2009 Third IEEE International Symposium on Theoretical Aspects of Software Engineering

Fault-based Test Case Generation for Component Connectors*

Bernhard K. Aichernig[1,2], Farhad Arbab[3], Lăcrămioara Aştefănoaei[3],
Frank S. de Boer[3], Sun Meng[3†] and Jan Rutten[3]
[1]*UNU-IIST, P.O.Box 3058, Macao S.A.R., China*
bka@iist.unu.edu
[2]*Institute for Software Technology, Graz University of Technology, Austria*
aichernig@ist.tugraz.at
[3]*CWI, Kruislaan 413, Amsterdam, The Netherlands*
{*Farhad.Arbab, L.Astefanoaei, F.S.de.Boer, M.Sun, Jan.Rutten*}*@cwi.nl*

Abstract

The complex interactions appearing in service-oriented computing make coordination a key concern in service-oriented systems. In this paper, we present a fault-based method to generate test cases for component connectors from specifications. For connectors, faults are caused by possible errors during the development process, such as wrongly used channels, missing or redundant subcircuits, or circuits with wrongly constructed topology. We give test cases and connectors a unifying formal semantics by using the notion of design, and generate test cases by solving constraints obtained from the specification and faulty connectors. A prototype symbolic test case generator serves to demonstrate the automatizing of the approach.

1. Introduction

With the growth of interest in service oriented computing (SOC) [15], a key aspect of aggregating business processes and web services is the coordination among services. Services are autonomous, platform-independent computational entities that can be described, published, categorized, discovered, and dynamically assembled for developing complex and evolvable applications that may run on large-scale distributed systems. Such systems, which typically are heterogeneous and geographically distributed, usually exploit communication infrastructures whose topology frequently varies and components can, at any moment, connect to or detach from. Coordination and composition have a key role in systems based on the notion of service: Services can be invoked by other services or simply interact with each other in order to carry on a task. Compositional coordination models and languages provide a formalization of the "glue code" that interconnects the constituent components / services and organizes the communication and cooperation among them in a distributed environment. They support large-scale distributed applications by allowing construction of complex component connectors out of simpler ones. As an example, Reo [5, 7] offers a powerful glue language for implementation of coordinating component connectors based on a calculus of mobile channels.

The complexity and importance of coordination models in service-oriented applications necessarily lead to a higher relevance of testing issues for connectors during development of systems. Testing is a widely used and accepted approach for validation and verification of software systems, and can be regarded as the ultimate review of their specifications, designs and implementations. Testing is applied to generate modes of behavior on the final product that show whether it is conforming to its original requirement specification, and to support the confidence in its safe and correct operation. Appropriate testing should always aim to show conformance or non-conformance of the final software system with some requirements or specifications. Since the behavior of connectors generally describes the manifold interactions among components / services rather than simple input-output behavior, where different input and output actions can be synchronized, we can use not simply sequences of input and output, but relations on different input / output sequences as test cases for connectors.

In this paper, we discuss the problem on *fault-based test case generation* for component connectors with Reo as our target implementation language. Similar to the mutation testing approach for programs in [3], the specification and implementation of connectors are given by pairs of pre-

* The work in this paper is supported by a grant from the GLANCE funding program of the Dutch National Organization for Scientific Research (NWO), through project CooPer (600.643.000.05N12).

† Corresponding author.

978-0-7695-3757-3/09 $25.00 © 2009 IEEE
DOI 10.1109/TASE.2009.14

and post-condition, and we adopt the design semantics for Reo connectors as used in Unifying Theories of Programming (UTP) [14]. Every connector can be modelled as a *design*, i.e., a pair of predicates $P \vdash Q$ where the assumption P is what the designer can rely on when the communicating operation is initiated by inputs to the connectors, and the commitment Q must be true for the outputs when the communicating operation terminates.

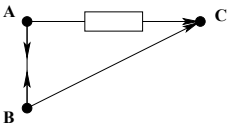

Figure 1. Example

Consider the example connector given in Figure 1. The behavior of this connector can be seen as imposing an order on the flow of data items written to **A** and **B**, through **C**: The data items obtained by successive take operations on **C** consist of the first data item written to **A**, followed by the first data item written to **B**, followed by the second data item written to A, followed by the second data item written to **B**, and so on. This connector can be mutated introducing possible faults like changing the topology of the connector, as shown in Figure 2. Such a faulty connector has a different behavior as the one in Figure 1. In other words, the sequence of values that appear through **C** consist of zero or more repetitions of the pairs of values written to **B** and **A**, in a different order. The idea is to automatically generate a test case that has different input data sequences to **A** and **B**, then it can detect such errors and exclude faulty implementations.

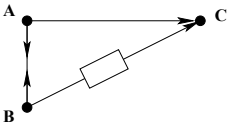

Figure 2. A Faulty Implementation for the Alternator Connector

The theoretical foundation of this work is based on the first author's earlier work [2, 3], in which it can be formally proven that test cases will detect certain faults by using refinement calculus. This paper is the first time that the mutation testing approach is extended to formal coordination language. Furthermore, we have a new model of Reo circuits, given by using the notion of designs. Comparing with the other semantic models of Reo, like constraint automata [7] or the coalgebraic model [6], in which the underlying time streams are infinite and thus the finite behavior (and connectors which exhibit finite behavior on any of their ports) are excluded by the model, the timed data sequence in our model can be either finite or infinite, and thus more expressive the the coalgebraic model. A prototype tool for test case generation for connectors has been developed by using the Maude language [1].

