2012 IEEE Sixth International Symposium on Theoretical Aspects of Software Engineering

Connectors as Designs: the Time Dimension

Sun Meng[1,2]
[1]*LMAM & Department of Informatics, School of Mathematical Science, Peking University*
[2]*State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences*
Beijing, China
sunmeng@math.pku.edu.cn

***Abstract*—Compositional coordination models and languages serve as a means to formally specify and implement component and service connectors. They support large-scale distributed applications by allowing construction of complex component connectors out of simpler ones. In this paper, we extend the design model for the channel-based coordination language Reo by introducing designs for timed connectors. Design is a key concept in Unifying Theories of Programming (UTP), which is used to describe the contract between programmer and client. The model developed in this paper specifies properties of timed channels and timed component connectors properly. Implementation of the design model developed in `JTom` is provided.**

***Keywords*-Coordination, Timed Reo Connector, UTP, Design, JTom**

I. Introduction

Complex distributed applications are typically heterogeneous and geographically distributed, and usually exploit communication infrastructures whose topology varies and components can, at any moment, connect to or detach from. The development of such distributed applications requires a coordination model that formalizes the orchestration among the components. Compositional coordination models and languages provide a formalization of the "glue code" that interconnects the constituent components and organizes the mutual interactions among them in a distributed processing environment. They support large-scale distributed applications by allowing construction of complex component connectors out of simpler ones. As an example, Reo [2], [7] offers a powerful glue language for implementation of coordinating component connectors. Primitive connectors called *channels* in Reo, such as synchronous channels and FIFO channels, can be composed to build circuit-like connectors which serve as the glue code to exogenously coordinate the behavior of components in distributed applications.

The problem that we identify and address in this paper is the time aspects of timed Reo connectors under the UTP semantic framework. UTP (Unifying Theories of Programming) is proposed by Hoare and He in [13], which can present a formal semantics for various programming languages and specification languages like Circus and timed Circus [18], [20], TCOZ [19] and rCOS [14]. In [17] we have shown that untimed Reo connectors can be translated into an essentially equivalent UTP *design*, i.e., a pair of predicates $P \vdash Q$ where P specifies what the designer can rely on when the communicating operation is initiated by input to the connector, and Q is the condition on output that must be true when the communicating operation terminates. Connectors also admit intermediate observations on suitable occasions between initiation and termination. Additional variables and naming conventions are used to denote the results of such observations. In this paper the design model for untimed Reo is extended such that a family of timed channels in Reo can be specified by the UTP design model, and we also show how the design models of such timed channels can be used in construction of more complex timed connectors. Furthermore, the UTP design model for timed connectors are implemented in `JTom` (`Tom` built on top of `Java`)[21].

To give a semantic model for timed Reo, which clearly separates data and time, we need to choose an appropriate model of time. There are two typical models: discrete time and dense time. The discrete time model has been adopted in [20], [19] on the semantics for timed Circus and TCOZ respectively. The dense time model, which is adopted in the timed data stream semantic model for Reo [6], seems to be more appropriate since it is very 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, such timing constraints are not explicit for the input providers. Therefore, we choose the dense time model in this paper.

The formal semantics for Reo makes it possible to specify and analyze the behavior of a connector precisely. There are different formal semantics for Reo in literature. For example, a coalgebraic semantics for Reo in terms of relations on infinite timed data streams has been developed by Arbab and Rutten [6], but the causality between input and output is not clear (In general one cannot separate the input and output of a connector from its semantics). An operational semantics for Reo using constraint automata is provided by Baier et al. [7]. However, modeling unbounded primitives or even bounded primitives with unbounded data domains is impossible with finite constraint automata. Bounded large data

978-0-7695-4751-0/12 $26.00 © 2012 IEEE
DOI 10.1109/TASE.2012.36

domains cause an explosion in the constraint automata model which becomes problematic. A model for Reo connectors based on the idea of coloring a connector with possible data flows to resolve synchronization and exclusion constraints is presented by Clarke et al. [9]. Unlike the coalgebraic and operational semantics, data sensitive behavior, which is supported by filter channels in Reo, are not captured in the coloring approach. The real-time aspects of Reo has been investigated in [3], which uses timed constraint automata (TCA) as the operational model for Reo connectors and provides a variant of LTL to serve as a specification formalism for timed Reo connectors. In [15], the TCA model is translated into mCRL2 for model checking timed Reo connectors. However, these works suffer from the same problem as in the coalgebraic approach that the casuality between input and output is not clear.

