Reo2PVS: Formal Specification and Verification of Component Connectors

M. Saqib Nawaz and Meng Sun
LMAM & Department of Informatics, School of Mathematical Sciences, Peking University, Beijing, China
{msaqibnawaz, sunm}@pku.edu.cn

***Abstract*—Compositional coordination models such as Reo provide powerful support for the development of large-scale distributed systems by allowing construction of complex connectors that coordinate behavior among different components. The reliability of such distributed systems highly depends on the correctness of connectors. In this paper, we use the proof assistant PVS for formal modeling, analysis and verification of component connectors. We first present the modeling of primitive channels and the composition operators that are used to combine channels for building complex connectors. Furthermore, we show how to model and analyze connector's behavior in PVS and prove some interesting connector properties. The model reflects the original topological structure of connectors simply and clearly. With the provided approach, different kinds of connector properties can be naturally formalized and proved in PVS.**

***Index Terms*—Reo, Connector, PVS, UTP, Design**

I. Introduction

Nowadays, most modern software systems are distributed over large networks of computing devices. However, software components that comprise the whole system usually do not fit together exactly and leave significant interfacing gaps among them. Such gaps are generally filled with additional "*glue code*". Compositional coordination languages offer such a glue code among components and facilitate the mutual interactions between components in a distributed environment. Reo [2] and Linda [11] are two popular examples of such compositional coordination languages, which have played an important role in the success of component-based systems in the past decades.

Reo is a channel-based exogenous coordination language where complex component connectors are orchestrated from channels via certain composition operators. Exogenous coordination [1] means coordination from outside, where the primitives that support the coordination of an entity with others reside outside of that entity. Connectors in Reo provide the protocols that control and organize the communication, synchronization and cooperation among the components that they interconnect. Despite its simplicity, Reo has been used successfully in various application domains, such as service-oriented computing [10], [21], [22], business processes [25] or biological systems [8].

The reliability of component-based systems highly depend on the correctness of connectors. Formal analysis and verification of connectors is gaining more interest in recent years with the evolution of software systems and advancements in Cloud and Grid computing technologies. Furthermore, the increasing growth in size and complexity of computing infrastructure has made the modeling and verification of connector properties a more difficult and challenging task. From the modeling and analysis context, the formal semantics for Reo allows us to specify and analyze the behavior of connectors precisely. In literature, different formal semantics have been proposed for Reo [13], such as the co-algebraic semantics in terms of relations on infinite timed data streams [3], operational semantics using constraint automata [6], the coloring semantics by coloring a connector with possible data flows [9] in order to resolve synchronization and exclusion constraints, and the UTP (Unified Theories of Programming) semantics [20], [23].

In the past decade, a lot of work has been done towards formal verification and analysis of Reo connectors. A symbolic model checker "Vereofy" has been developed in [5] to check the CTL-like properties of systems with exogenous coordination. Another approach is to take advantage of existing verification tools by translating Reo model to other formal models such as Alloy [14], mCRL2 [15], etc. Since infinite behavior is usually taken into consideration for connectors, the modeling and analysis of connectors are expected to be achieved efficiently in theorem provers. In [16], a method for formal modeling and verification of Reo connectors in Coq is provided. Reo connectors were represented in a constructive way and verification was based on the simulation of the behavior and output of Reo connectors. A different modeling and analysis framework in Coq was proposed in [24], which adopted the UTP design model for Reo connectors developed in [20], [23], i.e., a pair of predicates $P \vdash Q$ where the predicate 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. Work done in [24] was extended in [12] to cover the modeling and verification of timed channels and connectors in Coq.

In this paper, the aim is to provide an approach for formal modeling and reasoning about Reo connectors constructed from primitive (both untimed and timed) channels under the UTP semantic framework using the proof assistant PVS [17]. We first provide the modeling for a family of primitive channels and compositional operators in PVS. Then we show how to model and reason about more complex connectors. The basic idea is to model the observable behavior of a connector as a relation on the timed data sequences being observed

DOI reference number: 10.18293/SEKE2018-024

at its input and output nodes. In PVS, this is achieved by representing a connector as a logical predicate that describes the relation among the timed data sequences on its input and output nodes. The model makes it possible to prove complex and generic connectors' properties easily in PVS. The PVS dump file for this work can be found at [19].