The remainder of this paper is organized as follows: Section 2 contains a brief summary of Reo. We present the design semantics for Reo connectors in Section 3. In Section 4, we explain the relation between testing and the theory of designs and present the algorithm that generates test cases to find anticipated errors in connectors. The use of a prototype for a simple example is presented in Section 5. In Section 6, we present related work and compare it with our approach. Finally, Section 7 concludes the paper.

2. Reo

Reo [5] is a channel-based exogenous coordination model wherein complex coordinators, called connectors, are compositionally built out of simpler ones. Reo relies on a very liberal and simple notion of channels and can be used to model any kind of peer-to-peer communication. We summarize only the main concepts in Reo here. The semantics of Reo connectors are given by relations on input / output timed data sequences in Section 3. Further details about Reo and its semantics can be found in [5, 6, 7].

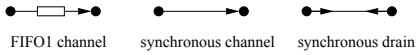

Figure 3. Some Basic Channels in Reo

Complex connectors in Reo are organized in a network of primitive connectors, called *channels*, that serve to provide the protocol that controls and organizes the communication, synchronization and cooperation among the components/services that they interconnect. Each channel has two *channel ends*. There are two types of channel ends: *source* and *sink*. A source channel end accepts data into its channel, and a sink channel end dispenses data out of its channel. It is possible for the ends of a channel to be both sinks or both sources. Reo places no restriction on the behavior of a channel and thus allows an open-ended set of different channel types to be used simultaneously together. Each channel end can be connected to at most one component instance at any given time. Figure 3 shows the graphical representation of some simple channel types that will be used in this paper. More detailed discussion can be found in [5].

A complex connector has a graphical representation, called *Reo circuit* or *network*. Reo circuits are constructed

by composing simpler ones via the *join* and *hiding* operations. A node in a Reo circuit represents a set of channel ends. A node arises through the *join* operator and can be categorized as a *source*, *sink* or *mixed* node, depending on whether all channel ends that coincide on the node are source ends, sink ends or a combination of both. The hiding operation is used to hide the internal topology of a component connector. The hidden (mixed) nodes can no longer be accessed or observed from outside.

A component can write data items to a source node that it is connected to. The write operation succeeds only if all (source) channel ends coincident on the node accept the data item, in which case the data item is transparently written to every source end coincident on the node. A source node, thus, acts as a replicator. A component can obtain data items, by an input operation, from a sink node that it is connected to. A take operation succeeds only if at least one of the (sink) channel ends coincident on the node offers a suitable data item; if more than one coincident channel end offers suitable data items, one is selected nondeterministically. A sink node, thus, acts as a nondeterministic merger. A mixed node nondeterministically selects and takes a suitable data item offered by one of its coincident sink channel ends and replicates it into all of its coincident source channel ends. Note that a component cannot connect to, take from, or write to mixed nodes. At most one component can be connected to a (source or sink) node at a time. The I/O operations are performed through interface nodes of components which are called ports.

3. Connectors as Designs

For an arbitrary connector $\mathbf{R}$, the relevant observations come in pairs, with one observation on the source nodes of $\mathbf{R}$, and one observation on the sink nodes of $\mathbf{R}$. For every node N, the corresponding observation on N is given by a timed data sequence, which is defined as follows:

Let D be an arbitrary set, the elements of which is called data elements. The set DS of data sequences is defined as $DS = D^*$, i.e., the set of all sequences $\alpha = (\alpha(0), \alpha(1), \alpha(2), \cdots)$ over D. Let $\mathbb{R}_+$ be the set of non-negative real numbers, which can be used in the present context to represent the time moments[1]. Let $\mathbb{R}_+^*$ be the set of sequences $a = (a(0), a(1), a(2), \cdots)$ over $\mathbb{R}_+$, and for all $a = (a(0), a(1), a(2), \cdots)$ and $b = (b(0), b(1), b(2), \cdots)$ in $\mathbb{R}_+^*$, if $|a| = |b|$, then

$$\begin{array}{lll} a < b & \text{iff} & \forall 0 \leq n \leq |a|, a(n) < b(n) \\ a \leq b & \text{iff} & \forall 0 \leq n \leq |a|, a(n) \leq b(n) \end{array}$$

where $|a|$ returns the length of sequence a.

The set TS of time sequences is defined by

$$TS = \{a \in \mathbb{R}_+^* \mid \forall 0 \leq n < |a|.a(n) < a(n+1)\}$$

Thus any time sequence $a \in TS$ consists of increasing time moments $a(0) < a(1) < a(2) < \cdots$.

The set TDS of timed data sequences is defined by $TDS \subseteq DS \times TS$ that contains pairs $\langle \alpha, a \rangle$ consisting of a data sequence α and a time sequence a with $|\alpha| = |a|$. Similar as the discussion in [6], timed data sequences can be alternatively and equivalently defined as (a subset of) $(D \times \mathbb{R}_+)^*$ because of the existence of isomorphism

$$\langle \alpha, a \rangle \mapsto (\langle \alpha(0), a(0) \rangle, \langle \alpha(1), a(1) \rangle, \langle \alpha(2), a(2) \rangle, \cdots)$$

For connectors, once a data is written or taken at some node, there is a time moment that the operation happens, which is modelled by one element in the timed data sequence for that node, i.e., a pair of data element and time moment.

