Using Coq for Formal Modeling and Verification of Timed Connectors

Weijiang Hong, M. Saqib Nawaz, Xiyue Zhang, Yi Li and Meng Sun

Department of Informatics and LMAM, School of Mathematical Sciences,
Peking University, Beijing, China
{wj.hong,msaqibnawaz,zhangxiyue,liyi_math,sunm}@pku.edu.cn

Abstract. Formal modeling and verification of connectors in component-based software systems are getting more interest with recent advancements and evolution in modern software systems. In this paper, we use the proof assistant Coq for modeling and verification of timed connectors. We first present the definition of timed channels and the composition operators for constructing timed connectors in Coq. Basic timed channels are interpreted as axioms and inference rules are used for the specification of composition operators. Furthermore, timed connectors being built by composing basic timed / untimed channels, are defined as logical predicates which describe the relations between inputs and outputs. Within this framework, timed connector properties can be naturally formalized and proved in Coq.

Keywords: Reo, Timed Connector, Coq, Modeling, Verification

1 Introduction

Most modern software systems today 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 interfacing gaps are generally filled with additional code known as "glue code". Compositional coordination languages offer such a "glue code" for components and facilitate the mutual interactions between components in a distributed processing environment. Reo [1] and Linda [12] 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. Connectors 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 [11, 20], business processes [23] or biological systems [7].

The criterion that how much we can rely on component-based systems is drastically based on correctness of component 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 connectors properties a more difficult and challenging task. From modeling and analysis context, the formal semantics for Reo allow us to specify and analyze the behavior of connectors precisely. In literature, different formal semantics have been proposed for Reo, such as the coalgebraic semantics in terms of relations on infinite timed data streams [3], operational semantics using constraint automata [5], the coloring semantics by coloring a connector with possible data flows [8] in order to resolve synchronization and exclusion constraints, and the UTP (Unified Theories of Programming) semantics [21].

Our aim in this paper is to provide an approach for formal modeling and reasoning about timed Reo connectors under the UTP semantic framework [19] in the proof assistant Coq [13]. Our mechanized verification of connectors in Coq is certainly not the first one. Much work has been carried out in the past for formal verification of connectors. Baier et al. [4] developed a symbolic model checker Vereofy for checking 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. However, since infinite behavior is usually taken into consideration for connectors during the modeling and verification process of their properties, the analysis and verification can not be achieved efficiently in model checking approach because of the large number of states that may lead to state-space explosion problem [9]. By contrast, theorem proving can handle infinite system behavior efficiently. In [16], we provided a method for formal modeling and verification of Reo connectors in Coq. Reo connectors were represented in a constructive way and verification was based on the simulation of the behavior and output of Reo connectors. Later in [22] a different approach was proposed, where primitive channels and connectors in Reo were modeled and analyzed in Coq, based on the UTP semantics.

Both [16] and [22] only took untimed connectors into consideration. A family of timed channels and connectors has been provided in [2,19], which can be used to measure the time elapsed between two events at input/output nodes and specify timed behavior happening in coordination. In this paper, the modeling and verification framework for connectors in [22] is extended to cover timed connectors as well. We first provide the definition for a family of timed channels in Coq, then we present our approach on how to model and reason about timed connectors. The basic idea is to model the observable behavior of a (timed) connector as a relation on the timed data streams as its input and output. In Coq, this can be naturally achieved by representing a connector as a logical predicate that describes the relation among the timed data streams on its input and output nodes. The details of the implementation in Coq can be found at [10].

The rest of this paper is organized as follows: Reo and Coq are briefly discussed in Section 2. Specifications for timed data streams, some pre-defined auxiliary functions and predicates in Coq are presented in Section 3, followed by the

formal modeling of basic timed channels and compositional operators in Section 4. Section 5 provides the approach for reasoning about connector properties in our framework. Finally, Section 6 concludes the paper and discusses some future research directions.

2 Preliminaries

A brief introduction to the coordination language Reo and the proof assistant Coq is provided in this section.

