# 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.** Reo is a channel-based exogenous coordination language for structuring interactions in distributed systems. This paper proposes a unified framework for modeling, refinement verification, and fault-based test case generation of Reo connectors based on Unifying Theories of Programming (UTP) and the SMT solver Z3. Reo connectors, including basic, timed, and probabilistic ones, are modeled as UTP designs with explicit separation of preconditions and postconditions, making connector causality precise. Refinement is defined via predicate implication in UTP and automatically checked using Z3, which also generates fault-based test cases by searching for bounded counterexamples. To support scalable construction of complex connectors, we introduce `automerger` and `multi-merger` extensions. Experimental results show that the framework effectively supports connector modeling, automated refinement checking, and test case generation.

**Keywords:** Connector, UTP, Z3, Refinement, Test Case Generation

## 1 Introduction

With the rapid growth of service-oriented computing and distributed systems, the coordination of heterogeneous components has become a fundamental challenge. Conventional ad hoc "glue code" for component interaction lacks a rigorous semantic foundation, often resulting in ambiguous designs and error-prone implementations. Reo, a channel-based exogenous coordination language, addresses this problem by supporting the compositional construction of connectors from primitive channels such as synchronous and FIFO channels, thereby separating coordination concerns from component logic and promoting loose coupling and reusability.

The correctness of Reo connectors is crucial to the reliability of distributed systems and calls for formal verification techniques. Existing semantic and verification approaches, however, exhibit notable limitations. In particular, some semantic models do not make the causal relationship between inputs and outputs explicit, while automata-based approaches suffer from scalability issues when handling unbounded data domains or timed and probabilistic behaviors. Moreover, test case generation for Reo connectors remains insufficiently integrated with formal semantic models and automated verification tools.

Unifying Theories of Programming (UTP) offers a uniform relational framework for modeling and reasoning about programs and specifications via pre/post-condition designs. Prior work has demonstrated that Reo connectors can be interpreted as UTP designs, enabling formal reasoning about equivalence and refinement, and has further extended this approach to timed and probabilistic connectors. Nevertheless, these approaches largely rely on interactive theorem proving, which limits automation and does not readily provide counterexamples or systematic test cases for invalid refinements.

SMT solvers, such as Z3, have shown strong capabilities in automated constraint solving and bounded verification. While recent work has applied Z3 to refinement checking of Reo connectors, support for fault-based test case generation and scalable construction of complex connectors remains limited.

To overcome these limitations, this paper presents a unified UTP- and Z3-based framework for the formal modeling, refinement verification, and fault-based test case generation of Reo connectors. Our main contributions are:

- modeling basic, timed, and probabilistic Reo connectors in Z3;
- automated refinement/equivalence checking using Z3, including fault-based test case generation approach for detecting common connector design errors;
- automated support for connector construction via an automerger algorithm and a multi-merger extension.

The remainder of the paper is organized as follows. Section 2 introduces Reo, UTP designs, and Z3. Section 3 presents the UTP-based modeling of connectors. Section 4 describes Z3-based refinement checking, test case generation, and implementation details. Section 5 concludes the paper and outlines future work.

## 2 Preliminaries

### 2.1 The coordination language Reo

Reo is a channel-based exogenous coordination language in which complex coordinators, called connectors, are compositionally constructed from simpler ones [3]. Exogenous coordination enforces a local interpretation of inter-component communication as pure I/O operations, enabling anonymous interaction through the exchange of untargeted data.

In Reo, complex connectors are formed as networks of primitive connectors, called *channels*, with well-defined behavior (e.g., synchronous and FIFO channels). A connector specifies communication, synchronization, and data-flow protocols independently of component logic. Each channel has two *channel ends*, classified as *source* or *sink* ends, which respectively accept and dispense data. Channel ends may be of the same or different types, and Reo imposes no restriction on channel behavior, allowing heterogeneous channel types to coexist within a single connector. Each channel end is connected to at most one component instance at a time.