In our semantic model, the observational semantics for a Reo connector will be described by a design, i.e., a relation expressed as $P \vdash Q$, where P is the predicate specifying the relationship among the timed data sequences on the input ends of the connector, and Q is the predicate specifying the condition that should be satisfied by the timed data sequences on the output ends of the connector. Note that a design $P \vdash Q$ has the following meaning:

$$P \vdash Q =_{df} (ok \wedge P \Rightarrow ok' \wedge Q)$$

where ok and ok' are two variables being used to analyze explicitly the phenomena of communication initialization and termination. The variable ok stands for a successful initialization and the start of a communication. When ok is **false**, the communication has not started, so no observation can be made. The variable ok' denotes the observation that the communication has terminated. The communication is divergent when ok' is **false**. To specify input and output explicitly, for a connector $\mathbf{R}$, we use $in_{\mathbf{R}}$ and $out_{\mathbf{R}}$ for the lists of timed data sequences on the input ends and output ends of $\mathbf{R}$ respectively and do not use primed variables. So the alphabet is different with that for design, as in [14]. Furthermore, healthiness conditions, which embody aspects about the model being studied, are taken as **true** here.

Since we are considering test case generation, all of which should be finite sequences, all timed data sequences in the following are finite. However, the semantic definition can be generalized to infinite sequences, which are timed data streams as proposed in [6]. We use $\mathcal{D}$ for a predicate of well-defined timed data sequence types. In other words,

1 Here we use the continuous time model since it is expressive and closer to the nature of time in the real world for connectors. For example, for a FIFO1 channel, if we have a sequence of two inputs, the time moment for the output should be between the two inputs. If we use a discrete time model like $\mathbb{N}$, and have the first input at time point 1, then the second input can only happen at a time point greater than 2, i.e., at least 3. But in general, this is not explicit for the input providers.

we define the behavior only for valid sequences expressed via a predicate $\mathcal{D}$. Then every connector $\mathbf{R}$ can be represented as the following design,

$$\mathbf{R}(in : in_{\mathbf{R}}; out : out_{\mathbf{R}})$$
$$P(in_{\mathbf{R}}) \vdash Q(in_{\mathbf{R}}, out_{\mathbf{R}})$$

where $P(in_{\mathbf{R}})$ is the precondition that should be satisfied by inputs $in_{\mathbf{R}}$ on the source nodes of $\mathbf{R}$, and $Q(in_{\mathbf{R}}, out_{\mathbf{R}})$ is the postcondition that should be satisfied by outputs $out_{\mathbf{R}}$ on the sink nodes of $\mathbf{R}$.

Here are some simple examples:

- Synchronous channel $\longrightarrow$:

$$\mathbf{Sync}(in : (\langle \alpha, a\rangle); out : (\langle \beta, b\rangle))$$
$$\mathcal{D}\langle \alpha, a\rangle \vdash \mathcal{D}\langle \beta, b\rangle \wedge \beta = \alpha \wedge b = a$$

  The synchronous channel just transfers the data without delay in time. So it behaves just as the identity function. The pair of I/O operations on its two ends can succeed only simultaneously, and the input is not taken until the output can be delivered, which is captured by the variable ok.

- FIFO1 channel $-\square\rightarrow$:

$$\mathbf{FIFO1}(in : (\langle \alpha, a\rangle); out : (\langle \beta, b\rangle))$$
$$\mathcal{D}\langle \alpha, a\rangle \vdash \mathcal{D}\langle \beta, b\rangle \wedge \beta = \alpha \wedge a < b \wedge$$
$$(tail(b^R))^R < tail(a)$$

  where for a sequence $a = (a(0), a(1), \cdots, a(n))$, $a^R = (a(n), \cdots, a(1), a(0))$ returns the reverse of a, and $tail(a) = (a(1), \cdots, a(n))$. For simplicity of notations, we use a' to denote $tail(a)$ in the rest of this paper. For a FIFO1 channel, when the buffer is not filled, the input is accepted without immediately outputting it. The accepted data item is kept in the internal FIFO buffer of the channel. The next input can only happen after an output occurs. Note that here we use $(tail(b^R))^R < tail(a)$ to represent the relationship between the time moments for outputs and the corresponding next inputs. This notation can be simplified to $b < tail(a)$ when we consider infinite sequences of inputs and outputs.

- Synchronous drain $\rightarrow\!\leftarrow$:

$$\mathbf{SyncDrain}(in : (\langle \alpha, a\rangle, \langle \beta, b\rangle); out : (\ ))$$
$$\mathcal{D}\langle \alpha, a\rangle \wedge \mathcal{D}\langle \beta, b\rangle \wedge a = b \vdash \mathbf{true}$$

  This channel has two input ends. The pair of input operations on its two ends can succeed only simultaneously. All data items written to this channel are lost. the predicate $\mathbf{true}$ in Q means the communication terminates.