The rest of this paper is organized as follows: Reo and PVS are briefly introduced in Section II. PVS specifications for basic definitions being used in the modeling of primitive channels are presented in Section III. In Section IV, formal modeling of primitive channels is described, followed by the modeling of compositional operators. Section V shows how to verify and reason about connector properties in PVS by several examples. Finally, Section VI concludes the paper.

II. Preliminaries

In this section, a brief introduction to the coordination language Reo and the PVS system is provided.

A. Reo

Reo is a channel-based exogenous coordination language where complex *connectors* are compositionally constructed out of simpler ones. Further details on Reo can be found in [2], [6]. Connectors provide the protocol to control and organize the communication, synchronization and cooperation among different components. The simplest connectors are channels with well-defined behavior. Each Reo channel has two channel ends, which can be of type *source* or *sink*. A source channel end accepts data into the channel and a sink channel end dispenses data out of the channel. Few primitive channel types in Reo are shown in Figure 1.

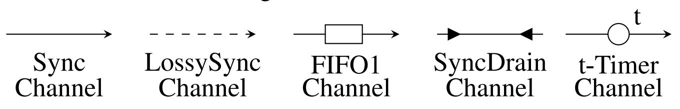

Figure 1. Some primitive channels in Reo

A *synchronous (Sync) channel* has one source and one sink end. I/O operations can succeed only if the writing operation at source end is synchronized with the read operation at its sink end. A *lossy synchronous (LossySync) channel* is a variant of synchronous channel that accepts all data through its source end. The written data is lost immediately if no corresponding read operation is available at its sink end. A *FIFO1 channel* is an asynchronous channel with one buffer cell, one source end and one sink end. The channel accepts a data item whenever the buffer is empty. The data item is kept in the buffer and dispensed to the sink end later in the FIFO order. A *synchronous drain (SyncDrain) channel* has two source ends and no sink end, which means that no data can be obtained from such channels. The write operation on both source ends should happen simultaneously and the data items written to this channel are irrelevant. A *t-timer channel* accepts any data item at its source end and produces a *timeout* signal on its sink end after a delay of t time units.

Complex connectors are constructed by composition of different channels with *join* and *hiding* operations. The result can be represented visually as a graph where a node represents a set of channel ends that are combined together through the join operation, while the edges in the graph represent the channels between the corresponding nodes. Nodes are categorized into *source, sink* or *mixed nodes*, depending on whether the node contains only source channel ends, sink channel ends, or both. Source nodes are analogous to input ports, sink nodes to output ports and mixed nodes are internal details of a connector that are hidden. The internal topology of any connector can be hidden from outside by applying the hiding operation. The behavior of a connector can be captured by the data-flow on its source and sink nodes. The hidden nodes can not be accessed or observed from outside.

B. PVS

PVS (Prototype Verification System) offers a formal specification language and a mechanical theorem proving environment. The PVS system consists of a specification language, a parser, a type-checker, a prover, specification libraries, and various browsing tools. Specification language of PVS is build on a higher order logic and type system of PVS supports predicate sub-typing and other type dependencies. The type system of PVS is not algorithmically decidable and theorem proving may be required to establish the type-consistency of a PVS specification. Theorems that need to be proved are called type-correctness conditions (TCC's). Here, we give a simple example for factorial function that is defined recursively in PVS.

```
factorial(n:nat): RECURSIVE posnat =
 IF n = 0 THEN 1 ELSE factorial(n-1)*n ENDIF
MEASURE n

the: THEOREM FORALL(k:nat): factorial(k) >= k
```

In PVS, recursive definitions are treated as constant declarations and it must be total, so that the function is defined for every value of its domain. In order to ensure this, recursive functions must be specified with a measure (n in the factorial function). Theorem *the* in this example shows that factorial of a number should be greater than or equal to that number. PVS offers inference rules, proof commands and decision procedures that can be used to prove theorems. PVS prover is based on *sequent calculus* where each proof goal is a *sequent* consisting of a sequence of formulas called *antecedents* and a sequence of formulas called *consequents*. The intuitive interpretation of a sequent is that the *conjunction* of the antecedents implies the *disjunction* of the consequents. During proof construction, PVS builds a graphical proof tree in which remaining proof obligations are at the leaves of tree. If a proof gets stuck, then this tree helps to see where the proof goes wrong. Further details on PVS can be found in [18].

