Connectors as designs: Modeling, refinement and test case generation

Sun Meng\textsuperscript{a,b,*}, Farhad Arbab\textsuperscript{b}, Bernhard K. Aichernig\textsuperscript{c}, Lăcrămioara Aștefănoaei\textsuperscript{b}, Frank S. de Boer\textsuperscript{b}, Jan Rutten\textsuperscript{b}

\textsuperscript{a} LMAM, School of Mathematical Science, Peking University, Beijing, 100871, China
\textsuperscript{b} CWI, Amsterdam, The Netherlands
\textsuperscript{c} Institute for Software Technology, Graz University of Technology, Austria

\textbf{A B S T R A C T}

Over the past years, the need for high-confidence coordination mechanisms has intensified as new technologies have appeared for the development of service-oriented applications, making formalization of coordination mechanisms critical. Unifying Theories of Programming (UTP) provide a formal semantic foundation not only for programming languages but also for various expressive specification languages. A key concept in UTP is design: the familiar pre/post-condition pair that describes a contract. In this paper we use UTP to formalize Reo connectors, whereby connectors are interpreted as designs in UTP. This model can be used as a semantic foundation for proving properties of connectors, such as equivalence and refinement relations between connectors. Furthermore, it can be used as a reference document for developing tool support for Reo, such as test case generators. A fault-based method to generate test cases for component connectors from specifications is also provided in this paper. 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 in UTP, and generate test cases by solving constraints obtained from a specification and a faulty implementation. A prototype serves to demonstrate the automatization of the approach.

\section{1. Introduction}

With the growth of interest in service-oriented computing (SOC) \cite{26}, 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 mutual interactions among them in a distributed processing environment. They support large-scale distributed applications by allowing construction of complex component
connectors out of simpler ones. As an example, Reo [7,12] offers a powerful glue language for implementation of coordinating component connectors.

To support rigorous development of loosely coupled, large-scale distributed applications, we need to investigate the formal semantics of coordination languages, which provide the foundations for understanding and reasoning about them and allow the construction of supporting tools. Hoare and He’s Unifying Theories of Programming (UTP) [22] can present a formal semantics for various programming languages as well as specification languages like Circus and timed Circus [32,37], TCOZ [35], rCOS [23] and CSP [21]. We believe UTP is also well suited for developing the formal foundation for coordination languages like Reo.

The behavior of a connector generally describes the manifold interactions among components/services that it interconnects, rather than simple input–output behavior on one individual interface. Thus, a connector may synchronize different input and output actions, and therefore instead of sequences of input and output, we must use relations on different input/output sequences to describe the behavior of connectors. The communications via connectors can be modeled as designs in UTP, i.e., a pair of predicates \( P \models Q \) 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.

The point of UTP is to formalize the similar features of different languages in a similar style, and on that basis to analyze and connect different languages. One potential benefit of a UTP semantics for Reo is the possibility to integrate reasoning about Reo with reasoning about component specifications/implementations in other languages for which UTP semantics is available, such as CSP, Circus and rCOS. Another possible benefit of the result in this paper is that it provides a semantic model in which the causality of connector behavior is made explicit by separation of the assumption and the commitment in the design model. The accounting of assumptions and commitments can enable a large team of engineers to collaborate successfully in the design of huge connectors. The UTP approach also makes it possible to check connector properties by assume–guarantee reasoning. Properties of a complex connector can be decomposed into properties of its subconnectors and each subconnector can be checked separately.

UTP has been successfully applied in defining the semantics of channel-based data-flow models [22]. However, as discussed in [9], Reo is more general than data-flow models, Kahn networks and Petri nets, which can all be viewed as special channel-based models that incorporate certain basic constructs for primitive coordination. For example, in data-flow models, the channels are all buffered so that they can accept a data item at any time, storing it until the sink end is ready to take it. Reo supports a much more general notion of channel, which makes it more difficult to model Reo connectors than data-flow networks in UTP.

The complexity and importance of coordination models 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 can be used to guarantee that no mutated models exist in the implementation. 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. The theoretical foundation of the test case generation approach in this paper is based on the third author’s earlier work [3,5], in which it can be formally proven that test cases will detect certain faults by using refinement calculus. This paper shows that the mutation testing approach can be extended to formal coordination language, and we are doing mutation testing on the specification level for connectors. Our work has been presented in previous publications [4,27]. This paper unifies and extends the results presented in these publications: we add (1) a section on refinement of connectors based on the design model, including examples and proofs for refinement; and (2) a section on implementation for the design model and its application in refinement and test case generation for connectors that has been developed in JTom.\(^1\)

This paper is structured as follows. In Section 2 we briefly summarize the coordination language Reo. Then, in Section 3, we present the UTP observation model with metavariables and introduce the UTP design model used throughout the rest of the paper. In Section 4, we present the UTP design semantics for connectors in Reo and discuss refinement of connectors. In Section 5 we present the approach for fault-based test case generation for connectors. In Section 6, we discuss the implementation of the prototype. In Section 7, we present related works and compare them with our approach. Finally, Section 8 concludes with some further research directions.

2. A Reo primer

In this section we provide a brief introduction to Reo [7]. Reo is a channel-based exogenous coordination model wherein complex coordinators, called connectors, are compositionally built out of simpler ones. Exogenous coordination imposes a purely local interpretation on each inter-component communication, engaged in as a pure I/O operation on each side, that

\(^1\) The implementation is provided at http://homepages.cwi.nl/~astefano/reospec.
allows components to communicate anonymously, through the exchange of untargeted passive data. We summarize only the main concepts in Reo here. Further details about Reo and its semantics can be found elsewhere [7,9,12].

Complex connectors in Reo are organized in a network of primitive connectors, called channels. A connector provides 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. Fig. 1 shows the graphical representation of some simple channel types in Reo. More detailed discussion about these channels is provided in Section 4.

Complex connectors are constructed by composing simpler ones via the join and hiding operations. Channels are joined together at nodes. A node consists of a set of channel ends. The set of channel ends coincident on a node A is disjointly partitioned into the sets \( \text{Src}(A) \) and \( \text{Snk}(A) \), denoting the sets of source and sink channel ends that coincide on A, respectively. Nodes are categorized into source, sink and mixed nodes, depending on whether all channel ends that coincide on a node are source ends, sink ends or a combination of the two. The hiding operation is used to hide the internal topology of a component connector. The hidden nodes can no longer be accessed or observed from outside. A complex connector has a graphical representation, called a Reo circuit, which is a finite graph where the nodes are labeled with pair-wise disjoint, non-empty sets of channel ends, and the edges represent their connecting channels. The behavior of a Reo circuit is formalized by means of the data-flow at its sink and source nodes. Intuitively, the source nodes of a circuit are analogous to the input ports, and the sink nodes to the output ports of a component, while mixed nodes capture its hidden internal details.

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 non-deterministically. A sink node, thus, acts as a non-deterministic merger. A mixed node non-deterministically 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.

3. The UTP observational model

Specialization and unification phases can be observed in every scientific discipline. During specialization scientists focus on some narrowly defined phenomenon and aim to discover the laws governing that phenomenon, which, typically, are special cases of a more general theory. Unification aims at a unifying theory that clearly and convincingly explains a broader range of phenomena. A proposed unification of theories often receives spectacular confirmation and reward by the prediction of new discoveries or by the development of new technologies. However, a unifying theory is usually complementary to the theories that it links, and does not seek to replace them. In [22] Hoare and He aim at unification in computer science. They saw the need for a comprehensive theory of programming that

- includes a convincing approach to the study of a range of languages in which computer programs may be expressed,
- must introduce basic concepts and properties that are common to the whole range of programming methods and languages,
- must deal separately with the additions and variations that are particular to specific groups of related languages,
- should aim to treat each aspect and feature in the simplest possible fashion and in isolation from all the other features with which it may be combined or confused.

Our theory of Reo connectors originated out of these motivations for unification. In this section we introduce the observational model for connectors and the theory of UTP designs briefly. More details about UTP can be found in [22].

3.1. Observational model

UTP adopts the relational calculus as the foundation to unify various programming theories. All kinds of specifications, designs and programs are interpreted as relations between an initial observation and a subsequent (intermediate, stable or final) observation of the behavior of their executions. Program correctness and refinement can be represented by inclusion of relations, and all laws of the relational calculus are valid for reasoning about correctness.
Collections of relations form a theory of the paradigm being studied, and it contains three essential parts: an alphabet, a signature, and healthiness conditions.

During observations it is usual to wait for some initial transient behavior to stabilize before making any further observation. In order to express this, we introduce two variables \( ok, ok' : \text{Boolean} \). The variable \( ok \) stands for a successful initialization and the start of a communication. When \( ok \) is \textit{false}, the communication has not started, so no observation can be made. The variable \( ok' \) denotes the observation that the communication has either terminated or reached an intermediate stable state. The communication is divergent when \( ok' \) is \textit{false}.

Connectors describe the coordination among components/services. We use \( \text{in}_R \) and \( \text{out}_R \) to denote what happens on the source nodes and the sink nodes of a connector \( R \), respectively, instead of using unprimed variables for initial observations (inputs) and primed variables for subsequent ones (outputs) as in [22]. Thus, the alphabet, i.e., the set of all observation-capturing variables, used in this paper is different from that for a design in [22]. The signature gives the rules for the syntax for denoting the elements of the theory. Note that in modeling of connectors not every possible predicate is useful. It is necessary to restrict ourselves to predicates that satisfy certain healthiness conditions which embody aspects of the model being studied: e.g., a predicate describing a connector that produces output without being started should be excluded from the theory \((\neg ok \land \text{out}_R = (d, 1))\). In addition, the results of the theory must match the expected observations in reality, e.g., merging the sink node of a connector that fails to terminate with the source node of any other connector must always lead to non-termination of the whole composed connector (this is the technical motivation for introducing \( ok, ok' \)). The subset of predicates that meet our requirements are called designs.

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

Let \( D \) be an arbitrary set, the elements of which are called data elements. The set \( DS \) of data sequences is defined as

\[
DS = D^* 
\]

i.e., the set of all sequences \( \alpha = (\alpha(0), \alpha(1), \alpha(2), \ldots) \) over \( D \). Let \( R^+ \) be the set of non-negative real numbers, which in the present context can be used to represent time moments.\(^2\) For a sequence \( s \), we use \( |s| \) to denote the length of \( s \), and if \( s \) is an infinite sequence, then \( |s| = \infty \). Let \( R^+_{\alpha} \) be the set of sequences \( a = (a(0), a(1), a(2), \ldots) \) over \( R^+ \), and for all \( a = (a(0), a(1), a(2), \ldots) \) and \( b = (b(0), b(1), b(2), \ldots) \) in \( R^+_{\alpha} \), if \( |a| = |b| \), then

\[
\begin{align*}
& a < b \quad \text{iff} \quad \forall 0 \leq n < |a|. \ a(n) < b(n) \\
& a \leq b \quad \text{iff} \quad \forall 0 \leq n < |a|. \ a(n) \leq b(n)
\end{align*}
\]

The set \( TS \) of time sequences is defined as

\[
TS = \{ a \in R^+_{\alpha} \mid (\forall 0 \leq n < |a|. \ a(n) < a(n+1)) \land (|a| = \infty \Rightarrow \forall t \in R^+. \ \exists k \in \mathbb{N}. a(k) > t) \} 
\]

Thus, a time sequence \( a \in TS \) consists of increasing and diverging time moments \( a(0) < a(1) < a(2) < \cdots \).

