Connectors as designs: Refinement and test case generation using Z3

Sihan Wu[1] and Xueyi Tan[2]

[1] Yuanpei College, Peking University, No.5 Yiheyuan Road, Haidian District, Beijing 100871, China,
2300017743@stu.pku.edu.cn,
[2] School of Mathematical Sciences, Peking University, No.5 Yiheyuan Road, Haidian District, Beijing 100871, China,
2300010816@stu.pku.edu.cn

Abstract. test. test.

Keywords: test, test, test

1 Introduction

2 Preliminaries

2.1 The coordination language Reo

Reo is a channel-based exogenous coordination language where complex coordinators, called connectors, are compositionally built out of simpler ones [3]. Exogenous coordination imposes a purely local interpretation on each inter-component communication, engaged in as a pure I/O operation on each side, that allows components to communicate anonymously, through the exchange of untargeted passive data.

Complex connectors in Reo are organized in a network of primitive connectors with well-defined behavior, called *channels*, such as synchronous channels, FIFO channels, etc. A connector provides the protocol that controls and organized the communication, synchronization and cooperation among the components/services that they interconnect. Each channel has two *channel ends*: *source* ends and *sink* ends. A source channel end accepts data into the channel, and a sink channel end dispenses data out of the 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. Some simple channel types in Reo are shown in sectionModeling Connectors.

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 Src(A) and 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 nondeterministically selects and takes a suitable data item offered by one of its coincident sink channel ends and replicates it into all of its coincident source channel ends. Note that a component cannot connect to, take from, or write to mixed nodes.

2.2 The UTP observational model

Specification and unification phases can be observed in every scientific discipline. During specification sicentists 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 unifying theory that clearly and convincingly exlains a broader range of phenomena. A proposed unification of theories often receives spectacular confirmation and reward complementary to 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 [6] 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 theory of UTP designs and the observational model for connectors briefly. More details about UTP can be found in [6].

A theory of designs 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 execution. 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 varaibles *ok*, *ok'*: **Boolean**. The variable *ok* stands for a successful initialization and the start of a communication. When *ok* is **false**, the communication has not started, so no observation can be made. The variable *ok'* denotes the observation that the communication has either terminated or reached an intermediate stable state. The communication is divergent when *ok'* is **false**.

In our semantic model, the observational semantics for a Reo connector is described by a design, i.e., a relation expressed as $P \vdash Q$, where P is the predicate specifying the relationship among the observations on the source nodes of the connector, and Q is the predicate specifying the condition that should be satisfied by the observations on the sink nodes of the connector. Such a design $P \vdash Q$ is defined as follows:

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

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

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 $P \lhd b \rhd Q$ 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:*

$$P \lhd b \rhd Q =_{df} (\textbf{\textit{true}} \vdash (b \wedge P \vee \neg b \wedge Q))$$

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 ($out_\alpha P$) and the input alphabet of Q ($in_\alpha Q$) must be the same.

Definition 3. *Let* $out_\alpha P = \{v'\}$, $in_\alpha Q = \{v\}$, *then*

$$P(in : u; out : v'); Q(in : v; out : w) =_{df} \exists v_0 \cdot P(in : u; out : v_0) \wedge Q(in : v; out : w)$$

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

$$(P_1 \vdash Q_1) \lhd b \rhd (P_2 \vdash Q_2) = ((P_1 \lhd b \rhd P_2) \vdash (Q_1 \lhd b \rhd Q_2))$$
$$(P_1 \vdash Q_1); (P_2 \vdash Q_2) = (P_1 \wedge \neg(Q_1; \neg P_2) \vdash (Q_1; Q_2))$$

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. $[(P_1 \vdash Q_1) \sqsubseteq (P_2 \vdash Q_2)]$ ***iff*** $[P_1 \Rightarrow P_2] \wedge [P_1 \wedge Q_2 \Rightarrow Q_1]$

The theory of designs forms a complete lattice, with miracle $\top_{\mathbf{D}}$ as the top element, and abort $\bot_{\mathbf{D}}$ as the bottom element.

$$\top_{\mathbf{D}} =_{df} (\textbf{true} \vdash \textbf{false}) \text{ and } \bot_{\mathbf{D}} =_{df} (\textbf{false} \vdash \textbf{true})$$

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

$$(P_1 \vdash Q_1) \sqcap (P_2 \vdash Q_2) = (P_1 \wedge P_2 \vdash Q_1 \vee Q_2)$$
$$(P_1 \vdash Q_1) \sqcup (P_2 \vdash Q_2) = (P_1 \vee P_2 \vdash ((P_1 \Rightarrow Q_1) \wedge (P_2 \Rightarrow Q_2)))$$

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.

