r$
```

In addition, the association between contracts and requirement identifiers enables tracing violations caught during consistency checking and test case execution back to the conflicting/violated 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 [19], in order to compute a minimal set of contracts $C' \subset C$ 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.

A test case generated from an interface A that gives a **fail** verdict while being executed on the SUT, can be traced back to the set of requirements that were violated. For observable interfaces, the tracing is straightforward, because every trace corresponds to at most one execution, and the violated requirements are obtained from the labels on the path.

In the presence of hidden variables in the interface, a violation can be traced to multiple sets of requirements, depending on the assumed valuations of the hidden variables along the trace. Algorithm 4 computes the set of requirements that might be linked to the implementation fault: it takes as input a trace that leads the SUT to the **fail** verdict of the test case and the interface A from which the test case was generated from. The algorithm builds a term t comprising of the initial contracts (Line 1) and the step relation unrolled for each step of the trace (Line 3). The function *sub* used in Lines 1, 3, 7 replaces the input and output variables within the contracts by their concrete values along the trace. For each step along the trace, the algorithm tests for each contract, whether there exists a valid valuation for the hidden variables (Line 7), that still enables the previously defined term and enables the contract (Line 7). If there is, the contract might have been executed by the SUT and its corresponding requirements are part of the diagnosis set returned by our tool.

4 Implementation and Experimental Results

Implementation and experimental setup. We present a prototype tool that we developed and that implements our test case generation and consistency checking framework, introduced in Section 3.. The implementation uses Scala 2.10 and the SMT solver Z3. The tool implements both *monolithic* and *incremental* approaches to test case generation and consistency checking. All experiments were run on a MacBook Pro with a 2.53 GHz Intel Code 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 and *Buffer 3* models a buffer that is composed of three internal buffers of size 50. 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).

Table 1. Run-time comparison between inconsistency detection in single and conjuncted interfaces in seconds

	Fault 1 (behavior)		Fault 2 (power)	
	single	conjuncted	single	conjuncted
Buffer 1	0.7	3.6	1	7.3
Buffer 2	5.3	13.4	1	26.7
Buffer 3	7.2	13.8	1	13

Consistency Checking. In order 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 and incremental consistency checking. 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 conjunct them one by one, checking consistency of partial conjunctions. Table 1 summarizes the time to find an inconsistency and compute the minimal inconsistent set of requirements in a single view and in the conjunction of the interfaces modeling the buffer. Note that neither approach was able to find an inconsistency resulting from *Fault 3*, hence we omitted it in the table.

The bounded consistency checking is very sensible to the search depth. Setting the bound to 5 increases the run-time to 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.

To summarize, the incremental approach can significantly improve the performance of consistency checking, see the case of the inconsistent model *Fault 2* in Table 1. However, the approach still poorly scales when the inconsistency is deep in the model.

Test Case Generation. We compare the monolithic and incremental approach to test case generation. Table 2 summarizes the results. The three examples diverge in complexity, expressed in the number of contracts and variables, where multi-buffer models require a separate model for each internal buffer. Our results show that the incremental approach outperforms the monolithic one, resulting in speed-ups from 33% to 68%.

Table 2. Run-time comparison between the incremental test-case generation approach and the monolithic approach in seconds

	# Contracts	# Vars	t_{inc}	t_{mon}	speed-up
Single buffer	6	6	10	16.8	68 %
Two buffers	15	12	36.7	48.8	33 %
Three buffers	20	15	69	115.6	68 %

5 Related Work

Compositional modeling for open systems used in our framework was mainly inspired by interface theories [12, 13] and contract-based design [6] in general. In particular, we exploit the properties of the conjunction operation, introduced in [14] as *shared refinement*, in the context of multiple viewpoint modeling with synchronous interfaces [9]. The properties of the conjunction were further studied in [18], and the operation was also introduced for the A/G contracts theory [5]. In a recent work [24], basic properties of multiple viewpoint modeling are studied in an abstract formal framework. We also mention sociable interfaces, that allow combining communication through events and shared variables [11]. While highly relevant, none of this work considers multiple viewpoint modeling in the context of testing. In addition, none of the synchronous interface theories allows hiding of internal variables in their models.

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 [16]. The Lustre language was used as the basis for the commercial SCADE[8] language, extensively used to model safety critical systems in railway and avionics industry. These languages are implementation languages. Requirement interfaces can be seen as the high-level specification formalism for expressing correctness properties of Lustre/SCADE and SIGNAL programs, and the test cases generated from our specifications can be executed on them. We also mention reactive modules [4] that can be used to model synchronous data-flow systems and that partly inspired the syntax of requirement interfaces.