2.1 Reo

Reo is a channel-based exogenous coordination language where complex *component connectors* are compositionally constructed out of simpler ones. Further details on Reo can be found in [**?**, 3, 5]. Connectors provide the protocol to control and organize the communication, synchronization and cooperation between concurrent components. The simplest connectors are channels with well-defined behavior. Each channel has two channel ends and there are two types of channel ends in Reo: *source* ends and *sink* ends. A source channel end accepts data into the channel and a sink channel end dispense data out of the channel. Few examples of basic channel types in Reo are shown in Fig 1.

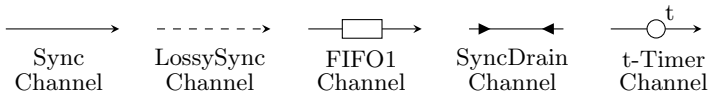

Fig. 1. Some basic channels in Reo

A *synchronous (Sync) channel* has one source and one sink end. I/O operations can succeed only if writing operation at source end is synchronized with 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 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 sourced ends should happen simultaneously and the data items written to this channel are irrelevant. A *t-timer* channel accepts any data items at its source end and produces a *timeout* signal after a delay of t time units on its sink end.

In Reo, complex connectors are constructed by composition of different channels with *join* and *hide* operations on the channel ends. 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 contain only source channel ends, sink channel neds, or both source and sink channel ends.

The internal topology of any component connector can be hidden with hiding operations. The hidden nodes can not be accessed or observed from outside. The behavior of a Reo connector can be formalized by data-flow at its source and sink nodes. Source nodes are analogous to input ports, sink nodes to output ports and mixed nodes are internal details of a connector that is hidden.

2.2 Coq Proof Assistant

Coq is a widely used proof assistant that is based on higher order logic and λ-calculus. It offers a formal specification language called *Gallina* and a mechanical (semi-interactive) theorem proving environment. *Gallina* can be used for writing specifications, mathematical definitions, propositions and functions, executable algorithms and theorems, for example:

```
(* Variables declaration *)
Variables a b: nat.
(* Factorial Function *)
Fixpoint fact(n: nat): nat=
 match n with
  | O => 1
  | S n‘ => n * fact n‘
 end
(* Theorem Declaration *)
Theorem fact_gre n m: n <= m -> fact n <= fact m
 Proof.
 (* interactive theorem proving *)
 auto.
 Qed.
```

This example describes a recursive function for computing factorial of a natural type number. Theorem can be proved interactively with tactics that Coq offers. Coq is also equipped with a rich set of standard libraries. Some of the libraries that we used in this work include *Reals*, *Stream*, *Arith* and *Logic*. Proofs can be reconstructed in other proof assistants such as Isabelle [17] and PVS [18]. Further details on Coq can be found in [6, 13].

3 Basic Definitions in Coq

The notion of timed data (TD) streams and some pre-defined auxiliary functions and predicates in Coq are briefly introduced in this section. These functions and predicates are used in the following sections for modeling timed channels and compositional operators.

In our previous work [22], behavior of connectors is formalized by means of data flows at its sink and source nodes. In Coq, such behavior over infinite data-flows is modeled by defining *TD streams* as follows:

```
Definition Time := R.
Definition Data := nat.
(*Inductive Data : Set :=
   |Natdata : nat-> Data
   |Empty : Data.*)
Definition TD := Time * Data.
```

Time is represented by real numbers ($\mathbb{R}$) and data by natural numbers ($\mathbb{N}$). The continuous time model captured by $\mathbb{R}$ is suitable as it is very expressive and close to the nature of time in the real world. Therefore, the time sequence consists of increasing and diverging time moments. Representation of data with natural numbers enables us to expand the model for different application domains by *Inductive*. Cartesian product of time and data items defines a TD object. The stream module in Coq is used to produce streams of TD objects.

