Input-output Conformance Testing for Channel-based Service Connectors

Natallia Kokash Farhad Arbab Behnaz Changizi
CWI, P.O. Box 94079, 1090 GB Amsterdam, The Netherlands
firstName.lastName@cwi.nl

Leonid Makhnist
Brest State Technical University, Department of Higher Mathematics, Moskovskaya 267, 224017 Brest, Republic of Belarus

Service-based systems are software systems composed of autonomous components or services provided by different vendors, deployed on remote machines and accessible through the web. One of the challenges of modern software engineering is to ensure that such a system behaves as intended by its designer. The Reo coordination language is an extensible notation for formal modeling and execution of service compositions. Services that have no prior knowledge about each other communicate through advanced channel connectors which guarantee that each participant, service or client, receives the right data at the right time. Each channel is a binary relation that imposes synchronization and data constraints on input and output messages. Furthermore, channels are composed together to realize arbitrarily complex behavioral protocols. During this process, a designer may introduce errors into the connector model or the code for their execution, and thus affect the behavior of a composed service. In this paper, we present an approach for model-based testing of coordination protocols designed in Reo. Our approach is based on the input-output conformance (ioco) testing theory and exploits the mapping of automata-based semantic models for Reo to equivalent process algebra specifications.

1 Introduction

Business process modeling is part of software development lifecycle which is primarily concerned with capturing the behavior of organizational business processes in a form that simplifies their analysis, fostering communication with various process stakeholders and helping to identify the requirements for the development of supporting software. Typically models are written using some (preferably standard) language or notation such as BPMN or UML diagrams. Once a process model has been constructed, it can be analyzed to uncover logical flaws in a process or optimize its functional or non-functional characteristics [26, 25].

While popular high-level modeling notations like BPMN or UML are suitable for fast prototyping and capturing system requirements, they are rather ambiguous and imprecise to be used for rigorous process analysis. Modeling languages should operate on the level of abstraction that allows designers to focus on the essence of the problem without being lost in technical details and at the same time provide sufficient precision and expressiveness to avoid ambiguities in the model or failure to describe certain important concepts. Multiple efforts on creating such modeling languages resulted into formalisms such as Petri nets and various process algebra-based languages often empowered with graphical syntax to simplify the process of unambiguous system description. These models are more difficult to use compared to high-level notations. However, their handicap of usability is compensated by automated validation and verification tools that provide powerful support for process analysis and quality assurance. More-

L. Aceto and M.R. Mousavi (Eds.): PACO 2011
EPTCS 60, 2011, pp. 19–35, doi:10.4204/EPTCS.60.2

over, various model-based transformation tools have been developed for major notations to assist process designers with converting high-level process models into more rigorous ones.

Reo [3] is an extensible model for coordination of software components or services wherein complex connectors are constructed out of simple primitives called channels. A channel is a binary relation that defines synchronization and data constraints on its input and output parameters. By composing basic channels, arbitrarily complex interaction protocols can be realized. Previous work shows that most of the behavioral patterns expressible in BPMN or UML notations can be modeled with Reo [6]. We have also developed a set of tools for automated conversion of such models to Reo[1]. Each Reo channel has a graphical representation and associated semantics. The most basic semantic model that currently exists for Reo relies on constraint automata [9]. Action constraint automata [22] constitute a model that generalizes constraint automata by allowing more detailed observations on connector ports. When channels with timed, context-sensitive and probabilistic behavior are used to design a connector, more expressive models to represent the semantics of the connector are required [4, 7, 10].

When using just a minimal set of channel types, it may happen that a substantial number of channels are required to construct a circuit with certain behavior. In general, it is not a trivial task to create a connector that implements a certain behavioral protocol. As any laborious process, connector implementation is error-prone and requires validation of the connector's behavior. There are several tools that can help to detect possible errors in Reo connectors. One of them is the animation engine [5]. This tool shows flash animated simulation of designed connectors and enables quick validation of connector designs. However, for complex connectors the number of possible traces is large and they are hard to analyze manually. Moreover, the current implementation of the animation engine is based on coloring semantics and cannot be used for reliable validation of data-dependent connectors. A more efficient analysis of Reo models can be performed with the help of simulation and model-checking tools, both specifically developed for Reo [8, 11, 20] and external [24, 21]. For example, simulation tools *lpsxsim* and *ocis* from `mCRL2` [2] and CADP [19] toolsets can be used to visualize execution traces of data-aware Reo networks followed by a user. Model checking tools *pbes2bool* and *evaluator* can be used to check the validity of connector properties expressed in the modal μ-calculus formulae.

Both kinds of tools require substantial effort from the designer to analyze simulation traces or correctly express complex properties using the intricate μ-calculus syntax. Yet another limitation of the aforementioned tools is their inability to analyze actual coordination code or protocol implementations. For example, in the context of the EU FP7 COMPAS project[2] we used Reo to design business process fragments and verify their conformance to various requirements extracted from compliance documents [28]. These fragments are further implemented in BPEL and stored in a repository to enable their on-demand retrieval and reuse in service-based systems. While we can verify the correctness of Reo models in this scenario, we cannot judge the correctness of fragment implementations.

In this paper, we extend our previous work on verification of Reo with model-based testing facilities to automatically derive tests from connector specifications and execute them to test service coordination code or protocol implementations. We enable testing of connector designs given specifications of their expected behavior in the form of constraint automata extended with inputs and outputs. Test generation is based on the *ioco*-testing theory which uses labelled transition systems (LTS) to represent system specifications, implementations and tests and defines a formal implementation relation called *ioco* to show conformance between implementations and specifications. The encoding of automata-based behavioral semantics for Reo in process algebra `mCRL2` is exploited to obtain LTS models suitable for testing Reo.

[1] http://reo.project.cwi.nl/cgi-bin/trac.cgi/reo/wiki/Converters
[2] http://www.compas-ict.eu/

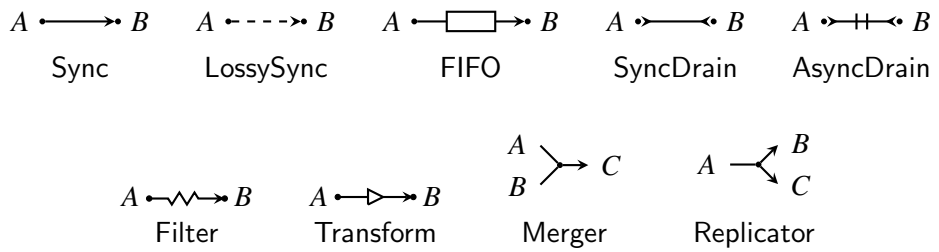


Figure 1: Graphical representation of basic Reo channels and nodes

Together with previously developed tools for converting specifications in high-level process modeling notations such as BPMN and UML to Reo, graphical Reo networks can be used as a formal specification of business process models. In this case, Reo connectors are seen as formal specifications of processes and used to automatically derive tests to check the quality of process implementations. Since the *ioco*-testing theory can be used to generate tests given specifications in any language with the LTS-based formal semantics, we can apply it to derive tests for any systems specified in Reo.

The remainder of this paper is organized as follows. In Section 2, we explain the basics of Reo. In Section 3, we briefly summarize the basics of input-output conformance (*ioco*) testing theory. In Section 4, we explain how this theory can be used to test Reo. In Section 5, we illustrate the use of model-based testing tools to analyze Reo connectors. Finally, in Section 6, we conclude the paper and outline our future work.

2 The Reo Coordination Language

Reo is a coordination language in which components and services are coordinated exogenously by channel-based connectors [3]. Connectors are essentially graphs where the edges are user-defined communication channels and the nodes implement a fixed routing policy. Channels in Reo are entities that have exactly two ends, also referred to as ports, which can be either source or sink ends. Source ends accept data into, and sink ends dispense data out of their respective channels. Although channels can be defined by users, a set of basic Reo channels (see Figure 1) with predefined behavior suffices to implement rather complex coordination protocols. Among these channels are (i) the Sync channel, which is a directed channel that accepts a data item through its source end if it can instantly dispense it through its sink end; (ii) the LossySync channel, which always accepts a data item through its source end and tries to instantly dispense it through its sink end. If this is not possible, the data item is lost; (iii) the SyncDrain channel, which is a channel with two source ends through which it accepts data simultaneously and loses them subsequently; (iv) the AsyncDrain channel, which accepts data items through only one of its two source channel ends at a time and loses them; and (v) the FIFO channel, which is an asynchronous channel with a buffer of capacity one. Additionally, there are channels for data manipulation. For instance, the Filter channel always accepts a data item at its source end and synchronously passes or loses it depending on whether or not the data item matches a certain predefined pattern or data constraint. Finally, the Transform channel applies a user-defined function to the data item received at its source end and synchronously yields the result at its sink end.

