Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 255 (2009) 119–135
www.elsevier.com/locate/entcs

# Connectors as Designs

Sun Meng[1] Farhad Arbab[2]

*CWI, Science Park 123, Amsterdam, The Netherlands*

**Abstract**

The complex interactions that appear in service-oriented computing make coordination a key concern in service-oriented systems. Over the past years, the need for high-confidence coordination mechanisms has intensified as new technologies have appeared for the development of service-oriented applications, making formalization of coordination mechanisms critical. Unifying Theories of Programming (UTP) provide a formal semantic foundation not only for programming languages but also for various expressive specification languages. A key concept in UTP is *design*: the familiar pre/post-condition pair that describes the contract. In this paper we use UTP to formalize Reo connectors, whereby connectors are interpreted by designs in UTP. This model can be used as a reference document for developing tool support for Reo, such as a test case generator. It can also be used as a semantic foundation for proving properties of connectors, such as equivalence and refinement relations between connectors.

*Keywords:* Connector, Reo circuits, Timed Data Sequence, Design

## 1 Introduction

Coordination models and languages are gaining more prominence in software engineering, especially in Service-Oriented Computing. Coordination models and languages provide a formalization of the "glue code" that interconnects the services/components, and organizes the mutual interactions among them in a distributed processing environment. For example, Reo [6,11] offers a powerful glue language for the implementation of coordinating component connectors based on a calculus of mobile channels. To support rigorous development of service-oriented applications, we need to investigate the formal semantics of coordination languages, which provide the foundations for understanding and reasoning about them and allow the construction of supporting tools. Hoare and He's Unifying Theories of Programming (UTP) [14] can present a formal semantics for various programming languages as well as specification languages like Circus and timed Circus [18,20],

[1] Email: M.Sun@cwi.nl
[2] Email: Farhad.Arbab@cwi.nl

1571-0661 © 2009 Elsevier B.V. Open access under CC BY-NC-ND license.
doi:10.1016/j.entcs.2009.10.028

TCOZ [19], rCOS [15] and CSP [13]. We believe UTP is also well suited for developing the formal foundation for coordination languages like Reo.

The behavior of a connector generally describes the manifold interactions among components / services that it interconnects, rather than simple input-output behavior on one individual interface. Thus, a connector may synchronize different input and output actions, aand therefore instead of sequences of input and output, we must use relations on different input / output sequences to describe the behavior of connectors. The communications via connectors can be modeled as *designs* in UTP, i.e., a pair of predicates $P \vdash Q$ where the assumption $P$ is what the designer can rely on when the communicating operation is initiated by inputs to the connectors, and the commitment $Q$ must be true for the outputs when the communicating operation terminates.

The semantics of Reo has been well investigated earlier. For example, a coalgebraic semantics for Reo in terms of relations on infinite timed data streams has been developed by Arbab and Rutten [8], but the casuality between input and output is not clear in this semantics. An operational semantics for Reo using constraint automata is provided by Baier et al. [11]. A model for Reo connectors based on the idea of coloring a connector with possible data flows to resolve synchronization and exclusion constraints is presented by Clarke et al. [12]. The semantic model of Reo provided in this paper is based on the UTP framework [14]. Other semantic models of Reo, like constraint automata [11] or the coalgebraic model [8], define behavior using infinite streams, which exclude a "natural" description of finite behavior (and connectors that exhibit finite behavior on any of their ports). In contrast, the timed data sequence in our model can be either finite or infinite, which makes it more expressive than the coalgebraic model. Furthermore, the UTP approach provides a family of algebraic operators, which can be used to interpret the composition of connectors explicitly.

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

UTP has been successfully applied in defining the semantics of channel-based dataflow models [14]. However, as discussed in [8], Reo is more general than dataflow models, Kahn-networks and Petri nets, which can all be viewed as special channel-based models that incorporate certain basic constructs for primitive coordination.

For example, in dataflow models, the channels are all buffered so that they can accept a data item at any time, storing it until the sink end is ready to take it. However, Reo supports a much more general notion of channel, which makes it more difficult to model Reo connectors than dataflow networks in UTP.

This paper is structured as follows. In Section 2 we briefly summarize the coordination language Reo. Then, in Section 3, we present the UTP observation model with meta variables and introduce the UTP design model used throughout the rest of the paper. In Section 4, we present the UTP design semantics for basic connectors in Reo. In Section 5, the composition of connectors is discussed. In Section 6 we discuss refinement and testing of connectors. Finally, Section 7 concludes with some further research directions.