Observational model for connectors Connectors describe the coordination among components/services. We use in_R and out_R to denote what happens on the source nodes and the sink nodes of a connector $\mathbf{R}$, respectively, instead of using unprimed variables for initial observations (inputs) and primed variables for subsequent ones (outputs) as in [6]. Thus, the alphabet, i.e., the set of all observationcapturing variables, used in this paper is different from that for a design in [6]. 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 ($\neg ok \wedge out_{\mathbf{R}} = \langle d, 1\rangle$). 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 $\mathbf{R}$, the relevant observations come in pairs, with one observation on the source nodes of $\mathbf{R}$, and one observation on the sink nodes of $\mathbf{R}$. For every node N, the corresponding observation on N is given by a (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 $\mathbb{R}_+^*$ be the set of non-negative real numbers, which in the present context can be used to represent time moments.[3] For a sequence s, we use $|s|$ to denote the length of s, and if s is an infinite sequence, then $|s| = \infty$. Let $\mathbb{R}_+^*$ be the set of sequences $a = (a(0), a(1), a(2), \ldots)$ over $\mathbb{R}_+$, and for all $a = (a(0), a(1), a(2), \ldots)$ and $b = (b(0), b(1), b(2), \ldots)$ in $\mathbb{R}_+^*$, if $|a| = |b|$, then

$$\begin{aligned} a < b \quad &\text{iff} \quad \forall 0 \leq n < |a|.a(n) < b(n) \\ a \leq b \quad &\text{iff} \quad \forall 0 \leq n < |a|.a(n) \leq b(n) \end{aligned}$$

For a sequence $a = (a(0), a(1), a(2), \ldots) \in \mathbb{R}_+^*$, and $t \in \mathbb{R}_+$, $a[+t]$ is a sequence defined as follows:

$$a[+t] = (a(0) + t, a(1) + t, a(2) + t, \ldots)$$

[3] 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 FIFO1 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.

Furthermore, the element $a(n)$ in a sequence $a = (a(0), a(1), a(2), \ldots)$ can also be expressed in terms of derivatives $a(n) = a^{(n)}(0)$, where $a^{(n)}$ is defined by

$$a^{(0)} = a, a^{(1)} = (a(1), a(2), \ldots), a^{(k+1)} = (a^{(k)})^{(1)}$$

The set TS of time sequences is defined as

$$\begin{aligned} TS = \{a \in \mathbb{R}_+^* \mid & (\forall 0 \leq n < |a|.a(n) < a(n+1)) \\ & \wedge (|a| = \infty \Rightarrow \forall t \in \mathbb{R}_+^*.\exists k \in \mathbb{N}.a(k) > t)\} \end{aligned}$$

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 $\overrightarrow{a}$ denote the reverse and the tail of a, respectively, defined as:

$$a^R = \begin{cases} () & \text{if } a = () \\ (a')^{R^\frown}(a(0)) & \text{if } a = (a(0))^\frown a' \end{cases}$$

$$\overrightarrow{a} = \begin{cases} () & \text{if } a = () \\ a' & \text{if } a = (a(0))^\frown a' \end{cases}$$

where $^\frown$ 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 α and a time sequence a with $|\alpha| = |a|$. Similar to the discussion in [4], timed data sequences can be alternatively and equivalently defined as (a subset of) $(D \times \mathbb{R}_+)^*$ because of the existence of the isomorphism

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

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.

2.3 Z3

Z3 [10] if an efficient SMT (Satisfiability Modulo Theories) solver freely available from Microsoft. It has been used in various software verification and analysis applications. Z3 expands to deciding the satisfiability (or dully the validity) of first order formulas with respect to combinations of theories such as: arithmetic, bit-vectors, arrays, and uninterpreted functions. Given the data and time constraints of connectors, it allows us to verify the satisfiability of properties or refinement relations. Z3 provides bindings for several programming languages. In this paper, we use *Z3 python-bindings* to construct the models and carry out refinement checking.

3 Modeling Connectors in UTP

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 [4]. We use $\mathcal{D}$ 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 $\mathcal{D}$. Then, every connector $\mathbf{R}$ can be represented by the design $P(in_{\mathbf{R}}) \vdash Q(in_{\mathbf{R}}, out_{\mathbf{R}})$ as follows:

$$\begin{aligned} \mathbf{con} &: \mathbf{R}(in : in_{\mathbf{R}}; out : out_{\mathbf{R}}) \\ \mathbf{pre} &: P(in_{\mathbf{R}}) \\ \mathbf{post} &: Q(in_{\mathbf{R}}, out_{\mathbf{R}}) \end{aligned}$$

where $\mathbf{R}$ is the name of the connector, $P(in_{\mathbf{R}})$ is the precondition that should be satisfied by inputs $in_{\mathbf{R}}$ on the source nodes of $\mathbf{R}$, and $Q(in_{\mathbf{R}}, out_{\mathbf{R}})$ is the post-condition that should be satisfied by outputs $out_{\mathbf{R}}$ on the sink nodes of $\mathbf{R}$.[4] Let $\mathcal{N}_{in}$ and $\mathcal{N}_{out}$ be the set of source and sink node names of $\mathbf{R}$, respectively, then $in_{\mathbf{R}}$ and $out_{\mathbf{R}}$ are defined as the following mappings from the corresponding sets to TDS:

$$\begin{aligned} in_{\mathbf{R}} &: \mathcal{N}_{in} \to TDS \\ out_{\mathbf{R}} &: \mathcal{N}_{out} \to TDS \end{aligned}$$

3.1 Basic connectors

We first develop the design model for a set of basic Reo connectors, i.e., channels.

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

$$\begin{aligned} \mathbf{con} &: \mathbf{Sync}(in : (A \mapsto \langle \alpha, a \rangle); out : (B \mapsto \langle \beta, b \rangle)) \\ \mathbf{pre} &: \mathcal{D}\langle \alpha, a \rangle \\ \mathbf{post} &: \mathcal{D}\langle \beta, b \rangle \wedge \beta = \alpha \wedge b = a \end{aligned}$$

FIFO1: The simplest form of an asynchronous channel is a FIFO channel with one buffer cell, which is denoted as **FOFI1**. The FIFO1 channel whose buffer is not full accepts input without immediately outputting it. The accepted data item

[4] Note that ok and 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 $\mathbf{pre} \vdash \mathbf{post}$. According to Definition 1, it means $ok \wedge \mathbf{pre} \Rightarrow ok' \wedge \mathbf{post}$.