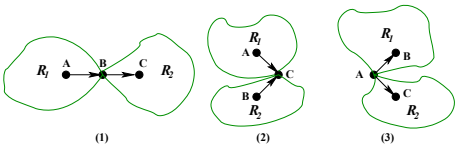

Figure 4. Connector composition

Different connectors can be composed. Since connectors can be modelled by designs, their composition can be naturally modelled by design composition. According to the node types in Reo, we have three types of composition, which can be captured by the three configurations as shown in Figure 4. Note that these composition operations can be easily generalized to the case for merging multiple nodes, and merging different nodes of the same connector. Due to the length limitation, here we use a simple example to show our approach for composition of connectors, instead of giving the formal definition and explanations for all the operations. More details can be found in [17].

Example 3.1 *Figure 1 shows a Reo circuit consisting of three channels* **AB**, **AC** *and* **BC** *which are of types* **SyncDrain**, **FIFO1** *and* **Sync**, *respectively. By composing the channels, we can get the connector as*

$$\mathbf{Order}(in : (\langle \alpha, a\rangle, \langle \beta, b\rangle); out : \langle \gamma, c\rangle)$$
$$P \vdash Q$$

where $P = \mathcal{D}\langle \alpha, a\rangle \wedge \mathcal{D}\langle \beta, b\rangle \wedge a = b$, *and*

$$\begin{aligned} Q = {} & \mathcal{D}\langle \gamma, c\rangle \wedge \exists \langle \gamma_1, c_1\rangle, \langle \gamma_2, c_2\rangle. \\ & \mathcal{D}\langle \gamma_1, c_1\rangle \wedge \mathcal{D}\langle \gamma_2, c_2\rangle \wedge \\ & \gamma_1 = \alpha \wedge a < c_1 \wedge ((c_1^R)')^R < a' \wedge \gamma_2 = \beta \wedge c_2 = b \wedge \\ & M(\langle \gamma_1, c_1\rangle, \langle \gamma_2, c_2\rangle, \langle \gamma, c\rangle) \end{aligned}$$

*In this design, a ternary relation M on timed data sequences is used to represent the merging the sink nodes of two channels (***AC** *and* **BC** *in this example), which is formally defined as*

$$M(\langle \gamma_1, c_1\rangle, \langle \gamma_2, c_2\rangle, \langle \gamma, c\rangle)$$
$$= \begin{cases} \langle \gamma, c\rangle = \langle \gamma_1, c_1\rangle & \textit{if } |\langle \gamma_2, c_2\rangle| = 0 \\ \langle \gamma, c\rangle = \langle \gamma_2, c_2\rangle & \textit{if } |\langle \gamma_1, c_1\rangle| = 0 \\ c_1(0) \neq c_2(0) \wedge & \\ \quad \begin{cases} \gamma(0) = \gamma_1(0) \wedge c(0) = c_1(0) \wedge & \\ M(\langle \gamma_1', c_1'\rangle, \langle \gamma_2, c_2\rangle, \langle \gamma', c'\rangle) \textit{ if } c_1(0) < c_2(0) & \\ \gamma(0) = \gamma_2(0) \wedge c(0) = c_2(0) \wedge & \\ M(\langle \gamma_1, c_1\rangle, \langle \gamma_2', c_2'\rangle, \langle \gamma', c'\rangle) \textit{ if } c_2(0) < c_1(0) & \end{cases} & \\ \qquad \textit{otherwise} \end{cases}$$

Implication of predicates establishes a refinement order over connectors. Thus, more concrete implementations im-

ply more abstract specifications. For two connectors

$$\mathbf{R}_i(in : in_{\mathbf{R}_i}; out : out_{\mathbf{R}_i})$$
$$P_i(in_{\mathbf{R}_i}) \vdash Q_i(in_{\mathbf{R}_i}, out_{\mathbf{R}_i})$$

where $i = 1, 2$, if $in_{\mathbf{R}_1} = in_{\mathbf{R}_2}$ and $out_{\mathbf{R}_1} = out_{\mathbf{R}_2}$, then

$$\mathbf{R}_1 \sqsubseteq \mathbf{R_2} \;=_{df}\; (P_1 \Rightarrow P_2) \wedge (P_1 \wedge Q_2 \Rightarrow Q_1) \qquad (1)$$

In other words, preconditions on inputs of connectors are weakened under refinement, and postconditions on outputs of connectors are strengthened.

4. Fault-based Test Case Generation

4.1. Test Cases for Connectors

In this paper, we aim to generate test cases on the basis of possible errors during the design of connectors. Examples of such errors might be a wrongly used channel, a missing subcircuit, or a circuit with wrongly constructed topology, etc. Both specifications and implementations of connectors are represented by using designs. As recommended in [2, 4], refinement is the central notion to discuss the roles and consequences of certain faults and design predicates that are most suitable for representing faults.

Definition 4.1 (faulty connector) *Given an intended connector specification*

$$\mathbf{R}(in : in_{\mathbf{R}}; out : out_{\mathbf{R}})$$
$$P(in_{\mathbf{R}}) \vdash Q(in_{\mathbf{R}}, out_{\mathbf{R}})$$

and a connector implementation

$$\mathbf{R}'(in : in_{\mathbf{R}}; out : out_{\mathbf{R}})$$
$$P'(in_{\mathbf{R}}) \vdash Q'(in_{\mathbf{R}}, out_{\mathbf{R}})$$

with the same input and output nodes as in $\mathbf{R}$*, which may contain an error.* $\mathbf{R}'$ *is called a faulty connector if and only if* $\mathbf{R} \not\sqsubseteq \mathbf{R}'$.