To capture the timed behavior of connectors, we extend our model in [22] from primitive channels to timed channels. One of the important differences between primitive channels and timed channels is the time requirement of input TD streams. For primitive channels, we only require that time of input TD streams is a simulation of real time which means that time stream is increasing as time passes by. Thus, we have the following representative definition about time stream where the terms *PrL* and *PrR* take a pair of values (a, b) as argument and return the first or second value of the pair, respectively.

```
Axiom Inc_T : forall (T: Stream TD) (n:nat),
PrL(Str_nth n T) < PrL(Str_nth n (tl T)).
```

However, for basic timed channels, we require that another input data is not accepted when there is no timeout signal for the last data. This constraint is reflected in the construct of basic timed channels as follows (the variable t is defined in the respective timed channel):

```
forall n:nat,PrL(Str_nth n Input) + t < PrL(Str_nth n (tl Input))
```

We use the basic predicate formulas of judgment about time and data in [22] and introduce some new judgment definitions for timed channels. As defined in [22], predicate *Teq* means that the time components of two streams are equal. *Tlt* means that each time dimension of the first stream is less than the other stream while *Tgt* means that every time dimension of the first stream is greater than the other stream. The judgment about equality of data which is defined as *Deq* is analogous to the judgment of time. The following three definitions that serve to facilitate the modeling of timed channels are plain and easy to understand with one of the time streams is added by a t time delay. An extra t is appended to the names of these new predicates about judgment of time to distinguish them from the original ones.

```
Definition Teqt(s1 s2:Stream TD)(t:Time): Prop :=
 forall n:nat, PrL(Str_nth n s1) + t = PrL(Str_nth n s2)
Definition Tltt(s1 s2:Stream TD)(t:Time): Prop :=
 forall n:nat, PrL(Str_nth n s1) + t < PrL(Str_nth n s2)
Definition Tgtt(s1 s2:Stream TD)(t:Time): Prop :=
 forall n:nat, PrL(Str_nth n s1) + t > PrL(Str_nth n s2)
```

Teqt means that time of the second stream is equal to time of the first stream with an addition of t time units. *Tltt* represents that time of the first stream with an addition of t is less than the second stream and *Tgtt* has the opposite meaning to *Tltt*.

4 Basic Timed Channels and Operators

In this section, we describe the modeling of a few timed channels in Reo that can be used to measure the time between two events and produce timeout signals. We also show the definition of composition operators to construct timed connectors.

4.1 Basic Timed Channels

Predicates are used to describe the constraint on time and data for timed channels in Coq, and such predicates can be combined (with intersection) together to provide the complete specification of timed channels. This approach is not only simple but it also offers convenience for the analysis and proof of timed connector properties. In the following, we present the formal definition of some basic timed channels in Coq.

t-Timer: The basic *t-timer channel* $A \overset{t}{\text{—}\circ\longrightarrow} B$ accepts any input value through its source end A and returns a *timeout* signal on its sink end B exactly after a delay of t time units. The following definition specifies the design model for the *t-timer* channel in Coq.

```
Parameter timeout: Data
Definition Timert (Input Output: Stream TD)(t: Time): Prop:=
  (forall n:nat,
  PrL(Str_nth n Input) + t < PrL(Str_nth n (tl Input)))
  /\ Teqt Input Output t
  /\ forall n:nat, PrR (Str_nth n Output) = timeout
```

In this specification, the first branch requires that an input data item cannot be accepted by the channel when there is no timeout signal for the previous data item it receives. This means that there should be no other input actions during the delay of t time units. The second requirement describes the relation on the time dimension of the input and output streams. The last one presents that after a delay of t time units for every data item it received at the source end, a timeout signal will be generated at the sink end.