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 $(\overrightarrow{b^R})^R < \overrightarrow{a}$ to represent the relationship between the time moments for outputs and their corresponding next inputs. This notation can be simplicified to $b < \overrightarrow{a}$ if we consider infinite sequences of inputs and outputs.[5]

$$\begin{aligned}
&\textbf{con} : \textbf{FIFO1}(in : (A \mapsto \langle \alpha, a \rangle); out : (B \mapsto \langle \beta, b \rangle)) \\
&\textbf{pre} : \mathcal{D}\langle \alpha, a \rangle \\
&\textbf{post} : \mathcal{D}\langle \beta, b \rangle \wedge \beta = \alpha \wedge a < b \wedge (\overrightarrow{b^R})^R < \overrightarrow{a}
\end{aligned}$$

FIFO1e: On the other hand, for the FIFO1 channel $A - \boxed{\text{e}} \rightarrow B$ 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

$$\begin{aligned}
&\textbf{con} : \textbf{FIFO1}(in : (A \mapsto \langle \alpha, a \rangle); out : (B \mapsto \langle \beta, b \rangle)) \\
&\textbf{pre} : \mathcal{D}\langle \alpha, a \rangle \\
&\textbf{post} : \mathcal{D}\langle \beta, b \rangle \wedge \beta = (e)^\frown \alpha \wedge a < \overrightarrow{b} \wedge (\overrightarrow{b^R})^R < a
\end{aligned}$$

SyncDrain: An exotic channel in Reo is the synchronous drain channel $A \rightarrow\!\!\leftarrow B$ that has two source ends. Because a drain has no sink end, no data value can ever be obtained from this channel. Thus, all data items written to this channel are lost. A synchronous drain accepts a data item through one of its ends iff a data item is also available for it to simultaneously accept through another end, the predicate **true** in the post-condition means the communication terminates.

$$\begin{aligned}
&\textbf{con} : \textbf{SyncDrain}(in : (A \mapsto \langle \alpha, a \rangle); out : (B \mapsto \langle \beta, b \rangle)) \\
&\textbf{pre} : \mathcal{D}\langle \alpha, a \rangle \wedge \mathcal{D}\langle \beta, b \rangle \wedge a = b \\
&\textbf{post} : \textbf{true}
\end{aligned}$$

LossySync: The lossy synchronous channel $A - \!\!\rightarrow B$ is similar to a synchronous channel, except that it always accepts all data items through its source end. If it

[5] Note that $(\overrightarrow{b^R})^R < \overrightarrow{a}$ denotes the sequence obtained by removing the nth 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 **pre** and **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.

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.

$$\begin{aligned}\textbf{con} &: \textbf{LossySync}(in : (A \mapsto \langle \alpha, a\rangle); out : (B \mapsto \langle \beta, b\rangle))\\ \textbf{pre} &: \mathcal{D}\langle \alpha, a\rangle\\ \textbf{post} &: \mathcal{D}\langle \beta, b\rangle \wedge L(\langle \alpha, a\rangle, \langle \beta, b\rangle)\end{aligned}$$

where

$$L(\langle \alpha, a\rangle,\langle \beta, b\rangle) \equiv (\beta = () \wedge b = ()) \vee \left(a(0) \leq b(0) \wedge \begin{cases} \alpha(0) = \beta(0) \wedge L(\langle \overrightarrow{\alpha}, \overrightarrow{a}\rangle, \langle \overrightarrow{\beta}, \overrightarrow{b}\rangle) & \text{if } a(0) = b(0) \\ L(\langle \overrightarrow{\alpha}, \overrightarrow{a}\rangle, \langle \beta, b\rangle) & \text{if } a(0) < b(0) \end{cases} \right)$$

Filter: The filter channel $A - \{p\} \rightarrow B$ 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.

$$\begin{aligned}\textbf{con} &: \textbf{Filter}p(in : (A \mapsto \langle \alpha, a\rangle); out : (B \mapsto \langle \beta, b\rangle))\\ \textbf{pre} &: \mathcal{D}\langle \alpha, a\rangle\\ \textbf{post} &: \mathcal{D}\langle \beta, b\rangle \wedge F(\langle \alpha, a\rangle, \langle \beta, b\rangle)\end{aligned}$$

where $F(\langle \alpha, a\rangle, \langle \beta, b\rangle) \equiv$

$$\begin{cases} \beta = () \wedge b = () & \text{if } \alpha = () \wedge a = () \\ \beta(0) = \alpha(0) \wedge b(0) = a(0) \wedge F(\langle \overrightarrow{\alpha}, \overrightarrow{a}\rangle, \langle \overrightarrow{\beta}, \overrightarrow{b}\rangle) & \text{if } \alpha(0) \in p \\ F(\langle \overrightarrow{\alpha}, \overrightarrow{a}\rangle, \langle \beta, b\rangle) & \text{if } a(0) \notin p \end{cases}$$

p-Producer: The p-Producer channel $A \text{—}p\text{—}\!\!\rightarrow B$ 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.

$$\begin{aligned}\textbf{con} &: \textbf{Producer}p(in : (A \mapsto \langle \alpha, a\rangle); out : (B \mapsto \langle \beta, b\rangle))\\ \textbf{pre} &: \mathcal{D}\langle \alpha, a\rangle\\ \textbf{post} &: \mathcal{D}\langle \beta, b\rangle \wedge \beta \in p^* \wedge b = a\end{aligned}$$