III. Basic Definitions in PVS

The behavior of a connector can be formalized by means of data-flows on its sink and source nodes which are essentially infinite sequences. In PVS, record structure is used to represent *timed data (TD)* sequences on the sink and source nodes, where *time* is defined as *positive real numbers* ($\mathbb{R}^+$) and *data* is defined as a *positive* type. The advantage of using the record structure for representing a TD sequence is that it offers names for both time and data of the sequence, which makes the specification more convenient and understandable.

```
Time: Type = posreal
Data: TYPE+
TD: TYPE = [# T: sequence[Time],
              D: sequence[Data] #]
Input, Output: VAR TD
```

A TD is a record structure type that has two components: *T* and *D*. The *D* component is a sequence of *data items*. The *T* component is a sequence of *time points* being used to represent the *time* when the data items in the *D* component being observed. *Input* and *Output* are declared as variables of type TD. The following predicates are used for primitive channels specification later:

```
Teq(Input,Output):bool = T(Input) = T(Output)
Tle(Input,Output):bool = T(Input) < T(Output)
Tgt(Input,Output):bool = T(Input) > T(Output)
Deq(Input,Output):bool = D(Input) = D(Output)
```

Teq takes two TD sequences and returns *true* if the time of two sequences are exactly equal to each other. *Tle* represents that time of the first sequence is strictly less than the second sequence and *Tgt* means that the time of the first sequence is strictly greater than the second sequence. *Deq* shows the equality of data: data sequence at *Input* is equal to data sequence at *Output*. *Teq, Tle* and *Tgt* only checks the time component of the record structure, whereas, *Deq* checks the data field. Since the type of component *T* in TD is defined as $sequence[Time]$, we have to define the operators "$<$" and "$>$" for sequences of times. A strict order (that is both transitive and irreflexive) is assumed for "$<$" and "$>$".

```
<: (strict_order?[sequence[Time]])
>: (strict_order?[sequence[Time]]) =
     LAMBDA (s1, s2: sequence[Time]): s2 < s1
```

Defining "<,>" for sequence of time generated two TCC's. Proof steps for these two TCC's can be found at [19].

For timed channels, three new predicate formulas are introduced, which are similar as the previous definitions for primitive untimed channels with one of the time sequences is added by a t time delay. An extra t is appended to the names of these new predicates to distinguish them from the ones for untimed channels. Definitions Teq, Tle and Tgt can also be specified with the terms used in the $Teqt$, $Tltt$ and $Tgtt$.

```
Teqt(T1,T2)(t:Time): bool = FORALL (n:nat):
  FrS(str_nth(n,T1)) + t = FrS(str_nth(n,T2))

Tltt(T1,T2)(t:Time): bool = FORALL (n:nat):
  FrS(str_nth(n,T1)) + t < FrS(str_nth(n,T2))

Tgtt(T1,T2)(t:Time): bool = FORALL (n:nat):
  FrS(str_nth(n,T1)) + t > FrS(str_nth(n,T2))
```

IV. Reo Channels and Operators

The modeling of primitive untimed / timed channels and operators for connectors composition is presented in this section. These channels and operators are used later in the modeling and analysis of complex connectors.