Channels can be joined together using nodes. A node can be a *source*, a *sink* or a *mixed* node, depending on whether all of its coinciding channel ends are source ends, sink ends or a combination of both. Source and sink nodes together form the boundary nodes of a connector, allowing interaction

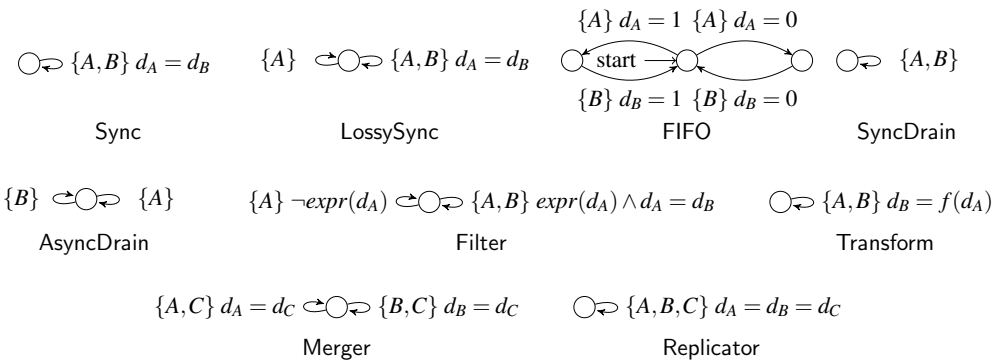


Figure 2: Constraint automata for basic Reo channels and nodes

with its environment. Source nodes act as synchronous replicators, and sink nodes as non-deterministic mergers. A mixed node combines these two behaviors by atomically consuming a data item from one of its sink ends at the time and replicating it to all of its source ends.

2.1 Automata-based Semantics for Reo

The most basic model expressing formally the semantics of Reo is constraint automata [9]. Transitions in a constraint automaton are labeled with sets of ports that fire synchronously, as well as with data constraints on these ports. The constraint automata-based semantics for Reo is compositional, meaning that the behavior of a complex Reo circuit can be obtained from the semantics of its constituent parts using the product operator. Furthermore, the hiding operator can be used to abstract from unnecessary details such as dataflow on the internal ports of a connector.

Definition 2.1 (Constraint automaton (CA)) *A constraint automaton $\mathscr{A} = (S, \mathscr{N}, \rightarrow, s_0)$ consists of a set of states S, a set of port names $\mathscr{N}$, a transition relation $\rightarrow \subseteq S \times 2^{\mathscr{N}} \times DC \times S$, where DC is the set of data constraints over a finite data domain Data, and an initial state $s_0 \in S$.*

We write $q \xrightarrow{N,g} p$ instead of $(q,N,g,p) \in \rightarrow$. Figure 2 shows the constraint automata for the basic Reo channels. Note that we use the set $Data = \{0,1\}$ as data domain for the FIFO channel. The behavior of any Reo circuit composed from these channels can be obtained by computing the product of the corresponding automata.

Constraint automata in their basic form do not express all the information about Reo node communication and fail to represent the behavior of e.g. context-dependent channels. An elemental example of such channels is a LossySync channel that loses a data item only if the environment or subsequent channels are not ready to consume it, i.e., it needs the information about the states of other channels or services to decide locally what to do with its data input. To address this problem, several other semantic models for Reo were introduced.

In intentional automata [16] we distinguish two sets of ports in their transition labels, a request set and a firing set. The request set models the context, i.e., the readiness of the channel ports to accept/dispense data, while the firing set models the actual flow of data through the circuit ports. Accounting for the requests that have arrived but have not been fired yet introduces additional states in the model. Due to this fact, intentional automata rapidly become large and difficult to manipulate.

Connector coloring [14] describes the behavior of Reo in a compositional fashion by coloring the parts of the circuit using different colors that match on connected ports. The basic idea in this model

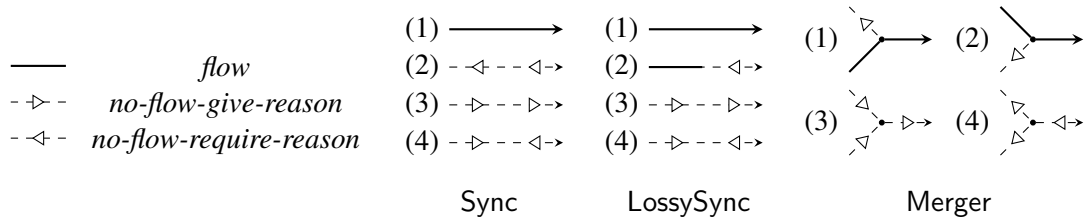


Figure 3: Examples of coloring semantics for Reo channels and nodes

is to associate *flow* and *no-flow* colors to channel ends. When three colors are used, the model captures context-dependent behavior by propagating negative information about the exclusion of dataflow through the connector. This model is used currently as a theoretical basis for Reo circuit animation and simulation tools. Figure 3 shows examples of coloring semantics for basic Reo channels and connectors.

In action constraint automata (ACA) [22], we distinguish several kinds of actions triggered on channel ports to signal the state changes of the channel. Formally, ACA can be defined as follows:

Definition 2.2 (Action constraint automaton (ACA)) *An action constraint automaton* $\mathscr{A} = (S, \mathscr{N}, \rightarrow, s_0)$ *consists of a set of states S, a set of action names* $\mathscr{N}$ *derived from a set of port names* $\mathscr{M}$ *and a set of admissible action types* $\mathscr{T}$*, a transition relation* $\rightarrow \subseteq S \times 2^{\mathscr{N}} \times DC \times S$*, where DC is the set of data constraints over a finite data domain Data, and an initial state* $s_0 \in S$*.*

We introduce an injective function $act : \mathscr{M} \times \mathscr{T} \rightarrow \mathscr{N}$ to define action names for each pair of a port name and an action type observed on the port. For example, the function $act(m, \alpha) = \alpha \cdot m$, for $m \in \mathscr{M}, \alpha \in \mathscr{T}$, where '$\cdot$' is a standard concatenation operator, can be used to obtain a set of unique action names given sets of distinctive Reo port names and types of observable actions. This model can be used, e.g., to represent a sequential data flow within a synchronous region and account for time delays in synchronous channels by distinguishing port blocking and unblocking events as well as the start and the end of data transfer through a port. Coloring semantics can also be represented in a form of ACA using three actions to convey the possibility of data *flow* as well as *requiring* and *giving reasons* for *no-flow*.

2.2 Process algebra-based Semantics for Reo

In our recent work [24], we represented the aforementioned semantic models for Reo using the process algebra `mCRL2`. This allowed us to apply a set of verification tools developed for this specification language to analyze Reo connectors.

The basic notion in `mCRL2` is the action. Actions represent atomic events and can be parameterized with data. Actions in `mCRL2` can be synchronized. In this case, we speak of multiactions which are constructed from other actions or multiactions using the so-called synchronization operator $|$, such as the multiaction $a|b|c$ of simultaneously performing the actions a, b and c. The synchronization operator is commutative, i.e., multiactions $a|b$ and $b|a$ are equivalent. The special action τ (tau) is used to refer to an internal, unobservable action. Processes are defined by process expressions, which are compositions of actions and multiactions using a number of operators. Among the basic operators are the following: (i) *deadlock* or *inaction* δ, which does not display any behavior; (ii) *alternative composition*, written as $p + q$, which represents a non-deterministic choice between the processes p and q; (iii) *sequential composition*, written $p \cdot q$, which means that q is executed after p, assuming that p terminates; (iv) the *conditional operator* or the *if-then-else* construct, written as $c \rightarrow p \diamond q$, where c is a data expression that evaluates to true or false; (v) *summation* $\Sigma_{d:D}\ p$ where p is a process expression in which the data