For a sequence \( a \), the two operators \( a^R \) and \( \overline{a} \) denote the reverse and the tail of \( a \), respectively, defined as:

\[
\begin{align*}
&a^R = \begin{cases} \\
&() \quad \text{if} \ a = () \\
&(a')^R = (a(0)) \quad \text{if} \ a = (a(0))^R a'
\end{cases} \\
&\overline{a} = \begin{cases} \\
&() \quad \text{if} \ a = () \\
&a' \quad \text{if} \ a = (a(0))^{-} a'
\end{cases}
\end{align*}
\]

where \( ^- \) is the concatenation operator on sequences. The concatenation of two sequences produces a new sequence that starts with the first sequence followed by the second sequence.

The set \( TDS \) of timed data sequences is defined as \( TDS \subseteq DS \times TS \) of pairs \( \langle \alpha, a \rangle \) consisting of a data sequence \( \alpha \) and a time sequence \( a \) with \( |\alpha| = |a| \). Similar to the discussion in [9], timed data sequences can be alternatively and equivalently defined as \( (D \times R^+)^\alpha \) because of the existence of the isomorphism

\[
\langle \alpha, a \rangle \mapsto \langle (\alpha(0), a(0)), (\alpha(1), a(1)), (\alpha(2), a(2)), \ldots \rangle 
\]

The occurrence (i.e., taking or writing) of a data item at some node of a connector is modeled by an element in the timed data sequence for that node, i.e., a pair of a data element and a time moment.

In our semantic model, the observational semantics for a Reo connector is described by a design, i.e., a relation expressed as \( P \models Q \), where \( P \) is the predicate specifying the relationship among the timed data sequences on the source nodes of the connector, and \( Q \) is the predicate specifying the condition that should be satisfied by the timed data sequences on the sink nodes of the connector. The following section gives a brief introduction to the theory of designs in UTP.

---

\(^2\) Here we use the continuous time model for connectors since it is expressive and closer to the nature of time in the real world. For example, for a FIFO channel, if we have a sequence of two inputs, the time moment for the output should be between the two inputs (The semantics we use for FIFO1 in this paper disallows output and input to happen at the same time moment.). 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.
3.2. A theory of designs

An important subtheory of relations allows the separation of preconditions from post-conditions, in the manner of the well-known formal methods like VDM [24], B [2], RAISE [38], refinement calculus [10] and more recently OCL [31]. This allows us to model the total correctness of programming constructs using relations. This section offers an introduction to the relational calculus on designs in UTP.

**Definition 1.** A design is a pair of predicates $P \vdash Q$, where neither predicate contains an $o$ or $o'$, and $P$ has only unprimed variables. It has the following meaning:

$$P \vdash Q \equiv (o \land P \Rightarrow o' \land Q)$$

A design predicate represents a pre/post-condition specification. The separation of precondition from post-condition allows us to write a specification that has a more general precondition than simply the domain of the relation used as a specification. Implementing a design, we are allowed to assume that the precondition holds, but we have to satisfy the post-condition. Moreover, we can rely on the system having been started, but we must ensure that it terminates. If the precondition does not hold, or the system does not start, we are not committed to establish the post-condition nor even to make the system terminate.

Any non-trivial system requires a facility to select between alternatives according to the truth or falsehood of some guard condition $b$. The restriction that $b$ contains no primed variables ensures that it can be checked before starting either of the actions. The conditional expression $P \prec b \succ Q$ describes a system that behaves like $P$ if the initial value of $b$ is true, or like $Q$ otherwise. It can be defined as follows:

**Definition 2.** The conditional expression is defined as follows:

$$P \prec b \succ Q \equiv (true \vdash (b \land P \lor \neg b \land Q))$$

The sequential composition $P; Q$ denotes a system that first executes $P$, and when $P$ terminates executes $Q$. This system is defined via existential quantification to hide its intermediate observation, and to remove the variables that record this observation from the list of free variables of the predicate. To accomplish this hiding, we introduce a fresh set of variables $v_0$ to denote the intermediate observation. These fresh variables replace the input variables $v$ of $Q$ and the output variables $v'$ of $P$, thus the output alphabet of $P$ (out$aP$) and the input alphabet of $Q$ (in$aQ$) must be the same.

**Definition 3.** Let out$aP = \{v'\}$, in$aQ = \{v\}$, then

$$P(in : u; out : v'); Q(in : v; out : w) \equiv \exists v_0 \bullet P(in : u; out : v_0) \land Q(in : v_0; out : w)$$

If the conditional and sequential operators are applied to designs, the result is also a design. This follows from the laws below.

$$(P_1 \vdash Q_1) \prec b \succ (P_2 \vdash Q_2) = ((P_1 \prec b \succ P_2) \vdash (Q_1 \prec b \succ Q_2))$$

$$(P_1 \vdash Q_1); (P_2 \vdash Q_2) = (P_1 \land \neg Q_2) \vdash (Q_1; Q_2)$$

A reassuring result about a design is the notion of refinement, which is defined via implication. In UTP, we have the well-known property that under refinement, preconditions are weakened and post-conditions are strengthened. This is established by the following definition:

**Definition 4.** $[(P_1 \vdash Q_1) \subseteq (P_2 \vdash Q_2)]$ iff $[P_1 \Rightarrow P_2] \land [P_1 \land Q_2 \Rightarrow Q_1]$

The theory of designs forms a complete lattice, with miracle $\top_D$ as the top element, and abort $\bot_D$ as the bottom element.

$$\top_D \equiv (true \vdash false) \quad \text{and} \quad \bot_D \equiv (false \vdash true)$$

The meet and join operations in the lattice of designs are defined as follows, which represent internal (non-deterministic, demonic) and external (angelic) choices.

$$(P_1 \vdash Q_1) \cap (P_2 \vdash Q_2) = (P_1 \land P_2 \vdash Q_1 \lor Q_2)$$

$$(P_1 \vdash Q_1) \cup (P_2 \vdash Q_2) = (P_1 \lor P_2 \vdash ((P_1 \Rightarrow Q_1) \land (P_2 \Rightarrow Q_2)))$$

Finally, iteration is expressed by means of recursive definitions. A recursively defined design has as its body a function on designs; as such, it can be seen as a (monotonic) function on pre/post-condition pairs $(X, Y)$, and iteration is defined as the least fixed point of the monotonic function.

The theory of designs can be taken as a tool for representing specifications, programs, and, as in the following sections, connectors.
4. Connectors as designs

Since we aim at specifying both finite and infinite behavior of connectors, we use relations on timed data sequences to model connectors. In the following, we assume that all timed data sequences are finite. However, the semantic definition can be easily generalized to infinite sequences, which are timed data streams as proposed in [9]. We use \( D \) for a predicate of well-defined timed data sequence types. In other words, we define the behavior only for valid sequences expressed via a predicate \( D \). Then, every connector \( R \) can be represented by the design \( P(\text{in}_R) \rightarrow Q(\text{out}_R) \) as follows:

\[
\begin{align*}
\text{con} & : R(\text{in} : \text{in}_R; \text{out} : \text{out}_R) \\
\text{pre} & : P(\text{in}_R) \\
\text{post} & : Q(\text{out}_R)
\end{align*}
\]

where \( R \) is the name of the connector, \( P(\text{in}_R) \) is the precondition that should be satisfied by inputs \( \text{in}_R \) on the source nodes of \( R \), and \( Q(\text{out}_R) \) is the post-condition that should be satisfied by outputs \( \text{out}_R \) on the sink nodes of \( R \). Let \( \mathcal{N}_{\text{in}} \) and \( \mathcal{N}_{\text{out}} \) be the set of source and sink node names of \( R \), respectively, then \( \text{in}_R \) and \( \text{out}_R \) are defined as the following mappings from the corresponding sets to TDS:

\[
\begin{align*}
\text{in}_R : \mathcal{N}_{\text{in}} & \rightarrow \text{TDS} \\
\text{out}_R : \mathcal{N}_{\text{out}} & \rightarrow \text{TDS}
\end{align*}
\]

4.1. Basic connectors

We first develop the design model for a set of basic Reo connectors, i.e., channels. More complex connectors obtained by composing channels are discussed in the next section.

- **Synchronous channel** \( A \rightarrowrightarrow B \):

  \[
  \begin{align*}
  \text{con} & : \text{Sync}(\text{in} : (A \leftrightarrow \langle \alpha, a \rangle); \text{out} : (B \leftrightarrow \langle \beta, b \rangle)) \\
  \text{pre} & : D(\alpha, a) \\
  \text{post} & : D(\beta, b) \land \beta = \alpha \land b = a
  \end{align*}
  \]

  The synchronous channel 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 \( \text{ok} \).

- **FIFO1 channel** \( A \rightarrowrightarrow \leftarrowrightarrow B \):

  \[
  \begin{align*}
  \text{con} & : \text{FIFO1}(\text{in} : (A \leftrightarrow \langle \alpha, a \rangle); \text{out} : (B \leftrightarrow \langle \beta, b \rangle)) \\
  \text{pre} & : D(\alpha, a) \\
  \text{post} & : D(\beta, b) \land \beta = \alpha \land a < b \land \langle \beta^R \rangle \rhd \langle a \rangle
  \end{align*}
  \]

  A FIFO1 channel whose buffer is not full accepts input without immediately outputting it. The accepted data item is kept in the internal FIFO buffer of the channel. The next input can happen only after an output occurs. Note that here we use \( \langle \beta^R \rangle \rhd \langle a \rangle \) to represent the relationship between the time moments for outputs and their corresponding next inputs. This notation can be simplified to \( b < a \) if we consider infinite sequences of inputs and outputs.\(^4\)

  On the other hand, for the FIFO1 channel \( A \rightarrowrightarrow \leftarrowrightarrow B \) where the buffer contains the data element \( e \), the communication can be initiated only if the data element \( e \) can be taken via the sink end. In this case, we have

  \[
  \begin{align*}
  \text{con} & : \text{FIFO1e}(\text{in} : (A \leftrightarrow \langle \alpha, a \rangle); \text{out} : (B \leftrightarrow \langle \beta, b \rangle)) \\
  \text{pre} & : D(\alpha, a) \\
  \text{post} & : D(\beta, b) \land \beta = \langle e \rangle \land \alpha \land a < \langle b \rangle \land \langle \beta^R \rangle \rhd \langle a \rangle
  \end{align*}
  \]

\(^3\) Note that \( \text{ok} \) and \( \text{ok}' \) do not show up in this (and the following) formulation. In fact, Such a formulation for a connector gives its specification as a design \( \text{pre} \rightarrow \text{post} \). According to Definition 1, it means \( \text{ok} \land \text{pre} \rightarrow \text{ok}' \land \text{post} \).