A. Primitive Channels

For the *Sync* channel, the time and data of a sequence that flows into the channel are exactly the same as those of the sequence flowing out. The channel is modeled as follows in PVS:

```
Sync(Input,Output):bool = Teq(Input,Output) &
                          Deq(Input,Output)
```

The *SyncDrain* channel has two source ends and no sink end. It works as a synchronous channel that allows pairs of write operations pending on its two ends to succeed simultaneously. In this channel, all written data items are consumed and lost. Thus, this channel is used just for synchronizing two TD sequences being observed on its two ends. This channel is specified in PVS as follows:

```
SyncD(Input1,Input2):bool= Teq(Input1,Input2)
```

A *LossySync* channel is analogous to the *Sync* channel, except that the write operation on the source end always succeeds immediately. If a corresponding read operation is already pending on the sink end, then the written data item is transferred to the sink end and both operations succeed. Otherwise, only the write operation on the source end succeeds and the data item is lost. *LossySnc* channel is defined inductively as follows: [*]

```
Lossysync(Input,Output)(n:nat):INDUCTIVE bool
 =  ( nth(Output,n)= nth(Input,n)
    & Lossysync(next(Input),next(Output))(n)
   OR Lossysync(next(Input),Output)(n))
```

Another important channel in Reo is the asynchronous one with buffering capacity 1, known as *FIFO1* channel (—☐→). The time when a *FIFO1* channel takes a data item at its source end is earlier than the time when the data item is delivered at its sink end. Furthermore, the time of the next data item that flows in at the source end should be later than the time when the data in the buffer is delivered at the sink end. The buffer is empty if no data item is in the buffer, and it contains a data element d after d is written through the source end and before d is taken out through the sink end.

```
Fifo1(Input,Output):bool= Tle(Input,Output) &
  Tle(Output,next(Input)) & Deq(Input,Output)
```

A *FIFO1e* channel is a variant of *FIFO1* where the buffer already contains a data element "e". The communication can only be initiated when e is taken out through the sink end. So the data sequence that flows out of the channel always get an extra element e settled at the beginning of the sequence.

[*]In PVS, inductive definitions are similar to recursive definitions, in that both involve induction and must satisfy additional constraints to guarantee that they are total.

Moreover, the time of the sequence that flows into the channel should be earlier than time of the tail of the sequence that flows out. As the buffer contains *e*, new data can be written into the channel only after the element *e* has been taken. Therefore, time of the sequence that flows out is earlier than time of the sequence that flows in.

```
Fifo1e(Input,Output)(e:Data)(n:nat): bool =
  Tgt(Input,Output) & Tle(Input,next(Output)
                    & e?(nth(Output,n))(e) &
  Deq(Input,(next(Output)))
```

The *t-timer channel* accepts input data through its source end and returns a *timeout* signal on its sink end exactly after a duration of *t* time units. It is specified in PVS as follows:

```
Timert(Input,Output)(t:Time)(d:Data): bool =
 FORALL(n:nat): FrS(str_nth(n,Input)) + t <
                FrS(str_nth(n,(next(Input))))
 & Teqt(Input,Output)(t)
 & SrF(str_nth(n,Output)) = timeout(d))
```

The definitions of more channels can be found at [19]. Defining primitive channels by intersection and disjunction of predicates in PVS makes the modeling of channels more concise, easy to understand as each predicate describes a simple order relation (requirement) on time or data. Furthermore, we can easily split the predicates for proofs of different properties which can make the reasoning and proving process simpler.

B. Operators Modeling in PVS

Three main composition operators (shown in Figure 2) are used in Reo for connector construction, which are (i) flow-through, (ii) replicate and (iii) merge.

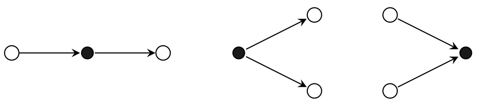
Figure 2. Operators for channel composition

The *flow-through* operator simply allows data items to pass the mixed node. It can be achieved explicitly without specifying it in PVS. This is explained with the simple example in Figure 3, which represents a flow-trough operator that connects two channels $Sync(A,B)$ and $FIFO1(B,C)$ at node B. Such a flow-through operation at node B can be implicitly implemented by just writing the connector as "$Sync(A,B) \wedge FIFO1(B,C)$".