Table 1: mCRL2 encoding for channels and nodes: CA semantics

$$\begin{aligned}
\mathsf{Sync} &= \Sigma_{d:Data}\, A(d)|B(d)\cdot \mathsf{Sync}\\
\mathsf{LossySync} &= \Sigma_{d:Data}\, (A(d)|B(d)+A(d))\cdot \mathsf{LossySync}\\
\mathsf{SyncDrain} &= \Sigma_{d_1,d_2:Data}\, A(d_1)|B(d_2)\cdot \mathsf{SyncDrain}\\
\mathsf{AsyncDrain} &= \Sigma_{d:Data}\, (A(d)+B(d))\cdot \mathsf{AsyncDrain}\\
\mathsf{FIFO}(f:DataFIFO) &= \Sigma_{d:Data}\\
&\quad (isEmpty(f)\rightarrow A(d)\cdot \mathsf{FIFO}(full(d))\diamond B(e(f))\cdot \mathsf{FIFO}(empty))\\
\mathsf{Filter} &= \Sigma_{d:Data}\, (expr(d)\rightarrow A(d)|B(d)\diamond A(d))\cdot \mathsf{Filter}\\
\mathsf{Transform} &= \Sigma_{d:Data}\, A(d)|B(f(d))\cdot \mathsf{Transform}
\end{aligned}$$

$$\begin{aligned}
\mathsf{Merger} &= \Sigma_{d:Data}\, (A(d)|C(d)+B(d)|C(d))\cdot \mathsf{Merger}\\
\mathsf{Replicator} &= \Sigma_{d:Data}\, A(d)|B(d)|C(d)\cdot \mathsf{Replicator}\\
\mathsf{Router} &= \Sigma_{d:Data}\, (A(d)|B(d)+A(d)|C(d))\cdot \mathsf{Router}
\end{aligned}$$

variable d may occur, used to quantify over a data domain D; (vi) *parallel composition* or *merge* $p \parallel q$, which interleaves and synchronizes the multiactions of p with those of q, where synchronization is governed by a communication function (see below); (vii) *allow* $\nabla_V(p)$, where only actions in p from the set V are allowed to occur; (viii) the *encapsulation* $\partial_H(p)$, where H is a set of action names that are not allowed to occur; (ix) the *renaming operator* $\rho_R(p)$, where R is a set of renamings of the form $a \rightarrow b$, meaning that every occurrence of the action a in p is replaced by the action b; (x) the *communication operator* $\Gamma_C(p)$, where C is a set of communications of the form $a_0|...|a_n \mapsto c$, which means that every group of actions $a_0|...|a_n$ within a multiaction is replaced by the action c; (xi) *hiding* $\tau_I(p)$, which renames all actions in I of p into τ. It is possible to define recursive processes in mCRL2. However, allow, encapsulation, hiding and communication operators can not be used within recursive processes. Structured operational semantics for the aforementioned mCRL2 operators can be found in [2].

The mCRL2 language provides a number of built-in datatypes (e.g., boolean, natural, integer) with a set of usual arithmetic operations. Moreover, an arbitrary structured type in mCRL2 can be declared by a construct of the form

$$\textbf{sort}\, S = \textbf{struct}\; c_1(p_1^1{:}S_1^1,\ \ldots,\ p_1^{k1}{:}S_1^{k1})?r_1 \;\mid\; \ldots \;\mid\; c_n(p_n^1{:}S_n^1,\ \ldots,\ p_n^{kn}{:}S_n^{kn})?r_n;$$

This construct defines the type S together with constructors $c_i\colon S_i^1 \times \ldots \times S_i^{ki} \rightarrow S$, projections $p_i^j\colon S \rightarrow S_i^j$, and type recognition functions $r_i\colon S \rightarrow Bool$.

The mCRL2 toolset allows users to verify software models specified in the mCRL2 language. It includes a tool for converting mCRL2 specifications into linear form (a compact symbolic representation of the corresponding LTS), a tool for generating explicit LTSs from linear process specifications (LPS), tools for optimizing and visualizing these LTSs, and many other useful facilities. A detailed overview of the available software can be found at the mCRL2 web site[3].

The presence of multiactions in mCRL2 makes it possible to compositionally map Reo to process specifications and compose a connector by synchronizing actions on joint ports. Thus, mCRL2 models for Reo circuits are generated in the following way: observable events, i.e., data flow on the channel ends, are represented as atomic actions, while data items observed at these ports are modeled as parameters of these actions. Analogously, we introduce a process for every node and actions for all channel ends meeting at the node. The encodings for the basic Reo channels and nodes are listed in Table 1.

[3] http://www.mcrl2.org/

Given process definitions for all channels and nodes, a composite process that models a complete Reo connector is built by forming a parallel composition of these processes and synchronizing actions for coinciding node/channel ends. Node/channel end synchronization is enforced using the mCRL2 operators communication and encapsulation. For example, an mCRL2 process for the replicator circuit in Figure 1 can be formed from three synchronous channels

$$\mathsf{Sync1} = A|X_1 \,.\, \mathsf{Sync1}\,, \quad \mathsf{Sync2} = Y_1|B \,.\, \mathsf{Sync2}\,, \quad \mathsf{Sync3} = Z_1|C \,.\, \mathsf{Sync3}$$

and a replicator node

$$\mathsf{ReplicatorNode} = X_2|Y_2|Z_2 \,.\, \mathsf{ReplicatorNode}$$

applying the communication and blocking operators to their parallel composition:

$$\begin{aligned}\mathsf{ReplicatorCircuit} = \partial_{\{X_1,Y_1,Z_1,X_2,Y_2,Z_2\}}\big(\Gamma_{\{X_1|X_2\to\tau,Y_1|Y_2\to\tau,Z_1|Z_2\to\tau\}}(\\ \mathsf{Sync1} \parallel \mathsf{Sync2} \parallel \mathsf{Sync3} \parallel \mathsf{ReplicatorNode}\big)\big);\end{aligned}$$

Here we assume that the sink end X_1 of the channel Sync1 is connected to the source end X_2 of the node ReplicatorNode, while sink ends Y_2 and Z_2 of the node are connected to source ends Y_1 and Z_1 of channels Sync2 and Sync3. Optionally, the mCRL2 hiding operator can be employed for abstracting the flow in selected nodes. For simplicity, we omitted the encoding of data parameters in this example.

For the treatment of data we assume, in the context of a given connector, a global datatype given as the custom sort *Data*. Given such a datatype, we can use the mCRL2 summation operator to define data dependencies imposed by channels. For the FIFO channel we additionally define the datatype

$$\textbf{sort}\ \mathit{DataFIFO} = \textbf{struct}\ \mathit{empty}?\mathit{isEmpty} \mid \mathit{full}(e{:}\mathit{Data})?\mathit{isFull}$$

which allows us to specify whether the buffer of the FIFO channel is empty or full, and if it is full, what value is stored in it. Additionally, we introduce a special kind of node, Join, which synchronizes all ends of incoming channels, forms a tuple of data items received and replicates it to the source ends of all outgoing channels. More details on data handling in Reo and mCRL2 can be found in [24].

Table 2 shows the mCRL2 encodings for the basic Reo channels and nodes according to the ACA model with four actions: *block* and *unblock* actions are used to establish port communication within a single transaction and release channel ports involved in such a transaction, respectively. The *start* and *finish* actions are used to represent the start and the end of dataflow through a blocked channel port. In our encoding, we use prefix letters b, u, s and f in front of Reo port names to denote block, unblock, start and finish actions observed on these ports. Since data support in the new translation is analogous to the case of the CA-based translation, we omit its discussion here and for simplicity show only the data-agnostic mapping. As in the CA approach, we construct nodes compositionally. Given process definitions for all channels and nodes, a composite process that models the complete Reo connector is built by forming a parallel composition of these processes and synchronizing communicating actions for the coinciding node/channel ends.

To incorporate the colorings in our encoding in mCRL2, we represent colors as data parameters of actions [24]. However, since the summation over a finite domain in mCRL2 is just an alternative choice of the same action with various parameters, we can represent every parameterized action as an alternative choice of several non-parameterized actions. This allows us to represent coloring semantics as shown in Table 3. For every port X, we consider three actions, wX, rX and gX which are abbreviations for actions *flow*, *no-flow-require-reason*, and *no-flow-give-reason* observations on channel ports. The advantage of this approach over the use of parameterized actions is the possibility to hide no-flow labels.

Thus, the process algebra mCRL2 provides a common ground for expressing most important semantic models for Reo preserving their compositionality.