## 2 A Reo Primer

In this section we provide a brief introduction to Reo [6]. Reo is a channel-based exogenous coordination model wherein complex coordinators, called connectors, are compositionally built out of simpler ones. Exogenous coordination imposes a purely local interpretation on each inter-components 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. We summarize only the main concepts in Reo here. Further details about Reo and its semantics can be found elsewhere [6,8,11].

Complex connectors in Reo are organized in a network of primitive connectors, called *channels*. A connector provides the protocol that controls and organizes the communication, synchronization and cooperation among the components/services that they interconnect. Each channel has two *channel ends*. There are two types of channel ends: *source* and *sink*. A source channel end accepts data into its channel, and a sink channel end dispenses data out of its channel. It is possible for the ends of a channel to be both sinks or both sources. Reo places no restriction on the behavior of a channel and thus allows an open-ended set of different channel types to be used simultaneously together. Each channel end can be connected to at most one component instance at any given time. Figure 1 shows the graphical representation of some simple channel types in Reo. More detailed discussion about these channels are given in Section 4.

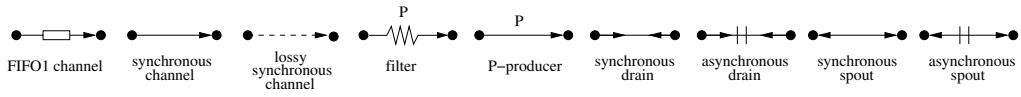


Fig. 1. Some basic channels in Reo

Complex connectors are constructed by composing simpler ones via the *join* and *hiding* operations. Channels are joined together in 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 $\mathsf{Src}(A)$ and $\mathsf{Snk}(A)$, 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 the 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 are its hidden internal details.

A component can write data items to a source node that it is connected to. The write operation succeeds only if all (source) channel ends coincident on the node accept the data item, in which case the data item is transparently written to every source end coincident on the node. A source node, thus, acts as a replicator. A component can obtain data items, by an input operation, from a sink node that it is connected to. A take operation succeeds only if at least one of the (sink) channel ends coincident on the node offers a suitable data item; if more than one coincident channel end offers suitable data items, one is selected non-deterministically. A sink node, thus, acts as a non-deterministic merger. A mixed node non-deterministically selects and takes a suitable data item offered by one of its coincident sink channel ends and replicates it into all of its coincident source channel ends. Note that a component cannot connect to, take from, or write to mixed nodes.

# 3 The UTP Observational Model

The Unifying Theories of Programming (UTP) were first proposed by Hoare and He [14]. In the following we introduce the observational model for Reo connectors and the theory of designs briefly. More details about UTP can be found in Hoare and He's book [14].

## *3.1 Observational Model*

UTP adopts the relational calculus as the foundation to unify various programming theories. All kinds of specifications, designs and programs are interpreted as relations between an initial observation and a subsequent (intermediate, stable or final) observation of the behavior of their executions. Program correctness and refinement can be represented by inclusion of relations, and all laws of the relational calculus are valid for reasoning about correctness.

Collections of relations form a theory of the paradigm being studied, and it contains three essential parts: an alphabet, a signature, and healthiness conditions.

Connectors describe the coordination among components / services. We use $in_{\mathbf{R}}$ and $out_{\mathbf{R}}$ to denote what happens on the input ends and the output ends of a connector $\mathbf{R}$, respectively, instead of using unprimed variables for initial observations and primed variables for subsequent ones as in [14] [3]. Thus, the alphabet, i.e., the

[3] There are two kinds of primed variables in our model: the auxiliary variable $ok'$ is used for successful

set of all observation-capturing variables, used in this paper is different from that for a design in [14]. The signature gives the rules for the syntax for denoting the elements of the theory. Healthiness conditions, which embody aspects of the model being studied, are taken as **true** here.

For an arbitrary connector **R**, the relevant observations come in pairs, with one observation on the source nodes of **R**, and one observation on the sink nodes of **R**. For every node $N$, the corresponding observation on $N$ is given by a 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 $\alpha = (\alpha(0), \alpha(1), \alpha(2), \cdots)$ 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 [4]. Let $\mathbb{R}_+^*$ be the set of sequences $a = (a(0), a(1), a(2), \cdots)$ over $\mathbb{R}_+$, and for all $a = (a(0), a(1), a(2), \cdots)$ and $b = (b(0), b(1), b(2), \cdots)$ in $\mathbb{R}_+^*$, if $|a| = |b|$, then

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

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

The set $TS$ of time sequences is defined as

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

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

The set $TDS$ of timed data sequences is defined 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 [8], timed data sequences can be alternatively and equivalently defined as (a subset of) $(D \times \mathbb{R}_+)^*$ because of the existence of the isomorphism

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

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 data element and time moment.