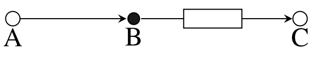

Figure 3. A connector composed of a Sync and a FIFO1 channel

The *replicate* operator puts the source ends of different channels together into one source node. Write operation on this node succeeds only if all the channels are capable of consuming a copy of the written data. Similar to the flow-through operator, it can be implicitly represented by the structure of connectors. For example, If we put one $Sync(A,B)$ channel and one $FIFO1(C,D)$ channel together, we can simply write $Sync(A,B) \wedge FIFO1(A,D)$ in PVS instead of defining a recursive or inductive function, and the replicate operator is achieved directly by renaming C with A for the $FIFO1$ channel. The use of conjunction and node renaming for flow-through and replicate operator allows us to define connectors directly in lemmas and theorems.

The modeling of *merge* operator is a bit more complicated. When the merge operator acts on two channels, it leads to a choice of taking the data item from one of them. The merge operator is defined inductively as the intersection of two predicates.

```
Merge(s1,s2,s3)(n:nat): INDUCTIVE bool =
 ( NOT (FrS(nth(s1,n))) = (FrS(nth(s2,n)))
 AND (((FrS(nth(s1,n)) < (FrS(nth(s2,n))))
   IMPLIES nth(s3,n) = nth(s1,n))
    & Merge(next(s1), s2, next(s3))(n))
 AND (((FrS(nth(s1,n)) > (FrS(nth(s2,n))))
  IMPLIES nth(s3,n) = nth(s2,n))
   & Merge(s1, next(s2), next(s3))(n)))
```

V. Reasoning about Connectors

In this section, we investigate and prove some interesting properties for connectors in PVS.

Example 1. *We first consider the connector shown in Figure 3. Let a,b,c denote the time sequences when the corresponding data sequence flows through nodes A, B and C. According to the semantics of Sync and $FIFO1$ channels, we know that $a = b < c$. Let α, β represent the data sequence being observed at the source node (A) and the sink node (C) respectively, we have $\alpha = \beta$. In PVS, these results are proved with the following theorem.*

Theorem 1. *Sync(A,B) $\wedge$ Fifo1(B,C) $\Rightarrow$ Tle(A,C) $\wedge$ Teq(A,B) $\wedge$ Deq(A,C)*

Proof. When the PVS proof checker is run on this theorem, it gives the following sequent (proof goal) which consists of no antecedent and one consequent formula:

```
|-------
{1} FORALL (A,B,C:TD):
    Sync(A,B) & Fifo1(B,C) => Tle(A,C)
    & Teq(A,B) & Deq(A,C)
```

The "*skolem!*" command is used first that creates a fresh free *skolem* variable for the universal quantifier ($\forall$) in consequent. Then the "$flatten$" command is used to simplify the proof goal by removing $=>$ from consequent. This changes the formula to:

```
{-1} Sync(A!1, B!1)
{-2} Fifo1(B!1, C!1)
  |-------
{1} Tle(A!1,C!1) & Teq(A!1,B!1) & Deq((A!1,C!1)
```

It now consists of two antecedent formulas and one consequent formula. In next steps, the definitions of $Sync$ and $FIFO1$ channels, as well as predicates Tle, Teq, Deq and

$next$ are expanded with the command ($expand$ "*id*"). Conjunction (&) in the antecedents are removed with the command "$flatten$" . The formula now simplifies to:

```
{-1} T(A!1) = T(B!1)
{-2} D(A!1) = D(B!1)
{-3} T(B!1) < T(C!1)
{-4} T(C!1) < (suffix(B!1`T, 1))
{-5} D(B!1) = D(C!1)
  |-------
[1] T(A!1)<T(C!1) & T(A!1)=T(B!1) & D(A!1)=D(C!1)
```

The sequent is then divided (with "*split*" command) into three sub-sequents (sub-goals), which are all proved with decision procedure command "$assert$". [†] □

Example 2. *We now consider the Lower Bounded FIFO1 connector as given in Figure 4. This connector has one source node A and one sink node B. It ensures the lower bound "$>t$" for the take operation on node B. Every data item received by this connector need to stay in its buffer for more than t time units. Let α, β represents the data sequences being observed at nodes A and B, and a, b represents the time sequences corresponding to α and β, i.e., the i-th element $a(i)$ in a (and $b(i)$ in b) denotes exactly the time moment of the occurrence of $\alpha(i)$ (and $\beta(i)$). For this connector, it is proved in theorem 2 that $\alpha = \beta$ and $a + t < b$.*