Complex connectors are constructed using the *join* and *hiding* operations. Channels are joined at nodes, each consisting of a set of coincident channel ends that are partitioned into source ends $\mathrm{Src}(A)$ and sink ends $\mathrm{Snk}(A)$. Depending on their incident channel ends, nodes are classified as *source*, *sink*, or *mixed* nodes. The hiding operation abstracts away internal nodes, making them unobservable from outside the connector. A connector is graphically represented as a *Reo circuit*, i.e., a finite graph whose nodes are labeled with disjoint, non-empty sets of channel ends and whose edges represent channels. The observable behavior of a Reo circuit is defined by the data-flow at its source and sink nodes.

## 2.2 The UTP observational model

Specification and unification are recurring phases in scientific inquiry. Specification focuses on narrowly defined phenomena and the laws governing them, while unification seeks general theories that account for a broader range of observations. Such unifying theories complement, rather than replace, the individual theories they relate. In computer science, Hoare and He introduced this unifying perspective through the Unifying Theories of Programming (UTP) [6].

Our theory of Reo connectors is motivated by this unification principle. In this section, we briefly introduce the theory of UTP designs and the observational model used for Reo connectors. Further details on UTP can be found in [6].

**A theory of designs** A UTP theory is defined by a collection of relations together with three components: an alphabet, a signature, and a set of healthiness conditions.

A direct and obvious way to represent the observable behavior of a connector is to model it as a relation on its inputs and outputs. Because the inputs/outputs take place through the nodes of the connector, sequences of data items that pass through a node together with the moments in time that the data items are observed emerge as the key building blocks for describing connector behavior.

In our semantic model, we use two auxiliary Boolean variables *ok* and *ok'* to analyze explicitly the phenomena of communication initialization and termination. The variable *ok* indicates successful initialization and the start of communication; when *ok* is **false**, no observation is possible. The variable *ok'* represents termination or the attainment of a stable intermediate state; divergence is indicated when *ok'* is **false**. The observational semantics for a Reo connector is described by a design, i.e., a relation expressed as $P \vdash Q$, where $P$ specifies constraints on observations at the source nodes and $Q$ specifies the required conditions on observations at the sink nodes. Such a design $P \vdash Q$ is formally 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:*

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

The separation of condition on inputs from condition on outputs in our model allows us to write a specification that has a more generous precondition than simply the domain of a relation used as a specification. We are allowed to assume that the condition on inputs holds, and we have to satisfy the condition on its outputs. Moreover, we can rely on the behavior of a connector having been started, but we must ensure that it terminates. If the condition on inputs does not hold, or the connector does not start its behavior, we are not committed to establish the condition on the outputs nor even to make the connector behavior terminate.

In UTP, we have the well-known property for refinement, which is established by the following definition:

**Definition 2.** $[(P_1 \vdash Q_1) \sqsubseteq (P_2 \vdash Q_2)]$ ***iff*** $[P_1 \Rightarrow P_2] \wedge [P_1 \wedge Q_2 \Rightarrow Q_1]$

Furthermore, there are a family of operators on designs, such as conditional choice, sequential composition, iteration, etc., and a lot of algebraic laws being satisfied by these operators. More detailed discussion on designs can be found in [6]. 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 $in_{\mathbf{R}}$ and $out_{\mathbf{R}}$ 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].

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 $\alpha = (\alpha(0), \alpha(1), \alpha(2), \dots)$ 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

[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.

$s$, 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:

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

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 $\alpha$ 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.

[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 $ok \wedge \mathbf{pre} \Rightarrow ok' \wedge \mathbf{post}$.

**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*.

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

**FIFO1:** The simplest form of an asynchronous channel is a FIFO channel with one buffer cell, which is denoted as FIFO1. The FIFO1 channel $A \longrightarrow\boxed{\phantom{e}}\longrightarrow B$ : 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 $(\overrightarrow{b^R})^R < \overrightarrow{a}$ 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 $A \longrightarrow\boxed{e}\longrightarrow B$ : 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 $A \rightarrowtail\!\leftarrowtail B$ : 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.

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

[5] Note that $(\overrightarrow{b^R})^R < \overrightarrow{a}$ denotes the sequence obtained by removing the $n$-th element from $a$ sequence $b$ with length $n$. For infinite sequences, we have $(\overrightarrow{b^R})^R = b$.

**LossySync:** The lossy synchronous channel $A \dashrightarrow B$ : 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.