In order to analyze explicitly the phenomena of communication initialization and termination, we introduce two variables $ok, ok'$ : **Boolean**. The variable $ok$

termination, and for a sequence $a$, $a'$ is used to return the tail of $a$.

[4] 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. 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.

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 whether the communication has terminated or has 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 timed data sequences on the input ends of the connector, and $Q$ is the predicate specifying the condition that should be satisfied by the timed data sequences on the output ends of the connector. The following section gives a brief introduction to the theory of designs in UTP.

### *3.2 A Theory of Designs*

An important subtheory of relations allows the separation of preconditions from postconditions, in the manner of the well-known formal methods like VDM [16], B[2], RAISE [21], refinement calculus [9] and more recently OCL [17]. This allows us to model the total correctness of programming constructs using relations. This section is an introduction to the relational calculus on designs in UTP.

**Definition 3.1** 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:

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

As can be seen, a design predicate represents a pre/post-condition specification. The separation of precondition from postcondition allows us to write a specification that has a more generous 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 postcondition. 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 postcondition nor even to make the system terminate.

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 postconditions are strengthened. This is established by the following definition:

**Definition 3.2** $[(P_1 \vdash Q_1) \sqsubseteq (P_2 \vdash Q_2)]$ **iff** $[P_1 \Rightarrow P_2] \wedge [P_1 \wedge Q_2 \Rightarrow Q_1]$

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.

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

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

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

**Definition 3.3** The conditional expression is defined as follows:

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

Sequential composition is defined via the existence of an intermediate state $v_0$ of a variable $v$. Here the existential quantification hides the intermediate observation $v_0$. In addition, the output alphabet of $P$ ($out\alpha P$) and the input alphabet of $Q$ (with all variables primed, $in\alpha Q'$) must be the same [5].

**Definition 3.4** $P(v'); Q(v) =_{df} \exists v_0 \bullet P(v_0) \wedge Q(v_0)$ provided $out\alpha P = in\alpha Q' = \{v'\}$.

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

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

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

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

## 4 Connectors as Designs

Since we aim at specifying both finite and infinite behavior of connectors, we use relations on timed data sequences to model connectors. In the following, we assume that all timed data sequences are finite. However, the semantic definition can be easily generalized to infinite sequences, which are timed data streams as proposed in [8]. 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 $\mathcal{D}$.

[5] For a predicate $P$, the sets $in\alpha P$ and $out\alpha P$ are the sets of unprimed and primed variables in $P$ respectively, standing for initial and final values, while $in\alpha P'$ is obtained by just putting a prime symbol on all the variables of the input alphabet $in\alpha P$.

Then, every connector $\mathbf{R}$ can be represented as the following design,

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

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

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

- Synchronous channel $\longrightarrow$:

$$\begin{aligned} \mathbf{con}: &\quad \mathbf{Sync}(in : (\langle \alpha, a\rangle); out : (\langle \beta, b\rangle)) \\ P: &\quad \mathcal{D}\langle \alpha, a\rangle \\ Q: &\quad \mathcal{D}\langle \beta, b\rangle \wedge \beta = \alpha \wedge b = a \end{aligned}$$

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

- FIFO1 channel $-\!\square\!\rightarrow$:

$$\begin{aligned} \mathbf{con}: &\quad \mathbf{FIFO1}(in : (\langle \alpha, a\rangle); out : (\langle \beta, b\rangle)) \\ P: &\quad \mathcal{D}\langle \alpha, a\rangle \\ Q: &\quad \mathcal{D}\langle \beta, b\rangle \wedge \beta = \alpha \wedge a < b \wedge (tail(b^R))^R < tail(a) \end{aligned}$$

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

  On the other hand, for the FIFO1 channel $-\!\boxed{e}\!\rightarrow$ 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

$$\begin{aligned} \mathbf{con}: &\quad \mathbf{FIFO1}e(in : (\langle \alpha, a\rangle); out : (\langle \beta, b\rangle)) \\ P: &\quad \mathcal{D}\langle \alpha, a\rangle \\ Q: &\quad \mathcal{D}\langle \beta, b\rangle \wedge \beta = (e)^\frown \alpha \wedge a < b' \wedge ((b^R)')^R < a \end{aligned}$$

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.