AsynSpout: The asynchronous spout channel $A \leftarrow\!\!\|\!\!\rightarrow B$ 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.

$$\begin{aligned}\textbf{con} &: \textbf{AsynSpout}(in : (A \mapsto \langle \alpha, a\rangle); out : (B \mapsto \langle \beta, b\rangle))\\ \textbf{pre} &: \textbf{true}\\ \textbf{post} &: \mathcal{D}\langle \alpha, a\rangle \wedge \mathcal{D}\langle \beta, b\rangle \wedge |a| = |b| \wedge a \bowtie b\end{aligned}$$

where $\bowtie \subseteq TS \times TS$ is defined as

$$a \bowtie b \equiv a = () \vee b = () \vee \left(a(0) \neq b(0) \vee \begin{cases} \overrightarrow{a} \bowtie b & \text{if } a(0) < b(0) \\ a \bowtie \overrightarrow{b} & \text{if } b(0) < a(0) \end{cases} \right)$$

The **asynchronous drain** channel $A \rightarrow\leftarrow B$ inputs two sequences of data items at its two input (source) ends, but the data items on the two ends are never delivered at the same time. Moreover, the **synchronous spout** channel $A \leftarrow\mapsto B$ has two sink ends, and outputs data sequence simultaneously, i.e., the two ends must deliver data items 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.

3.2 Probabilistic connectors

The specification of channels with probabilistic behavior can be captured by the disjunction or conjunction of different predicates about time and data distributions. We consider four types of probabilistic channels: *message-corrupting synchronous channel*, *randomized synchronous channel*, *probabilistic lossy synchronous channel*, and *faulty FIFO1 channel*. Specifications of other primitive channels are ignored here and can be found at [2].

CptSync: The message-corrupting synchronous channel $A \xrightarrow{p} B$ is a synchronous channel which has an extra parameter p compared with the primitive synchronous channel. The delivered message can be corrupted with probability p. Hence, if a data item flows into the channel through the source end, then the correct data value will be obtained at the sink end with probability $1 - p$ and a corrupted data value $\perp$ will be obtained with probability p.

RandomSync: The randomized synchronous channel $A \xrightarrow{rand(0,1)} B$ can generate a random number $b \in \{0, 1\}$ with equal probability when it is activated through an arbitrary write operation on its source end, and this random number will be taken on the sink end simultaneously.

ProbSync: The message transmitted by the probabilistic synchronous channel $A -\!\!\overset{q}{-}\!\!\rightarrow B$ can get lost with a certain probability q. It can also act like a *Sync* channel and the message will be delivered successfully with probability $1 - q$.

FaultyFIFO1: The messages flowing into a faulty FIFO1 channel $A \cdot\!\overset{r}{\cdot}\!\cdot \square \rightarrow B$ can get lost with probability r when it is inserted into the buffer. In this case, the buffer remains empty. It can also behave as a normal *FIFO1* channel when the insertion of data into the buffer is successful with probability $1 - r$.

Another kind of faulty FIFO1 channel named LossyFIFO1 channel $A \rightarrow \overset{r}{\square} \cdots B$ works perfectly on the insertion of data item into its buffer but may loose messages from the buffer before being taken on the sink end. The difference between this channel and FtyFIFO1 is that the loss of data items happens in different steps. Loss behavior in this channel arises in the process of being taken from the buffer, while loss behavior in FtyFIFO1 arises in the process of storage into the buffer.

3.3 Timer connectors

We now describe the design models of a few timer channels in Reo that can serve to measure the time between two events and produce timeout signals.

t-Timer: The source end of a t-timer channal $A - \overset{t}{\circ} \rightarrow B$ accepts any input value d and returns on its sink end B a timeout signal after a delay of t time units.

$$\begin{aligned} \textbf{con} &: \textbf{Timer}t(in : (A \mapsto \langle \alpha, a \rangle); out : (B \mapsto \langle \beta, b \rangle)) \\ \textbf{pre} &: \mathcal{D}\langle \alpha, a \rangle \wedge a[+t] \leq \overrightarrow{a} \\ \textbf{post} &: \mathcal{D}\langle \beta, b \rangle \wedge \beta \in \{timeout\}^* \wedge b = a[+t] \end{aligned}$$

Off-Timer The t-timer with the off-option $A - \ominus \textcircled{t} \rightarrow B$ allows the timer to be stopped before the expiration of its delay when a special "*off*" value is consumed through its source end.

$$\begin{aligned} \textbf{con} &: \textbf{OFFTimer}t(in : (A \mapsto \langle \alpha, a \rangle); out : (B \mapsto \langle \beta, b \rangle)) \\ \textbf{pre} &: \mathcal{D}\langle \alpha, a \rangle \wedge \forall i < |a|.(a[+t](i) \leq \overrightarrow{a}(i) \vee \overrightarrow{a}(i) = off) \\ \textbf{post} &: \mathcal{D}\langle \beta, b \rangle \wedge \beta \in \{timeout\}^* \wedge OFF(\langle \alpha, a \rangle, \langle \beta, b \rangle) \end{aligned}$$

where $OFF(\langle \alpha, a \rangle, \langle \beta, b \rangle) \equiv$

$$\begin{cases} \langle \beta, b \rangle = \langle (), () \rangle & \text{if } \langle \alpha, a \rangle = \langle (), () \rangle \\ OFF(\langle \overrightarrow{\alpha}, \overrightarrow{a} \rangle, \langle \beta, b \rangle) & \text{if } a(1) < a(0) + t \wedge \alpha(1) = off \\ OFF(\langle \overrightarrow{\alpha}, \overrightarrow{a} \rangle, \langle \overrightarrow{\beta}, \overrightarrow{b} \rangle) \wedge \beta(0) = a(0) + t & \text{if } a(0) + t \leq a(1) \vee |a| = 1 \end{cases}$$