Testing of Lustre/SCADE programs was studied in [23, 22]. Compositional properties of specifications in the context of testing were studied in [27, 21, 25,

[8] www.esterel-technologies.com/products/scade-suite/

3, 10]. None of this work considers synchronous data-flow specification, and the compositional properties are investigated with respect to the parallel composition and hiding operations. This work is orthogonal and complementary to studying compositional properties of the conjunction and multiple viewpoint modeling in general. A different and complementary notion of conjunction is introduced in [17] for the test case generation with SAL. In that work, the authors encode test purposes as trap variables, and conjunct the variables in order to drive the test case generation process towards reaching all the test purposes with a single test case. Model checking (including bounded) techniques to generate test cases have been extensively studied and used, see [15] for a survey. We mention our previous work [2, 20] in which we use bounded model checking in the context of real-time mutation-based test case generation.

6 Conclusions and Future Work

We presented a framework for requirement-driven modeling and testing of complex systems. We believe that the multiple viewpoint approach, supported by requirement interfaces, is a natural way to model complex systems in a compositional manner from the requirements document. Exploiting the structure of the requirements, we showed that checking consistency of requirements and generating test cases can be done in a smarter and incremental manner. In addition, our requirement-driven approach to testing enables natural association between informal requirements, their formal models and the test cases (tracing).

Our requirement-driven framework opens many future directions. The test cases that we generate are non-adaptive – we fix the input vector that could drive the SUT to the test purpose, but we do not guarantee that the test purpose will be actually reached (due to the implementation freedom given by the specification). We will extend this work to support generation of adaptive test cases. In order to achieve this goal, we need to synthesize controllers that are able to adapt the input to the SUT based on its actual observed output. We believe that the controller synthesis for symbolic specifications is an interesting and important problem on its own. We focused in this paper on the multiple viewpoint modeling of a system using the conjunction operation. We will investigate in the future the composition of sub-system specifications, and the relations between the conjunction and the composition operations in the context of model-based testing. In this paper, we assumed that the requirements document is already decomposed into views. We plan to study whether this process can be automated, by analyzing dependencies between input and output variables of individual requirements. In the same spirit, we will investigate whether sensible test purposes can be directly inferred from the requirement interface. We will use our implementation to generate test cases for larger industrial-size systems from our automotive, avionics and railways partners, and will integrate our tracing capabilities with a requirement management tool.

References

1. ISO/DIS 26262-1 - Road vehicles Functional safety Part 1 Glossary. Technical report, Geneva, Switzerland, July 2009.
2. Bernhard K. Aichernig, Florian Lorber, and Dejan Nickovic. Time for mutants - model-based mutation testing with timed automata. In Margus Veanes and Luca Viganò, editors, *TAP*, volume 7942 of *Lecture Notes in Computer Science*, pages 20–38. Springer, 2013.
3. Marc Aiguier, Frédéric Boulanger, and Bilal Kanso. A formal abstract framework for modelling and testing complex software systems. *Theor. Comput. Sci.*, 455:66–97, 2012.
4. Rajeev Alur and ThomasA. Henzinger. Reactive modules. *Formal Methods in System Design*, 15(1):7–48, 1999.
5. Albert Benveniste, Benoît Caillaud, Alberto Ferrari, Leonardo Mangeruca, Roberto Passerone, and Christos Sofronis. Multiple viewpoint contract-based specification and design. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem P. de Roever, editors, *FMCO*, volume 5382 of *Lecture Notes in Computer Science*, pages 200–225. Springer, 2007.
6. Albert Benveniste, Benoit Caillaud, Dejan Nickovic, Roberto Passerone, Jean-Baptiste Raclet, Philipp Reinkemeier, Alberto Sangiovanni-Vincentelli, Werner Damm, Thomas Henzinger, and Kim G. Larsen. Contracts for System Design. Rapport de recherche RR-8147, INRIA, November 2012.
7. Albert Benveniste, Paul Caspi, Paul Le Guernic, and Nicolas Halbwachs. Data-flow synchronous languages. In J. W. de Bakker, Willem P. de Roever, and Grzegorz Rozenberg, editors, *REX School/Symposium*, volume 803 of *Lecture Notes in Computer Science*, pages 1–45. Springer, 1993.
8. Paul Caspi, Daniel Pilaud, Nicolas Halbwachs, and John Plaice. Lustre: A declarative language for programming synchronous systems. In *POPL*, pages 178–188. ACM Press, 1987.
9. Arindam Chakrabarti, Luca de Alfaro, Thomas A. Henzinger, and Freddy Y. C. Mang. Synchronous and bidirectional component interfaces. In Ed Brinksma and Kim Guldstrand Larsen, editors, *CAV*, volume 2404 of *Lecture Notes in Computer Science*, pages 414–427. Springer, 2002.