\(^4\) Note that \( \langle b^R \rangle \rangle \) denotes the sequence obtained by removing the \( n \)th element from a sequence \( b \) with length \( n \). This operation is defined only for finite sequences. We allow both finite and infinite sequences in our definition since theoretically, infinite sequence is possible, but in implementation (especially for test cases) we can only use finite sequences. For other channels, the representation for infinite or finite sequences in the \( \text{pre} \) and \( \text{post} \) conditions are not different. But for FIFO channels, the format being used in the definition is for finite sequence because we also use it in the implementation. The definition for infinite case is similar, the only difference is because we cannot make reverse of an infinite sequence. This is not a big problem, the reader can easily find out which case is discussed and which format can be used according to the context.
• Synchronous drain \( A \rightarrow B \):

- **con**: \( \text{SyncDrain}(\text{in} : (A \mapsto \langle \alpha, a \rangle, B \mapsto \langle \beta, b \rangle); \text{out} : (\ ))\)
- **pre**: \( D(\alpha, a) \land D(\beta, b) \land a = b \)
- **post**: \( \text{true} \)

This channel has two source channel 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 \( \text{true} \) in the post-condition means the communication terminates.

• Lossy Synchronous Channel \( A \rightarrow B \):

- **con**: \( \text{LossySync}(\text{in} : (A \mapsto \langle \alpha, a \rangle); \text{out} : (B \mapsto \langle \beta, b \rangle))\)
- **pre**: \( D(\alpha, a) \)
- **post**: \( D(\beta, b) \land L((\alpha, a), (\beta, b)) \)

where

\[
L((\alpha, a), (\beta, b)) \equiv (\beta = ( ) \land b = ( )) \lor (a(0) \leq b(0) \land \left\{ \begin{array}{ll}
\alpha(0) = \beta(0) \land L((\overrightarrow{\alpha}, \overrightarrow{a}), (\overrightarrow{\beta}, \overrightarrow{b})) & \text{if } a(0) = b(0) \\
L((\overrightarrow{\alpha}, \overrightarrow{a}), (\beta, b)) & \text{if } a(0) < b(0)
\end{array} \right.)
\]

This channel is similar to a synchronous channel, except that it always accepts all data items through its source end. If it is possible for it to simultaneously dispense the data item through its sink end, the channel transfers the data item. Otherwise the data item is lost.

• Filter \( A \rightarrow\{p\} \rightarrow B \):

- **con**: \( \text{Filter}(\text{in} : (A \mapsto \langle \alpha, a \rangle); \text{out} : (B \mapsto \langle \beta, b \rangle))\)
- **pre**: \( D(\alpha, a) \)
- **post**: \( D(\beta, b) \land F((\alpha, a), (\beta, b)) \)

where

\[
F((\alpha, a), (\beta, b)) \equiv \left\{ \begin{array}{ll}
\beta = ( ) \land b = ( ) & \text{if } \alpha = ( ) \land a = ( ) \\
F((\overrightarrow{\alpha}, \overrightarrow{a}), (\beta, b)) & \text{if } \alpha(0) \in p
\end{array} \right.
\]

This channel specifies a filter pattern \( p \) which is a set of data values. It transfers only those data items that match with the pattern \( p \) and loses the rest. A write operation on the source end succeeds only if either the data item to be written does not match with the pattern \( p \) or the data item matches the pattern \( p \) and it can be taken synchronously via the sink end of the channel.

• \( p \)-Producer \( A \rightarrow\{p\} \rightarrow B \):

- **con**: \( \text{Producer}(\text{in} : (A \mapsto \langle \alpha, a \rangle); \text{out} : (B \mapsto \langle \beta, b \rangle))\)
- **pre**: \( D(\alpha, a) \)
- **post**: \( D(\beta, b) \land \beta \in p^* \land b = a \)

This channel specifies a producer pattern \( p \) which is a set of data values. Once it accepts a data item from the source end, it produces a data item in the set \( p \) which is taken synchronously via the sink end.

• Asynchronous Spout \( A \leftarrow\{p\} \rightarrow B \):

- **con**: \( \text{AsynSpout}(\text{in} : (\ ); \text{out} : (A \mapsto \langle \alpha, a \rangle, B \mapsto \langle \beta, b \rangle))\)
- **pre**: \( \text{true} \)
- **post**: \( D(\alpha, a) \land D(\beta, b) \land |a| = |b| \land a \Rightarrow b \)

where \( \Rightarrow TS \times TS \) is defined as

\[
a \Rightarrow b \equiv a = ( ) \lor b = ( ) \lor \left\{ \begin{array}{ll}
a(0) \neq b(0) \land \overrightarrow{a} \Rightarrow \overrightarrow{b} & \text{if } a(0) < b(0) \\
a \Rightarrow \overrightarrow{b} & \text{if } b(0) < a(0)
\end{array} \right.
\]

This channel outputs two sequences of data items at its two output (sink) ends, but the data items on the two ends are never delivered at the same time.

Similar to the definition for synchronous drain and asynchronous spout, we can easily derive the design models for asynchronous drain and synchronous spout channels as well, which we skip here.
4.2. Composing connectors

Different connectors can be composed to build more complex connectors. Since basic channels can be modeled by
designs, their composition can be modeled by design composition, and the resulting connector is still a design. According
to the node types in Reo, we have three types of composition, which can be captured by the three configurations as shown in
Fig. 2. In the following, for \( i = 1, 2 \), we use

\[
\begin{align*}
\text{con} & : R_i(\text{in} : \text{in}_{R_i}; \text{out} : \text{out}_{R_i}) \\
\text{pre} & : P_i(\text{in}_{R_i}) \\
\text{post} & : Q_i(\text{in}_{R_i}, \text{out}_{R_i})
\end{align*}
\]

to denote the two connectors being composed.

Fig. 2(1) shows the case of flow-through composition of connectors. For the two connectors \( R_1 \) and \( R_2 \), suppose one
sink node of \( R_1 \) and one source node of \( R_2 \) are joined together into a mixed node \( B \). When we compose connectors, we
want the events on the mixed nodes to happen silently and automatically whenever they can, without the participation or
even the knowledge of the environment of the connector. Such mixed nodes are hidden (encapsulated) by using existential
quantification. Let \( B \mapsto \langle \beta_1, b_1 \rangle \in \text{out}_{R_1} \) and \( B \mapsto \langle \beta_2, b_2 \rangle \in \text{in}_{R_2} \) be the output on the node \( B \) in \( R_1 \) and input on the node
\( B \) in \( R_2 \), respectively. Then, the resulting connector is denoted by \( R = R_1 \bowtie R_2 \), and the corresponding design is given as\(^5\)

\[
\begin{align*}
\text{con} & : R \left( \text{in} : \bigcup_{i=1,2} \text{in}_{R_i} \setminus \{ B \mapsto \langle \beta_2, b_2 \rangle \}; \text{out} : \bigcup_{i=1,2} \text{out}_{R_i} \setminus \{ B \mapsto \langle \beta_1, b_1 \rangle \} \right) \\
\text{pre} & : P_1 \land \neg (Q_1(\beta_1, b_1) ; \langle \beta_2, b_2 \rangle \mapsto \neg P_2) \\
\text{post} & : Q_1(\beta_1, b_1) \land \langle \beta_2, b_2 \rangle \land Q_2
\end{align*}
\]

in which the union and subtraction operators are defined as follows: For two mappings \( f : M \to S \) and \( g : N \to T \), and
\( M \cap N = \emptyset, f \cup g : M \cup N \to S \cup T \), where

\[
(f \cup g)(e) = \begin{cases} f(e) & \text{if } e \in M \\ g(e) & \text{if } e \in N \end{cases}
\]

and for \( L \subseteq M \), let \( f_L : L \to S \) where for all \( e \in L, f_L(e) = f(e) \), then \( f \setminus f_L = f_{M \setminus L} \). Furthermore, the sequential composition of predicates is defined similarly as in Definition 3. If one sink node of \( R_1 \) and one source node of \( R_2 \) are joined together into a
mixed node \( B \), \( \langle \beta_1, b_1 \rangle \) and \( \langle \beta_2, b_2 \rangle \) are the timed data sequences on \( B \) in \( R_1 \) and \( R_2 \) respectively, for the two predicates \( P \) on
\( \langle \beta_1, b_1 \rangle \) and \( Q \) on \( \langle \beta_2, b_2 \rangle \),

\[
P_{\langle \beta_1, b_1 \rangle} \land \langle \beta_2, b_2 \rangle \land Q = \exists(\beta, b).P[\langle \beta, b \rangle / \langle \beta_1, b_1 \rangle] \land Q[\langle \beta, b \rangle / \langle \beta_2, b_2 \rangle]
\]

For a predicate \( P \), if \( v \) is a variable in \( P \), \( P[u/v] \) is the predicate obtained by replacing all occurrence of \( v \) in \( P \) by \( u \). Note that the condition on \( \text{out}_u P \) and \( \text{in}_u Q \) in Definition 3 is not needed here, since not all output nodes of \( R_1 \) and input nodes of \( R_2 \) are composed with each other, leaving some of them to be used as the external nodes of the composed connector.

Fig. 2(2) shows the case of merging two sink nodes of the connectors \( R_1 \) and \( R_2 \). Let \( \langle y_i, c_i \rangle \) for \( i = 1, 2 \) be the timed
data sequences on the node \( C \) in \( R_1 \) and \( R_2 \), respectively. Then the resulting connector is denoted by \( R = R_1 \bowtie_C R_2 \), and the corresponding design is given as

\(^5\) The existential quantification in the definition for sequential composition is on a pair of streams. But this definition can be easily generalized to
connectors composed on multiple nodes. If we have multiple nodes to be composed, the user should decide each pair of nodes to be composed together.
Then the only change that we need to make on the definition is to replace \( B \) with a sequence of nodes, and \( \langle \beta_i, b_i \rangle (i = 1, 2) \) with two sequences of TDS. The order in the two sequences implies the way of composition. In other words, we can have \( P_{\langle \beta_i, b_i \rangle} \land Q = \exists(\beta^1, \beta^2, \ldots, \beta^n, b^1, b^2) . \prod_{i=1,2} \langle \beta_i, b_i \rangle \land Q_1 \land Q_2 \) where \( \prod \) is the sequence \((\beta^1, b^1), \ldots, (\beta^n, b^n)\) and \( \pi_1, \pi_2 \) are two sequences of TDS: \( \pi_i = (\langle \beta_i, b_i \rangle) \).
In this relation, two timed data sequences \( \langle y_1, c_1 \rangle \) and \( \langle y_2, c_2 \rangle \) are merged together into one single timed data sequence \( \langle y, c \rangle \) where the order of elements in the sequence is decided by the time moments.