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

where

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

**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.

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

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 $A \xrightarrow{\cancel{p}} B$ 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.

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

**AsynSpout:** The asynchronous spout channel $A \leftarrow\!\!\Vert\!\!\rightarrow B$ 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 $A \overset{r}{\cdots} \square \rightarrow B$ 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 $A \rightarrow \square \overset{r}{\cdots} B$ 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 $A{\dashv}\!\textcircled{t}\!\rightarrow B$ 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 $A \vec{\vdash}\!\textcircled{t}\!\rightarrow B$ makes the timer produce its timeout signal through its sink and reset itself when it consumes a special "expire" value through its source end.

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

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 discussed above, UTP-based modeling of Reo connectors supports reasoning about a broad class of properties. However, for refinement and equivalence verification, this approach is limited to establishing valid refinement relations and cannot automatically produce counterexamples when refinement does not hold. Since refinement plays a central role in the formal analysis of connectors, the absence of such counterexamples hinders effective diagnosis of incorrect designs. To address this limitation, we employ the SMT solver Z3 to perform bounded searches for counterexamples. The existence of a counterexample conclusively demonstrates the absence of a refinement relation between the given connectors.

### 4.1 Formalization of channels in Z3

We first implement connector construction and define refinement checking in a form amenable to Z3. Connector construction is based on formal definitions of channels and composition operators, which serve as the basis for building and analyzing more complex connectors. The framework is designed to minimize complexity in subsequent verification tasks.

Consistent with the UTP-based channel definitions, both temporal and data constraints are encoded in Z3 at the basic level. A correspondence between UTP constructs and their Z3 representations is summarized in [1].

An auxiliary function `Conjunction` is introduced to compute the conjunction of constraint sets. For example, the behavior of the *Sync* channel is specified by data and time constraints, expressing equivalence between input and output timed data streams (represented by `node[0]` and `node[1]`, respectively). Since both categories of constraints must hold simultaneously, they are combined into a single conjunctive constraint.

Having defined channel behavior in UTP, we illustrate its encoding in Z3 using the *Sync* channel as a representative example.

```
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)
```

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.

The Z3 definitions of the remaining connectors can be found in the source code [1].

## 4.2 Composing operators

While basic channels constitute the foundation of the connector construction framework, composition operators are essential for assembling large-scale connectors, playing a role analogous to binding elements. We consider three classes 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 operators, *merge* is the most involved. In our framework, it is modeled explicitly by a generalized *Merger* channel that supports an arbitrary number of input connectors, rather than being restricted to two. For each output item produced by the *Merger*, one of the available input streams is selected nondeterministically. To capture this behavior, the *Merger* channel is defined recursively, in a manner similar to the *LossySync* channel. The resulting constraints faithfully reflect the semantics of the merge operator, while avoiding additional assumptions such as fixed priorities among input streams. Together with *replicate* and *flow-through*, this generalized merge operator provides a complete and flexible basis for constructing complex Reo connectors, consistent with the original composition principles of Reo [3,9].

### 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 3 (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** Now we address fault-based test case generation for Reo connectors. Reo circuits are treated as specification models that capture the intended coordination behavior among components, while mutated circuits represent fault models describing potential coordination or protocol errors. By applying refinement checking between the specification and its mutants, counterexamples are generated that distinguish correct from faulty behaviors. These counterexamples are then used to derive executable test cases, ensuring that the corresponding faulty coordination models are not implemented. Since test cases are systematically derived from faults introduced at the specification level, the approach is referred to as fault-based.

For a given connector, a test case is defined as a specification of the expected timed data sequences at the output nodes, corresponding to prescribed timed data sequences at the input nodes.

**Definition 4 (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 5 (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 preceding discussion, test cases, specifications, and implementations can all be uniformly represented as designs. Test cases are derived as abstractions of the specification and therefore capture selected aspects of its intended behavior. A correct implementation must refine its specification; consequently, it must also refine the derived test cases. Passing all test cases is thus a necessary condition for the correctness of an implementation with respect to the specification. 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 6 (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

Based on the defined basic channels and composition operators, we construct Reo connectors within a unified framework. In our implementation, `Connector` is defined as a class that supports the construction of complex connectors from primitive channels and composition operators. The class encapsulates the corresponding construction and analysis functions.