10. Przemyslaw Daca, Thomas A Henzinger, Willibald Krenn, and Dejan Ničković. Compositional specifications for ioco testing: Technical report. Technical report, IST Austria, 2014. `http://repository.ist.ac.at/152/`.
11. Luca de Alfaro, Leandro Dias da Silva, Marco Faella, Axel Legay, Pritam Roy, and Maria Sorea. Sociable interfaces. In Bernhard Gramlich, editor, *FroCoS*, volume 3717 of *Lecture Notes in Computer Science*, pages 81–105. Springer, 2005.
12. Luca de Alfaro and Thomas A. Henzinger. Interface automata. In *Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering*, ESEC/FSE-9, pages 109–120, New York, NY, USA, 2001. ACM.
13. Luca de Alfaro and Thomas A. Henzinger. Interface theories for component-based design. In Thomas A. Henzinger and Christoph M. Kirsch, editors, *EMSOFT*, volume 2211 of *Lecture Notes in Computer Science*, pages 148–165. Springer, 2001.
14. Laurent Doyen, Thomas A. Henzinger, Barbara Jobstmann, and Tatjana Petrov. Interface theories with component reuse. In *Proceedings of the 8th ACM international conference on Embedded software*, EMSOFT '08, pages 79–88, New York, NY, USA, 2008. ACM.

15. G. Fraser, F. Wotawa, and P. E. Ammann. Testing with model checkers: a survey. *Softw. Test. Verif. Reliab.*, 19(3):215–261, September 2009.
16. Thierry Gautier and Paul Le Guernic. Signal: A declarative language for synchronous programming of real-time systems. In Gilles Kahn, editor, *FPCA*, volume 274 of *Lecture Notes in Computer Science*, pages 257–277. Springer, 1987.
17. Grégoire Hamon, Leonardo De Moura, and John Rushby. Automated test generation with sal. *CSL Technical Note*, 2005.
18. Thomas A. Henzinger and Dejan Nickovic. Independent implementability of viewpoints. In Radu Calinescu and David Garlan, editors, *Monterey Workshop*, volume 7539 of *Lecture Notes in Computer Science*, pages 380–395. Springer, 2012.
19. Ulrich Junker. Quickxplain: Preferred explanations and relaxations for over-constrained problems. In *Proceedings of the 19th National Conference on Artifical Intelligence*, AAAI'04, pages 167–172. AAAI Press, 2004.
20. Willibald Krenn, Dejan Nickovic, and Loredana Tec. Incremental language inclusion checking for networks of timed automata. In Víctor A. Braberman and Laurent Fribourg, editors, *FORMATS*, volume 8053 of *Lecture Notes in Computer Science*, pages 152–167. Springer, 2013.
21. Moez Krichen and Stavros Tripakis. Conformance testing for real-time systems. *Formal Methods in System Design*, 34(3):238–304, 2009.
22. V. Papailiopoulou. Automatic test generation for lustre/scade programs. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, ASE '08, pages 517–520, Washington, DC, USA, 2008. IEEE Computer Society.
23. Pascal Raymond, Xavier Nicollin, Nicolas Halbwachs, and Daniel Weber. Automatic testing of reactive systems. In *RTSS*, pages 200–209. IEEE Computer Society, 1998.
24. Jan Reineke and Stavros Tripakis. Basic problems in multi-view modeling. Technical Report UCB/EECS-2014-4, EECS Department, University of California, Berkeley, Jan 2014.
25. Augusto Sampaio, Sidney Nogueira, and Alexandre Mota. Compositional verification of input-output conformance via csp refinement checking. In *ICFEM*, volume 5885 of *LNCS*, pages 20–48. Springer, 2009.
26. Jan Tretmans. Test generation with inputs, outputs and repetitive quiescence. *Software - Concepts and Tools*, 17(3):103–120, 1996.
27. Machiel van der Bijl, Arend Rensink, and Jan Tretmans. Compositional testing with ioco. In *FATES*, volume 3395 of *LNCS*, pages 86–100. Springer, 2003.

A Proofs

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

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 $X_I^1 = X_I^2 = \{x\}$, $X_O^1 = X_O^2 = \{y\}$, $X_H^1 = X_H^2 = \emptyset$, $\hat{C}^1 = C^1 = \{c^1\}$ and $\hat{C}^2 = C^2 = \{c^2\}$, where $c^1 = \textbf{true} \vdash y' = 0$ and $c^2 = \textbf{true} \vdash y' = 1$. 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 simulataneously ($y' = 0 \wedge y' = 1$).