**Example 3** shows the case of merging two source nodes of the connectors \( R_1 \) and \( R_2 \). Let \((α_i, a_i)\) for \(i = 1, 2\) be the timed data sequences on the node \( A \) in \( R_1 \) and \( R_2 \), respectively. Then the resulting connector is denoted by \( R = R_1 \times_A R_2 \), and the corresponding design is given as

\[
\begin{align*}
\text{con} & : \quad R \left( \text{in} : \bigcup_{i=1,2} \text{in}_{R_i} \setminus \{ A \mapsto \langle α_i, a_i \rangle \}; \text{out} : \bigcup_{i=1,2} \text{out}_{R_i} \right) \\
\text{pre} & : \quad \bigwedge_{i=1,2} P_i(\text{in}_{R_i})[\langle α, a \rangle / (α_i, a_i)] \\
\text{post} & : \quad \bigwedge_{i=1,2} Q_i(\text{in}_{R_i}, \text{out}_{R_i})[\langle α, a \rangle / (α_i, a_i)]
\end{align*}
\]

According to the design semantics, we can easily establish a number of algebraic laws relating the composition patterns. For example, one gets associativity and commutativity for \( \times \) and \( \times_C \):

\[
\begin{align*}
(R_1 \times_C R_2) \times_C R_3 & \equiv R_1 \times_C (R_2 \times_C R_3) \\
(R_1 \times_A R_2) \times_A R_3 & \equiv R_1 \times_A (R_2 \times_A R_3) \\
R_1 \times_C R_2 & \equiv R_2 \times_C R_1 \\
R_1 \times_A R_2 & \equiv R_2 \times_A R_1
\end{align*}
\]

Furthermore, for connectors \( R_1, R_2 \) and \( R_3 \), if \( R_2 \) has \( A (B) \) as one of its source (sink) nodes, while \( A (B) \) is also a sink (source) node of \( R_1 (R_3) \), then we have

\[
(R_1 \times_A R_2) :_A R_3 \equiv R_1 :_A (R_2 :_A R_3)
\]

An arbitrary \( m \times n \)-mixed node in a connector, which connects \( m \) sink channel ends and \( n \) source channel ends, is expressed in terms of an \( m \)-ary merger connected to an \( n \)-ary replicator. In other words, for such \( m \times n \)-mixed nodes, the operations of merging source nodes and merging sink nodes have higher priority than flow-through composition. For example, the node \( A \) in **Fig. 3(a)** is a mixed node connecting 3 sink channel ends and 2 source channel ends. When we make the composition, we first merge the sink ends of \( BA, CA \) and \( DA \) into one sink node, and the source ends of \( AE \) and \( AF \) into a source node, respectively, and then merge the resulting sink node and the resulting source node together, as shown in **Fig. 3(b)**. Thus, a mixed node cannot be composed with other nodes. This way, any Reo connector can be modeled by designs composed out of the designs of basic channels.

Note that the composition operations can be easily generalized to the case for merging multiple nodes, and merging different nodes of the same connector. The definitions are similar. Therefore, we use the following two simple examples to show our approach instead of giving all the technical details.

**Example 1.** **Fig. 4** shows a Reo circuit consisting of three channels \( AB, AC \) and \( BC \) which are of types \textbf{SyncDrain}, \textbf{FIFO1} and \textbf{Sync}, respectively. By composing the channels, we can get the connector as

Fig. 3. Mixed node in Reo.

Fig. 4. Alternator.

Fig. 5. Exclusive router.

\[ \text{con : Alternator} \left( \text{in : } (A \mapsto \langle \alpha, a \rangle, B \mapsto \langle \beta, b \rangle); \text{ out : } C \mapsto \langle \gamma, c \rangle \right) \]
\[ \text{pre : } D(\alpha, a) \land D(\beta, b) \land a = b \]
\[ \text{post : } D(\gamma, c) \land \exists(\gamma_1, c_1), (\gamma_2, c_2). D(\gamma_1, c_1) \land D(\gamma_2, c_2) \land \]
\[ \gamma_1 = \alpha \land a < c_1 \land \left(\frac{c_1}{c_2}\right)^k < \frac{a}{\delta} \land \gamma_2 = \beta \land c_2 = b \land \]
\[ M(\langle \gamma_1, c_1 \rangle, \langle \gamma_2, c_2 \rangle, \langle \gamma, c \rangle) \]

where the post-condition \text{post} happens to be equivalent to

\[ \text{post : } D(\gamma, c) \land \gamma(2n) = \beta(n) \land \gamma(2n + 1) = \alpha(n) \land \]
\[ c(2n) = a(n) \land a(n) < c(2n + 1) < a(n + 1) \]

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.

\textbf{Example 2.} Fig. 5 shows an implementation of an exclusive router connector. A data item arriving at the input port A flows through to only one of the output ports B or C, depending on which one is ready to consume it. If both output ports are prepared to consume a data item, then one of the output ports is selected non-deterministically. We can parameterize this connector to have as many output nodes as we want simply by inserting more LossySync and pairs of Sync channels, as required.

By composing the channels together, we can get the connector as

\[ \text{con : EXRouter} \left( \text{in : } (A \mapsto \langle \alpha, a \rangle); \text{ out : } (B \mapsto \langle \beta, b \rangle, C \mapsto \langle \gamma, c \rangle) \right) \]
\[ \text{pre : } D(\alpha, a) \]
\[ \text{post : } D(\beta, b) \land D(\gamma, c) \land L(\langle \alpha, a \rangle, \langle \beta, b \rangle) \land L(\langle \alpha, a \rangle, \langle \gamma, c \rangle) \land \]
\[ M(\langle \beta, b \rangle, \langle \gamma, c \rangle, \langle \alpha, a \rangle) \]

where \text{L} and \text{M} were introduced previously.

\subsection{4.3. Refinement of connectors}

Implication of predicates establishes a refinement order over connectors. Thus, more concrete implementations imply more abstract specifications. For two connectors
We first consider the connector (a) in \( \text{a} \) which is equivalent to \( \text{D} \). Similarly, we have, for \( \text{b} \) in \( \text{D} \) equivalence can be proved easily by applying the UTP semantics of connectors and the definition of refinement in \( \text{D} \).

Proof. We first consider the connector (a) in Fig. 6, which can be constructed by the following steps:

\[
\text{con} : R_i \langle \text{in} : \text{in} R_i; \text{out} : \text{out} R_i \rangle
\]
\[
\text{pre} : P_i (\text{in} R_i)
\]
\[
\text{post} : Q_i (\text{in} R_i, \text{out} R_i)
\]

where \( i = 1, 2 \), if \( \text{in} R_1 = \text{in} R_2 \) and \( \text{out} R_1 = \text{out} R_2 \), then

\[
R_1 \sqsubseteq R_2 \iff (P_1 \Rightarrow P_2) \land (P_1 \land Q_2 \Rightarrow Q_1)
\]

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

Based on the design model of connectors, we can develop various (equivalence and refinement) laws for connector constructs and encode them as theorems to support a reasoning system. For example, the result of connecting the sink node of an arbitrary connector \( \text{c} \) to the source end of a synchronous channel is equal to the connector \( \text{c} \).

One more interesting example of such connector refinement laws is as shown in Fig. 6. The connector (a) on the left side enables the data written to the source node \( \text{A} \) to be asynchronously taken out via the two sink nodes \( \text{B} \) and \( \text{C} \), and the connector (b) in the middle refines this behavior by synchronizing the two sink nodes, which means that the two output events must happen simultaneously. Finally, the connectors (b) and (c) have identical behavior. Both the refinement and equivalence can be proved easily by applying the UTP semantics of connectors and the definition of refinement in (1).

As an example of constructing the refinement relations, here we show the proof of \( \text{a} \sqsubseteq \text{b} \) for the connectors (a) and (b) in Fig. 6.

In the first step, we apply the flow-through composition operation to the synchronous channel \( \text{D}_2 \text{E} \) and the FIFO1 channel \( \text{EB} \), and to the synchronous channel \( \text{D}_3 \text{F} \) and the FIFO1 channel \( \text{FC} \), respectively. Taking \( \text{D}_2 \text{B} \) into consideration, we have

\[
\text{con} : \text{D}_2 \text{B} \langle \text{in} : \text{D}_2 \mapsto (\delta_2, d_2); \text{out} : (\beta \mapsto (\beta, \beta)) \rangle
\]
\[
\text{pre} : \text{D} (\delta_2, d_2) \land \neg \text{D} (\epsilon, \epsilon). (\text{D} (\epsilon, \epsilon) \land \epsilon = \delta_2 \land \epsilon = d_2 \land \neg \text{D} (\epsilon, \epsilon))
\]
\[
\text{post} : \exists (\epsilon, \epsilon). (\text{D} (\epsilon, \epsilon) \land \epsilon = \delta_2 \land \epsilon = d_2 \land \text{D} (\beta, \beta) \land \beta = \epsilon \land \epsilon < b \land
\]
\[
(\beta^R)^R < \frac{b^R}{\epsilon^R}
\]

which is equivalent to

\[
\text{con} : \text{D}_2 \text{B} \langle \text{in} : \text{D}_2 \mapsto (\delta_2, d_2); \text{out} : (\beta \mapsto (\beta, \beta)) \rangle
\]
\[
\text{pre} : \text{D} (\delta_2, d_2)
\]
\[
\text{post} : \text{D} (\beta, \beta) \land \beta = \delta_2 \land d_2 < b \land (\beta^R)^R < \frac{d^R}{\delta_2^R}
\]

Similarly, we have, for \( \text{D}_3 \text{C} \),

\[
\text{con} : \text{D}_3 \text{C} \langle \text{in} : \text{D}_3 \mapsto (\delta_3, d_3); \text{out} : (\gamma \mapsto (\gamma, \gamma)) \rangle
\]
\[
\text{pre} : \text{D} (\delta_3, d_3)
\]
\[
\text{post} : \text{D} (\gamma, \gamma) \land \gamma = \delta_3 \land d_3 < c \land (\gamma^R)^R < \frac{d_3^R}{\gamma^R}
\]

Then we merge \( \text{D}_2 \) and \( \text{D}_3 \) into node \( \text{D}_4 \) in the second step. According to the definition of merging source nodes, we get

\[\text{D}_4 \langle \text{in} : \text{D}_4 \mapsto \langle \delta_2, d_2, \delta_3, d_3 \rangle; \text{out} : \langle \beta, \gamma \rangle \rangle\]
Both specifications and implementations of connectors are represented as designs. As recommended in [4], such errors include a wrongly used channel, a missing subcircuit, or a circuit with wrongly constructed topology, etc.