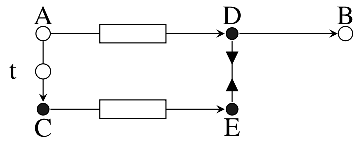

Figure 4. Lower Bounded FIFO1 Connector

Theorem 2. *∀ A,B,C,D,E ∈ TD, t ∈ Time, d ∈ Data:*

Timert(A,C)(t)(d) ∧ Fifo1(A,D) ∧ SyncD(D,E) ∧ Fifo1(C,E) ∧ Sync(D,B) ⇒ Deq(A,B) ∧ Tltt(A,B)(t)

Proof. After applying $skolemization$, $expansion$ and $flattening$, the main goal is split into two sub-goals. The first sub-goal is for the data dimension, i.e., the data sequence being received at A should be equal to the data sequence being taken at B. The $Fifo1(A, D)$ channel satisfies the predicate $Deq(A, D)$ and the $Sync(D, B)$ channel satisfies $Deq(D, B)$. The conjunction of both predicates results in $Deq(A, B)$. For the time dimension, we have predicates $Teqt(A, C, t)$, $Tlt(C, E)$, $Teq(E, D)$ and $Teq(D, B)$, which can be obtained from the definitions of $Timert$, $Fifo1$, $SyncD$ and $Sync$ channels respectively. These four predicates introduce the constraints $A+t = C$, $C < E$, $E = D$, $D = B$ for time. The combination of these predicates results in $Tltt(A, B, t)$, such that $A + t < B$ holds for time. □

Example 3. *Figure 5 shows an expiring FIFO1 connector that can be constructed with a normal* FIFO1 *channel and a t-timer (and some other channels). In this connector, a data item received through the source node A is dropped from the buffer if it is not taken out through the sink node B within t time units.*

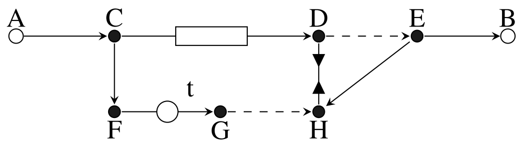

Figure 5. Expiring *FIFOn* channel

Theorem 3. *∀ A,B,C,D,E,F,G,H ∈ TD, t ∈Time, d ∈ Data, n ∈ nat:*

Sync(A,C) ∧ Sync(C,F) ∧ Fifo1(C,D) ∧ Timert(F,G)(t)(d) ∧ Lossysync(G,H)(n) ∧ SyncD(D,H) ∧ Lossysync(D,E)(n) ∧ Sync(E,H) ∧ Sync(E,B) ⇒ Teqt(A,B)(t) ∧ Tgt(B,A)

Proof. The PVS proof process of this theorem is presented as following:
($induct$ "n") (The proof starts by applying induction on n)
Main goal is split into two sub-goals. For the first sub-goal (the base case):
($skosimp$) ($expand$ $Fifo$ -3) ($expand$ "*Lossysync*") ($expand$ "*Sync*") ($expand$ "*SyncD*") ($expand$ "*Timert*") ($expand$ "Teq") ($expand$ "$Teqt$") ($expand$ "Tgt") ($assert$) ($split$)
The first sub-goal is split into two more sub-goals. For the first sub-sub-goal:
($skosimp$) ($inst?$ -10) ($skosimp$) ($assert$).
This proves the first sub-sub-goal.
For the second sub-sub-goal:
($assert$)($inst?$ -5) ($skosimp$) ($assert$).
This proves the second sub-sub-goal and the proof of the first sub-goal is complete.
For the second sub-goal:
($skosimp*$) ($inst?$ -1) ($expand$ "$Fifo1$") ($assert$) ($expand$ "$Teqt$") ($expand$ "Tgt") ($skosimp$) ($assert$) ($split$)
The second sub-goal is divided into two more sub-goals. For the first sub-sub-goal:
($skosimp$)($typepred$ "t!1") ($inst?$ -3) ($assert$) ($grind$).
This proves the first sub-sub-goal.
For the second sub-sub-goal:
($assert$) ($typepred$ "$>$") ($expand$ "*strict_order*") ($flatten$) ($expand$ "*transitive*") ($assert$) ($grind$).
This proves the second sub-sub-goal and the proof of the second sub-goal is complete.
This completes the proof of the theorem. □