Comparing with previous work on (both untimed and timed) Reo semantics, the UTP approach provides a family of algebraic operators for different kinds of composition of connectors (merger, replicator and flow-through composition), which can be used to interpret the composition of connectors more explicitly than other approaches, such as TDS in the coalgebraic semantics. Moreover, by separation of the assumption (input) and the commitment (output) in the design model in our approach, the connector behavior becomes more clear and easy to be composed. 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.

The paper is structured as follows. After this general introduction, we briefly summarize the coordination language Reo and introduce the notion of design in UTP being used throughout the rest of the paper in Section II. Section III presents the model of observations on connector nodes. Section IV briefly summarizes the UTP design model for basic (untimed and timed) Reo channels and timed Reo circuits. In Section V, we discuss the implementation of the design model in `Tom`. Finally, Section VI concludes with some further research directions.

II. Preliminaries

In this section, we briefly review basic concepts of Reo and UTP design. The discussion on Reo is mainly based on [2], [7] and the overview of UTP design is from [13].

A. Reo

Reo [2] is a channel-based exogenous coordination language wherein complex coordinators, called *connectors*, are compositionally constructed from simpler ones. We summarize only the main concepts in Reo here. Further details about Reo can be found in [2], [7].

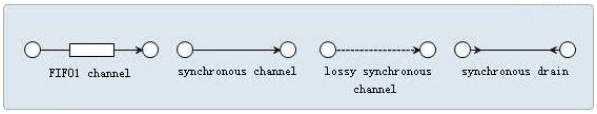

Figure 1. Some basic channels in Reo

A Reo connector consists of 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 communicate through the connector. 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. 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 whose composition allows for expressing a rich set of coordination patterns [2].

A *FIFO1 channel* represents an asynchronous channel with one buffer cell which is empty if no data item is shown in the box (this is the case in Figure 1). If a data element d is contained in the buffer of a FIFO1 channel then d is shown inside the box in its graphical representation. A *synchronous channel* has a source and a sink end and no buffer. It accepts a data item through its source end iff it can simultaneously dispense it through its sink. A *lossy synchronous channel* is similar to a synchronous channel except that it always accepts all data items through its source end. The data item is transferred if it is possible for the data item to be dispensed through the sink end, otherwise the data item is lost. *Synchronous drain* has two source ends and no sink end. A synchronous drain can accept a data item through one of its ends iff a data item is also available for it to simultaneously accept through its other end as well, and all data accepted by the channel are lost. More exotic channels permitted in Reo will be omitted here and can be found in [2]. Note that the set of channel types is not fixed in Reo, and new ones can be defined freely by users with their own interaction policies.

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 of source and sink channel ends that coincide on A. 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*, whose behavior 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.

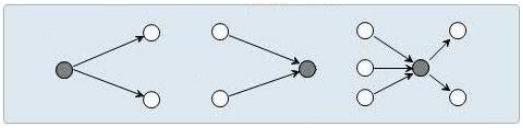

Figure 2. Source, Sink and Mixed Nodes in Reo

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.

B. A Theory of Designs

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 variables ok and ok' to analyze explicitly the phenomena of communication initialization and termination. The variable ok stands for a successful initialization and the start of a communication. When ok is **false**, no observation can be made. The variable ok' denotes the observation that the communication has either terminated or reached an intermediate stable state. The communication is divergent when ok' is **false**. The observational semantics for a Reo connector is described by a design, i.e., a relation expressed as $P \vdash Q$, where P is the predicate specifying the relationship among the observations on the source nodes of the connector, and Q is the predicate specifying the condition that should be satisfied by the observations on the sink nodes of the connector. Such a design $P \vdash Q$ is defined as follows:

Definition 2.1: A design is a pair of predicates $P \vdash Q$, where neither predicate contains ok or ok', and P has only input 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 2.2: $[(P_1 \vdash Q_1) \sqsubseteq (P_2 \vdash Q_2)]$ **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 [13].