5.1. Testcases for connectors

We expand on this idea and present the theory our test case generator is built on.

In this section, we set out to generate test cases to detect possible errors in the designs of connectors. Examples of such errors include a wrongly used channel, a missing subcircuit, or a circuit with wrongly constructed topology, etc.

### Fault-based test case generation

In this section, we discuss the problem of fault-based test case generation for component connectors with Reo as our target implementation language. Similar to the mutation testing approach for programs in [5], the specification, implementation and test cases for connectors are all given by designs, and we are doing mutation testing on the specification level.

Consider the example connector given in Fig. 4. This connector can be mutated to introduce possible faults such as a change in the topology of the connector, as shown in Fig. 7. Such a faulty connector has a different behavior than that of the one in Fig. 4: 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, using which it can detect such errors and exclude faulty implementations.

The idea is to test against such coordination or protocol problems. The Reo circuit serves as a specification model representing the expected coordination among a set of components. A number of mutated Reo circuits represent the fault models encoding situations of what can go wrong. A refinement checker will generate counter-examples distinguishing the correct from the mutated (faulty) behavior. From this counterexample a test case is generated and executed on the implementation. Each test case will prevent that the mutated coordination model has been implemented. Hence, the test cases cover certain faults modeled at the specification level. This is why the approach is called fault-based. In the following, we expand on this idea and present the theory our test case generator is build on.

#### 5.1. Test cases for connectors

In this section, we set out to generate test cases to detect possible errors in the designs of connectors. Examples of such errors include a wrongly used channel, a missing subcircuit, or a circuit with wrongly constructed topology, etc.

Both specifications and implementations of connectors are represented as designs. As recommended in [3,6], 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 5** (Faulty Connector). Given an intended connector specification

\[
\text{con : } R(in : i_{inR}; out : o_{outR}) \\
\text{pre : } P(i_{inR}) \\
\text{post : } Q(i_{inR}, o_{outR})
\]

and a connector implementation

\[
\text{con : } R'(in : i_{inR}; out : o_{outR}) \\
\text{pre : } P'(i_{inR}) \\
\text{post : } Q'(i_{inR}, o_{outR})
\]

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

Note that not all errors in the design of a connector lead to a faulty connector. 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, the relation is a bidirectional refinement which is stronger than mere refinement.). However, swapping the nodes leads to a faulty connector. For example, if we swap \( A \) and \( B \) in Fig. 4, the resulting connector Alternator’ as shown in Fig. 7 is a faulty connector with respect to the specification Alternator.

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

**Definition 6** (Deterministic Test Case). For a connector \( R(in : i_{inR}; out : o_{outR}) \), let \( i \) be the input and \( o \) be the output, both are mapping from set of node names to timed data sequences with the same domains as \( i_{inR} \) and \( o_{outR} \) respectively. A deterministic test case for \( R \) is defined as

\[
t_d(i_{inR}, o_{outR}) = (i_{inR} = i) \vdash (o_{outR} = o)
\]

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

**Definition 7** (Test Case). For a connector \( R(in : i_{inR}; out : o_{outR}) \), let \( i \) be the input and \( O \) be a possibly infinite set containing the expected output(s). Both \( i \) and any \( o \in O \) are mapping from set of node names to timed data sequences with the same domains as \( i_{inR} \) and \( o_{outR} \) respectively. A test case for \( R \) is defined as

\[
t(i_{inR}, o_{outR}) = i_{inR} = i \vdash o_{outR} \in O
\]

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

**Definition 8.** For a connector specification \( S \), its implementation \( R \) and a test case \( t \), which satisfy

\[
t \sqsubseteq S \sqsubseteq R
\]

- \( t \) is called a correct test case with respect to \( S \).
- \( R \) passes the test case \( t \) and conforms to the specification \( S \).

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

**Definition 9** (Fault-Adequate Test Case). Let \( t(i_{inR}, o_{outR}) \) be a test case (which can be either deterministic or non-deterministic), \( R \) an expected connector, and \( R' \) its faulty implementation. Then \( t \) is a fault-adequate test case if and only if

\[
t \sqsubseteq R \land t \not\sqsubseteq R'
\]

A fault-adequate test case detects a fault in \( R' \). Alternatively, we can say that the test case distinguishes \( R \) and \( R' \). All test cases that detect a certain fault form a fault-adequate equivalence class.
5.2. Test case generation

A strategy for generating test cases is always based on a hypothesis about faults. **Definition 9** shows that a test case for finding errors in a faulty connector has to, first, be a correct test case of the intended connector; and second, it must not be an abstraction of the faulty connector. Given \( R \) and \( R' \), if \( R' \) is a faulty implementation of \( R \) then no refinement relation holds between them, i.e., \( R \nsubseteq R' \). By the definition of refinement in (1), a test case is a solution to the negated implications \( P \Rightarrow P' \) and \( P \land Q' \Rightarrow Q \), and this is what the following algorithm exploits.

**Algorithm 1.** Consider an intended connector

\[
\text{con} : \ R\text{(in : in}_R\text{; out : out}_R) \\
\text{pre} : \ P\text{(in}_R) \\
\text{post} : \ Q\text{(in}_R\text{, out}_R)
\]

and its possibly/assumed faulty implementation

\[
\text{con} : \ R'\text{(in : in}_R\text{; out : out}_R) \\
\text{pre} : \ P'\text{(in}_R) \\
\text{post} : \ Q'\text{(in}_R\text{, out}_R)
\]

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

1. A test case \( t \) is searched by
   
   1. find a pair \((i, o)\) as a solution of
      \( P(i) \land Q(i, o) \land \neg Q(i, o) \)
   
   2. if it exists, then the test case \( t(i, O) \) is generated by finding the maximal set \( O \) of outputs, such that for all \( o \in O \), \( i = \text{con}(\text{pre}(i, o)) \land \text{post}(i, o)\) holds.

2. If the previous step does not succeed, then look for a test case \( t(i, O) \) with \( O \) the maximal set of outputs, such that for all \( o \in O \), \( \neg P(i) \land P(i) \land Q(i, o) \) holds.

**Theorem 1** (Correctness). Given an intended connector

\[
\text{con} : \ R\text{(in : in}_R\text{; out : out}_R) \\
\text{pre} : \ P\text{(in}_R) \\
\text{post} : \ Q\text{(in}_R\text{, out}_R)
\]

its faulty implementation

\[
\text{con} : \ R'\text{(in : in}_R\text{; out : out}_R) \\
\text{pre} : \ P'\text{(in}_R) \\
\text{post} : \ Q'\text{(in}_R\text{, out}_R)
\]

and a test case \( t(i, O) \) generated by **Algorithm 1**,

\[
t(i, O) \subseteq R
\]

**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 \( o \in O \), \( P(i) \land Q(i, o) \) is satisfied, we have

   \[
t(i, O) \subseteq R \\
   \equiv\{\text{(1) and Definition 7}\} \\
   (\text{in}_R = i \Rightarrow P\text{(in}_R)) \land \\
   (Q\text{(in}_R\text{, out}_R) \land \text{in}_R = i \Rightarrow \text{out}_R \in O) \\
   \equiv\{O \text{ is the maximal set such that } P(i) \land Q(i, o) \text{ is satisfied for all } o \in O\} \\
   \equiv\{\text{true} \land \text{true}\} \\
   \equiv\text{true}
   \]

2. Since \( O \) is the maximal set of outputs \( o \), each of which satisfies \( \neg P'(i) \land P(i) \land Q(i, o) \), it follows that \( P(i) \land Q(i, o) \) 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

   \[
t(i, O) \subseteq R \equiv \text{true} \quad \square
   \]
Theorem 2 (Fault Coverage). Given an intended connector

\[ \text{con} : \ R(in : in_R ; out : out_R) \]
\[ \text{pre} : \ P(in_R) \]
\[ \text{post} : \ Q(in_R, out_R) \]

its faulty implementation

\[ \text{con} : \ R'(in : in_R ; out : out_R) \]
\[ \text{pre} : \ P'(in_R) \]
\[ \text{post} : \ Q'(in_R, out_R) \]

and a test case \( t(i, O) \) generated by Algorithm 1,

\[ t(i, O) \not\in R' \]

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

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

\[ t(i, O) \not\in R' \]
\[ \equiv \{1\} \text{ and Definition 7} \]
\[ \neg(in_R = i \Rightarrow P'(in_R)) \lor \]
\[ \neg(in_R = i \wedge Q'(in_R, out_R) \Rightarrow out_R \in O) \]
\[ \equiv (in_R = i \wedge \neg P'(in_R)) \lor \]
\[ \exists in_R, out_R.(in_R = i \wedge Q'(in_R, out_R) \wedge out_R \notin O) \]
\[ \equiv \{\text{let in}_R = i, \text{ out}_R = \widehat{o} \} \]
\[ (in_R = i \wedge \neg P'(in_R)) \lor \text{true} \]
\[ \equiv \text{true} \]

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

\[ t(i, O) \not\in R' \]
\[ \equiv \{1\} \text{ and Definition 7} \]
\[ \neg(in_R = i \Rightarrow P'(in_R)) \lor \]
\[ \neg(in_R = i \wedge Q'(in_R, out_R) \Rightarrow out_R \in O) \]
\[ \equiv \exists in_R.(in_R = i \wedge \neg P'(in_R)) \lor \]
\[ \neg(in_R = i \wedge Q'(in_R, out_R) \Rightarrow out_R \in O) \]
\[ \equiv \{\text{let in}_R = i \} \]
\[ \text{true} \lor \neg(in_R = i \wedge Q'(in_R, out_R) \Rightarrow out_R \in O) \]
\[ \equiv \text{true} \]

From a mathematical point of view, the theorems are enough for Algorithm 1: the characterization that “test cases are solutions to a satisfiability problem” is correct. However, this is not sufficient from a computational point of view. To actually compute test cases, one needs to proceed further and investigate the decidability of the logical theory underlying the satisfaction problem. Since this is a research topic of its own, we will not detail it in this paper. We only state that test cases are, in fact, effectively computable, since the logic of connectors can be “massaged” such as to reduce it to a fragment of quantifier free first order logic, which is well-known to be decidable.

5.3. Discussion of the approach

As explained, in this test case generation approach, we assume a Reo model specifying the expected coordination. The generated test cases will cover certain faults modeled as mutations. As a consequence, we rely on two assumptions common to all mutation testing approaches at the modeling level.

1. The number and efficiency of the test cases depend on the fault models, i.e. the number and the kind of mutations. Mutation operators are a common way to generate mutants for a given language. A mutation operator is a rewriting rule that
transforms a given language construct into a faulty one. We have not yet identified a number of useful mutation operators for Reo connectors. This is future work, since it needs a number of empirical studies with our new test case generator.