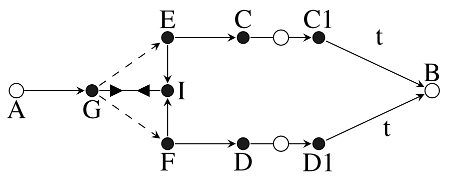

Figure 6. $2 \times t$ Timer Connector

Example 4. *A timed connector $\xrightarrow{n\times t}$ can be built by using n t-timer channels and an exclusive router (with n sink nodes).*

[†]Note that a theorem in PVS can be proved in different ways that depends on the proof commands, inference rules and decision procedures being used.

Such a connector produces a $timeout$ signal after a delay t for every input it receives. The duration between the arrival time for the i-th input and that for the $(i+j)$-th input for $j < n$ can be less than t whereas the duration between the arrival time for the i-th input and that for the $(i+n)$-th input should be at least t. Figure 6 shows the topology structure of $\overset{2\times t}{\longrightarrow\!\!\bullet\!\!\longrightarrow}$.

Let a, b represent the time sequences corresponding to the data sequences flowing into A and out of B, respectively. Theorem 4 states the property that $a + t = b$ for the $2 \times t$ timed connector.

Theorem 4. *∀ A,B,C,D,E,F,G,I,C1,D1 ∈ TD, t ∈Time, d ∈ Data, n ∈ nat:*

Sync(A,G) ∧ Lossysync(G,E)(n) ∧ Lossysync(G,F)(n) ∧ Sync(E,I) ∧ Sync(F,I) ∧ SyncD(G,I) ∧ Merge(E,F,I)(n) ∧ Sync(E,C) ∧ Sync(F,D) ∧ Timert(C,C1)(t)(d) ∧ Timert(D,D1)(t)(d) ∧ Merge(C1,D1,B)(n) ⇒ Teqt(A,B)(t)

Since the $Lossysync$ channel and the composition operator $merge$ are both defined inductively, this theorem is proved by induction on the parameter n. The main goal is divided into two sub-goals. The first sub-goal is for the base case and the second subgoal is for the inductive step. The details of the proof process is similar to the proofs for previous theorems in this section, and we omit it here due to the length limitation.

VI. Conclusion

This paper presents a method for formal modeling of Reo connectors and reasoning about Reo connectors in PVS. The formalization is based on the UTP design semantics for Reo and preserves the original structure and behavior semantics of Reo channels and composition operators, which makes their description in PVS reasonably readable. Connector properties are specified with predicates which offer an appropriate description of the relations between different timed data sequences being observed on the nodes of a connector. The proofs of connector properties are completed with the help of PVS proof-commands, inference rules and decision procedures. Generalized property for connectors for arbitrary n, which cannot be verified explicitly with model checkers, can be proved here as well.

The main problem of this approach is that the analysis and proof process of complex connector properties in PVS always requires heavy interactions between users and the proof assistant, and thus consumes a lot of time. Even for simple properties, the proof process can become hard and requires the users to have good knowledge on PVS to make the proof successfully. In the future, efforts will be made to encapsulate frequently-used proof patterns as PVS strategies in order to make the proof process easier and reduce repetitive work. Machine learning techniques are also expected to provide some help to automate the proof process and reduce the amount of human efforts. On the other hand, extension of the approach to deal with probabilistic [4] or stochastic [7] behavior of connectors is in our plan for future work as well.

Acknowledgement. The work was partially supported by the National Natural Science Foundation of China under grant no. 61772038, 61532019, 61202069 and 61272160.

References

[1] F. Arbab. The IWIM Model for Coordination of Concurrent Activities. In *Proceedings of COORDINATION 1996*, pages 34–56, 1996.

[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 and J. Rutten. A Coinductive Calculus of Component Connectors. In *Proceedings of WADT 2002*, volume 2755 of *LNCS*, pages 34–55. Springer-Verlag, 2002.