Table 2: mCRL2 encoding for channels and nodes: ACA semantics

$$
\begin{aligned}
\mathsf{Sync} &= bA|bB \cdot sA|sB \cdot fA|fB \cdot uA|uB \cdot \mathsf{Sync} \\
\mathsf{LossySync} &= (bA|bB \cdot sA|sB \cdot fA|fB \cdot uA|uB + bA \cdot sA \cdot fA \cdot uA) \cdot \mathsf{LossySync} \\
\mathsf{SyncDrain} &= bA|bB \cdot (\\
&\quad sA \cdot (sB \cdot (fA \cdot fB + fB \cdot fA + fA|fB) + fA \cdot sB \cdot fB + sB|fA \cdot fB) + \\
&\quad sB \cdot (sA \cdot (fA \cdot fB + fB \cdot fA + fA|fB) + fB \cdot sA \cdot fA + sA|fB \cdot fA) + \\
&\quad sA|sB \cdot (fA \cdot fB + fB \cdot fA + fA|fB)) \cdot uA|uB \cdot \mathsf{SyncDrain} \\
\mathsf{AsyncDrain} &= (bA \cdot sA \cdot fA \cdot uA + bB \cdot sB \cdot fB \cdot uB) \cdot \mathsf{AsyncDrain} \\
\mathsf{FIFO}(f : \mathit{DataFIFO}) &= \mathit{isEmpty}(f) \rightarrow bA \cdot sA \cdot fA \cdot uA \cdot \mathsf{FIFO}(\mathit{full}) \diamond \\
&\qquad\qquad bB \cdot sB \cdot fB \cdot uB \cdot \mathsf{FIFO}(\mathit{empty})
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{Merger} &= (bA|bC \cdot sA|sC|fA|fC.uA|uC + \\
&\qquad bB|bC \cdot sB|sC|fB|fC \cdot uB|uC) \cdot \mathsf{Merger} \\
\mathsf{Replicator} &= bA|bB|bC \cdot sA|sB|sC \cdot fA|fB|fC \cdot uA|uB|uC \cdot \mathsf{Replicator}
\end{aligned}
$$

Table 3: mCRL2 encoding for channels and nodes: coloring semantics

$$
\begin{aligned}
\mathsf{Sync} &= (wA|wB + rA|gB + gA|rB + gA|gB) \cdot \mathsf{Sync} \\
\mathsf{LossySync} &= (wA|wB + wA|gB + gA|rB + gA|gB) \cdot \mathsf{LossySync} \\
\mathsf{SyncDrain} &= (wA|wB + rA|gB + gA|rB + gA|gB) \cdot \mathsf{SyncDrain} \\
\mathsf{AsyncDrain} &= (wA|gB + gA|wB + rA|wB + rB|wA + gA|gB) \cdot \mathsf{AsyncDrain} \\
\mathsf{FIFO}(f : \mathit{DataFIFO}) &= \mathit{isEmpty}(f) \rightarrow ((wA|rB + wA|gB) \cdot \mathsf{FIFO}(\mathit{full}) + \\
&\qquad (gA|rB + gA|gB) \cdot \mathsf{FIFO}(\mathit{empty})) \diamond \\
&\qquad ((rA|wB + gA|wB) \cdot \mathsf{FIFO}(\mathit{empty}) + \\
&\qquad (rA|gB + gA|gB) \cdot \mathsf{FIFO}(\mathit{full}))
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{Merger} &= wA|gB|wC + gA|wB|wC + rA|rB|gC + gA|gB|rC) \cdot \mathsf{Merger} \\
\mathsf{Replicator} &= (wA|wB|wC + rA|rB|gC + rA|gB|rC + gA|gB|gC) \cdot \mathsf{Replicator}
\end{aligned}
$$

3 Input-output Conformance Testing

In this section, we briefly introduce a model-based test generation theory for testing input-output conformance (*ioco*) of an implementation and a given specification [30]. Transition labels in (action) constraint automata represent sets of simultaneously observable actions on Reo ports with enabling guards while in the original definitions on LTS each transition refers to a single observable action. As follows from our mapping of constraint-automata-based semantics of Reo to LTS, each set of transition labels $\{A,B,C\}$ in a CA corresponds to a transition with a unique action label $A|B|C$ in the corresponding LTS, which further can be renamed to an action ABC. Assuming that the semantics of Reo is given in a form of such LTS, we can apply the *ioco* testing theory to test Reo. In the following we redefine all necessary concepts of the *ioco* testing theory using CA, the original definitions on LTS can be found in [30].

Let L^* be the set of all finite sequences over a set L and ε denote the empty sequence. Given finite sequences σ_1 and σ_2, we denote their concatenation $\sigma_1 \cdot \sigma_2$. If for some automaton there exists a trace $q \xrightarrow{N_1 \cdot \tau \cdot \tau \cdot N_2 \cdot \tau \cdot N_3 \cdot \tau} p$, where $N_1, N_2, N_3 \in L$ are sets of actions representing constraint automata labels and τ is a special action that refers to any set of unobservable constraint automata ports, we write $p \overset{N_1 \cdot N_2 \cdot N_3}{\Longrightarrow} q$ for the $\tau-$abstracted sequence of observable actions and say that p is able to perform the trace $N_1 \cdot N_2 \cdot N_3 \in L^*$. As we demonstrated in [23], every state s of a CA can be identified with a behaviorally equivalent

mCRL2 process p. We exploit this correspondence in the rest of the paper and do not distinguish between CA states and processes associated with these states. The following definitions are needed to formally define the *ioco* testing relation for a given specification and a system implementation.

Definition 3.1 *Let p be a process associated with the initial state s_0 of a constraint automaton $\mathscr{A} = (S, \mathscr{N}, \rightarrow, s_0)$ and $\sigma \in L^*$ where $L = 2^{\mathscr{N}} \times DC$ is a set of the constraint automaton labels.*

1. $init(p) = \{\rho \in L \cup \tau \mid p \xrightarrow{\rho}\}$.
2. $traces(p) = \{\sigma \in L^* \mid p \stackrel{\sigma}{\Longrightarrow}\}$
3. p **after** $\sigma = \{p' \mid p \stackrel{\sigma}{\Longrightarrow} p'\}$
4. P **after** $\sigma = \bigcup p$ **after** $\sigma \mid p \in P$, *where $P \subseteq S$ is a set of states.*
5. P **refuses** $A = \exists p \in P, \forall \rho \in A \cup \tau : p \stackrel{\rho}{\not\rightarrow}$, *where $P \subseteq S$ and $A \subseteq L$.*
6. $der(p) = \{p' \mid \exists \sigma \in L^* : p \stackrel{\sigma}{\Longrightarrow} p'\}$
7. *p has* finite behavior *if there is a natural number n such that all traces in $traces(p)$ have length smaller than n.*
8. *p is a* finite state *if the number of reachable states $der(p)$ is finite.*
9. *p is* deterministic *if, for all $\sigma \in L^*$, p **after** σ has at most one element.*
10. *p is* image finite *if, for all $\sigma \in L^*$, p **after** σ is finite.*
11. *p is* strongly convergent *if there is no state of p that can perform an infinite sequence of internal transitions.*
12. *$\mathscr{CA}(L)$ is the class of image finite and strongly convergent constraint automata with labels in L.*

Definition 3.2 (Constraint automaton with Inputs and Outputs) *A* constraint automaton with inputs and outputs *is a constraint automaton $\mathscr{A} = (S, \mathscr{N}, \rightarrow, s_0) \in \mathscr{CA}(L_I \cup L_U)$, where L_I and L_U, $L_I \cap L_U = \emptyset$ are countable sets of disjoint input and output labels.*

LTS with inputs and outputs are used as formal specifications for *ioco* testing theory. Being a variant of LTS, constraint automata with inputs and outputs are used in our work to represent system-under-test specifications. This does not mean that these specifications have to be written explicitly in a form of automata: it suffices that a specification language, e.g., Reo, had semantics expressed in the form of constraint automata with inputs and outputs.

Definition 3.3 (Input-Output Constraint Automaton) *An* input/output constraint automaton *(IOCA) is a constraint automaton with inputs and outputs $\mathscr{A} = (S, \mathscr{N}, \rightarrow, s_0)$ where all inputs are enabled in any reachable state, i.e., $\forall s \in der(s_0), \forall N \subseteq L_I : s \stackrel{N}{\Longrightarrow}$.*

Let $\mathscr{CA}(L_I, L_U)$ denote the class of all constraint automata with inputs in L_I and outputs in L_U. The class of input-output constraint automata with inputs in L_I and outputs in L_U is denoted by $\mathscr{IOCA}(L_I, L_U) \subseteq \mathscr{CA}(L_I, L_U)$. A constraint automaton with inputs and outputs can be converted to an input-output constraint automaton by adding a self-loop transition with labels from L_I to every reachable state. This operation is called *angelic completion* [30]. Input-output constraint automata are used to model systems in which inputs are initiated by the environment and never refused by the system and outputs are initiated by the system and never refused by the environment. The input enabledness of system implementations

is required in *ioco* testing theory to define the relation between the inputs generated by the tester and the observable outputs.