(2) By covering the simple faults at the modeling level, we assume to detect additional, and perhaps more subtle, faults at the implementation level as well. This is similar to the assumption of the coupling effect in classical program mutation testing, where one assumes that simple mutations are sufficient to find more subtle bugs. Experiments will show, if this assumption is also valid for coordination models.

Different scenarios are possible for the testing approach. For instance, the Reo model may serve as an abstract specification of the protocol for a given implementation under test. Here, the test cases would check if the protocol was correctly implemented, i.e. that it conforms to the Reo model. More precisely, the tests guarantee that no mutated model has been implemented.

In a second scenario, the protocol implementation would be generated from the Reo circuit. Here, the test cases serve (a) to test the code generator, (b) as integration tests for the component network, and (c) as regression tests for future versions.

Traditionally, the communication/synchronization directives are spread over an implementation. The advantage of the Reo modeling language is that it makes the communication/synchronization explicit in a connector language. This enables to apply mutation testing to coordination patterns, which opens a new field of research and needs further studies. In the following, we describe the implementation of a test case generator that will drive these future experiments.

6. Implementation

Higher-level languages like Maude [15] and Tom [19] offer suitable means to quickly prototype one’s favorite (declarative) language. In contrast to dedicated languages like Java, the advantages include easy prototyping, clear separation between syntax and semantics, and high-level mechanisms like strategies for meta-control of the executions of the programs. To give a short illustrating example, imagine a Java class implementing the syntax of Reo connectors and to a priori compare it to the corresponding Tom code from Section 6.1.

Before presenting our Tom6 implementation of the design model and the theory of refinement for connectors, we briefly motivate why we changed from Maude7 (the language of choice in the conference version [4] of this implementation) to Tom. Maude is a high-level 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 that facilitates the use of a wide range of formal methods. For example, the language has been shown to be suitable both as a logical and as a semantic framework [28]. Our current implementation is in Tom, a powerful and efficient pattern matching engine on top of conventional programming languages like C or Java. Maude was a good candidate in the initial phase, when we were experimenting and getting accustomed with the formal definitions and concepts of our theory. However, Tom is much more flexible. By design, it is meant to support pattern matching against native data structures like objects or records [13]. For example in JTom (Tom built on top of Java) we can use the facilities provided by Java and rewriting logic. This is a strong point since we can take advantage of existing efficient implementations of data structures like Arraylist or HashMap and the corresponding functions to manipulate them. In contrast, when working with Maude one needs to adapt to the basic constructions provided by the system or implement new ones. This is because Maude is meant to facilitate the prototyping of modeling and not programming languages. However, since the basic ingredients in the theory of connectors are sequences and streams, being able to manipulate Arraylist in an efficient manner offers a clear argument in favor of JTom. Using a combination of declarative and imperative languages as a laboratory for studying “new” languages is, in our opinion, still a fresh topic, which deserves more attention. Thus, we think that a separate section discussing implementation issues in more detail is justified to provide a clear illustration of the usefulness of such an approach.

To get a glimpse of how JTom turns out to be useful for implementing the theory of connectors, we focus on two main aspects. On the one hand, we illustrate how the syntax and the semantics of connector designs are implemented in JTom. This is of concern especially for future developers. This part should be seen as a short documentation for a library providing user facilities for manipulating connector specifications. On the other hand, we present concrete examples to clarify what users need to know in order to perform their own experiments, that is, how users can take advantage of the interface of the mentioned library.

6.1. Implementing the syntax

The basic facilities that Tom provides allow to define data types, to construct data, and to transform such data using pattern matching. We take, for example, the following JTom code where the module Connector from the construction \%gom\{ . . . \} specifies the BNF grammar of connectors:

6 Tom can be downloaded from http://tom.loria.fr.
7 The corresponding Maude code can be found at http://homepages.cwi.nl/~astefano/roespec.
import testconnector.connector.types.*;
public class TestConnector{
    %gom{
        module Connector
            imports Stream Logic
            abstract syntax
                Node = node(name:String, s:StreamId) | source(n:Node) | sink(n:Node*)
                Ins = ins(Node*)
                Outs = outs(Node*)
                Config = R(ins:Ins, outs:Outs)
                ChannelType = sync() | fifo() | syncDrain() | ...
                Pre = pre(p:Pred)
                Post = post(p:Pred)
                CSpec = spec(p:Pre, q:Post)
                | getSpecBC(c:Connector)
                | constraint(t:ChannelType, s1:StreamId, s2:StreamId)
                Connector = channel(ct:ChannelType, n1:Node, n2:Node)
                | connector(cf:Config, cs:CSpec)
                ConnectorList = connectorList(Connector*) }
    public final static void main(String[] args) {
        Connector ab;
        StreamId sa, sb;
        sa = 'sId("sa"); sd = 'sId("sb");
        ab = 'channel(sync(), source(node("a", sa)), sink(node("b", sb))); } }

In order to use the Java code generated by the %gom{...} construction we need to import the package testconnector.connector.types.* where the path corresponds to the name of the class (TestConnector), followed by the name of the module (Connector), both in lowercase, and ending with types.*. Furthermore, the module Connector imports the module Stream, where the BNF grammar of streams is defined as pairs of data and time sequences, and the module Logic, which defines predicates as basic blocks for building connector specification. These predicates include, besides the booleans True() and False(), D for the well-definedness of streams and M for the merging operation. The %gom{...} construct defines sorts for data structures, e.g., Node, Connector, and operators on these sorts. The operators node, connector and channel are called constructors because they are used to "construct" connectors either from configurations and specifications or from nodes, in the case of basic channels. The main method illustrates the declaration and definition of connectors and streams. The definitions use the back-quote expression ' to construct data. The operator sId is declared in the module Stream to identify streams. Note that we work symbolically with identifiers instead of real values because we do not actually need them. As we will see later, whenever the identifier of a data or time sequence is needed, we only need to call a corresponding function which unfolds streams. Note further the use of the constructor channel for defining ab as a basic channel of type sync() that has a source node a with a stream sa and a sink node b with a stream sb. We distinguish between the two constructors connector and channel because we do not want users to explicitly write the specification of basic channels; to define a channel, one only needs to specify the type and the nodes of the channel, and this is what the definition ab illustrates.

6.2. Implementing the semantics

In this section we describe how the semantics of connector designs is implemented in JTom. We first consider the specification of connectors. Connectors are either basic channels, or have the generic format of pairs of configurations and specifications. To extract the specification we define a Java function which uses one of the main features of Tom, the construction %match:

public static CSpec getSpec(Connector c) {
    %match(c) {
        connector(_, spec) -> { return 'spec; }
        channel(sync(), source(_, sa), sink(_, sb)) && comp(da, ta) << getUnfold(sa)
        && comp(db, tb) << getUnfold(sb)
        -> { return 'spec(pre(D(comp(da, ta))),
                    post(andL(D(comp(db, tb)), eqT(ta, tb), eqD(da, db)))); } } }

where "_" denotes an arbitrary variable and can be used when the name of the variable is not needed. The %match construct is similar to a switch/case mechanism in imperative languages or to a rule in declarative languages: given a subject, if it matches a pattern, the associated action is executed. For example, when the subject c matches the pattern connector(_, spec) denoting a connector in the general format, the associated action is to return the specification of c. Besides patterns,
Tom allows the definition of constraints. For example, the expression \( \text{comp}(\text{da}, \text{ta}) \ll \text{getUnfold}(\text{sa}) \) denotes a constraint between the pattern \( \text{comp}(\text{da}, \text{ta}) \) and the subject \( \text{getUnfold}(\text{sa}) \). It basically says that the stream identified by \( \text{sa} \) has a data (time) sequence identified by \( \text{da} \). The boolean connector \&\& is used to combine multiple constraints. Thus, when the subject \( c \) matches the pattern \( \text{channel}(\text{sync}(), \text{source}(\_\_, \text{sa}), \text{sink}(\_\_, \text{sb})) \) which denotes a \textit{Sync} channel with a stream \( \text{sa} \) (\( \text{sb} \)) on its source (sink) node, and the constraints on the streams are satisfied, then the action returns the specification of the \textit{Sync} channel, i.e., the precondition of the well-definedness of \( \text{sa} \) and the post-condition of the well-definedness of \( \text{sb} \), together with the equalities between the data (time) sequences. Observe that the advantage of the Tom syntax is its modularity and expressiveness. This leads to more clear and concise code.

The next step in implementing the semantics of Reo is to consider the operators for connector composition, i.e., sequencing, merging and replicating. Since we consider connector compositions as elementary transformations, we implement them as elementary strategies by means of the Tom construct %strategy. In this way, we maintain a clear separation between \textit{transformation} and \textit{control}. Strategies also afford us a great degree of flexibility and this makes it possible to experiment with different alternative scheduling policies for compositions, merely by choosing alternative definitions of strategies, without changing any Java code. As an example, consider the strategy implementing the sequential composition:

```java
%strategy Seq() extends Fail(){
    visit ConnectorList {
        connector(R(\text{ins}(i*), \text{outs}(o1*, \text{node}(\_\_, o2*)), spec1), l2*,
        connector(R(\text{ins}(i1*, \text{node}(\_\_, i2*), o*)), spec2), l3*) -> {
            Pred p1, p2, q1, q2;
            p1 = getPre('spec1); q1 = getPost('spec1);
            p2 = getPre('spec2); q2 = getPost('spec2);
            CSpec specR = 'spec(pre(and(p1, p2)), post(and(q1, q2)));
            CSpec specRS = 'Repeat(SimpSpec()).visit(specR);
            Connector cRes =
                'connector(R(\text{ins}(i*, i1*, i2*), \text{outs}(o*, o1*, o2*)), specRS);
            return 'connectorList(l1*, cRes, l2*, l3*); }
    } }
```

In Tom, each user-defined strategy has a name, followed by a mandatory pair of parentheses.\(^8\) The keyword extends defines the behavior of the default strategy that will be applied, in our case, \textit{Fail()}, which means that \textit{Seq()} always fails when no rule is applicable. The body of the strategy is a list of visited sorts, in our case, \textit{visit ConnectorList}. Each “visit strategy” contains a list of “visit actions” that will be applied to the corresponding sort. Visit actions follow the format of match constructions (rules). In our case, the rule included in the above code states that when the subject being visited matches\(^9\) the pattern of a connector list, and when this list contains two connectors \( c1 \) and \( c2 \) where \( c1 \) has an output with the same name as one input in \( c2 \), then \( c1 \) and \( c2 \) are replaced by a new connector \( cRes \). The dots at the end of the strategy elide the code corresponding to the symmetric case where \( c1 \) has an input and \( c2 \) has an output with the same name. This symmetric rule is necessary because although the reasoning is analogous, we work with lists of connectors, which are non-commutative structures. Restricting only to the first (or second) case would be fine if we intended to fix a direction (from left to right) in which the compositions are performed.