Note that not all errors during the design of connectors lead to faulty connectors. To contain a fault, a possible external observation of the fault must exist. For example, adding by mistake redundant synchronous channels to some input/output nodes does not result in a faulty connector since refinement holds (in fact, this results in a bisimilar connector, which is a stronger relation than refinement). However, swapping the nodes leads to a faulty connector. For example, if we swap **A** and **B** in Figure 1, the resulting connector $\mathbf{Order}'$ is a faulty connector with respect to the specification $\mathbf{Order}$.

For connectors, we consider test cases as specifications that define the expected list of timed data sequences on the output nodes for a given list of timed data sequences on the input nodes.

Definition 4.2 (deterministic test case) *For a connector* $\mathbf{R}(in : in_{\mathbf{R}}; out : out_{\mathbf{R}})$*, let* i *be the input vector and* o *be the output vector, both lists of timed data sequences with the same lengths as* $in_{\mathbf{R}}$ *and* $out_{\mathbf{R}}$ *respectively. A deterministic test case for* $\mathbf{R}$ *is defined as*

$$t_d(in_{\mathbf{R}}, out_{\mathbf{R}}) = in_{\mathbf{R}} = i \vdash out_{\mathbf{R}} = o$$

Sometimes the behavior of a connector can be nondeterministic. In this case, we can generalize the notion of test case as follows:

Definition 4.3 (test case) *For a connector* $\mathbf{R}(in : in_{\mathbf{R}}; out : out_{\mathbf{R}})$*, let* i *be the input vector and* O *be a possibly infinite set containing the expected output vector(s). Both* i *and any* $o \in O$ *are lists of timed data sequences with the same lengths as* $in_{\mathbf{R}}$ *and* $out_{\mathbf{R}}$ *respectively. A test case for* $\mathbf{R}$ *is defined as*

$$t(in_{\mathbf{R}}, out_{\mathbf{R}}) = in_{\mathbf{R}} = i \vdash out_{\mathbf{R}} \in O$$

From the previous discussion, we know that test cases, as well as connector specifications and implementations, can be specified by designs. It is obvious that an implementation that is correct with respect to its specification should refine its test cases. Therefore, test cases are abstractions of an implementation if and only if the implementation passes the test cases. Taking specifications into consideration, test cases should also be abstractions of a specification if they are properly derived from the specification.

Definition 4.4 *For a connector specification* $\mathbf{S}$*, its implementation* $\mathbf{R}$ *and a test case* t*, which satisfy*

$$t \sqsubseteq \mathbf{S} \sqsubseteq \mathbf{R}$$

- t *is called a correct test case with respect to* $\mathbf{S}$.
- $\mathbf{R}$ *passes the test case* t *and conforms to the specification* $\mathbf{S}$.

Finding a test case t that detects a given fault is the central strategy in fault-based testing. For connectors, a fault-based test case is defined as follows:

Definition 4.5 (fault-adequate test case) *Let* $t(in_{\mathbf{R}}, out_{\mathbf{R}})$ *be a test case (which can be either deterministic or non-deterministic),* $\mathbf{R}$ *an expected connector, and* $\mathbf{R}'$ *its faulty implementation. Then* t *is a fault-adequate test case if and only if*

$$t \sqsubseteq \mathbf{R} \wedge t \not\sqsubseteq \mathbf{R}'$$

A fault-adequate test case detects a fault in $\mathbf{R}'$*. Alternatively we can say that the test case distinguishes* $\mathbf{R}$ *and* $\mathbf{R}'$*. All test cases that detect a certain fault form a fault-adequate equivalence class.*

4.2. Test Case Generation

A strategy for generating test cases is always based on a hypothesis about faults. Definition 4.5 shows that a test case for finding errors in a faulty connector has to, first, be a correct test case of the intended connector; second, it must not be an abstraction of the faulty connector. For a given intended connector $\mathbf{R}$ and its faulty implementation $\mathbf{R}'$, the following algorithm generates a test case.

Algorithm 4.1 *Consider an intended connector*

$$\mathbf{R}(in : in_{\mathbf{R}}; out : out_{\mathbf{R}})$$
$$P(in_{\mathbf{R}}) \vdash Q(in_{\mathbf{R}}, out_{\mathbf{R}})$$

and its faulty implementation

$$\mathbf{R}'(in : in_{\mathbf{R}}; out : out_{\mathbf{R}})$$
$$P'(in_{\mathbf{R}}) \vdash Q'(in_{\mathbf{R}}, out_{\mathbf{R}})$$

as inputs. A test case t is generated by the following steps:

1. *A test case t is searched by*
   1. *find a pair $\langle \widehat{i}, \widehat{o} \rangle$ as a solution of*
   $$P(\widehat{i}) \wedge Q'(\widehat{i}, \widehat{o}) \wedge \neg Q(\widehat{i}, \widehat{o})$$
   2. *if it exists, then the test case $t(i, O)$ is generated by finding the maximal set O of output vectors, such that for all $o \in O$, $i = \widehat{i} \wedge P(i) \wedge Q(i, o)$ is satisfied.*
2. *If the previous step does not succeed, then look for a test case $t(i, O)$ with O the maximal set of output vectors, such that for all $o \in O$, $\neg P'(i) \wedge P(i) \wedge Q(i, o)$ is satisfied.*

Theorem 4.1 (Correctness) *Given an intended connector*

$$\mathbf{R}(in : in_{\mathbf{R}}; out : out_{\mathbf{R}})$$
$$P(in_{\mathbf{R}}) \vdash Q(in_{\mathbf{R}}, out_{\mathbf{R}})$$

its faulty implementation

$$\mathbf{R}'(in : in_{\mathbf{R}}; out : out_{\mathbf{R}})$$
$$P'(in_{\mathbf{R}}) \vdash Q'(in_{\mathbf{R}}, out_{\mathbf{R}})$$

and a test case $t(i, O)$ generated by Algorithm 4.1,

$$t(i, O) \sqsubseteq \mathbf{R}$$

Proof: The proof is divided into two cases, corresponding to steps 1 and 2 of the algorithm, respectively.