Refinement checking is realized by the function `isRefinementOf`, which returns a Boolean value indicating whether one connector refines another. According to the refinement definition in Section 4.3, a connector $P$ refines $Q$, written $P \sqsubseteq Q$, if the behavior of $P$ is implied by that of $Q$, i.e., $Q \rightarrow P$. This condition can be encoded as $\neg Q \vee P$. If Z3 finds a model satisfying the negation of this formula, the refinement relation is disproved by a counterexample; otherwise, the refinement is considered to hold within a given bound. The formal definition of `isRefinementOf` is presented in Algorithm 1, with the full implementation available in [1].

Algorithm 1 is sound with respect to counterexample generation: all counterexamples identified within the bounded model correspond to genuine counterexamples of the unbounded model. Given a bound $b$, only constraints over prefixes of timed data streams are submitted to Z3. Let $\Sigma$ denote the full set of connector constraints and $\sigma$ their bounded counterparts. If a bounded prefix $\pi$ violates $\sigma$, then any extension of $\pi$ to a complete timed data stream necessarily violates $\Sigma$. Consequently, no spurious counterexamples are introduced by bounding, and all counterexamples produced by Algorithm 1 are valid.

### 4.5 Implementation details

In order to ensure the correctness of the implementation, we need to pay attention to some details when constructing connectors in Z3.

**Black boxes:** 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.

**Automerger:** In practical usage, a merger may combine channels of multiple types. To maintain the original functionality of each channel, a user has to add hidden nodes manually to separate the functioning part and the merging part. Fig.1 shows an example of merging a Sync channel and a FIFO1 channel. A hidden node B' should be added after the FIFO1 channel to ensure the data flows out of the FIFO1 channel before being merged.

To address this predicament, we develop an automatic merging algorithm `automerger` that enables users to input the nodes only in their original forms without manually adding hidden nodes. The input includes the basic channels and the nodes to be merged. The algorithm will then search for a merger's participant channels, automatically add a hidden node with a unique name to each one (to satisfy the black box checking), and construct a new Sync channel between the hidden node and the merged node. This algorithm simplifies the process of