Observe that we explicitly simplify \( \text{specR} \) before retrieving the new connector. We do this by applying a predefined strategy \textit{Repeat} which repeatedly applies its parameter strategy \textit{SimpSpec} until it is no longer possible to do so. We do not give the details of \textit{SimpSpec} here, but only informally describe what it does. The strategy \textit{SimpSpec} simplifies specifications by deleting the stream, data, and time identifiers belonging to hidden nodes after it eventually replaces formulas like \( \text{da} = \text{db} \) and \( \text{db} = \text{dc} \) by predicates like \( \text{da} = \text{dc} \) when \( \text{db} \) is hidden.

Replicating and merging are implemented similarly and we do not describe them here. To understand the rest of the section it suffices to know that their corresponding strategies are \textit{Replicate()} and \textit{Merge()}. Taking into account that merging and replicating have a higher priority than sequencing, we implement a function that takes a list of connectors as input and returns its corresponding “normalized” connector, obtained by applying all possible compositions:

```java
public static ConnectorList fixpoint(ConnectorList cl){
    try{
        cl = 'Repeat(Choice(Merge(), Replicate())).visit(cl);
        cl = 'Repeat(Seq()).visit(cl);
    } catch (VisitFailure e) { System.out.println("fixpoint: strategy failed"); }
    return cl; }
```

\(^8\) Inside these parentheses, there may be optional parameters.

\(^9\) JTom supports list matching through the so-called “variable-star” construct like \( l1* \), which are instantiated by a comma-separated sequence of connector terms instead of a single connector term (if the “*” were not present).
In the above code, Choice is a basic strategy combinator with intuitive meaning and visit is the method for applying a strategy to a term, in our case to the list of connectors cl. A nice property of the fixpoint function is that it relieves the programmer from the burden of explicitly specifying the order of compositions. For example, consider the connector $\text{ABC}_a$, described in Section 4.3. Its definition in Tom is as follows:

```
Connector ad, de, df, eb, fc, ABCa;
ad = 'channel(sync(), source(node("a", sa)), sink(node("d", sd)));
de = 'channel(sync(), source(node("d", sd)), sink(node("e", se)));
df = 'channel(sync(), source(node("d", sd)), sink(node("f", sf)));
eb = 'channel(fifo(), source(node("e", se)), sink(node("b", sb)));
f = 'channel(fifo(), source(node("f", sf)), sink(node("c", sc)));
ConnectorList l = 'connectorList(ad, de, df, eb, fc);

ABCa = fixpoint(l);
```

If we pretty print ABCa we obtain the following output:

```
pretty(ABCa) = R((a, sa) : (b, sb), (c, sc)),
D< comp(da,ta) > | - D< comp(db,tb) > and D< comp(dc,tc) > and
between(ta) = t tb and da = d db and between(ta) = t tc and da = d dc
```

where between(ta) is a predicate that we use in order to denote a time sequence whose elements have arbitrary values between successive values of the time sequence denoted by ta; in other words, between(ta) = t tb symbolically denotes the formula $a < b \land (b^t) < c$. Thus ABCa corresponds precisely to $\text{ABC}_a$ in Section 4.3.

### 6.3. Refinement and testing

Recall that a connector $R'$ with $P'$ and $Q'$ as its pre- and post-conditions is a refinement of a connector $R$ with $P$ and $Q$ as its pre- and post-conditions if and only if the implications $P \Rightarrow P'$ and $P \land Q' \Rightarrow Q$ are both true. This is exactly what the following function implements:

```
public static Pred isRefinement(Connector r, Connector r') {
    Pred p = getPre(getSpec(r));
    Pred q = getPost(getSpec(r));
    Pred p' = getPre(getSpec(r'));
    Pred q' = getPost(getSpec(r'));
    Pred implic1 = 'implic(p, p');
    Pred implic2 = 'implic(and(p, q'), q);
    try{
        Pred res1 = 'Repeat(SimpImplication()).visit(implic1);
        Pred res2 = 'Repeat(SimpImplication()).visit(implic2);
        return 'and(res1, res2);
    } catch(VisitFailure e) { System.out.println("strategy failed"); }
    return 'and(implic1, implic2); }
```

SimpImplication is a strategy that we define for simplifying implications. This strategy is similar to the one we used in the implementation of the sequencing composition, SimpSpec. We distinguish between them mainly to have a clear separation of concerns: SimpSpec is concerned with substituting elements of hidden nodes, while SimpImplication is concerned with deleting formulas that are equal from the left- and the right-hand sides of implications. For example, the following code shows some simplification cases:

```
%strategy SimpImplication() extends Fail()
visit Pred {
    (1) implic(_, andL()) -> { return 'True(); }
    (2) implic(p, p) -> { return 'True(); }
    (3) implic(and(p, q), p) -> { return 'implic(q, True()); }
    (4) implic(p1, p2) && andL(p11*, D(s1), p12*) << getAndL(p1)
        && slD(s) << getFold(s1) && slD(s) << getFold(s2) -> {
            return 'implic(andL(p11*, p12*), andL(p21*, p22*)); }
    (5) implic(p1, p2) && andL(p11*, eqD(e1, e2), p12*) << getAndL(p1)
        && slD(s) << getFold(s1) && slD(s) << getFold(s2) -> {
            return 'implic(andL(p11*, p12*), andL(p21*, p22*)); }
    (6) implic(p1, p2)
```

&& andL(p11*, e1@eqL(_*), p12*) << getEqL(getAndL(p1))
&& andL(p21*, e2@eqL(_*), p22*) << getEqL(getAndL(p2)) -> {
  if (equalsS('e1, 'e2))
    return 'implic(andL(p11*, p12*), andL(p21*, p22*));
}
(7) implic(p1, p2) && andL(p11*, p, p12*) << getAndL(p1)
&& andL(p21*, p, p22*) << getAndL(p2) -> {
  return 'implic(andL(p11*, p12*), andL(p21*, p22*));
}

The meaning of the cases (1)–(3) is intuitive. In case (4), the function getFold extracts from a pair \(\text{comp}(da, \text{ta})\) the corresponding stream identifier \(\text{sId}(sa)\); the function getAndL is used to flatten conjunctive formulas, for example \(\text{getAndL}(\text{and}(p, q), r))\) returns \(\text{andL}(p, q, r)\). This simplification is meant to delete, for instance, \(D(\text{sId}(sa))\) from \(p1\) and \(D(\text{comp}(da, \text{ta}))\) from \(p2\) if they both denote the same stream. Case (5) is needed because equality is only associative and not also commutative. In case (6), the function getEqL computes the transitive closure of the time equality relation, i.e., \(\text{getEqL}(\text{andL}(p*, \text{eqT}(\text{ta}, \text{tb}), r*, \text{eqT}(\text{tb}, \text{tc}), s*))\) returns \(\text{andL}(p*, \text{eqL}(\text{ta}, \text{tb}, \text{tc}), r*, s*)\). Furthermore, the function equalsS checks whether two sets of time sequence identifiers are equal. Case (7) recursively deletes a predicate if it appears on both the left- and the right-hand sides of an implication.

As an example, consider the scenario from Section 4.3. We are interested in the refinement relation between \(\text{ABC}_a\) and \(\text{ABC}_c\), and this is what we check by calling the function isRefinement with the parameters \(\text{ABC}_a\) and \(\text{ABC}_c\), where \(\text{ABC}_c\) is the result of applying the fixpoint function as we showed for \(\text{ABC}_a\):

```java
testRefinement of
  connector \text{ABC}_a = R(( \text{a}, \text{sa} ) : ( \text{c}, \text{sc} ), ( \text{b}, \text{sb} ) ), \text{D}_< \text{sa} > |-
  \text{D}_< \text{comp}(\text{dc},\text{tc}) > and \text{tc} = t between2(\text{ta}) and \text{D}_< \text{comp}(\text{db},\text{tb}) >
  and tb = t between2(\text{ta}) and \text{db} = d and \text{da} = d

and

  connector \text{ABC}_c = R(( \text{a}, \text{sa} ) : ( \text{b}, \text{sb} ), ( \text{c}, \text{sc} ) ),
  \text{D}_< \text{comp}(\text{da},\text{ta}) > |-
  \text{D}_< \text{comp}(\text{db},\text{tb}) > and between2(\text{ta}) = \text{tb} and
  \text{da} = d and \text{D}_< \text{comp}(\text{dc},\text{tc}) > and between2(\text{ta}) = t \text{tc} and \text{da} = d
is:
  True() and True()
```

The above pretty printed output illustrates that \(\text{ABC}_c\) refines \(\text{ABC}_a\). The function isRefinement is a main ingredient of the implementation of the test case generation:

```java
public static void generateTestCases(Connector r, Connector r') {
  Pred resRef = isRefinement(r, r');
  Bool resRefUnfold = unfoldResRef(resRef, 0);
  int i = 0;
  while(resRefUnfold) { i++;
    resRefUnfold = unfoldResRef(resRef, i); } }
```

Recall that a test case is a solution to the negated formula of the definition of the refinement relation. If we look at the result \(\text{resRef}\) returned by isRefinement\((\text{alternatorF}, \text{alternator})\), where alternator \((\text{respectively, alternatorF})\) corresponds to the connectors from Section 5, we obtain:

```java
pretty(\text{resRef}) = True() and
  (\text{ta} = \text{tb} and \text{D}_< \text{sa} > and \text{D}_< \text{sb} > and \text{M}(\text{comp}(\text{da},\text{between2}(\text{ta})), \text{comp}(\text{db}, \text{da}), \text{sc})
  => \text{M}(\text{comp}(\text{da}, \text{ta}), \text{comp}(\text{db},\text{between2}(\text{ta})), \text{sc}) )
```

which can no longer be simplified to True(). From this formula we derive test cases. We do this by symbolically unfolding formulas:

```java
public static boolean unfoldResRef(Pred p, int i){
  %match (p){
    implic(p1, p2) -> {
      String cv1 = symbolicUnfoldPred('p1, i);
      String cv2 = symbolicUnfoldPred('p2, i);
      String[] cv1Parts, cv2Parts;
      String delimiter = ":";
      cv1Parts = cv1.split(delimiter);
      cv2Parts = cv2.split(delimiter);
      if(cv1Parts != null && cv2Parts != null &&
        cv1Parts.length > 1 && cv2Parts.length > 1) {
        if(cv1Parts[0].equals(cv2Parts[0]) &&
```
The function unfoldResRef returns false and prints a counterexample whenever it encounters an implication whose left- and right-hand sides contain predicates that when unfolded i times lead to a contradiction. One illustrating example is the case where the unfolding of the predicates returns the strings (t:da) and (t:db) which state that at time t both data da and db are observable. The function symbolicUnfoldPred does a case analysis and returns strings denoting pairs of time elements and corresponding observed data elements. For example, the unfolding of the merging predicate M is as follows:

```java
import java.util.ArrayList;
import java.util.HashMap;

public static String symbolicUnfoldPred(Pred p, int i){
    %match (p) {
    M(comp(dId(da), tId(x)), comp(dId(db), tId(y)), c) -> {
        if('x.equals("between(" + 'y + ")")') {
            String sc = stream2String(getFold('c)) + "[" + i + "]";
            String tc = 'y + "[" + i + "]";
            String dc = ";";
            if (i % 2 == 0) dc = 'db + "[" + i + "]"; else dc = 'da + "[" + i + "]";
            String scV = sc + " = (" + tc + " : " + dc + ");";
            return scV; } }
    ...
    }
```

which states that if we have an input data stream whose time sequence is "between(y)" and the time sequence of the other input time data stream is y, then we can automatically compute the elements of the stream c. Since we know that y < between(y) < tail(y) we also know that for an even index i the data dc[i] is assigned the value of da[i] and for an odd index i the data dc[i] is assigned db[i]. The dots represent the symmetric case where a "between" sequence appears in the second time data stream.