Since input-output constraint automata are just a particular type of constraint automata, all definitions for the latter apply, including the definitions of product and hiding operations. A state q of a process p without output actions, i.e., $\forall \rho \in L_U \,|\, q \stackrel{\rho}{\not\rightarrow}$, is called *suspended* or *quiescent* and is denoted $\delta(q)$. The external observer of a system in a quiescent state does not see any outputs. Such a situation with no observations can be considered as a special action, denoted as δ. In our test cases, we allow system transitions $p \stackrel{\delta}{\longrightarrow}$ meaning that p cannot perform any output actions. It is also possible to extend traces with δ, e.g., $p \stackrel{N_1 \cdot \delta \cdot N_2 \cdot N_3}{\Longrightarrow}$, where $N_1, N_2 \in L_I, N_3 \in L_U$, expresses the fact that after the input N_1 was observed, the system remained quiescent, while after the input N_2, the system produced output N_3. The *quiescent traces* of p are those that may lead to quiescent states, i.e.,

$$Qtraces(p) = \{\sigma \in L^* | \exists p' \in (p \,\mathbf{after}\, \sigma) : \delta(p')\}.$$

Traces that may contain the quiescence action are called *suspension traces*. More formally, the suspension traces are

$$Straces(p) = \{\sigma \in L_\delta^* \,|\, p_\delta \stackrel{\sigma}{\longrightarrow}\},$$

where $L_\delta = L \cup \delta$ and p_δ is a process defined by a constraint automaton $\mathscr{A} = (S, \mathscr{N}, \rightarrow, s_0)$ with inputs L_I, outputs $L_U \cup \delta$ and a transition relation $\rightarrow \cup \rightarrow_\delta$, such that $\rightarrow_\delta = \{s \stackrel{\delta}{\longrightarrow} s \,|\, s \in S, \delta(s)\}$.

To test a system using the *ioco* testing theory, we assume that a tester is an environment which is able to provide inputs and observe system outputs including quiescence. This environment must be able to accept any output produced by the system. Thus, the behavior of a tester can be modeled as IOCA with inputs and outputs exchanged. The occurrence of a special symbol $\theta \notin L_I \cup L_U \cup \tau \cup \delta$ in tests indicates the detection of quiescence. Practically this means that the tester has to wait for a certain time-out to conclude that the system did not produce an output. Since test case execution must always lead to a verdict, we include two special states reachable from any other state of a testing IOCA: **fail**, **pass** $\in S$. Thus, a test case is defined as follows in *ioco*:

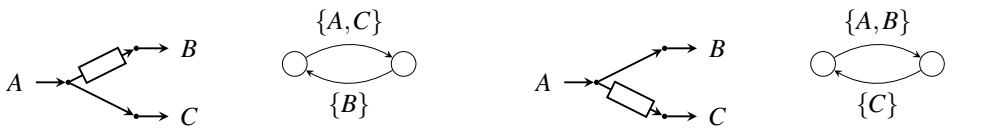
Definition 3.4 (Test case) *A* test case *t for an implementation with inputs in L_I and outputs in L_U is an IOCA $\mathscr{A} = (S, \mathscr{N}, \rightarrow, s_0) \in \mathscr{IOCA}(L_I, L_U \cup \theta)$ such that*

- *t is finite and deterministic;*
- *S contains two special states* **pass** *and* **fail**, **pass** $\neq$ **fail**;
- *t has no cycles except those in states* **pass** *and* **fail**;
- *$\forall s \in S$ it holds that $init(s) = a \cup L_U \,|\, a \in L_I$ or $init(s) = L_U \cup \theta$.*

The class of test cases for implementations with inputs in L_I and outputs in L_U is denoted $\mathscr{TTS}(L_U, L_I)$. A run of a test case $t \in \mathscr{TTS}(L_U, L_I)$ with an implementation under test $i \in \mathscr{IOCA}(L_I, L_U)$ corresponds to the parallel synchronization of behavior expressed by the tester and the system. However, the usual parallel synchronization needs to be extended to account for special labels δ and θ. Such an extension, denoted by $t \rceil | i$, is defined by the following inference rules:

$$\frac{i \stackrel{\tau}{\longrightarrow} i'}{t \rceil | i \stackrel{\tau}{\longrightarrow} t \rceil | i'} \quad \frac{t \stackrel{a}{\longrightarrow} t', \quad i \stackrel{a}{\longrightarrow} i'}{t \rceil | i \stackrel{a}{\longrightarrow} t \rceil | i'} \quad \frac{t \stackrel{\theta}{\longrightarrow} t', \quad i \stackrel{\delta}{\longrightarrow}}{t \rceil | i \stackrel{\theta}{\longrightarrow} t \rceil | i'}.$$

Here $a \in L_I \cup L_U$. The resulting system runs without deadlocks. This property follows immediately from the definition of test cases: since $\forall s \in S$ it holds that $init(s) = a \cup L_U \,|\, a \in L_I$ or $init(s) = L_U \cup \theta$, we



Reo model Constraint automaton Reo model Constraint automaton

(a) Specification (b) Implementation

Figure 4: Specifications of a Reo connector and its wrong implementation (Example 1)

can conclude that either an action a can always be performed on the implementation or i produces some output $x \in L_U \cup \theta$.

Definition 3.5 (*Ioco* relation) *Given a set of inputs L_I and a set of outputs L_U, the relation* **ioco** $\subseteq \mathscr{IOCA}(L_I, L_U) \times \mathscr{CA}(L_I, L_U)$ *is defined as follows:*

$$i \,\mathbf{ioco}\, s = \forall \sigma \in Straces(s) : out(i \,\mathbf{after}\, \sigma) \subseteq out(s \,\mathbf{after}\, \sigma)$$

where for any state s of a CA $out(s) = \{x \in L_U \mid s \xrightarrow{x}\} \cup \delta \mid \delta(s)$ and for a set of states S $out(S) = \cup\{out(s) \mid s \in S\}$

For more details about *ioco* testing theory, i.e., test generation algorithm and the analysis of its coverage, refer to [30]. The extension of *ioco* to test component-based systems is presented in [17]. Aichernig and Weiglhofer propose a unification of *ioco* relation by lifting the definition from LTS to reactive processes. In the rest of this paper, we discuss the application of the presented testing theory to detect errors in implementations of Reo coordination protocols. Given a Reo circuit specification, we use the *ioco*-based test generation algorithm to produce sets of inputs and judge the correctness of the implementations by observing its outputs. Inputs in our approach essentially represent sets of boundary ports of the circuit ready to accept data items while outputs are actual observations of dataflow on these ports.

4 Testing Channel-based Service Connectors

To enable testing of Reo connectors, we extend constraint automata with actions that represent input/output events. Figure 4 shows a Reo connector specification and an erroneous implementation where Sync and FIFO channels are swapped. Figure 5 shows another sample specification and a wrong implementation where the SyncDrain channel is erroneously added to the circuit. The goal of testing is to detect such errors automatically by providing inputs and observing outputs obtained from a wrong implementation which do not occur in the specification. Note that we use Reo to model both a specification and an erroneous implementation for illustration purposes only. In practice these errors may correspond to wrong implementation code such as e.g., wrong type of communication (synchronous vs. asynchronous) in the first example or wrongly enforced synchronization on two ports in the second example.