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

$$\begin{aligned} \textbf{con}: &\quad \textbf{SyncDrain}(in : (\langle \alpha, a\rangle, \langle \beta, b\rangle); out : (\,)) \\ P: &\quad \mathcal{D}\langle \alpha, a\rangle \wedge \mathcal{D}\langle \beta, b\rangle \wedge a = b \\ Q: &\quad \textbf{true} \end{aligned}$$

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

- Lossy Synchronous Channel $-\!\!\rightarrow$:

$$\begin{aligned} \textbf{con}: &\quad \textbf{LossySync}(in : (\langle \alpha, a\rangle); out : (\langle \beta, b\rangle)) \\ P: &\quad \mathcal{D}\langle \alpha, a\rangle \\ Q: &\quad \mathcal{D}\langle \beta, b\rangle \wedge L(\langle \alpha, a\rangle, \langle \beta, b\rangle) \end{aligned}$$

where

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

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

- Filter $-\{p\}\!\rightarrow$:

$$\begin{aligned} \textbf{con}: &\quad \textbf{Filter}p(in : (\langle \alpha, a\rangle); out : (\langle \beta, b\rangle)) \\ P: &\quad \mathcal{D}\langle \alpha, a\rangle \\ Q: &\quad \mathcal{D}\langle \beta, b\rangle \wedge F(\langle \alpha, a\rangle, \langle \beta, b\rangle) \end{aligned}$$

where

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

This channel specifies a filter pattern $p$ which is a set of data values. It transfers only those data items that match with the pattern $p$ and loses the rest. A write

operation on the source end succeeds only if either the data item to be written does not match with the pattern $p$ or the data item matches the pattern $p$ and it can be taken synchronously via the sink end.

- $p$-Producer $\dashrightarrow$:

$$\begin{aligned} \mathbf{con}: &\quad \mathbf{Producer}p(in : (\langle \alpha, a\rangle); out : (\langle \beta, b\rangle)) \\ P: &\quad \mathcal{D}\langle \alpha, a\rangle \\ Q: &\quad \mathcal{D}\langle \beta, b\rangle \wedge \beta \in p^* \wedge b = a \end{aligned}$$

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

- Asynchronous Spout $\longleftarrow\!\!\|\longrightarrow$:

$$\begin{aligned} \mathbf{con}: &\quad \mathbf{AsynSpout}(in : (); out : (\langle \alpha, a\rangle, \langle \beta, b\rangle)) \\ P: &\quad \mathbf{true} \\ Q: &\quad \mathcal{D}\langle \alpha, a\rangle \wedge \mathcal{D}\langle \beta, b\rangle \wedge |a| = |b| \wedge a \bowtie b \end{aligned}$$

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

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

  This channel outputs two sequences of data items at the two output ends, but the data items on the two ends are never outputed 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, which are not given here.

## 5 Composing Connectors

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

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

where $i = 1, 2$ to denote the two connectors being composed.

Figure 2(1) shows the case of flow-through composition of connectors. For the two connectors $\mathbf{R}_1$ and $\mathbf{R}_2$, suppose one sink node of $\mathbf{R}_1$ and one source node of $\mathbf{R}_2$ are joined together into a mixed node $B$. When we compose connectors, we

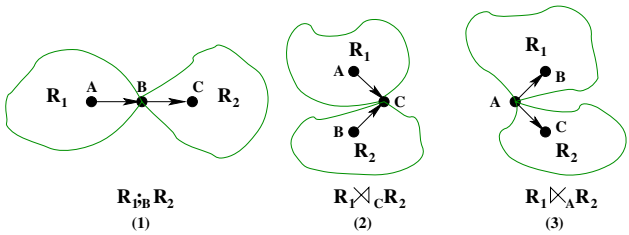


Fig. 2. Connector composition

want the events on the mixed nodes to happen silently and automatically whenever they can, without the participation or even the knowledge of the environment of the connector. Such mixed nodes are hidden (encapsulated) by using existential quantification. Let $\langle \beta_1, b_1 \rangle \in out_{\mathbf{R}_1}$ and $\langle \beta_2, b_2 \rangle \in in_{\mathbf{R}_2}$ be the timed data sequences on the node $B$ in $\mathbf{R}_1$ and $\mathbf{R}_2$, respectively. Then, the resulting connector is denoted by $\mathbf{R} = \mathbf{R}_1;_B \mathbf{R}_2$, and the corresponding design is given as [6]