In our case, the output of the pretty printing the result of calling generateTestCases for alternator and alternatorF is as follows:

```java
Counterexample: sc[0] = (tb[0] : db[0]) and sc[0] = (tb[0] : da[0])
generateTestCases: pretty(resRefUnfold) is: false
```

which illustrates that there is no refinement because in this result, at time tb[0] the data elements db[0] and da[0] are equal, which, in general, is false.

To help understanding, we tailored our explanations for the alternator example. To have a more global view of the underlying mechanism for generating cases, we summarize it as follows. Any formula resulting from calling isRefinement that is not trivially true and can no longer be simplified is further used to generate test cases. This formula is typically an implication where the atomic sentences are equalities on data (time) sequences and predicates like M, the merger. Test cases are found by looking at atomic sentences on the left-hand side of the implication that invalidate a predicate on the right-hand side. Since there is a finite number of predicates in the logic of connectors, exhaustively searching all combinations guarantees to eventually find a counterexample, that is, a test case.

6.4. Executing connector designs

In this section we describe how to "execute" concrete connector designs. For this, we will use the alternator example. First, we instantiate the streams on the nodes a and b with some values (for simplicity, integers for time and string for data). We do this by means of the existing Java structures ArrayList and HashMap. We associate to each data and time sequence identifier an array list of four values and we record them in a hash map env whose keys are denoted by the string identifiers. We further extend the environment to env1 by appending for each time sequence a "between" sequence. Given a time sequence taValues, its associated "between" sequence consists of randomly generated integers between taValues[i] and taValues[i+1], with i being an arbitrary index of taValues:

```java
ArrayList<String> daValues = new ArrayList<String>();
ArrayList<Integer> taValues = new ArrayList<Integer>();
ArrayList<Integer> tbValues = new ArrayList<Integer>();
ArrayList<String> dbValues = new ArrayList<String>();
Map<String, ArrayList> env = new HashMap<String, ArrayList>();
for (int i = 0; i < 4; i++) {
    daValues.add("da" + i);
    dbValues.add("db" + i);
```
taValues.add((Integer)((i+1)*10));
tbValues.add((Integer)((i+1)*10)); }
env.put("da", daValues);
env.put("ta", taValues);
env.put("db", dbValues);
env.put("tb", tbValues);
env1.addBetween(env);

Given an environment env, and a connector specification spec, its evaluation is returned by the following function:

```java
public static CSpec evalSpec(Map<String, ArrayList> env, CSpec spec) {
  %match (spec){
    spec(pre(p), post(q)) &&
    evalPreAndL(_, !False(), _) << getAndL(evalPred(env, p)) -> {
      return 'spec(pre(evalPre), post(evalPred(env, getAndL(q)))); }}
    return 'spec(pre(False()), post(q)); }
}
```

which in turn calls an evaluation function evalPred for the pre- and post-conditions. The evaluation of the post-condition happens only if the precondition does not evaluate to False(), i.e., !False(), where the symbol ! denotes an anti-pattern.\(^\text{10}\) The function evalPred is defined recursively to iterate through formulas until it reaches equations on time or data sequences. These equations can be evaluated to either true or false based on the values recorded in the environment.

To see how the evaluation function evalSpec works, we apply it for the alternator connector:

```java
CSpec spec = getSpec(alternator);
CSpec evalSpecRes = evalSpec(env1, spec);
System.out.println("evalSpec(env1, spec) = 
" + pretty(evalSpecRes));
```

The corresponding output is as follows:

```
bash-3.2$ java reospec.TestConnector
evalSpec(env1, spec) = True() |-
  D< sc > and valS("(10 : db0), (18 : da0), (20 : db1), (29 : da1),
                 (30 : db2), (34 : da2), (40 : db3), (41 : da3)"
```

which shows that the time data stream on the output c is obtained by merging the time data streams on the input a and b. This can also be observed if we print the environment:

```java
printEnv:
----------------------------------------------
dc : [db0, da0, db1, da1, db2, da2, db3, da3]
db : [db0, db1, db2, db3]
da : [da0, da1, da2, da3]
tb : [10, 20, 30, 40]
tc : [10, 18, 20, 29, 30, 34, 40, 41]
ta : [10, 20, 30, 40]
between(tb) : [11, 27, 36, 41]
between(ta) : [18, 29, 34, 41]
----------------------------------------------
```

The changes in the environment are the effect of the evaluation function evalPred which adds new data when it evaluates the merging predicate \(M(\text{comp}(da, \text{between}(ta)), sb, sc)\). The new time data stream \((tc, dc)\) on the output node c is constructed from the values \(da\), \text{between}(ta), db, tc which have been recorded in the environment after the instantiation.

### 7. Related work

The semantics of Reo has been well investigated earlier. For example, a coalgebraic semantics for Reo in terms of relations on infinite timed data streams has been developed by Arbab and Rutten [9], but the causality between input and output is not clear in this semantics. An operational semantics for Reo using constraint automata is provided by Baier et al. [12]. However, modeling unbounded primitives or even bounded primitives with unbounded data domains is impossible with finite automata. Bounded large data domains cause an explosion in the constraint automata model which becomes problematic. A model for Reo connectors based on the idea of coloring a connector with possible data flows to

\[^{10}\] The reader should note that anti-pattern matching, i.e., the support for complements, what one does not want to match, is a research topic in itself [25].
resolve synchronization and exclusion constraints is presented by Clarke et al. [14]. Unlike the coalgebraic and operational semantics, data sensitive behavior, which is supported by filters in Reo, are not captured in the coloring approach.

The semantic model of Reo provided in this paper is based on the Unifying Theories of Programming framework [22]. As we pointed out in Section 3, “unifying theories” does not mean to unify different existing semantics of one language, but to study a range of languages within a convincing framework. Thus, we are not aiming at “unifying” different Reo semantics in this paper. The semantics we adopted is closely related to the coalgebraic semantics in [9]. However, the coalgebraic model [9] defines behavior using infinite streams, which exclude a “natural” description of finite behavior (and connectors that exhibit finite behavior on any of their ports). In contrast, the timed data sequence in our model can be either finite or infinite, which makes it more expressive than the coalgebraic model. On the other hand, the separation of input and output in our model makes the causality of connector behavior more clear. Furthermore, the UTP approach provides a family of algebraic operators, which can be used to interpret the composition of connectors explicitly.

A UTP semantics of the data-flow model has been investigated in [22]. However, in the data-flow model, only asynchronous communication is allowed and the channels can buffer an arbitrary number of messages, a behavior that corresponds to only a special kind of channel in Reo, i.e., the unbound FIFO channel. In other words, Reo is more expressive than the data-flow model. Indeed, Reo allows an open-ended set of user-defined channel types and supports arbitrary combinations of synchrony and asynchrony, loose coupling, distribution and mobility, exogenous coordination by third parties and dynamic reconfigurability. All this makes the UTP semantics of Reo more challenging than that of classical data-flow models.

Test case generation from specifications has been developed as a prominent technique in testing of software systems. A large body of literature and several tools for generation of test cases from model-based specifications [17,16] already exist. 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 [18,36]. When the specification has data variables, constraint solving techniques can be used to find input values that drive the execution in a desired direction [29].

The problem of deriving tests from state-based models that distinguish between input and output actions has been widely investigated [17,34]. 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) models, which are generally generated from formal specifications. 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 its output in reaction to the previous input. However, in Reo, such assumptions do not hold and thus, the FSM approach is not appropriate for generating test cases for connectors.

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 [20]. Since then it has found many applications and has become the major assessment technique in empirical studies on new test case selection techniques [41]. Recently, the first author has developed a general theory of fault-based testing, in particular mutation testing, using Refinement Calculus [3] and Unifying Theories of Programming [5], 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 [30] 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 [21] 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 labeled transition systems [39,40] and test driver implementation [33]. 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 [30]), or to the testing of very abstract (finite) models for which exhaustive testing is feasible (like in protocol testing [40]).

8. Conclusion

This paper demonstrates that UTP can be applied not only for giving semantics of specific programming languages and specification languages, but also for providing a formal semantic foundation for coordination languages. In particular, the unified semantic model for different kinds of channels and composite connectors in Reo covers different communication mechanisms encoded in Reo, and allows the combination of synchronous and asynchronous channels as in Reo. In our work, we model basic connectors in Reo as designs in UTP, and the composition of connectors is specified by design composition. Our semantic model offers potential benefits in developing tool support for Reo, like test case generators. A fault-based mutation testing approach is proposed to check if connectors are correctly implemented with respect to the Reo model. The design model and the test case generator are implemented in Tom.

In future work, we will investigate the semantic model of timed connectors [8] and probabilistic connectors [11], and build links between these models and the model in this paper. This will make it possible to reason about some properties of the connectors in one model, given the design semantics in the other. On the other hand, we will investigate the relationship between the UTP semantics and other semantics of Reo that have been developed, and extend the UTP model to treat the
inherent dynamic topology and mobility in “full” Reo. We also plan to encode the UTP model into theorem provers to prove properties of connectors based on their UTP semantics. The development of refinement laws for connectors like in Fig. 6 and integration of these laws into our existing tools for Reo [1] are in our scope as well. Furthermore, the predicates used in UTP provide a possible symbolic representation of coloring for connectors [14], and thus make it possible to synthesize connectors from specifications more efficiently.

Acknowledgements

The authors are indebted to our colleagues, especially Milad Niqui and Christel Baier for helpful discussions, and Jifeng He for the constructive comments on an earlier version of the paper which helped to simplify the model and improved this paper. The work reported 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), and the DFG-NWO-project SYANCO.

References