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

[5] C. Baier, T. Blechmann, J. Klein, S. Klüppelholz, and W. Leister. Design and Verification of Systems with Exogenous Coordination using Vereofy. In *Proceedings of ISoLA 2010*, volume 6416 of *LNCS*, pages 97–111. Springer, 2010.

[6] 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.

[7] C. Baier and V. Wolf. Stochastic Reasoning about Channel-Based Component Connectors. In *Proceedings of COORDINATION 2006*, volume 4038 of *LNCS*, pages 1–15. Springer-Verlag, 2006.

[8] D. Clarke, D. Costa, and F. Arbab. Modelling Coordination in Biological Systems. In *Proceedings of ISoLA'04*, volume 4313 of *LNCS*, pages 9–25. Springer, 2004.

[9] D. Clarke, D. Costa, and F. Arbab. Connector Coloring I: Synchronization and Context Dependency. *Science of Computer Programming*, 66(3):205–225, 2007.

[10] N. Diakov and F. Arbab. Compositional Construction of Web Services using Reo. In *Proceedings of ICEIS 2004*, pages 13–14, 2004.

[11] D. Gelernter and N. Carriero. Coordination Languages and their Significance. *Communications of the ACM*, 35(2):96, 1992.

[12] W. Hong, M. S. Nawaz, X. Zhang, Y. Li, and M. Sun. Using Coq for Formal Modeling and Verification of Timed Connectors. In *Proceedings of SEFM 2017*, volume 10729 of *LNCS*, pages 558–573. Springer, 2017.

[13] S. T. Q. Jongmans and F. Arbab. Overview of Thirty Semantic Formalisms for Reo. *Scientific Annals of Computer Science*, 22(1):201–251, 2012.

[14] R. Khosravi, M. Sirjani, N. Asoudeh, S. Sahebi, and H. Iravanchi. Modeling and Analysis of Reo Connectors using Alloy. In *Proceedings of COORDINATION 2008*, volume 5052 of *LNCS*, pages 169–183. Springer, 2008.

[15] N. Kokash, C. Krause, and E. de Vink. Reo+mCRL2: A Framework for Model-checking Dataflow in Service Compositions. *Formal Aspects of Computing*, 24:187–216, 2012.

[16] Y. Li and M. Sun. Modeling and Verification of Component Connectors in Coq. *Science of Computer Programming*, 113(3):285–301, 2015.

[17] S. Owre, J. M. Rushby, and N. Shankar. PVS: A Prototype Verification System. In *Proceedings of CADE 1992*, pages 748–752. Springer, 1992.

[18] S. Owre, N. Shankar, J. M. Rushby, and D. W. J. Stringer-Calvert. PVS System Guide, PVS Prover Guide, PVS Language Reference. Technical report, NASA, November 2001.

[19] PVS dump files. Available at: https://github.com/saqibdola/Reo-in-PVS.

[20] M. Sun. Connectors as Designs: The Time Dimension. In *Proceedings of TASE 2012*, pages 201–208. IEEE Computer Society, 2012.

[21] M. Sun and F. Arbab. Web Services Choreography and Orchestration in Reo and Constraint Automata. In *Proceedings of SAC'07*, pages 346–353. ACM, 2007.

[22] M. Sun and F. Arbab. A Model for Web Service Coordination in Long-Running Transactions. In *Proceedings of SOSE'10*, pages 121–128. IEEE Computer Society, 2010.

[23] M. Sun, F. Arbab, B. K. Aichernig, L. Astefanoaei, F. S. de Boer, and J. Rutten. Connectors as Designs: Modeling, Refinement and Test Case Generation. *Science of Computer Programming*, 77(7-8):799–822, 2012.

[24] X. Zhang, W. Hong, Y. Li, and M. Sun. Reasoning about Connectors in Coq. In *Proceedings of FACS 2016*, volume 10231 of *LNCS*, pages 172–190. Springer, 2016.

[25] Z. Zlatev, N. Diakov, and S. Porkaev. Construction of Negotiation Protocols for E-Commerce Applications. *ACM SIGecom Exchanges*, 5(2):12–22, 2004.