III. Observations on Connectors

To specify inputs and outputs on connectors explicitly, for a connector $\mathbf{R}$, we use $in_{\mathbf{R}}$ and $out_{\mathbf{R}}$ to denote the observations on its source nodes and sink nodes, respectively, instead of using unprimed variables for initial observations (inputs) and primed variables for subsequent ones (outputs) as in [13] (the definition of $in_{\mathbf{R}}$ and $out_{\mathbf{R}}$ will be given later).

For every node N in a connector $\mathbf{R}$, 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. For a sequence s, the length of s is denoted by $|s|$. If s is an infinite sequence, then $|s| = \infty$. Let $\mathbb{R}_+^*$ be the set of sequences $a = (a(0), a(1), a(2), \cdots)$ over $\mathbb{R}_+$, and for all a, b in $\mathbb{R}_+^*$, if $|a| = |b|$, then

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

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

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

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

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

The set TS of time sequences is defined as

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

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

For a sequence a, the two operators a^R and $\overrightarrow{a}$ are used to denote the reverse and tail of a respectively, which are defined as:

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

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

where $\frown$ is the concatenation operator on sequences. The concatenation of two sequences produces a new sequence that starts with the first sequence followed by the second sequence.

The set TDS of timed data sequences is defined as $TDS \subseteq DS \times TS$ of pairs $\langle \alpha, a \rangle$ consisting of a data sequence α and a time sequence a with $|\alpha| = |a|$. Similar to the discussion in [6], 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 of a data item d at some node N of a connector (i.e., taking d from N or writing d to N) 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.

IV. Reo and its Design Model

In this section we provide an overview on how Reo connectors can be modeled as UTP designs. We first see how primitive untimed channels in Reo are specified as designs, and then study the design model of timed channels. Finally we show how composite connectors can be constructed from simpler ones structurally.

We use $\mathcal{WD}$ for a predicate of well-defined TDS types. In other words, we define the behavior only for valid sequences expressed via a predicate $\mathcal{WD}$. 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:

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

where $\mathbf{R}$ is the name of the connector, $P(in_{\mathbf{R}})$ is the condition 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 condition that should be satisfied by outputs $out_{\mathbf{R}}$ on the sink nodes of $\mathbf{R}$. 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:

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

A. Designs for Primitive Reo Channels

We now start by presenting a few examples of basic channels in Reo and their design model. Discussions on more channel types can be found in [17].

FIFO channels. The simplest form of an asynchronous channel is a FIFO channel with one buffer cell, which is denoted as **FIFO1**. A **FIFO1** channel with source end A and sink end B is graphically represented by $A{-}\square{\to}\ B$. The design model for a **FIFO1** channel is given as follows:

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

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 $(\overrightarrow{b^R})^R < \overrightarrow{a}$ to represent the relationship between the time moments for outputs and their corresponding next inputs. This notation can be simplified to $b < \overrightarrow{a}$ if we consider infinite sequences of inputs and outputs. [1]

For the **FIFO1** channel $A{-}\square e\square{\to}\ B$ 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 denote the channel by $\mathbf{FIFO1}e$ and have its design model as follows:

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

Synchronous channels. A 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. A synchronous channel with source end A and sink end B is graphically represented

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

as $A \longrightarrow B$. The design model for a synchronous channel is as follows:

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

A lossy synchronous channel (graphically depicted as $A- \rightarrow B$) is similar to a normal 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. Its design model is given as follows:

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

where

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

An exotic channel in Reo is the synchronous drain $A \rightarrow\leftarrow B$ that has two source ends. Because a drain has no sink end, no data value can ever be obtained from this channel. Thus, all data accepted by 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.

$$\begin{aligned}&\mathbf{con} : \mathbf{SyncDrain}(in : (A \mapsto \langle \alpha, a\rangle, B \mapsto \langle \beta, b\rangle); out\colon (\,))\\ &\quad \mathbf{in} : \mathcal{WD}\langle \alpha, a\rangle \wedge \mathcal{WD}\langle \beta, b\rangle \wedge a = b\\ &\mathbf{out} : \mathbf{true}\end{aligned}$$

B. Designs for Reo Timer Channels

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.