To obtain connector specifications suitable for testing, we combine the idea of explicit representation of pending requests introduced in intentional automata with constraint automata semantics for Reo. Thus, for every boundary Reo port A we introduce two actions $?A$ and $!A$ that represent an external request for this port to accept or dispense a data item and the actual observation of data flow on the circuit node A, respectively. Thus, our representation of boundary nodes in `mCRL2` will be as follows:

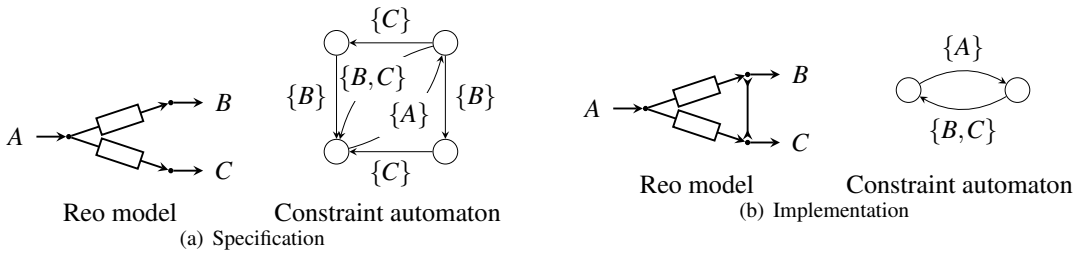


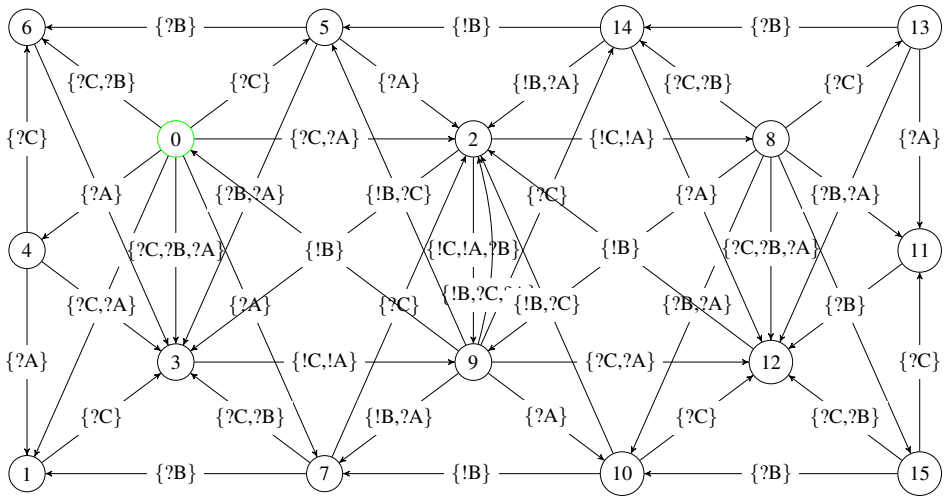
Figure 5: Specifications of a Reo connector and its wrong implementation (Example 2)

$$\begin{aligned}\mathsf{Merger} &= ?C \cdot (A|!C + B|!C) \cdot \mathsf{Merger};\\ \mathsf{Replicator} &= ?A \cdot !A|B|C \cdot \mathsf{Replicator};\\ \mathsf{Router} &= ?A \cdot (!A|B + !A|C) \cdot \mathsf{Router};\end{aligned}$$

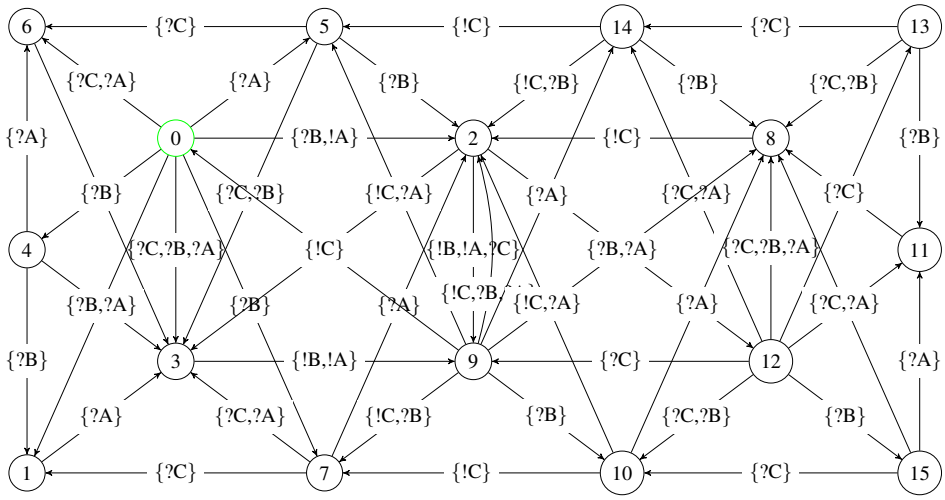
Here we assume that the merger node has two internal input ports A and B and a boundary output port C while the replicator and the router nodes have one input boundary port A and two internal output ports B and C. It is not allowed in Reo to have a boundary node which serves both as input and output port. Taking into account that we label input and output events on the same port using different action names (decorated with ? and !, respectively), we can conclude that for a Reo circuit with all disjoint port names the requirement $L_I \cap L_U = \emptyset$ holds. Figure 6 shows constraint automata with inputs and outputs for the specification and implementation of Reo connectors in Example 1.

Aichernig et al. [1] developed a tool for testing Reo based on the representation of connectors as designs and specifying them in Maude. The authors claim that testing theories based on finite-state machines are not suitable for testing Reo since in Reo not all input events are followed by output events. While this is true assuming that Reo circuit specifications are provided in the form of basic constraint automata, observe that with our mapping schema we can distinguish a situation when some input item is rejected by a circuit from the case when this item is accepted by the circuit but does not appear on any of the output ports, e.g., destroyed by a SyncDrain or LossySync channels. In fact, any data item supplied by an environment that enters a circuit through an input boundary port A generates an output event $!A$. Similarly, any output event $!B$ observed on the boundary output port B can only follow the preceding input event $?B$ triggered by the environment. Furthermore, in contrast to earlier approaches based on input/output finite state machines [12, 27], the *ioco* testing theory allows us to "observe" outputs with no data flow on Reo ports (quiescence). We now illustrate why such an extended semantic model is needed to test Reo. In Example 2, the behavior of the circuit in the specification is more general than the behavior of the implemented circuit: for any data input through the input boundary port A in the specification, data flow on the port B, port C or both of them simultaneously will be eventually observed. In contrast, in the implementation data flow on ports B and C will be always observed simultaneously. If we generate test cases based on constraint automata, we always observe outputs that are a subset of the admissible outputs in the specification. However, if we explicitly take into account requests from the environment to supply/consume data, we can detect the difference in the circuit implementation. Thus, after observing the input events $?A$ and $?B$ and the output event $!A$, the specification will expect the observation of the action $!B$ while the presented wrong implementation will be quiescent.

Many existing semantic models for Reo operate at the level of observable data flow on Reo ports and do not specify what happens with possibly multiple requests arriving at the boundary nodes. There are several strategies to handle these requests: for every port A with a pending request $?A$ on the arrival of another request $?A$ we can (a) ignore the second request, (b) substitute the initial request with the new

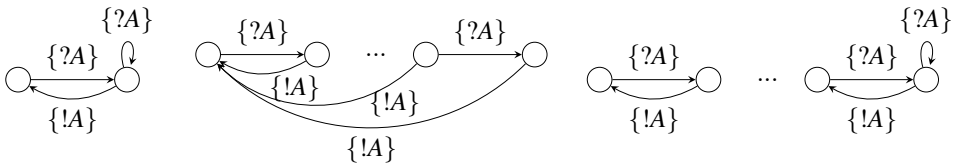


(a) Specification



(b) Implementation

Figure 6: Example 1: Constraint automaton with inputs and outputs for the specification of a Reo connector and its wrong implementation



(a) Ignore (b) Overwrite (c) Add

Figure 7: Input request handling

request, (c) add the second request to the waiting line to be processed by the circuit, e.g., on the FIFO basis. Figure 7 shows constraint automata with inputs and outputs-based specifications for the aforementioned strategies. Note that it makes sense to distinguish between the first and the second strategies only for data-aware requests. For data-agnostic circuits it matters only how many requests the circuit needs to process. In the third case, we have to assume that the queue for pending requests is bounded in order to keep the model finite, and after its limit is reached, the further requests are either ignored or overwrite previous ones. What is important is that in all three cases we can see that Reo connector specifications can be represented by constraint automata with inputs and outputs that are input enabled. Based on this observation, we can apply angelic completion for constraint automata with inputs and outputs generated from Reo circuits as discussed above to obtain an input-output constraint automaton without affecting the actual behaviour of the circuit: for any input request $?A$ a subsequent request can influence the behavior of the circuit only after the first request is processed, i.e., action $!A$ is observed, and, thus, adding loops with labels from L_I to each state does not change the semantics of the circuit.