Reset-Timer The reset-timer $A \circlearrowleft \textcircled{t} \rightarrow B$ allows the timer to be reset to 0 after it has been activated when a special "reset" value is consumed through its source end.

$$\begin{aligned} \textbf{con} &: \textbf{RSTTimer}t(in : (A \mapsto \langle \alpha, a \rangle); out : (B \mapsto \langle \beta, b \rangle)) \\ \textbf{pre} &: \mathcal{D}\langle \alpha, a \rangle \wedge \forall i < |a|.(a[+t](i) \leq \overrightarrow{a}(i) \vee \overrightarrow{a}(i) = reset) \\ \textbf{post} &: \mathcal{D}\langle \beta, b \rangle \wedge \beta \in \{timeout\}^* \wedge RT(\langle \alpha, a \rangle, \langle \beta, b \rangle) \end{aligned}$$

where $RT(\langle \alpha, a \rangle, \langle \beta, b \rangle)$ is similar to the definition of OFF except that it changes the prerequisite of the second line into $a(1) < a(0) + t \wedge \alpha(1) = reset$.

Expire-Timer The t-timer with early expiration $A \overset{\mathbb{P}}{\textcircled{t}}\!\!\rightarrow B$ makes the timer produce its timeout signal through its sink and reset itself when it consumes a special "expire" value through its source end.

$$\begin{aligned}\mathbf{con} &: \mathbf{EXPTimer}t(in : (A \mapsto \langle \alpha, a\rangle); out : (B \mapsto \langle \beta, b\rangle))\\ \mathbf{pre} &: \mathcal{D}\langle \alpha, a\rangle \wedge \forall i < |a|.(a[+t](i) \leq \overrightarrow{a}(i) \vee \overrightarrow{a}(i) = expire)\\ \mathbf{post} &: \mathcal{D}\langle \beta, b\rangle \wedge \beta \in \{timeout\}^* \wedge ET(\langle \alpha, a\rangle, \langle \beta, b\rangle)\end{aligned}$$

where $ET(\langle \alpha, a\rangle, \langle \beta, b\rangle) \equiv$

$$\begin{cases} \langle \beta, b\rangle = \langle (), ()\rangle & \text{if } \langle \alpha, a\rangle = \langle (), ()\rangle \\ ET(\langle \overrightarrow{\alpha}, \overrightarrow{a}\rangle, \langle \beta, b\rangle) \wedge b(0) = a(0) + t & \text{if } a(0) + t < a(1) \wedge |a| = 1 \\ ET(\langle \overrightarrow{\alpha}, \overrightarrow{a}\rangle, \langle \overrightarrow{\beta}, \overrightarrow{b}\rangle) \wedge b(0) = a(1) & \text{if } a(1) < a(0) + t \wedge a(1) = expire \end{cases}$$

4 Refinement checking and test case generation in Z3

As shown previously, modeling and reasoning of Reo connectors in UTP allows us to reason about a relatively wide range of properties. However, when dealing with verification of refinement/equivalence relations between connectors, it is only capable of proving that one connector is indeed a refinement of the other one but cannot automatically provide a counter example when one connector is not a refinement of the other connector. Moreover, the refinement relation is a fairly important part of properties, which has been investigated in many applications. When we fail to find a proof of the existence of refinement relations between connectors, we propose to use Z3, an SMT solver, to search for bounded counter examples. If a counter example is successfully located, it means that there is no refinement relation between the connectors.

4.1 Formalization of channels in Z3

We first need to implement the construction of connectors and also define the refinement checking function in a way that fits Z3. Typically, connector construction starts with the formal definition of channels and composition operators, which will be used hereafter to build and assess more complex models. Such a framework must be carefully developed so that the further model assessment can be simplified as much as possible.

Coinciding with the definition of channels in UTP, we dealt with both time and data relation constraints in the basic definition in Z3. A table with correspondence between basic definitions in UTP and Z3 can be found at [1].

An auxiliary function `Conjunction` is used to take the conjunction of every constraint in the constraint lists param-eterized by constraints. For example, the specific behavior constraints for *Sync* channel are classified into two types: data constraints and time constraints. The definition of *Sync* provides the behavior pattern it needs to follow: each item in timed data streams of output specified as `node[1]` is equivalent to the timed data items of input specified as `node[0]`,

which are exactly data and time constraints. As they are both requirements that we need to satisfy no matter data or time is related, each constraint in the list will all be combined finally.

Since we have already defined the behavior constraints for channels with UTP, now we take the *Sync* channel as an example to show the code of defining them in Z3.

```
def Sync(nodes, bound)
    assert len(nodes) == 2
    constraints = []
    for i in range(bound):
        constraints += [nodes[0]['data'][i]==nodes[1]['data'][i]]
        constraints += [nodes[0]['time'][i]==nodes[1]['time'][i]]
    return Conjunction(constraints)
```

SyncDrain channel is defined in a similar way. On the basis of the behavior of *SyncDrain* channel, only time related constraints are needed, i.e., equivalence of corresponding items in two time streams of the two inputs.