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

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

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

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

where

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

Similarly, the *reset*-option allows the timer to be reset to 0 after it has been activated when a special "reset" value is consumed through its source end. A t-timer with the *reset*-option is graphically represented as $A \overset{t}{\circlearrowleft\!\!\textcircled{t}\!\!\longrightarrow} B$ and its design model is given as follows:

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

where

$$\begin{aligned}&RT(\langle \alpha, a\rangle, \langle \beta, b\rangle)\\ \equiv&\begin{cases}\langle \beta, b\rangle = \langle (), ()\rangle & \text{if } \langle \alpha, a\rangle = \langle (), ()\rangle\\ RT(\langle \overrightarrow{\alpha}, \overrightarrow{a}\rangle, \langle \beta, b\rangle) & \text{if } a(1) < a(0) + t \wedge \alpha(1) = \mathit{reset}\\ RT(\langle \overrightarrow{\alpha}, \overrightarrow{a}\rangle, \langle \overrightarrow{\beta}, \overrightarrow{b}\rangle) \wedge b(0) = a(0) + t & \\ \qquad\qquad \text{if } a(0) + t \leq a(1) \vee |a| = 1 & \end{cases}\end{aligned}$$

A timer with early expiration makes the timer produce its timeout signal through its sink and reset itself when it consumes a special "expire" value through its source end. A t-timer with early expiration is graphically represented as $A \overset{t}{\rightleftharpoons\!\!\textcircled{t}\!\!\longrightarrow} B$ and its design model is given as follows:

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

where

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

C. Designs for Reo Circuits

Complex connectors in Reo are organized as a network of channels, which are called *Reo circuits*. Since basic channels can be modeled by designs, their composition can be defined as 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 3.

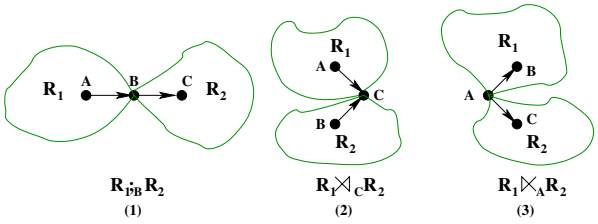

Figure 3. Connector composition

Different channel ends can be joined together into new nodes. If all channel ends coincident on a node are source (sink) channel ends, the node is called a *source (sink) node*. Figure 3(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 node B. In this case, the node B is called a *mixed node* which behaves as a self-contained pumping station. When we compose connectors, we want the events on the mixed nodes to happen silently and automatically whenever they can, without the participation or even the knowledge of the environment of the connector. Such mixed nodes are hidden (encapsulated) by using existential quantification. Figure 3(2) shows the case of merging two sink nodes of the connectors $\mathbf{R}_1$ and $\mathbf{R}_2$. In this case the node C behaves as a merger. An attempt to take a data item from the node succeeds when at least one of its coincident channel ends has a suitable data to offer, in which case the suitable data available through one of its coincident channel ends is non-deterministically selected to be taken. Figure 3(3) shows the case of merging two source nodes of the connectors $\mathbf{R}_1$ and $\mathbf{R}_2$. In this case the node acts as a replicator. Writing a data item to a source node succeeds when all of the coincident channel ends are capable of accepting the data item simultaneously, in which case the data item is atomically copied into every source ends coincident on A. Due to the length limitation, we will only give the definition for merging sink nodes operation here. Details for the other composition operators can be found in [17].

Let $\langle \gamma_i, c_i \rangle$ for $i = 1, 2$ be the timed data sequences on the sink node C in $\mathbf{R}_1$ and $\mathbf{R}_2$, respectively. Then by merging the two sink nodes, the resulting connector is denoted by $\mathbf{R} = \mathbf{R}_1 \rtimes_C \mathbf{R}_2$, and the corresponding design is given as

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

In this definition, the ternary relation M is defined as

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

In this relation M, two timed data sequences $\langle \gamma_1, c_1 \rangle$ and $\langle \gamma_2, c_2 \rangle$ are merged together into one single timed data sequence $\langle \gamma, c \rangle$ where the order of elements in the sequence is decided by the time moments. Furthermore, if we have three timed data sequences $\langle \gamma_1, c_1 \rangle$, $\langle \gamma_2, c_2 \rangle$ and $\langle \gamma, c \rangle$, such that $M(\langle \gamma_1, c_1 \rangle, \langle \gamma_2, c_2 \rangle, \langle \gamma, c \rangle)$ is satisfied, then we say that $\langle \gamma, c \rangle = merge(\langle \gamma_1, c_1 \rangle, \langle \gamma_2, c_2 \rangle)$. Moreover, the merging operation can also be defined on the data and time sequences separately: $\gamma = merge(\gamma_1, \gamma_2)$ and $c = merge(c_1, c_2)$.