$$\begin{aligned} \mathbf{con}: \quad & \mathbf{R}(in : (\bigcup_{i=1,2} in_{\mathbf{R}_i}) \setminus \{\langle \beta_2, b_2 \rangle\}; out : (\bigcup_{i=1,2} out_{\mathbf{R}_i}) \setminus \{\langle \beta_1, b_1 \rangle\}) \\ P: \quad & P_1 \wedge \neg(Q_1(\langle \beta_1, b_1 \rangle); \neg P_2(\langle \beta_2, b_2 \rangle)) \\ Q: \quad & Q_1(\langle \beta_1, b_1 \rangle); Q_2(\langle \beta_2, b_2 \rangle) \end{aligned}$$

in which the sequential composition of predicates is defined similarly as in Definition 3.4. If one sink node of $\mathbf{R}_1$ and one source node of $\mathbf{R}_2$ are joined together into a mixed node $B$, $\langle \beta_1, b_1 \rangle \in out_{\mathbf{R}_1}$ and $\langle \beta_2, b_2 \rangle \in in_{\mathbf{R}_2}$ are the timed data sequences on $B$ in $\mathbf{R}_1$ and $\mathbf{R}_2$ respectively, for the two predicates $P$ and $Q$,

$$P(\langle \beta_1, b_1 \rangle); Q(\langle \beta_2, b_2 \rangle) = \exists \langle \beta, b \rangle . P(\langle \beta, b \rangle) \wedge Q(\langle \beta, b \rangle)$$

Note that the condition on $out\alpha P$ and $in\alpha Q$ in Definition 3.4 is not needed here, since not all output nodes of $\mathbf{R}_1$ and input nodes of $\mathbf{R}_2$ are composed with each other, leaving some of them to be used as the external nodes of the composed connector.

Figure 2(2) shows the case of merging two sink nodes of the connectors $\mathbf{R}_1$ and $\mathbf{R}_2$. Let $\langle \gamma_i, c_i \rangle \in out_{\mathbf{R}_i}$, $i = 1, 2$, be the timed data sequences on the node $C$ in $\mathbf{R}_1$ and $\mathbf{R}_2$, respectively. Then the resulting connector is denoted by $\mathbf{R} = \mathbf{R}_1 \rtimes_C \mathbf{R}_2$,

[6] Note here and in the rest of this paper, we abuse the well-known notations in set theory a little with the input and output vectors to simplify the formulas. For example, we use $\bigcup_{i=1,2} in_{\mathbf{R}_i}$ to specify the input vector that contains the input timed data sequences in both $\mathbf{R}_1$ and $\mathbf{R}_2$, and $\setminus$ to remove a set of sequences from a given vector of sequences.

and the corresponding design is given as

$$
\begin{aligned}
\mathbf{con}: \quad & \mathbf{R}(in : \bigcup_{i=1,2} in_{\mathbf{R}_i}; out : (\bigcup_{i=1,2} (out_{\mathbf{R}_i} \setminus \{\langle \gamma_i, c_i\rangle\})) \cup \{\langle \gamma, c\rangle\}) \\
P: \quad & \bigwedge_{i=1,2} P_i(in_{\mathbf{R}_i}) \\
Q: \quad & \mathcal{D}\langle \gamma, c\rangle \wedge \exists \langle \gamma_1, c_1\rangle, \langle \gamma_2, c_2\rangle.(\bigwedge_{i=1,2} Q_i(in_{\mathbf{R}_i}, out_{\mathbf{R}_i})) \wedge M(\langle \gamma_1, c_1\rangle, \langle \gamma_2, c_2\rangle, \langle \gamma, c\rangle)
\end{aligned}
$$