The remaining basic channels are omitted here. For the construction of probabilistic channels, we adopted a method that involves generating a random number between 0 and 1 and comparing it with the corresponding probability. Specifically, for the *CptSync* channel, given the probability p of data corruption, we generate a random float number q. If q is less than p, the data is judged as corrupted; otherwise, the data is successfully transmitted.

4.2 Composing operators

While the basic channels are already well defined as the basis of the whole construction framework, the composition operators also need to be specified for large-scale connectors' construction just like cement for bricks. There are three kinds of composition operators: *replicate*, *flow-through*, and *merge*.

Both *replicate* and *flow-through* perators can be implicitly achieved when we compose channels to construct connectors using same node names. A simple example for the implementation of *flow-through* is `c1.connect('Sync', 'A', 'E')`, `c1.connect('Fifo1', 'E', 'F')`, `c1.connect('Fifo1', 'E', 'G')` where the message that node E receives from A is duplicated and sent to F and G simultaneously. As for *Flow-through* operator, a simple example is `c2.connect('Fifo1','A','D')`, `c2.connect('Sync', 'D', 'B')` where the timed data stream flows from node A to node B through the mixed node D. In these two examples, `c1` and `c2` both belong to the `Connector` class which will be elaborated in detail later.

Among the three types of composition operators, the most difficult one is *Merge*, which is specially dealt with as a channel *Merger* in our model. Together with the other two kinds of composition operators: *replicate* and *flow-through*, they provide a complete basis for the construction of any connector, coinciding with the original set of composition operations [3], [9]. For each item in the output

stream of the *Merger* channel, there is a non-deterministic choice between the two input nodes. Hence, the *Merger* channel also needs to be defined in a recursive way like the *LossySync* channel. Note that the constraints also conform well to the original semantics of the *merge* operator. Derived from the two recursive definitions, an additional advantage of defining *LossySync* and *Merger* channel in this way is that it can preserve the behavioral pattern to a great extent without any assumption such as priorities of reading or taking data from the two input streams.

4.3 Refinement and test cases for connectors

Refinement and equivalence A refinement relation between connectors which allows developing connectors systematically in a step-wise fashion, may help to bridge the gap between requirements and the final implementations. The notion of refinement has been widely used in different system descriptions. For example, in data refinement [5], the 'concrete' model is required to have *enough redundancy* to represent all the elements of the 'abstract' one. This is captured by the definition of a surjection from the former into the latter (the *retrieve map*). If models are specified in terms of pre and post-conditions, the former are weakened and the latter strengthened under refinement [7]. In process algebra, refinement is usually discussed in terms of several 'observation' preorders, and most of them justify transformations entailing *reduction of nondeterminism* (see, for example, [12]). For, the refinement relation can be defined as in [9], where a proper refinement order over connectors has been established based on the implication relation on predicates.

Connector Q is a refinement of another connector P if the behavior property of P can be derived from hypothesis Q, i.e., the behavior property of connector Q (denoted by $P \sqsubseteq Q$). Two connectors are equivalent if each one of them is a refinement of the other.

Definition 5 (Refinement). *For two connectors* $\boldsymbol{R}_1$ *and* $\boldsymbol{R}_2$

$$\begin{aligned} \textbf{con} &: \boldsymbol{R}_i(in : in_{\boldsymbol{R}_i}; out : out_{\boldsymbol{R}_i}) \\ \textbf{pre} &: P_i(in_{\boldsymbol{R}_i}) \\ \textbf{post} &: Q_i(in_{\boldsymbol{R}_i}, out_{\boldsymbol{R}_i}) \end{aligned}$$

where $i = 1, 2$*, then*

$$\boldsymbol{R}_1 \sqsubseteq \boldsymbol{R}_2 =_{df} (P_1 \Rightarrow P_2) \wedge (P_1 \wedge Q_2 \Rightarrow Q_1)$$

Test cases for connectors In this section, we discuss the problem of *fault-based test case* for component connectors with Reo as our target implementation language. 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.

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* $\boldsymbol{R}(in : in_{\boldsymbol{R}}; out : out_{\boldsymbol{R}})$*, 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* $in_{\boldsymbol{R}}$ *and* $out_{\boldsymbol{R}}$ *respectively. A deterministic test case for* $\boldsymbol{R}$ *is defined as*

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

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* $\boldsymbol{R}(in : in_{\boldsymbol{R}}; out : out_{\boldsymbol{R}})$*, 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* $in_{\boldsymbol{R}}$ *and* $out_{\boldsymbol{R}}$ *respectively. A deterministic test case for* $\boldsymbol{R}$ *is defined as*

$$t_d(in_{\boldsymbol{R}}, out_{\boldsymbol{R}}) = (in_{\boldsymbol{R}} = i) \vdash out_{\boldsymbol{R}} \in O$$

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 the implementation is correct, then it should pass the test cases.

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 8 (Fault-Adequeate Test Case). *Let* $t_d(in_{\boldsymbol{R}}, out_{\boldsymbol{R}})$ *be a test case (which can be either deterministic or non-deterministic),* $\boldsymbol{R}$ *an expected connector, and* $\boldsymbol{R}$*' its faulty implementation. Then* t *is a fault-adequate test case if and only if*

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

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.

4.4 Refinement checking

With the above basic channels and composition operators, the next step is to construct Reo connectors in this framework. In our model, `Connector` is defined as a class, which provides a method for constructing complex connectors out of the basic channels and composition operators. The statements inside the class definition are function definitions.