OFF-t-Timer: A *t-timer* with the *off*-option $A \multimap\!\!\textcircled{t}\!\!\longrightarrow B$ which allows the timer to be stopped before the expiration of its delay t is designed in case users require the timer to stop working as soon as possible. Under this circumstance, a special *off* value whose type is also assumed to be *Data* can be consumed through the source end. We define the *t-timer* with the *off*-option inductively as follows:

```
Parameter off: Data.
Parameter OFFTimert: Stream TD -> Stream TD -> Time -> Prop.
Axiom OFFTimert_coind:
 forall (Input Output: Stream TD)(t:Time),
 OFFTimert Input Output t ->
 (forall n:nat, PrR(Str_nth n Input) = off  \/
  PrL(Str_nth n Input) + t  < PrL(Str_nth n (tl Input)))
 /\ (  (PrR (hd (tl Input)) = off) ->
 (OFFTimert (tl (tl Input)) Output t))
 /\ ((~PrR (hd (tl Input)) = off) ->
 (PrL (hd Output) = PrL (hd Input) + t)
 /\ (PrR (hd Output) = timeout)
 /\ OFFTimert (tl Input) (tl Output) t).
```

The first predicate in the specification specifies the behavior of input, whereas the next two predicates specify the output behavior. There are some requirements on the inter-arrival time of the inputs that depends on the value of incoming data. So input need to meet such requirements which are specified in the first predicate. For the output, different actions are taken to deal with the data element to be accepted, which is $PrR\ (hd\ (tl\ Input))$. There are two cases which are captured by the two predicates respectively:

- Case I: If the data element to be accepted is *off*, the timer is stopped and we remove this data *off* and the current data without any outputs. Then resume a new input stream, i.e. *OFFTimert (tl (tl Input)) Output t*.
- Case II: If the data element to be accepted is not *off*, a *timeout* signal is produced as output after a delay of t time units. Then resume a new input stream, i.e *OFFTimert (tl Input) (tl Output) t*.

RST-t-Timer: Similarly, a *reset*-option is needed when the users require the timer to be reset to 0 at once. A *t-timer* channel with the *reset*-option $A \multimap\!\!\textcircled{t}\!\!\longrightarrow B$ is activated as soon as a special *reset* value is consumed through its source end.

EXP-t-Timer: When an early expiration is needed for a t-timer channel, like the above two cases, an *expire* value is needed for input. For the t-timer channel with *expire*-option $A \multimap\!\!\textcircled{t}\!\!\longrightarrow B$, once the *expire* value is consumed through the source end, a *timeout* signal is produced through the sink end instantaneously.

The modeling of the RST-t-Timer and the EXP-t-Timer in Coq are similar to the OFF-t-Timer. The details can be found at [10].

In Coq, specifying timed channels by intersection of predicates makes the model intuitive and concise as each predicate describes a simple order relation

on time or data. Moreover, we can easily split these predicates to make the process of proving connector properties simpler.

4.2 Modeling Operators in Coq

We now describe how composition operators for construction of complex connectors from basic channels can be modeled in Coq. We have three types of composition operators which are graphically represented in Fig. 2:

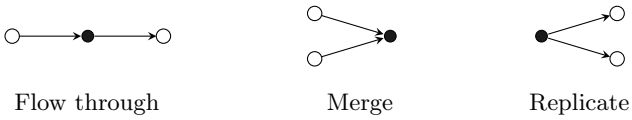

Fig. 2. Channel Composition Operators

The *flow-through* operator which acts on mixed nodes simply allows data items to flow through the junction node, from one channel to the other. Thus, we need not to give the *flow-through* operator a specific definition which can be achieved implicitly. For example, while we specify two channels $Sync(A, B)$ and $FIFO1(B, C)$, the *flow-through* operator that acts on node B for these two channels has been achieved.

Similar to the *flow-through* operator, the *replicate* operator can also be achieved implicitly by means of renaming. For example, for channels $Sync(A, B)$ and $FIFO1(C, D)$, we can illustrate $Sync(A, B)$ and $FIFO1(A, D)$ instead of defining a function like $rep(Sync(A, B), FIFO1(C, D))$ and the *replicate* operator is achieved directly by renaming C with A for the FIFO1 channel.