An interesting result follows from the precongruence property for input enabled specifications [30] (see Proposition 3) and the fact that our generated constraint automata-based specifications are input enabled.

Proposition 4.1 *For any two pairs of connector implementations and specifications, $i_k \in \mathscr{IOCA}(L_{I_k}, L_{U_k})$ and $s_k \in \mathscr{IOCA}(L_{I_k}, L_{U_k})$, $k = 1,2$ with disjoint sets of input/output labels, i.e., $L_{I_1} \cap L_{I_2} = L_{U_1} \cap L_{U_2} = \emptyset$, it holds that*

$$i_1 \,\mathbf{ioco}\, s_1 \text{ and } i_2 \,\mathbf{ioco}\, s_2 \text{ implies } \partial_H(\Gamma_{H\to\{\tau\}}(i_1||i_2)) \,\mathbf{ioco}\, \partial_H(\Gamma_{H\to\{\tau\}}(s_1||s_2)),$$

where $H = (L_{I_1} \cap L_{I_2}) \cup (L_{U_1} \cap L_{U_2})$ denotes the set of observed actions on their connected ports while $\partial_H(\cdot)$ and $\Gamma_C(\cdot)$ are the `mCRL2` encapsulation and communication operators introduced in Section 2.2.

Practically this means that the product operator on input-output constraint automata preserves the *ioco* relation and testing of Reo connectors can be performed compositionally.

5 Tool Support

To automate testing of Reo, we integrated the JTorX tool into the ECT environment. JTorX is a Java-based tool to test whether the *ioco* relation holds between a given specification and a given implementation. JTorX expects the specification to be given in a form of an LTS represented, e.g., in Aldebaran (.aut) or GraphML format. Thus, we employ our Reo to `mCRL2` conversion framework to generate LTSs that are behaviorally equivalent to constraint automata with inputs and outputs introduced in Section 3. A detailed description of Reo to `mCRL2` mapping plug-in is available in [24]. To include input/output actions into an `mCRL2` specification generated from the graphical Reo circuit, select the *I/O actions* check box on the mapping parameters panel. This option can be chosen in combination with coloring and ACA-based mappings. The corresponding `mCRL2` code will appear in the integrated text editor. An LTS with input and output events can be obtained from the generated `mCRL2` code by pressing the *Show LTS* button and saved in the .aut format afterwards. The JTorX tool does not recognize synchronized input and output actions in the form of `mCRL2` multiactions. Therefore, we additionally developed a simple script that converts labels of the form $iA|iB$ and $oA|oB$ into $\{?A, ?B\}$ and $\{!A, !B\}$, respectively. Similarly to `mCRL2`, all actions represented by a set of labels on a single transition in the LTS operated on by JTorX must happen simultaneously, and thus our transformation does not affect the outcome of testing.

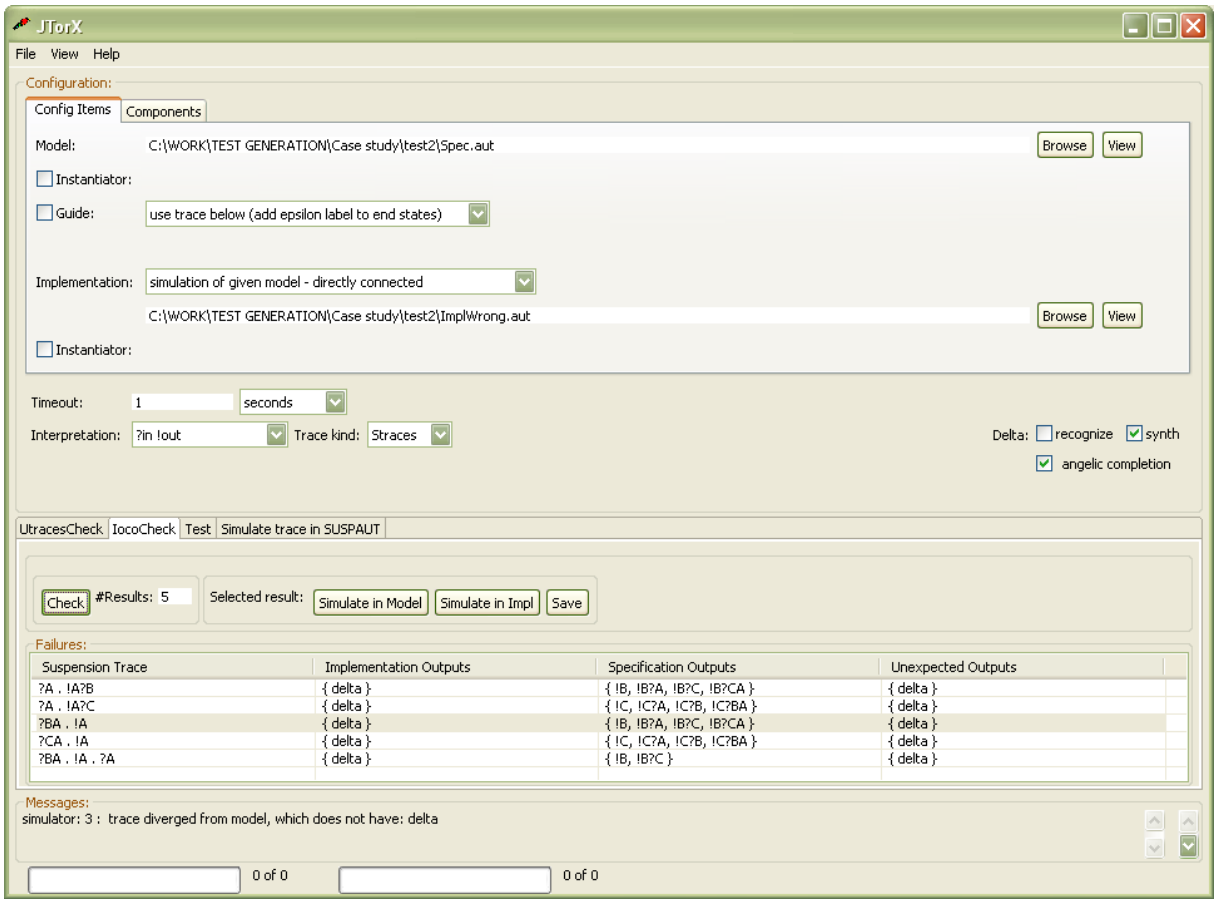


Figure 8: Testing Reo with JTorX: generated test cases for Example 2

The implementation is either given in a form of LTS or it is a real program. In the latter case, JTorX needs to be able to interact with it, e.g., via the TCP protocol or via an adapter. For testing connector implementations against constraint automata specifications, we can supply both the specification and the implementation in the form of LTS representing their input/output constraint automata semantics. Similarly, for testing implementations of business protocols modeled with Reo, we can obtain LTSs by converting execution code, i.e., BPEL, to Reo [29], and then to `mCRL2`, and, finally, to LTS as described above. However, as this approach requires each translation step to preserve the semantics of the original code, which is not always feasible, a more natural approach would be to develop adapters that execute tests generated by JTorX and observe outputs produced by the real system under test. There is an ongoing work on developing such an adapter for JTorX to communicate with the distributed implementation of Reo in Java [15].

Figure 8 shows a screenshot of the JTorX tool with tests generated for Example 2. The highlighted line shows a test case discussed in Section 4 on which the wrong implementation fails to yield the expected outputs and remains quiescent. Using JTorX, one can simulate test case execution to show traces corresponding to the violated test cases on both specification and implementation LTSs. In our future work, we will develop a plug-in to simulate such test violation traces using Reo animation engine [5].

6 Conclusions

In this paper, we presented an approach to testing models in the Reo coordination language using the *ioco* testing theory. The approach is based on mapping of automata-based semantic models for Reo to the process algebra mCRL2 and reuse of existing state-space generation and model-based testing tools. We extended the semantic model for Reo with input/output events and showed that the generated specifications are suitable for testing. In contrast to the previous work on testing Reo [1], where basic connectors are specified equationally and their composition is encoded by means of rewrite rules, no additional effort is required to obtain testable specifications and implementations in our framework. We also expect compositionality of testing Reo with *ioco* to be a useful property that will allow us to assure quality of large process models.

In our future work, we will investigate the applicability of several extensions of *ioco* relation, namely, *symbolic ioco* (*sioco*) [18] and *timed-ioco* (*tioco*) [13], to test time and data-aware Reo circuits.