The function `isRefinementOf` is used to express refinement checking. The result type of the function of Boolean, which shows whether or not a connector is a refinement of another. According to the definition of refinement relation in Section 4.3, we have a formula that represents it properly: $P \sqsubseteq Q$ if and only if the behavior property of connector P can be derived from the property of connector Q, i.e., $Q \rightarrow P$. If Q is indeed a refinement of P, it can be rephrased as $\neg Q \vee P$. If Z3 solver presents a model satisfying the negation of this formula, then the refinement relation is falsified by a counter example. On the contrary, if there doesn't exist a counter example within a given bound, we say that the refinement relation, $P \sqsubseteq Q$, holds within the bound. The formal definition of `isRefinementOf` is given in Algorithm 1. The complete definition of the function can be found at [1].

Algorithm 1 cannot produce any spurious counter examples, i.e. all the counter examples of the bounded model located by the algorithm are also correct counter examples of the original model. To illustrate this conclusion, first we assume the bound we set is b, for any channel, the constraints added to Z3 solver are bounded, i.e., only constraints on the prefixes of timed data streams are fed to Z3. Let Σ be the set of all constraints of connectors and σ be the set of bounded ones. If there exists a counter example π (π is a set of timed data streams' prefixed corresponding to the nodes) that fails to satisfy the weaker constraints: $\pi \not\models \sigma$, then we can deduce that $\pi \not\models \Sigma$. Let π be a prefix of the full timed data stream Π and $\Pi \setminus \pi$ satisfies all the left constraints $\Sigma \setminus \sigma$. A valid prefix leads to a valid timed data stream, hence we have $\Pi \not\models \Sigma$. Thus, the counter examples generated by Algorithm 1 are always sound and genuine.

4.5 Implementation

We use test cases to validate the correctness of our method, which can be found at [1]. Here we choose three representative examples to show how the system works.

There is one thing worth mentioning. In most cases, we intend to check the refinement relation just in terms of the input and output nodes of connectors, i.e., we view the connectors as black boxes. Therefore, when constructing connectors in Z3, we need to ensure the internal nodes obtain different node names so that the algorithm will ignore the distinction inside the connectors. Nevertheless, sometimes we may want to look into the internal behavior of connectors. In this case, we can name the internal nodes the same to observe the refinement relation including the hidden parts.

Algorithm 1: Q.isRefinementOf(P, bound).

Input: Both P and Q are connectors.
Output: A boolean result: `True` or `False` with a counter example

```
solver ← create a Z3 Solver instance;
nodes ← {};
for ch← channels in Q do
    for n ← channel ends in ch do
        if n ∉ nodes then
            tds_n ← {time : [n_t_0, ··· , n_t_(bound − 1)], data :
              [n_d_0, ··· _(bound − 1)]};
            nodes[n]← tds_n;
            add time constraints n_t_0 ≥ 0 ∧ n_t_i < n_t_i + 1 to solver
    add channel-specific constraints to solver according to the definition in
      Section 4.1
foralls ← {};
absGlobalConstr ← {};
absTimeConstr ← {};
for ch← channels in P do
    for n ← channel ends in ch do
        if n ∉ nodes∪ foralls then
            tds_n ← {time : [n_t_0, ··· , n_t_(bound − 1)], data :
              [n_d_0, ··· _(bound − 1)]};
            foralls[n]← tds_n;
            add time constraints n_t_0 ≥ 0 ∧ n_t_i < n_t_i + 1 to
              absTimeConstr
    add channel-specific constraints to absTimeConstr according to the
      definition in Section 4.1
absGlobalConstr← ¬(absTimeConstr∩absGlobalConstr);
let foralls be:
    {n_1, ··· , n_m}, add the following constraint to solver
      (∀n_1_t_0) ··· (∀n_1_t_(bound−1))(∀n_1_d_0) ··· (∀n_1_d_(bound−1)) ···
      (∀n_m_t_0) ··· (∀n_m_t_(bound − 1))(∀n_m_d_0) ··· (∀n_m_d_(bound −
      1))absGlobalConstr
solver.check()
```

Example 1. Fig.1 contains four similar connectors, the purpose of which is to output the data flow simultaneously from two sink nodes with a delay. Graph `b`, `c`, and `d` all complete the task perfectly by using a *SyncDrain* channel to synchronize the two flows or just making the delay happens in the same *FIFO1* channel. However, graph `a` fails to achieve the goal since the node B and C are not constrained to be simultanuous. Hence, `b`, `c`, and `d` are equivlent with each other, and are all refinements of `a`.

In our Z3 implementation, we can easily construct these four connectors and check their refinement relations. Graph `a` and `b` can be formalized into

```
c1 = ['Sync A D', 'Sync D E', 'Sync D F', 'Fifo1 E B', 'Fifo1 F C']
c2 = ['Sync A D', 'Sync D E', 'Sync D F', 'Fifo1 E G', 'Fifo1 F H',
      'SyncDrain G H','Sync G B', 'Sync H C']
```

After constructing them into the `Connector` class by the function `automerger`, the result aligns with what we expect: `c1.isRefinementOf(c2, bound)` returns `False` with a counter example, while `c2.isRefinementOf(c1, bound)` returns `True`. The equivalence between `c2`, `c3`, and `c4` can also be verified in the same way.