1. Since O is the maximal set such that for all $o \in O$, $P(i) \wedge Q(i, o)$ is satisfied, we have

$$\begin{aligned}
& t(i, O) \sqsubseteq \mathbf{R} \\
\equiv & \{(1) \text{ and Definition } 4.3\} \\
& (in_{\mathbf{R}} = i \Rightarrow P(in_{\mathbf{R}})) \wedge \\
& (Q(in_{\mathbf{R}}, out_{\mathbf{R}}) \wedge in_{\mathbf{R}} = i \Rightarrow out_{\mathbf{R}} \in O) \\
\equiv & \{O \text{ is the maximal set such that } P(i) \wedge Q(i, o) \\
& \text{ is satisfied for all } o \in O\} \\
& \mathbf{true} \wedge \mathbf{true} \\
\equiv & \mathbf{true}
\end{aligned}$$

2. Since O is the maximal set of output vectors o each of which satisfies $\neg P'(i) \wedge P(i) \wedge Q(i, o)$, it follows that $P(i) \wedge Q(i, o)$ is also satisfied for all $o \in O$ and O is still maximal. By the same style of reasoning as in the first case, we can derive that

$$t(i, O) \sqsubseteq \mathbf{R} \ \equiv \ \mathbf{true}$$

Theorem 4.2 (Fault coverage) *Given an intended connector*

$$\mathbf{R}(in : in_{\mathbf{R}}; out : out_{\mathbf{R}})$$
$$P(in_{\mathbf{R}}) \vdash Q(in_{\mathbf{R}}, out_{\mathbf{R}})$$

its faulty implementation

$$\mathbf{R}'(in : in_{\mathbf{R}}; out : out_{\mathbf{R}})$$
$$P'(in_{\mathbf{R}}) \vdash Q'(in_{\mathbf{R}}, out_{\mathbf{R}})$$

and a test case $t(i, O)$ generated by Algorithm 4.1,

$$t(i, O) \not\sqsubseteq \mathbf{R}'$$

Proof: This proof is also divided into the two cases of the algorithm.

1. When O is the maximal set such that for all $o \in O$, $i = \widehat{i} \wedge P(i) \wedge Q(i, o)$ is satisfied. The test case is generated only if there exists $(\widehat{i}, \widehat{o})$, such that $P(\widehat{i}) \wedge Q'(\widehat{i}, \widehat{o}) \wedge \neg Q(\widehat{i}, \widehat{o})$ is satisfied. Thus we have $i = \widehat{i}$, $\widehat{o} \notin O$ and

$$\begin{aligned}
& t(i, O) \not\sqsubseteq \mathbf{R}' \\
\equiv & \{(1) \text{ and Definition } 4.3\} \\
& \neg(in_{\mathbf{R}} = i \Rightarrow P'(in_{\mathbf{R}})) \vee \\
& \neg(in_{\mathbf{R}} = i \wedge Q'(in_{\mathbf{R}}, out_{\mathbf{R}}) \Rightarrow out_{\mathbf{R}} \in O) \\
\equiv & (in_{\mathbf{R}} = i \wedge \neg P'(in_{\mathbf{R}})) \vee \\
& \exists in_{\mathbf{R}}, out_{\mathbf{R}}.(in_{\mathbf{R}} = i \wedge Q'(in_{\mathbf{R}}, out_{\mathbf{R}}) \wedge out_{\mathbf{R}} \notin O) \\
\equiv & \{\text{let } in_{\mathbf{R}} = \widehat{i}, out_{\mathbf{R}} = \widehat{o}\} \\
& (in_{\mathbf{R}} = i \wedge \neg P'(in_{\mathbf{R}})) \vee \mathbf{true} \\
\equiv & \mathbf{true}
\end{aligned}$$

2. If O is the maximal set of output vectors o each of which satisfies $\neg P'(i) \wedge P(i) \wedge Q(i, o)$, then

$$\begin{aligned}
& t(i, O) \not\sqsubseteq \mathbf{R}' \\
\equiv & \{(1) \text{ and Definition } 4.3\} \\
& \neg(in_{\mathbf{R}} = i \Rightarrow P'(in_{\mathbf{R}})) \vee \\
& \neg(in_{\mathbf{R}} = i \wedge Q'(in_{\mathbf{R}}, out_{\mathbf{R}}) \Rightarrow out_{\mathbf{R}} \in O) \\
\equiv & \exists in_{\mathbf{R}}.(in_{\mathbf{R}} = i \wedge \neg P'(in_{\mathbf{R}})) \vee \\
& \neg(in_{\mathbf{R}} = i \wedge Q'(in_{\mathbf{R}}, out_{\mathbf{R}}) \Rightarrow out_{\mathbf{R}} \in O) \\
\equiv & \{\text{let } in_{\mathbf{R}} = i\} \\
& \mathbf{true} \vee \neg(in_{\mathbf{R}} = i \wedge Q'(in_{\mathbf{R}}, out_{\mathbf{R}}) \Rightarrow out_{\mathbf{R}} \in O) \\
\equiv & \mathbf{true}
\end{aligned}$$