The modeling of the *merge* operator in Coq is more complicated. When the *merge* operator acts on two channels AB and CD, it leads to a choice of data items being taken from AB or CD. Similar to the definition of timed channels, we define *merge* as the intersection of two predicates and use recursive definition:

```
Parameter merge:Stream TD -> Stream TD -> Stream TD -> Prop.
Axiom merge_coind:
   forall s1 s2 s3:Stream TD,
   merge s1 s2 s3->(~(PrL(hd s1) = PrL(hd s2))
   /\ (((PrL(hd s1) < PrL(hd s2))  ->
   ((hd s3 = hd s1)  /\ merge (tl s1) s2 (tl s3)))
   /\ ((PrL(hd s1) > PrL(hd s2))   ->
   ((hd s3 = hd s2)  /\ merge s1 (tl s2) (tl s3))))).
```

The three timed data streams *s1*, *s2*, *s3* are located in two source ports and one sink port, respectively. If time corresponding to the first data of *s1* is less than time corresponding to the first data of *s2*, then the first data and time elements of stream *s3* are equal to the elements of stream *s1*. Meanwhile, *Merge*

is called again for the next data and time elements, but the arguments need to be changed to *tl s1*, *s2* and *tl s3*. For the circumstance that time corresponding to the first data of *s1* is greater than time corresponding to the first data of *s2*, the constraints are similar to the first case.

5 Reasoning about Connectors

With time channels, we can analyze and prove some interesting and important properties of timed connectors in Coq. In this section, we give some examples to elucidate how to reason about timed connectors properties.

5.1 Derivation of Some Lemmas

In Coq, the proof process of a property (in the form of a theorem) is as follows: First the user states the proposition that needs to be proved, called a *goal*. Then user applies commands called *tactics* to decompose this goal into simpler subgoals or solve it directly. This decomposition process ends when all subgoals are completely solved. Before proving properties, we first introduce some lemmas that are used to facilitate the proofs.

```
[1]Lemma Eq_Tltt : forall(A B:Stream TD)(t:Time)(n:nat),
         Tltt A B t  ->  PrL(Str_nth n A) + t < PrL(Str_nth n B).

[2]Lemma Eq_Teqt : forall(A B:Stream TD)(t:Time)(n:nat),
         Teqt A B t  ->  PrL(Str_nth n A) + t = PrL(Str_nth n B).

[3]Lemma transfer_eqt_lt : forall(s1 s2 s3:Stream TD)(t:Time),
     (Teqt s1 s2 t) /\ (Tlt s2 s3)  ->  Tltt(s1 s3 t).

[4]Lemma transfer_gt_tl : forall s1 s2:Stream TD,
     Tgt s1 s2  ->  Tgt (tl s1) (tl s2).

[5]Lemma transfer_gt_gt : forall s1 s2 s3:Stream TD,
     (Tgt (tl s2) s1) /\ (Tgt (tl s3) s2) ->  (Tgt (tl(tl s3)) s1).

[6]Lemma transfer_eqt : forall(s1 s2 s3:Stream TD)(t:Time),
        (Teq s1 s2) /\ (Teqt s2 s3 t)  ->  (Teqt s1 s3 t).

[7]Lemma transfer_merge : forall (s11 s12 s13 s21 s22 s23:
    Stream TD)(t:Time),(merge s11 s12 s13)/\(Teqt s11 s21 t)
    /\(merge s21 s22 s23)/\(Teqt s12 s22 t)
    ->  (Teqt s13 s23 t).
```

Lemma 1 means that $PrL(Str\_nth\, n\, A) + t < PrL(Str\_nth\, n\, B)$ can be derived from $Tltt(A, B)$. Similarly, Lemma 2 means that $PrL(Str\_nth\, n\, A) + t =$

$PrL(Str\_nth\, n\, B)$ can be derived from $Teqt(A, B)$. Lemmas 3-6 describe the transitivity on the time dimension. Taking lemma $transf\_eqt\_lt$ as an example: If we have one function $Teqt$ with time streams $s1$ and $s2$, such that $s1 + t = s2$, and another function Tlt with time streams $s2$ and $s3$, such that $s2 < s3$, then we can deduce $Tltt(s1, s3)$, such that $s1 + t < s3$. Lemma 7 means that the transitivity for time also holds for the *merge* operator. We obtain $s13$ and $s23$ by merging $s11$, $s12$ and $s21$, $s22$ respectively. Then if we have $s11 + t = s21$ and $s12 + t = s22$, we can easily deduce $s13 + t = s23$.