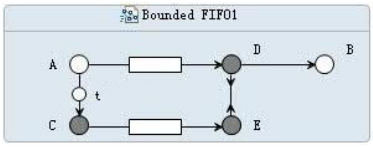

Figure 4. Lower Bounded FIFO1 Channel

Example 4.1: We consider the circuit given in Figure 4 as a simple example. This circuit ensures the lower bound "$> t$" for a take operation on node B; it yields a FIFO1 channel that guarantees every data item will remain in its buffer at least t time units. The design model for this connector can be easily obtained from the composition of the basic channels:

$$\begin{aligned}
\mathbf{con}: &\ \mathbf{LBFIFO1}(in : (A \mapsto \langle \alpha, a \rangle); out : B \mapsto \langle \beta, b \rangle)\\
\mathbf{in}: &\ \mathcal{WD}\langle \alpha, a \rangle \wedge a[+t] \leq \overrightarrow{a}\\
\mathbf{out}: &\ \mathcal{WD}\langle \beta, b \rangle \wedge \beta = \alpha \wedge a[+t] < b \wedge (\overrightarrow{b^R})^R < \overrightarrow{a}
\end{aligned}$$

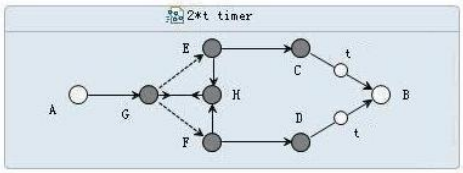

Figure 5. $2 * t$ Timer Connector

Example 4.2: An exciting behavior for a different timer channel $A \xrightarrow{n*t} B$ is to produce a timeout after a delay t for every input (for up to n such inputs), even if the inter-arrival time of the inputs is less than t. The corresponding

design model can be obtained by composition of the basic channels, which is similar as t-timer, but has a different condition on the inputs. In fact, $\overset{n*t}{\longrightarrow\!\bullet\!\longrightarrow}$ can be built by using n timer channels $\overset{t}{\longrightarrow\!\circ\!\longrightarrow}$ and an exclusive router (with n sink nodes). As an example, The connector in Figure 5 shows the construction of $A\overset{2*t}{\longrightarrow\!\bullet\!\longrightarrow}B$, and its design model is given as follows:

$$\begin{aligned}
&\mathbf{con}: \mathbf{ATimer}2t(in:(A\mapsto\langle\alpha,a\rangle);out:(B\mapsto\langle\beta,b\rangle))\\
&\quad\mathbf{in}: \mathcal{WD}\langle\alpha,a\rangle\wedge\exists c,d\in\mathbb{R}^*_+.(c[+t]\leq\overrightarrow{c}\wedge d[+t]\leq\overrightarrow{d}\wedge\\
&\qquad a=merge(c,d))\\
&\quad\mathbf{out}: \mathcal{WD}\langle\beta,b\rangle\wedge\beta\in\{timeout\}^*\wedge b=a[+t]
\end{aligned}$$

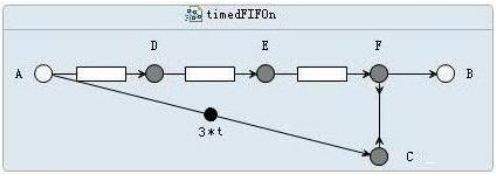

Figure 6. Timed FIFO3 Connector

Example 4.3: A useful connector that can be used to model a real-time network is a timed FIFOn that delays every input for t time units, even if the inter-arrival time of the inputs is less than t (for up to n such inputs). Such a connector can not be obtained by just composing n timed FIFO1 channels. However, it is still easy to be constructed by using $\overset{n*t}{\longrightarrow\!\bullet\!\longrightarrow}$. Figure 6 shows an example of such a timed FIFO3 connector. We can parameterize this connector to have as many buffers as we want simply by inserting more (or fewer) FIFO1 channels between nodes A and F and using a corresponding $\overset{n*t}{\longrightarrow\!\bullet\!\longrightarrow}$ where n is the number of FIFO1 channels, as required. The design model for this connector is as follows[2]:

$$\begin{aligned}
&\mathbf{con}: \mathbf{timedFIFO}n(in:(A\mapsto\langle\alpha,a\rangle);out:B\mapsto\langle\beta,b\rangle)\\
&\quad\mathbf{in}: \mathcal{WD}\langle\alpha,a\rangle\\
&\quad\mathbf{out}: \mathcal{WD}\langle\beta,b\rangle\wedge\beta=\alpha\wedge b=a[+t]\wedge b<a^{(n)}
\end{aligned}$$

Example 4.4: A variant of **FIFO1** is called *expiring FIFO1*, denoted by $A\overset{}{-\!\!\square\!\!\rightarrow_{\leq t}}B$, where a data item is lost if it is not taken out of the buffer through the sink end of the channel within t time units after it enters through its source end. Figure 7 shows how an expiring FIFO1 channel can be constructed out of a normal FIFO1 channel and a timer.

By composing the channels together (and after some equational transformations to simplify the expression), we can get the design model for the expiring FIFO1 as[3]:

$$\begin{aligned}
&\mathbf{con}: \mathbf{ExpFIFO1}(in:(A\mapsto\langle\alpha,a\rangle);out:B\mapsto\langle\beta,b\rangle)\\
&\quad\mathbf{in}: \mathcal{WD}\langle\alpha,a\rangle\wedge a[+t]\leq\overrightarrow{d}\\
&\quad\mathbf{out}: \exists\langle\pi,h\rangle.\mathcal{WD}\langle\beta,b\rangle\wedge\mathcal{WD}\langle\pi,h\rangle\wedge a<merge(b,h)\wedge\\
&\qquad(\overrightarrow{(merge(b,h))^R})^R<\overrightarrow{d}\wedge L(\langle\alpha,merge(b,h)\rangle,\langle\beta,b\rangle)\wedge\\
&\qquad L(\langle timeout^*,a[+t]\rangle,\langle\pi,h\rangle)
\end{aligned}$$

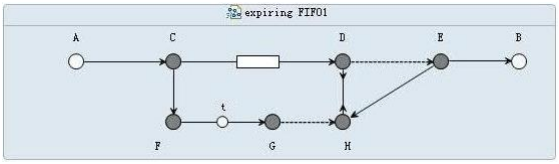

Figure 7. Expiring FIFO1 Channel

V. Implementation

`Tom` [12] is a language extension that can be used with different languages like `C++` or `Java`. In contrast to dedicated languages like `Java`, the advantages of `JTom` (`Tom` in `Java`) include easy prototyping, clear separation between syntax and semantics, and high-level mechanisms like strategies for meta-control of the executions of the programs. `JTom` offers suitable means to quickly prototype our design model for Reo connectors. The reason for choosing `JTom` instead of `Maude` (the language of choice in [1] for implementation of untimed Reo connectors) for implementing the design model for timed connectors is that `JTom` is much more flexible and support pattern matching against native data-structures like objects or records. This is a strong point since we can take advantage of existing efficient implementations of data structures like `ArrayList` or `HashMap` and the corresponding functions to manipulate the basic ingredients in the theory of connectors, which are sequences and streams.

Comparing with the ECT backends for Reo [11] and other tools like Vereofy [22], our implementation in `JTom` contains the refinement checking and test case generation for Reo connectors, which make it possible to verify and validate Reo connectors with respect to their specifications given as designs. [4]. Details about the implementation are not discussed here due to the length limitation.

VI. Conclusion and Future Work

This paper extends our previous work on the design model for (untimed) Reo connectors and introduces the UTP design model for timed Reo connectors. This approach provides a unified semantic model for different kinds of channels and composite connectors, covers different communication

[2]To simplify the expression we consider infinite sequences of inputs and outputs here. For finite sequences with length k, the only difference is to replace the condition $b<a^{(n)}$ by the relationship that $b(i)<a(n+i)$ for any $i\leq k-n$.

[3]Here we use $timeout^*$ to denote the sequence $s\in\{timeout\}^*$.