5. Example

We have developed a prototype tool for testing Reo connectors in Maude [1]. Maude is a high-performance reflective language and system supporting *equational and rewriting logic specification and programming*. One of the main advantages of Maude is that it provides a single framework in which the use of a wide range of formal methods is facilitated. For example, the language has been shown to be suitable both as a logical and as a semantic framework [8, 16]. This makes it easy to: (1) specify basic connectors equationally, (2) encode their composition by means of rewrite rules and (3) find test cases by rewriting constraints.

The source code of the tool is available at `http://homepages.cwi.nl/~astefano/testing-reo/`. In the current version, it is possible to specify and compose connectors as it is described in Section 3. Furthermore, since we have also prototyped Algorithm 4.1, one can experiment with searching test cases for faulty connectors. We provide as a default illustration the Maude implementation for the following scenario. We take the alternator connector in Example 3.1 to represent the intended connector specification. We recall that this connector can have faulty implementations where the topology of the channels is changed, or miss or wrongly use some channels. The faulty implementation that we consider is the one shown in Figure 2. As one might expect, running the search procedure for generating test cases for the faulty connector with respect to the correct one, we obtain, given as symbolic input a pair of one element timed data sequences $(\langle \alpha, a \rangle, \langle \beta, b \rangle)$, the result that the pair is a valid test case whenever the constraint $\alpha \neq \beta$ is satisfied. We note that lists of one element represent minimal test cases. Any pair of arbitrary length lists is, in fact, a test case, under the condition that they differ by at least one element.

The example, although trivial, demonstrates the possibility to automate the connector testing approach. Both behavior and data are crucial for successful test case generation. However, in most cases, connectors interact with an environment, which stimulates the execution of I/O operations on the ports of the connectors, where the data values being written/taken come from large or even infinite domains. Considering such data values at test generation stage may lead to non-terminate generation process or produce many unnecessary test cases. Therefore, we abstract away from concrete data values and concentrate on symbolic test case generation, and reintroduce data for test execution later.

6. Related Work

Test case generation from specifications has been developed as a prominent technique in testing of software systems. There is a large body of literature and several tools for generation of test cases from model-based specifications [10, 9]. In typical approaches, the selection of test cases uses partition analysis and Disjunctive Normal Form, and follows some particular coverage criterion, such as coverage of control states, edges, or an explicitly given set of test purposes [11, 22]. When the specification has data variables, constraint solving techniques can be used to find input values that drive the execution in a desired direction [18].

The problem of deriving tests from state-based models that distinguish between input and output actions has been widely investigated [10, 21]. According to the basic assumptions about the relationships between inputs and outputs being addressed, most of the approaches are based on input/output Finite State Machine (FSM) model, which are generally generated from formal specifications. However, this approach is not appropriate for generating test cases for connectors, since the FSM approach assumes that a pair of input and output constitutes an atomic action of a system, in other words, that the system cannot accept the next input before producing the output in reaction to the previous input. In Reo, such assumptions do not hold.

Fault-based testing was born in practice when testers started to assess the adequacy of their test cases by first injecting faults into their implementations, and then by observing if the test cases could detect these faults. This technique of mutating the source code became well-known as *mutation testing* and goes back to the late 1970s [12]. Since then it has found many applications and has become the major assessment technique in empirical studies on new test case selection techniques [25]. Recently, the first author has developed a general theory of fault-based testing, in particular mutation testing, by using Refinement Calculus [2] and Unifying Theories of Programming [3], which leads to the theoretical foundation of the work presented in this paper.

The relation between testing and refinement is not completely new. Hennessy and de Nicola [19] developed a testing theory that defines the equivalence and refinement of non-deterministic processes based on a notion of testing. Similarly, the failure-divergence refinement of CSP [13] is inspired by testing, since it is defined via the possible observations of a tester. Later, these theories led to the work on conformance testing based on labelled transition systems [23, 24] and test driver implementation [20]. However, these theories do not focus on the use of abstraction (the reverse of refinement) in order to select a subset of test cases. Furthermore, the existing testing theories focus on verification. This restricts their use either to the study of semantic issues (e.g., the observable equivalence of processes [19]), or to the testing of very abstract (finite) models for which exhaustive testing is feasible (like in protocol testing [24]).

A UTP semantics of data-flow model has been investigated in [14]. However, in data-flow model, only asynchronous communication is allowed and the channels can

buffer an arbitrary number of messages, which form a special kind of channel in Reo, i.e., the unbound FIFO channel. In other words, Reo is more expressive than data-flow model since Reo has more channel types and provides arbitrary combinations of synchrony and asynchrony, loose coupling, distribution and mobility, exogenous coordination by third parties and dynamic reconfigurability, and even allows an open set of user-defined channel types.