In this definition, the ternary relation $M$ is defined as

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

Figure 2(3) shows the case of merging two source nodes of the connectors $\mathbf{R}_1$ and $\mathbf{R}_2$. Let $\langle \alpha_i, a_i\rangle \in in_{\mathbf{R}_i}$ $i = 1, 2$, be the timed data sequences on the node $A$ in $\mathbf{R}_1$ and $\mathbf{R}_2$, respectively. Then the resulting connector is denoted by $\mathbf{R} = \mathbf{R}_1 \ltimes_A \mathbf{R}_2$, and the corresponding design is given as

$$
\begin{aligned}
\mathbf{con}: \quad & \mathbf{R}(in : (\bigcup_{i=1,2} in_{\mathbf{R}_i} \setminus \{\langle \alpha_i, a_i\rangle\}) \cup \{\langle \alpha, a\rangle\}; out : \bigcup_{i=1,2} out_{\mathbf{R}_i}) \\
P: \quad & \bigwedge_{i=1,2} P_i(in_{\mathbf{R}_i})[\langle \alpha, a\rangle / \langle \alpha_i, a_i\rangle] \\
Q: \quad & \bigwedge_{i=1,2} Q_i(in_{\mathbf{R}_i}, out_{\mathbf{R}_i})[\langle \alpha, a\rangle / \langle \alpha_i, a_i\rangle]
\end{aligned}
$$

For a predicate $P$, if $v$ is a variable in $P$, $P[u/v]$ is the predicate obtained by replacing all occurrance of $v$ in $P$ by $u$.

According to the design semantics, we can easily establish a number of algebraic laws relating the composition patterns. For example, one gets, associativity for all

the operators ;, $\rtimes$, $\ltimes$ and commutativity for $\rtimes$ and $\ltimes$:

$$(\mathbf{R}_1;_A \mathbf{R}_2);_B \mathbf{R}_3 \equiv \mathbf{R}_1;_A (\mathbf{R}_2;_B \mathbf{R}_3)$$
$$(\mathbf{R}_1 \rtimes_C \mathbf{R}_2) \rtimes_C \mathbf{R}_3 \equiv \mathbf{R}_1 \rtimes_C (\mathbf{R}_2 \rtimes_C \mathbf{R}_3)$$
$$(\mathbf{R}_1 \ltimes_A \mathbf{R}_2) \ltimes_A \mathbf{R}_3 \equiv \mathbf{R}_1 \ltimes_A (\mathbf{R}_2 \ltimes_A \mathbf{R}_3)$$
$$\mathbf{R}_1 \rtimes_C \mathbf{R}_2 \equiv \mathbf{R}_2 \rtimes_C \mathbf{R}_1$$
$$\mathbf{R}_1 \ltimes_A \mathbf{R}_2 \equiv \mathbf{R}_2 \ltimes_A \mathbf{R}_1$$

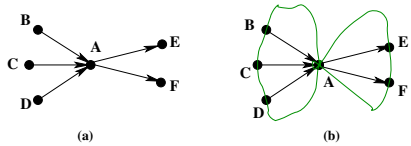


Fig. 3. Mixed node in Reo

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

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

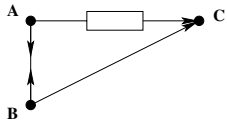


Fig. 4. Alternator

**Example 5.1** Figure 4 shows a Reo circuit consisting of three channels **AB**, **AC** and **BC** which are of types **SyncDrain**, **FIFO1** and **Sync**, respectively. By composing the channels, we can get the connector as

$$\begin{aligned}
\mathbf{con}: \quad & \mathbf{Alternator}(in : (\langle \alpha, a\rangle, \langle \beta, b\rangle); out : \langle \gamma, c\rangle) \\
P: \quad & \mathcal{D}\langle \alpha, a\rangle \wedge \mathcal{D}\langle \beta, b\rangle \wedge a = b \\
Q: \quad & \mathcal{D}\langle \gamma, c\rangle \wedge \exists \langle \gamma_1, c_1\rangle, \langle \gamma_2, c_2\rangle . \mathcal{D}\langle \gamma_1, c_1\rangle \wedge \mathcal{D}\langle \gamma_2, c_2\rangle \wedge \\
& \gamma_1 = \alpha \wedge a < c_1 \wedge ((c_1^R)')^R < a' \wedge \gamma_2 = \beta \wedge c_2 = b \wedge M(\langle \gamma_1, c_1\rangle, \langle \gamma_2, c_2\rangle, \langle \gamma, c\rangle)
\end{aligned}$$

where the postcondition $Q$ happens to be equivalent to

$$\begin{aligned} Q: \quad & \mathcal{D}\langle\gamma, c\rangle \wedge \gamma(2n) = \beta(n) \wedge \gamma(2n+1) = \alpha(n) \wedge \\ & c(2n) = a(n) \wedge a(n) < c(2n+1) < a(n+1) \end{aligned}$$