[4]The source code of the `JTom` implementation can be fetched from `http://www.math.pku.edu.cn/teachers/sunm/utp/`.

mechanisms encoded in Reo, and allows the combination of different untimed and timed channels as in Reo. In our work, we model basic channels in Reo as designs in UTP, and the composition of connectors is specified by design composition. Our semantic model offers potential benefits in developing tool support for Reo, such as test case generator and translator to other languages with UTP semantics like rCOS[14]. The syntax and design semantics for timed Reo connectors are implemented in `JTom`.

Incorporating more complex real-time (and other QoS) constraints on connectors [4], [5] into the design model is an interesting topic in our future work. For example, one can envision a synchronous channel where input and output are amended by real-time constraints, say a data item should be dispensed at the sink within 0.1 second after the input commences at the source. Such a constraint can be specified by replacing $b = a$ in the design model for Sync with $a \leq b \leq a[+0.1]$. The development of refinement and testing theories for timed connectors like refinement and testing for untimed connectors in [1], [17] and integration of such theories into the existing tools for Reo [11] are of special interest and in our scope as well. On the other hand, we will also investigate the relationship between the design model for timed Reo connectors and real-time property specification patterns [8], [16]. The choice of $\mathbb{R}_+$ to model time makes the decidability of the design model an interesting problem.

Acknowledgement.: The author is indebted to Lăcrămioara Aştefănoaei for her help in the `JTom` implementation, and Zongyan Qiu for the valuable questions and comments. The work is partially supported by the Macao Science and Technology Development Fund under the PEARL project, grant number 041/2007/A3.

REFERENCES

[1] B. K. Aichernig, F. Arbab, L. Astefanoaei, F. S. de Boer, S. Meng, and J. Rutten. Fault-based test case generation for component connectors. In *Proceedings of TASE 2009*, pages 147–154. IEEE Computer Society, 2009.

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

[3] 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, *Proceedings of SEFM2004*, pages 198–207. IEEE Computer Society, 2004.

[4] F. Arbab, T. Chothia, S. Meng, and Y.-J. Moon. Component Connectors with QoS Guarantees. In A. L. Murphy and J. Vitek, editors, *Proceedings of Coordination'07*, volume 4467 of *LNCS*, pages 286–304. Springer, 2007.

[5] F. Arbab, T. Chothia, R. van der Mei, S. Meng, Y.-J. Moon, and C. Verhoef. From coordination to stochastic models of qos. In J. Field and V. T. Vasconcelos, editors, *Proceedings of Coordination'09*, volume 5521 of *LNCS*, pages 268–287. Springer, 2009.

[6] F. Arbab and J. Rutten. A coinductive calculus of component connectors. In M. Wirsing, D. Pattinson, and R. Hennicker, editors, *WADT 2002*, volume 2755 of *LNCS*, pages 34–55. Springer-Verlag, 2003.

[7] 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.

[8] P. Bellini, P. Nesi, and D. Rogai. Expressing and organizing real-time specification patterns via temporal logics. *Journal of Systems and Software*, 82(2):183–196, 2009.

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

[10] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. F. Quesada. Maude: specification and programming in rewriting logic. *Theoretical Computer Science*, 285:187–243, 2002.

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

[12] J. Guyon, P.-E. Moreau, and A. Reilles. An integrated development environment for pattern matching programming. *Electronic Notes in Theoretical Computer Science*, 107:33–49, 2004.

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

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

[15] N. Kokash, C. Krause, and E. de Vink. Time and data aware analysis of graphical service models. In *Proceedings of SEFM'10*, pages 125–134. IEEE Computer Society, 2010.

[16] S. Konrad and B. H. C. Cheng. Real-time specification patterns. In *Proceedings of ICSE'05*, pages 372–381. ACM, 2005.

[17] S. Meng and F. Arbab. Connectors as Designs. In *Proceedings of FOCLASA'09*, volume 255 of *ENTCS*, pages 119–135, 2009.

[18] M. Oliveira, A. Cavalcanti, and J. Woodcock. A Denotational Semantics for Circus. *Electronic Notes in Theoretical Computer Science*, 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 *Proceedings of FME'03*, 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, *Proceedings of ICFEM 2002*, volume 2495, pages 613–624. Springer, 2002.

[21] Tom. http://tom.loria.fr/wiki/index.php5/Main_Page.

[22] Vereofy. http://www.vereofy.de.