References

[1] B. K. Aichernig, F. Arbab, L. Astefanoaei, F. S. de Boer, M. Sun & J. Rutten (2009): *Fault-Based Test Case Generation for Component Connectors*. In: *Proc. TASE 2009*, pp. 147–154, doi:10.1109/TASE.2009.14.

[2] J.F. Groote et al. (2007): *The Formal Specification Language mCRL2*. In E. Brinksma et al., editor: *Methods for Modelling Software Systems*, IBFI, Schloss Dagstuhl, pp. 1–34.

[3] F. Arbab (2004): *Reo: A Channel-based Coordination Model for Component Composition*. *Mathematical Structures in Computer Science* 14, pp. 329–366, doi:10.1017/S0960129504004153.

[4] F. Arbab, C. Baier, F. de Boer & J. Rutten (2007): *Models and Temporal Logical Specifications for Timed Component Connectors*. *Software and Systems Modeling* 6, pp. 59–82, doi:10.1007/s10270-006-0009-9.

[5] F. Arbab, C. Koehler, Z. Maraikar, Y.J. Moon & J. Proenca (2008): *Modeling, Testing and Executing Reo Connectors with the Eclipse Coordination Tools*. Tool demo session at FACS 2008.

[6] F. Arbab, N. Kokash & M. Sun (2008): *Towards Using Reo for Compliance-aware Business Process Modelling*. In T. Margaria & B. Steffen, editors: *Proc. ISoLA 2008*, *LNCS* 17, Springer, pp. 108–123, doi:10.1007/978-3-540-88479-8_9.

[7] C. Baier (2005): *Probabilistic Models for Reo Connector Circuits*. *Journal of Universal Computer Science* 11(10), pp. 1718–1748, doi:10.3217/jucs-011-10-1718.

[8] C. Baier, T. Blechmann, J. Klein & S. Klüppelholz (2009): *A Uniform Framework for Modeling and Verifying Components and Connectors*. In J. Field & V.T. Vasconcelos, editors: *Proc. COORDINATION 2009*, *LNCS* 5521, Springer, pp. 268–287, doi:10.1007/978-3-642-02053-7_13.

[9] C. Baier, M. Sirjani, F. Arbab & J. Rutten (2006): *Modeling Component Connectors in Reo by Constraint Automata*. *Science of Computer Programming* 61, pp. 75–113, doi:10.1016/j.scico.2005.10.008.

[10] M. Bonsangue, D. Clarke & A. Silva (2009): *Automata for Context-dependent Connectors*. In J. Field & V.T. Vasconcelos, editors: *Proc. COORDINATION 2009*, *LNCS* 5521, Springer, pp. 184–203, doi:10.1007/978-3-642-02053-7_10.

[11] M. Bonsangue & M. Izadi (2010): *Automata Based Model Checking for Reo Connectors*. In: *Proc. FSEN 2009*, *LNCS* 5961, Springer, pp. 260–275, doi:10.1007/978-3-642-11623-0_15.

[12] L. du Bousquet & N. Zuanon (1999): *An Overview of Lutess: A Specification-based Tool for Testing Synchronous Software*. In: *Proc. ASE'99*, IEEE Computer Society, pp. 208–215, doi:10.1109/ASE.1999.802255.

[13] L. Brandan Briones & E. Brinksma (2005): *A Test Generation Framework for Quiescent Real-Time Systems*. In J. Grabowski & B. Nielsen, editors: *Proc. FATES 2004*, *LNCS* 3395, Springer, pp. 64–78, doi:10.1007/b106767.

[14] D. Clarke, D. Costa & F. Arbab (2007): *Connector Coloring I: Synchronization and Context Dependency*. *Science of Computer Programming* 66, pp. 205–225, doi:10.1016/j.scico.2007.01.009.

[15] D. Clarke, J. Proenca, A. Lazovik & F. Arbab (2011): *Channel-based Coordination via Constraint Satisfaction*. *Science of Computer Programming* 76(8), pp. 681–710, doi:10.1016/j.scico.2010.05.004.

[16] D. Costa (2010): *Formal Models for Context Dependent Connectors for Distributed Software Components and Services*. PhD thesis, Vrije Universiteit Amsterdam.

[17] A. Faivre, C. Gaston & P. Le Gall (2007): *Symbolic Model based Testing for Component-oriented Systems*. In A. Petrenko et al., editor: *Proc. TestCom/FATES 2007*, *LNCS* 4581, Springer, pp. 90–106, doi:10.1007/978-3-540-73066-8_7.

[18] L. Frantzen, J. Tretmans & T.A.C. Willemse (2005): *Test Generation Based on Symbolic Specifications*. In J. Grabowski & B. Nielsen, editors: *Proc. FATES 2004*, *LNCS* 3395, Springer, pp. 1–15, doi:10.1007/978-3-540-31848-4_1.

[19] H. Garavel, R. Mateescu, F. Lang & W. Serwe (2007): *CADP 2006: A Toolbox for the Construction and Analysis of Distributed Processes*. In W. Damm & H. Hermanns, editors: *Proc. CAV 2007*, *LNCS* 4590, Springer, pp. 158–163, doi:10.1007/978-3-540-73368-3_18.

[20] S. Kemper (2011): *SAT-based Verification for Timed Component Connectors*. *Science of Computer Programming* doi:10.1016/j.scico.2011.02.003.

[21] R. Khosravi, M. Sirjani, N. Asoudeh, S. Sahebi & H. Iravanchi (2008): *Modeling and Analysis of Reo Connectors Using Alloy*. In D. Lea & G. Zavattaro, editors: *Proc. COORDINATION 2008*, LNCS 5052, pp. 169–183, doi:10.1007/978-3-540-68265-3_11.

[22] N. Kokash, B. Changizi & F. Arbab (2010): *A Semantic Model for Service Composition with Coordination Time Delays*. In Jin Song Dong & Huibiao Zhu, editors: *Proc. ICFEM 2010*, LNCS 6447, pp. 106–121, doi:10.1007/978-3-642-16901-4_9.

[23] N. Kokash, C. Krause & E.P. de Vink (2010): *Verification of Context-Dependent Channel-Based Service Models*. In F. de Boer et al., editor: *Proc. FMCO 2009*, LNCS 6286, pp. 21–40, doi:10.1007/978-3-642-17071-3_2.

[24] N. Kokash, C. Krause & E.P. de Vink (2011): *Reo + mCRL2: A Framework for Model-checking Dataflow in Service Compositions*. *Formal Aspects of Computing* doi:10.1007/s00165-011-0191-6.

[25] N. Lohmann, E. Verbeek & R. Dijkman (2009): *Petri Net Transformations for Business Processes - A Survey*. In K. Jensen & W. van der Aalst, editors: *Transactions on Petri Nets and Other Models of Concurrency (ToPNoC) II*, *LNCS* 5460, Springer, pp. 46–63, doi:10.1007/978-3-642-00899-3_3.

[26] S. Morimoto (2008): *A Survey of Formal Verification for Business Process Modeling*. In M. Bubak et al., editor: *Proc. ICCS 2008*, *LNCS* 5102, Springer, pp. 514–522, doi:10.1007/978-3-540-69387-1_58.

[27] A. Petrenko (2000): *Fault Model-Driven Test Derivation from Finite State Models: Annotated Bibliography*. In F. Cassez et al., editor: *Modeling and Verification of Parallel Processes*, *LNCS* 2067, Springer, pp. 196–205, doi:10.1007/3-540-45510-8_10.

[28] D. Schumm, O. Turetken, F. Leymann N. Kokash, A. Elgammal & W.-J. Heuvel (2010): *Business Process Compliance through Reusable Units of Compliant Processes*. In: *Current Trends in Web Engineering*, *LNCS* 6385, Springer, pp. 325–337, doi:10.1007/978-3-642-16985-4_29.

[29] S. Tasharofi, M. Vakilian, R. Z. Moghaddam & M. Sirjani (2008): *Modeling Web Service Interactions Using the Coordination Language Reo*. In: *Proc. WS-FM 2008*, *LNCS* 4937, Springer, pp. 108–123, doi:10.1007/978-3-540-79230-7_8.

[30] J. Tretmans (2008): *Model Based Testing with Labelled Transition Systems*. In: *Formal Methods and Testing*, *LNCS* 4949, Springer, pp. 1–38, doi:10.1007/978-3-540-78917-8_1.