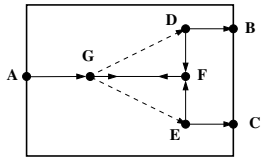


Fig. 5. Exclusive Router

**Example 5.2** Figure 5(a) shows an implementation of an exclusive router connector. A data item arriving at the input port **A** flows through to only one of the output ports **B** or **C**, depending on which one is ready to consume it. If both output ports are prepared to consume a data item, then one of the output ports is selected non-deterministically. We can parameterize this connector to have as many output nodes as we want simply by inserting more (or fewer) LossySync and pairs of Sync channels, as required.

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

$$\begin{aligned} \mathbf{con}: \quad & \mathbf{EXRouter}(in : (\langle\alpha, a\rangle); out : (\langle\beta, b\rangle, \langle\gamma, c\rangle)) \\ P: \quad & \mathcal{D}\langle\alpha, a\rangle \\ Q: \quad & \mathcal{D}\langle\beta, b\rangle \wedge \mathcal{D}\langle\gamma, c\rangle \wedge L(\langle\alpha, a\rangle, \langle\beta, b\rangle) \wedge L(\langle\alpha, a\rangle, \langle\gamma, c\rangle) \wedge M(\langle\beta, b\rangle, \langle\gamma, c\rangle, \langle\alpha, a\rangle) \end{aligned}$$

where $L$ and $M$ were introduced previously.

## 6 Refinement and Testing of Connectors

Implication of predicates establishes a refinement order over connectors. Thus, more concrete implementations imply more abstract specifications. For two connectors

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

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

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

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

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

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

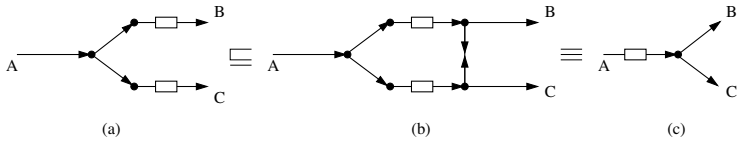


Fig. 6. Example of Connector Refinement

The design model of connectors has been successfully applied in generating test cases for connectors [4]. The theoretical foundation of the test case generation approach is based on [3,5], in which it can be formally proven that test cases will detect certain faults by using refinement calculus.

Test cases are generated on the basis of possible errors during the design of connectors. Examples of such errors might be a wrongly used channel, a missing subcircuit, or a circuit with wrongly constructed topology, etc. Note that not all errors during the design of connectors lead to faulty connectors. To contain a fault, a possible external observation of the fault must exist. For example, adding by mistake redundant synchronous channels to some input/output nodes does not result in a faulty connector since refinement holds between the connectors with and without redundant channels (in fact, such two connectors are bisimilar, which is a stronger relation than refinement). However, swapping different nodes can lead to a faulty connector. If we want to have a connector whose specification is given by a design $\mathbf{R}$, and implement the connector as $\mathbf{R}'$, then $\mathbf{R}'$ is called a faulty connector if and only if $\mathbf{R} \not\sqsubseteq \mathbf{R}'$.

For connectors, we consider test cases as specifications that define the expected list of timed data sequences on the output nodes for a given list of timed data sequences on the input nodes. For a connector $\mathbf{R}(in : in_{\mathbf{R}}; out : out_{\mathbf{R}})$, let $i$ be the input vector and $o$ be the output vector, both lists of timed data sequences with the same lengths as $in_{\mathbf{R}}$ and $out_{\mathbf{R}}$ respectively. A test case for $\mathbf{R}$ is defined as

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

Therefore, test cases, as well as connector specifications and implementations, can be specified by designs. It is obvious that an implementation that is correct with respect to its specification should refine the test cases of the latter. An implementation refines a test case if and only if the implementation passes the test case. Taking specifications into consideration, the specification should also be a refinement of a test case if the test case is properly derived from the specification. An algorithm

for fault-based test case generation has been developed in [4], and a prototype for test case generation has been developed using Maude.

# 7 Conclusion

This paper demonstrates that UTP can be applied not only for giving semantics of specific programming languages and specification languages, but also for providing a formal semantic foundation for coordination languages. In particular, the unified semantic model for different kinds of channels and composite connectors in Reo covers different communication mechanisms encoded in Reo, and allows the combination of synchronous and asynchronous channels as in Reo. Our semantic model offers potential benefits in developing tool support for Reo, like test case generators. Furthermore, the predicates used in UTP provide a possible symbolic representation of coloring for connectors [12], and thus make it possible to synthesize connectors from specifications more efficiently.