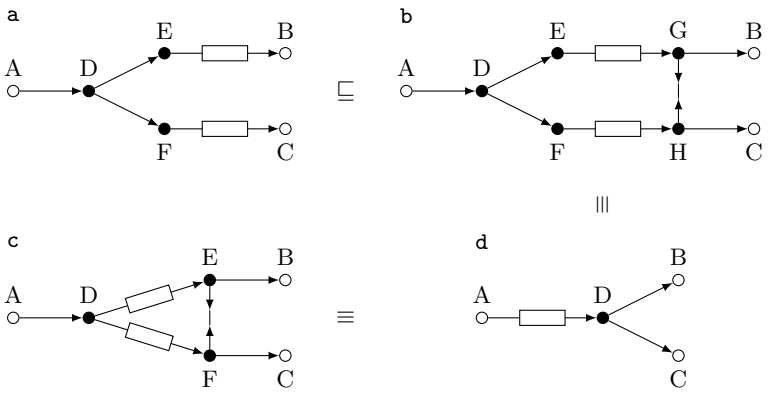
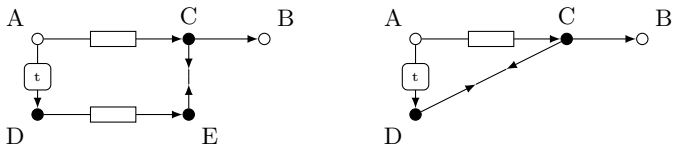


Fig. 1. Basic test case

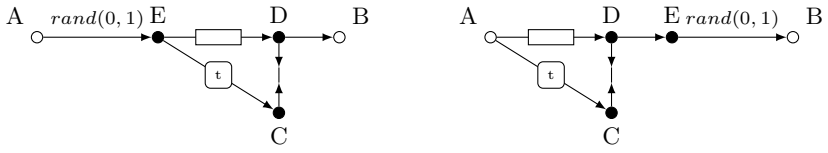
Example 2. The left graph in Fig.2 is called Lower Bounded FIFO1, which aims to output the data flow with a delay of at least t time units. The right graph is a mutated connector of the former one, since it requires the output time to be exactly t time units later than the input time. Hence in Z3 implementation, the result of `isRefinementOf` between the two connectors would be both `False` with counter examples.



Lower Bounded FIFO1 Mutated Lower Bounded FIFO1

Fig. 2. Time-based test case

Example 3. In the third example, we combine a timer channel and a probabilistic channel to form a complex connector and its mutation as shown in Fig.3. The two connectors are supposed to output a random signal between 0 and 1 after a delay of t time units. The two connectors only differ where the random signal is generated: before or after the delay. Therefore, they are equivalent, i.e., a refinement of each other. In Z3 implementation, we can verify this quivalence by verifying the `True` output of the `isRefinementOf` function in both directions.



Random tFIFO1 Mutated Random tFIFO1

Fig. 3. Probability-based test case

References

1. Package of source files., https://github.com/Rainco-S/Z3
2. The source code of probabilistic connectors., https://github.com/Zhang-Xiyue/Prob-Reo
3. Arbab, F.: Reo: A channel-based coordination model for component composition. Mathematical Structures in Computer Science 14, 329–366 (06 2004)
4. Arbab, F., Rutten, J.J.: A coinductive calculus of component connectors. In: International Workshop on Algebraic Development Techniques. pp. 34–55. Springer (2002)
5. De Roever, W.P., Engelhardt, K.: Data refinement: model-oriented proof methods and their comparison. No. 47, Cambridge University Press (1998)
6. Hoare, C.A.R.: Unified theories of programming. In: Mathematical methods in program development, pp. 313–367. Springer (1997)
7. Jones, C.B.: Systematic software development using VDM, vol. 66. Prentice Hall International Englewood Cliffs (1986)

8. Meng, S.: Connectors as designs: The time dimension. In: Margaria, T., Qiu, Z., Yang, H. (eds.) Sixth International Symposium on Theoretical Aspects of Software Engineering, TASE 2012, 4-6 July 2012, Beijing, China. pp. 201–208. IEEE (2012), http://doi.ieeecomputersociety.org/10.1109/TASE.2012.36
9. Meng, S., Arbab, F., Aichernig, B.K., Aştefănoaei, L., de Boer, F.S., Rutten, J.: Connectors as designs: Modeling, refinement and test case generation. Science of Computer Programming 77(7), 799–822 (2012), https://www.sciencedirect.com/science/article/pii/S0167642311001006, (1) FOCLASA'09 (2) FSEN'09
10. de Moura, L., Bjørner, N.: Z3: An efficient smt solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. pp. 337–340. Springer Berlin Heidelberg, Berlin, Heidelberg (2008)
11. Nawaz, M.S., Sun, M.: Using pvs for modeling and verification of probabilistic connectors. In: Hojjat, H., Massink, M. (eds.) Fundamentals of Software Engineering. pp. 61–76. Springer International Publishing, Cham (2019)
12. Roscoe, A.: The theory and practice of concurrency (1998)
13. Sun, M.: Towards formal modeling and verification of probabilistic connectors in coq xiyue zhang (2018), https://api.semanticscholar.org/CorpusID:85541303
14. Woodcock, J.: Uncertainty and probabilistic utp. In: The Practice of Formal Methods: Essays in Honour of Cliff Jones, Part II, pp. 184–205. Springer (2024)
15. Zhang, X., Hong, W., Li, Y., Sun, M.: Reasoning about connectors using coq and z3. Science of Computer Programming 170, 27–44 (2019), https://www.sciencedirect.com/science/article/pii/S0167642318304076