Assume that $A^1 \wedge A^2$ is consistent. We now prove that $L(A^1 \wedge A^2) \subseteq L(A^1) \cap L(A^2)$. 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 $\sigma \in L(A^1 \wedge A^2)$. By inductive hypothesis, $\sigma \in L(A^1)$ and $\sigma \in L(A^2)$. Consider an arbitrary w_{obs} such that $\sigma \cdot w_{obs} \in L(A^1 \wedge A^2)$. Let $V_{1\wedge 2} = \{v \mid \hat{v} \overset{\sigma}{\Rightarrow} v\}$. By the definition of semantics of requirement interfaces, it follows that $V'_{1\wedge 2} = \{v' \mid v \xRightarrow{w_{obs}}_{1\wedge 2} v'$ for some $v \in V_{1\wedge 2}\}$ is non-empty. Let v' be an arbitrary state in $V'_{1\wedge 2}$, hence we have that $v \rightarrow_{1\wedge 2} v'$. Let $C^i_* = \{(\varphi, \psi) \mid (\varphi, \psi) \in C^i$ 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 $(v, v') \models \bigwedge_{(\varphi,\psi)\in C^1_*} \psi \wedge \bigwedge_{(\varphi,\psi)\in C^2_*} \psi$. It follows that $(v, v') \models \bigwedge_{(\varphi,\psi)\in C^1_*} \psi$, and $(v, v') \models \bigwedge_{(\varphi,\psi)\in C^2_*} \psi$, hence we can conclude that $v \rightarrow_1 v'$ and $v \rightarrow_2 v'$, hence $\sigma \cdot w_{obs} \in L(A^1)$ and $\sigma \cdot w_{obs} \in L(A^2)$, which concludes the proof that $L(A^1 \wedge A^2) \subseteq L(A^1) \cap L(A^2)$.

We now show that $L(A^1 \wedge A^2) \supseteq L(A^1) \cap L(A^2)$. 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 $\sigma \in L(A^1)$ and $\sigma \in L(A^2)$. By inductive hypothesis, it follows that $\sigma \in L(A^1 \wedge A^2)$. Let $\sigma = \sigma' \cdot v$. Consider an arbitrary w_{obs} such that $\sigma \cdot w_{obs} \in L(A^1)$ and $\sigma \cdot w_{obs} \in L(A^2)$. It follows that $v \xRightarrow{w_{obs}}_1$ and $v \xRightarrow{w_{obs}}_2$. Let $C^i_* = \{(\varphi, \psi) \mid (\varphi, \psi) \in C^i$ and $(v, w_{obs}) \models \varphi\}$ 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 $(v, v') \models \bigwedge_{(\varphi,\psi)\in C^1_*} \psi$, and $(v, v'') \models \bigwedge_{(\varphi,\psi)\in C^2_*} \psi$. By th assumption that $A^1 \wedge A^2$ is consistent, we have that there exists v' such that $(v, v') \models \bigwedge_{(\varphi,\psi)\in C^1_*} \psi \wedge \bigwedge_{(\varphi,\psi)\in C^2_*} \psi$, $\sigma \cdot w_{obs} \in L(A^1 \wedge A^2)$, which concludes the proof that $L(A^1 \wedge A^2) \supseteq L(A^1) \cap L(A^2)$.

$\square$

Theorem 2. *Let A^1 and A^2 be two consistent requirement interfaces defined over the same alphabet such that A^1 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 $A \preceq A^1 \wedge A^2$.*

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.

$\square$

Theorem 3. *Let A^1 be an inconsistent requirement interface. Then for all consistent requirement interfaces A^2 with the same alphabet as A^1, $A^1 \wedge A^2$ is also inconsistent.*

Proof. Follows directly from the definition of conjunction, which constrains the guarantees of individual interfaces.

Proposition 1. *Let A, $T_{\sigma_I,A}$ and I be an arbitrary requirement interface, a test case generated from A and implementation, respectively. Then, we have that:*

1. *if $I \preceq A$, then $\mathit{TestExec}(I, T_{\sigma_I,A}) = \textbf{pass}$; and*

   Proof. We first proof the loop invariant that if $I \preceq A$, then $\mathrm{in} \neq \textbf{fail}$ 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 I. If $I \preceq A$ this extended trace is 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.*

   $\square$

2. *if $\mathit{TestExec}(I, T_{\sigma_I,A}) = \textbf{fail}$, then $I \not\preceq A$.*

   Proof. By negation we obtain the proposition: if $I \preceq A$, then $\mathit{TestExec}(I, T_{\sigma_I,A}) \neq \textbf{fail}$. This follows directly from the loop invariant established above.

   $\square$