5.2 Verification of Connector Properties

Example 1. We consider the timed connector shown in Fig. 3. The node A in the connector is a source node, whereas C, D and E are mixed nodes and B is a sink node. This connector consists of five channels AC, AD, CE, DE and DB with channel type *t-timer*, $FIFO1$, $FIFO1$, *SyncDrain* and *Sync* respectively. This connector ensures the lower bound "$> t$" for a take operation on node B. Every data item being received at A will be kept in the buffer of channel AD for more than t time units before it can be taken out at node B.

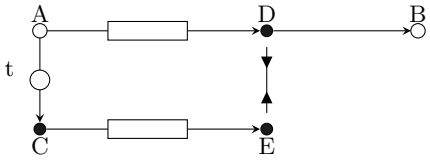

Fig. 3. Lower Bounded FIFO1 Channel

The relation between source node A and sink node B on both time and data dimensions for this connector is specified in Theorem 1. If we use α, β to represent the data streams that flow through the nodes A and B, and use a, b to denote the time stream corresponding to α and β respectively, i.e., the i-th element $a(i)$ in a denotes exactly the time moment of the occurrence of $\alpha(i)$, then Theorem 1 states the property that $\alpha = \beta$ and $a + t < b$ for the connector in Fig. 3.

The connector is built from axioms *Sync*, *SyncDrain*, $FIFO1$ and $Timert$ for its basic channels, which are used as hypothesis for the proof of Theorem 1.

Theorem 1 (LB FIFO1 Channel). $\forall A, B, C, D, E \in Stream\ TD,\ t \in Time$

$$Timert(A, C, t) \wedge FIFO1(A, D) \wedge SyncDrain(D, E) \wedge FIFO1(C, E) \wedge Sync(D, B) \rightarrow Deq(A, B) \wedge Tltt(A, B, t)$$

Proof. First we consider the case for data. In this connector, we have predicate $Deq(A, D)$ which can be obtained from $FIFO1(A, D)$ and predicate $Deq(D, B)$

which is specified by $Sync(D, B)$. The combination of both predicates with *flow_through* operator results in $Deq(A, B)$.

For the case of time, we have four predicates, $Teqt(A, C), Tlt(C, E), Teq(E, D)$ and $Teq(D, B)$ respectively, such that the constraints $A + t = C$, $C < E$, $E = D$, $D = B$ hold for time. The combination of these four predicates results in $Tltt(A, B)$, such that $A + t < B$ holds on time. The proof of Theorem 1 has been implemented in Coq with the help of *tactics*.

Note that we have a condition on the input for timed channel *t-timer*, which requires the inter-arrival time of the inputs is at least t. This condition may bring inconvenience if we hope to produce a timeout after a delay t for every input even if sometimes the inter-arrival time of the inputs is less than t. We show how to weaken this restriction in the following example.

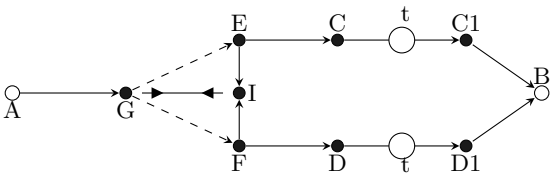

Fig. 4. $2 \times t$ Timed Connector

Example 2. A timed connector $A \xrightarrow{n \times t} B$ can be built by using n t-timer channels and an exclusive router (with n sink nodes), whose behavior is to produce a *timeout* after a delay t for every input (i). What we deserve to notice is that the arrival time between $input_i$ and $input_{i+j}$ $(j < n)$ can be less than t whereas the arrival time between $input_i$ and $input_{i+n}$ should be at least t. The connector in Fig. 4 shows the topology structure of $A \xrightarrow{2 \times t} B$ where A is a source node, B is a sink node and all other nodes are mixed nodes.