7. Conclusion

In this paper we define a semantic model of Reo connectors, and present a fault-based testing approach for test case generation from connector specifications. The idea is to model faults on the same level as the specification of connectors by using the notion of design. The timed data sequences being used in the semantic model make it possible to specify both finite and infinite behavior of connectors, and thus more expressive than previous semantic models using constraint automata or coalgebra. We generate test cases that can discover the faults in connectors. The algorithm for finding test cases is based on a general theory of refinement for pre- and post-condition specifications, and has been instantiated in a prototype tool for test case generation developed in Maude.

In future work we will investigate more complex case studies to gain more experience with the testing approach. We are also interested in generation of actual test cases, i.e., testing data selection based on constraint solving and its integration into the generated symbolic test cases. In addition, we are also planning to investigate specification-based testing of black-box connectors, where the specifications are given by either state machine models like constraint automata, or scenarios like UML sequence diagrams.

References

[1] The maude system. http://maude.cs.uiuc.edu/.

[2] B. K. Aichernig. Mutation testing in the refinement calculus. *Formal Aspects of Computing*, 15(2-3):280–295, 2003.

[3] B. K. Aichernig and H. Jifeng. Mutation Testing in UTP. *Formal Aspects of Computing*, 21(1-2):33–64, 2009.

[4] B. K. Aichernig and P. A. P. Salas. Test Case Generation by OCL Mutation and Constraint Solving. In *Proceedings of QSIC'05*, pages 64–71. IEEE Computer Society, 2005.

[5] F. Arbab. Reo: A Channel-based Coordination Model for Component Composition. *Mathematical Structures in Computer Science*, 14(3):329–366, 2004.

[6] F. Arbab and J. Rutten. A coinductive calculus of component connectors. In M. Wirsing, D. Pattinson, and R. Hennicker, editors, *WADT 2002, Revised Selected Papers*, volume 2755 of *LNCS*, pages 34–55. Springer-Verlag, 2003.

[7] C. Baier, M. Sirjani, F. Arbab, and J. Rutten. Modeling component connectors in Reo by constraint automata. *Science of Computer Programming*, 61:75–113, 2006.

[8] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. L. Talcott, editors. *All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic*, volume 4350 of *LNCS*. Springer, 2007.

[9] J. Dick and A. Faivre. Automating the generation and sequencing of test cases from model-based specifications. In J. C. P. W. P. G. Larsen, editor, *FME '93*, volume 670 of *LNCS*, pages 268–284. Springer, 1993.

[10] L. du Bousquet and N. Zuanon. An Overview of Lutess: A Specification-based Tool for Testing Synchronous Software. In *Proceedings of ASE'99*, pages 208–215. IEEE Computer Society, 1999.

[11] J.-C. Fernandez, C. Jard, T. Jéron, and C. Viho. An Experiment in Automatic Generation of Test Suites for Protocols with Verification Technology. *Science of Computer Programming*, 29(1-2):123–146, 1997.

[12] R. G. Hamlet. Testing Programs with the Aid of a Compiler. *IEEE Transactions on Software Engineering*, 3(4):279–290, 1977.

[13] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall International, 1985.

[14] C. A. R. Hoare and J. He. *Unifying Theories of Programming*. Prentice Hall International, 1998.

[15] M. P. Papazoglou and D. Georgakopoulos. Service Oriented Computing. *Comm. ACM*, 46(10):25–28, 2003.

[16] N. Martí-Oliet and J. Meseguer. Rewriting logic as a logical and semantic framework. In J. Meseguer, editor, *ENTCS*, volume 4. Elsevier Science Publishers, 2000.

[17] S. Meng and F. Arbab. Connectors as designs. Submitted.

[18] C. Meudec. Atgen: automatic test data generation using constraint logic programming and symbolic execution. *Software Testing, Verification and Reliability*, 11(2):81–96, 2001.

[19] R. D. Nicola and M. Hennessy. Testing Equivalences for Processes. *Theoretical Computer Science*, 34:83–133, 1984.

[20] J. Peleska and M. Siegel. From Testing Theory to Test Driver Implementation. In M.-C. Gaudel and J. Woodcock, editors, *FME'96, Proceedings*, volume 1051 of *LNCS*, pages 538–556. Springer, 1996.

[21] A. Petrenko. Fault Model-Driven Test Derivation from Finite State Models: Annotated Bibliography. In F. Cassez, C. Jard, B. Rozoy, and M. D. Ryan, editors, *Modeling and Verification of Parallel Processes*, volume 2067 of *LNCS*, pages 196–205. Springer, 2000.

[22] V. Rusu, L. du Bousquet, and T. Jéron. An Approach to Symbolic Test Generation. In W. Grieskamp, T. Santen, and B. Stoddart, editors, *IFM 2000, Proceedings*, volume 1945 of *LNCS*, pages 338–357. Springer, 2000.

[23] J. Tretmans. Test generation with inputs, outputs and repetitive quiescence. *Software – Concepts and Tools*, 17(3):103–120, 1996.

[24] J. Tretmans. Testing Concurrent Systems: A Formal Approach. In J. C. M. Baeten and S. Mauw, editors, *CONCUR '99, Proceedings*, volume 1664 of *LNCS*, pages 46–65. Springer, 1999.

[25] W. E. Wong, editor. *Mutation Testing for the New Century (Advances in Database Systems)*. Springer, 2001.