**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 $n \notin$ nodes then
            $tds_n \leftarrow \{time : [n\_t\_0, \cdots, n\_t\_(bound-1)], data : [n\_d\_0, \cdots\_(bound-1)]\}$;
            nodes[n]← $tds_n$;
            add time constraints $n\_t\_0 \geq 0 \wedge n\_t\_i < n\_t\_i+1$ 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 $n \notin$ nodes∪ foralls then
            $tds_n \leftarrow \{time : [n\_t\_0, \cdots, n\_t\_(bound-1)], data : [n\_d\_0, \cdots\_(bound-1)]\}$;
            foralls[n]← $tds_n$;
            add time constraints $n\_t\_0 \geq 0 \wedge n\_t\_i < n\_t\_i+1$ to absTimeConstr
    add channel-specific constraints to absTimeConstr according to the definition in Section 4.1
absGlobalConstr← ¬(absTimeConstr∩absGlobalConstr);
let foralls be:
    $\{n_1, \cdots, n_m\}$, add the following constraint to solver
    $(\forall n_1\_t\_0) \cdots (\forall n_1\_t\_(bound-1))(\forall n_1\_d\_0) \cdots (\forall n_1\_d\_(bound-1)) \cdots (\forall n_m\_t\_0) \cdots (\forall n_m\_t\_(bound-1))(\forall n_m\_d\_0) \cdots (\forall n_m\_d\_(bound-1))$absGlobalConstr
solver.check()
```

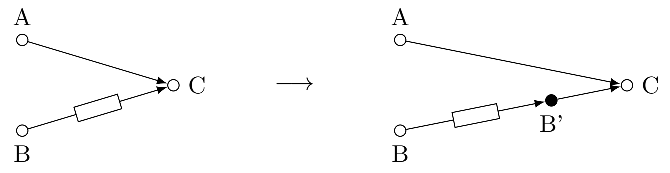


**Fig. 1.** Adding hidden node to a merger

building complex connectors involving mergers and makies the structure more intuitive and user-friendly, avoiding potential errors from manual node addition.

In addition, we upgrade the merger to a multi-merger that can merge more than two input nodes at the same time. The new merger can rearrange the data flows in a chronological order just like the original merger. The implementation of multi-merger can be found at [1]. The `automerger` algorithm works well with the multi-merger.

## 4.6 Examples of test cases

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.

*Example 1.* Fig.2 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.

*Example 2.* The left graph in Fig.3 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.

*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.4. 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

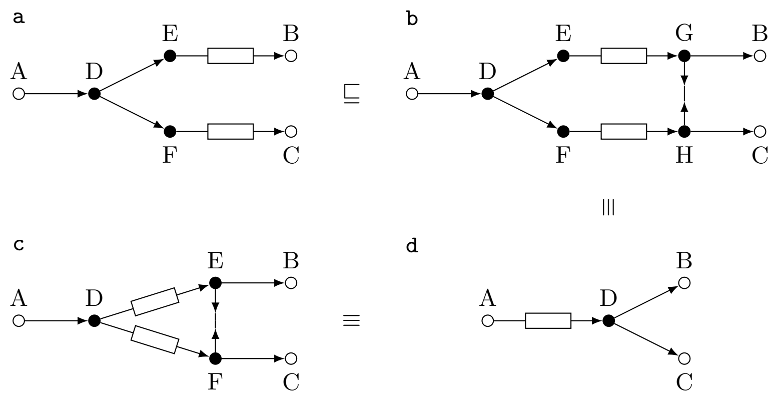


**Fig. 2.** Basic test case

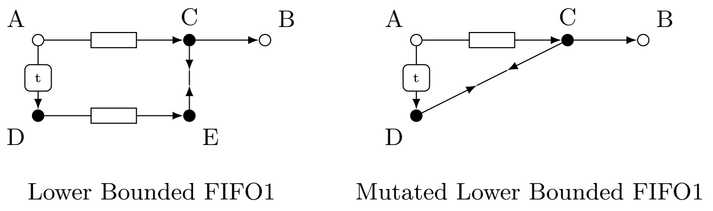
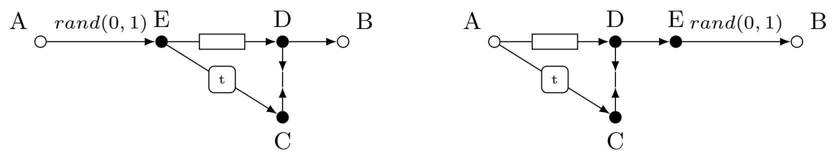


**Fig. 3.** Time-based test case

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. 4.** Probability-based test case

## 5 Conclusion and future work

This paper presents a unified framework for the formal modeling, refinement verification, and test case generation of Reo connectors by integrating UTP-based semantics with automated reasoning using Z3. The framework establishes a coherent foundation for reasoning about connector behavior, correctness, and test adequacy.

We extend the UTP design model to encompass basic, timed, and probabilistic Reo connectors. By explicitly distinguishing preconditions and postconditions, the proposed model makes input–output causality explicit and supports uniform reasoning about heterogeneous connector behaviors, including timing constraints and probabilistic effects.

The Z3-based realization enables automated refinement checking and fault-based test case generation. Refinement verification is reduced to constraint solving, allowing the automatic detection of invalid refinements through bounded counterexamples and alleviating the reliance on interactive theorem proving. Moreover, fault-based test cases are systematically derived from specifications and faulty implementations, facilitating the detection of common design errors such as incorrect channel types and topology mismatches.

To support scalable connector construction, we introduce an automerger algorithm and a multi-merger extension that automate node merging and hiding. These mechanisms reduce manual intervention and help ensure the correctness of complex connector compositions. Experimental results indicate that the framework effectively supports connector modeling, refinement and equivalence checking, and automated test case generation.

Future work includes extending the framework to richer quantitative constraints, improving the scalability of automated verification, and validating the approach in larger and more realistic application scenarios. Overall, this work provides a rigorous and practical basis for the formal development and validation of Reo-based coordination systems.

## 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