Let a and b represent the time streams that correspond to the data flows into A and out of B, respectively. Theorem 2 states the property that $a + t = b$ for the $2 \times t$ timed connector which can be derived from axioms $Sync$, $LossySync$, $SyncDrain$ and $Timert$.

Theorem 2 ($2 \times t$ Timed Channel). $\forall A, B, C, D, E, F, G, I \in Stream\ TD$, $t \in Time$

$$Sync(A, G) \wedge LossySync(G, E) \wedge LossySync(G, F) \wedge Sync(E, I) \wedge$$
$$Sync(F, I) \wedge SyncDrain(G, I) \wedge merge(E, F, I) \wedge Sync(E, C) \wedge$$
$$Sync(F, D) \wedge Timert(C, C1, t) \wedge Timert(D, D1, t) \wedge merge(C1, D1, B)$$
$$\rightarrow Teqt(A, B, t)$$

```
Section example2.
Theorem nt_Timed: forall (A B C D E F G I C1 D1: Stream TD)(t: Time),
  (Sync A G) /\ (LossySync G E) /\ (LossySync G F) /\ (Sync E I) /\
  (Sync F I) /\ (SyncDrain G I) /\ (merge E F I) /\ (Sync E C) /\
  (Sync F D) /\ (Timert C C1 t) /\ (Timert D D1 t) /\ (merge C1 D1 B)
   -> (Teqt A B t).

Proof.
intros.
destruct H. destruct H0.
destruct H1. destruct H2.
destruct H3. destruct H4.
(*Prepare for Lemma transfer_eqt*)
assert((Teq A I)/\ (Teqt I B t)).
split.
(*Proof for Teq A I*)
rewrite H.
apply H4.
(*Proof for Teqt I B t*)
  (*Prepare for Lemma transfer_merge*)
  assert((merge C D I) /\ (Teqt C C1 t) /\ (Teqt D D1 t)
                      /\ (merge C1 D1 B)).
  repeat split.
  (*Proof for merge C D I*)
  destruct H5. destruct H6.
  rewrite <- H6.
  destruct H7.
  rewrite <- H7.
  assumption.
  (*Proof for Teqt C C1 t*)
  destruct H5. destruct H6.
  destruct H7. destruct H8.
  destruct H8. destruct H10.
  assumption.
  (*Proof for Teqt D D1 t*)
  destruct H5. destruct H6.
  destruct H7. destruct H8.
  destruct H9. destruct H9.
  destruct H11.
  assumption.
  (*Proof for merge C1 D1 B*)
  destruct H5.
  destruct H6.
  destruct H7.
  destruct H8.
  destruct H9.
  assumption.
  generalize H6.
  apply transfer_merge.
  generalize H6.
apply transfer_eqt.

Qed.
End example2.
```

Fig. 5. Proof steps for Example 2 in Coq

Proof. The main *goal* $Teqt(A, B, t)$ is first split in two *subgoals* $Teq(A, I)$ and $Teqt(I, B, t)$ with Lemma 6. For the first subgoal $Teq(A, I)$, we have $Teq(A, G)$ ($A = G$ on the time dimension) which is derived from $Sync(A, G)$ and $Teq(G, I)$ ($G = I$ on the time dimension) which is derived from $SyncDrain(G, I)$. So we can easily deduce that $Teq(A, I)$. For the second subgoal $Teqt(I, B, t)$, the proof