In future work, we will investigate the semantic model of timed connectors [7] and probabilistic connectors [10], and build links between these models and the model in this paper. This will make it possible to reason about some properties of the connectors in one model, given the design semantics in the other. On the other hand, we will investigate the relationship between the UTP semantics and other semantics of Reo that have been developed, and extend the UTP model to treat the inherent dynamic topology and mobility in "full" Reo. We also plan to encode the UTP model into theorem provers to prove properties of connectors based on their UTP semantics. The development of refinement laws for connectors like in Figure 6 and integration of such laws into our existing tools for Reo [1] are in our scope as well.

# Acknowledgement

The authors are indebted to our colleagues, especially Bernhard Aichernig, Jan Rutten, Frank de Boer, Milad Niqui and Christel Baier for helpful discussions, and Jifeng He for the constructive comments on an earlier version of the paper which helped to simplify the model and improve this paper. We are grateful to Lacramioara Astefanoaei for her help in developing the test case generator. The work reported in this paper is supported by a grant from the GLANCE funding program of the Dutch National Organization for Scientific Research (NWO), through project CooPer (600.643.000.05N12), and the DFG-NWO-project SYANCO.

# References

[1] Eclipse Coordination Tools. `http://reo.project.cwi.nl/`.

[2] J. R. Abrial. *The B-Book : Assigning Programs to Meanings.* Cambridge University Press, 1996.

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

[4] B. K. Aichernig, F. Arbab, L. Astefanoaei, F. S. de Boer, S. Meng, and J. Rutten. Fault-based test case generation for component connectors. Accepted by *TASE'09*.

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

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

[7] F. Arbab, C. Baier, F. de Boer, and J. Rutten. Models and Temporal Logics for Timed Component Connectors. In J. R. Cuellar and Z. Liu, editors, *SEFM2004, 2nd International Conference on Software Engineering and Formal Methods*, pages 198–207. IEEE Computer Society, 2004.

[8] F. Arbab and J. Rutten. A coinductive calculus of component connectors. In M. Wirsing, D. Pattinson, and R. Hennicker, editors, *Recent Trends in Algebraic Development Techniques: 16th International Workshop, WADT 2002, Frauenchiemsee, Germany, September 24-27, 2002, Revised Selected Papers*, volume 2755 of *LNCS*, pages 34–55. Springer-Verlag, 2003.

[9] R.-J. Back and J. von Wright. *Refinement Calculus: A Systematic Introduction*. Springer, 1998.

[10] C. Baier. Probabilistic Models for Reo Connector Circuits. *Journal of Universal Computer Science*, 11(10):1718–1748, 2005.

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

[12] D. Clarke, D. Costa, and F. Arbab. Connector colouring I: Synchronisation and context dependency. *Science of Computer Programming*, 66:205–225, 2007.

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

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

[15] H. Jifeng, X. Li, and Z. Liu. rCOS: A refinement calculus of object systems. *Theoretical Computer Science*, 365(1-2):109–142, 2006.

[16] C. B. Jones. *Systematic Software Development using VDM*. Prentice-Hall, 1990.

[17] Object Management Group. *Unified Modeling Language: Superstructure - version 2.1.1*, 2007. `http://www.uml.org/`.

[18] M. Oliveira, A. Cavalcanti, and J. Woodcock. A Denotational Semantics for Circus. *Electr. Notes Theor. Comput. Sci.*, 187:107–123, 2007.

[19] S. Qin, J. S. Dong, and W.-N. Chin. A Semantic Foundation for TCOZ in Unifying Theories of Programming. In *FME 2003: Formal Methods, International Symposium of Formal Methods Europe, Pisa, Italy, September 8-14, 2003, Proceedings*, volume 2805, pages 321–340, 2003.

[20] A. Sherif and J. He. Towards a Time Model for Circus. In C. George and H. Miao, editors, *Formal Methods and Software Engineering, 4th International Conference on Formal Engineering Methods, ICFEM 2002 Shanghai, China, October 21-25, 2002, Proceedings*, volume 2495, pages 613–624. Springer, 2002.

[21] The RAISE Language Group. *The RAISE Specification Language*. Prentice Hall International, 1992.