is supported by Lemma 7. Consequently, we only need to prove the premise condition of the Lemma. Note that $merge(C, D, I)$ and $merge(C1, D1, B)$ are given by Hypothesis, and $Teqt(C, C1, t)$ and $Teqt(D, D1, t)$ can be easily obtained from the definition of channels $Timert(C, C1, t)$ and $Timert(D, D1, t)$ respectively. So for the time dimension, we have $merge(C, D, I)$, $merge(C1, D1, B)$, $Teqt(C, C1, t)$ and $Teqt(D, D1, t)$ such that $C \wedge D = I, C1 \wedge D1 = B, C + t = C1$, and $D + t = D1$. Then we can draw the conclusion that $I + t = B$, i.e. $Teqt(I, B, t)$. A sample of concrete proof steps in Coq is provided in Fig 5.

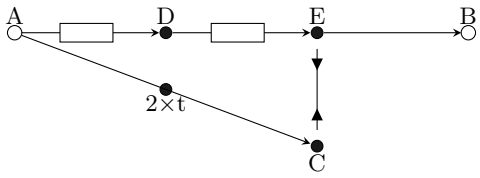

Fig. 6. Timer FIFO2 Channel

Example 3. Another 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 construct such connectors by using $\xrightarrow{n \times t}$. Fig. 6 shows an example of such a timed FIFO2 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 E and using a corresponding $\xrightarrow{n \times t}$ where n is the number of *FIFO1* channels.

Theorem 3 (Timed FIFO2 Channel). $\forall A, B, C, D, E \in Stream\ TD,\ t \in Time$

$$FIFO1(A, D) \wedge FIFO1(D, E) \wedge Sync(E, B) \wedge SyncDrain(C, E) \wedge Teqt(A, C, t) \rightarrow Teqt(A, B, t) \wedge Tlt(B, tl(tlA))$$

Proof. The proof process is based on divide and conquer technique. There are two goals in this theorem: $Teqt(A, B, t)$ and $Tlt(B, tl(tlA))$. The first goal $Teqt(A, B, t)$ is split into two subgoals $Teq(B, C)$ and $Teqt(A, C, t)$. Furthermore, the *subgoal* $Teq(B, C)$ can be split into $Teq(B, E)$ and $Teq(E, C)$. The second goal $Tlt(B, tl(tlA))$ is split into two subgoals $Teq(B, E)$ and $Tlt(E, (tl(tlA)))$ and $Tlt(E, (tl(tlA)))$ into $Tlt(E, (tlD))$ and $Tlt(D, (tlA))$. All of these six subgoals as shown in Fig. 7 are easy to be solved.

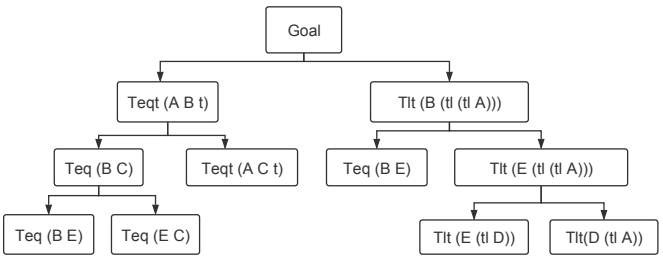

Fig. 7. Proof Steps of Theorem 3

6 Conclusion

As an extension of the work in [22], this paper presents a method for formal modeling of timed channels and reasoning about timed connectors properties in Coq. The model preserves the original structure of timed channels and composition operators, which makes their description reasonably readable. Connector properties are defined with predicates which offer an appropriate description of the relation between different TD streams on the nodes of a connector. The proofs of connector properties are completed with the help of pre-defined techniques and tactics that Coq offers.

Analysis and proof process of complex connectors in Coq is hard and time consuming as it depends on more tactics and decision procedures. The proof process can become easier by adding frequently-used proof patterns as new tactics. Furthermore, automation methods may also help us to avoid tons of hand-written proofs and Coq offers several auto tactics to solve proof goals. With proper configuration, perhaps such tactics will work well in our framework. Another possible future work is to prove more generic properties, like generalizations of theorems 2 and 3 for any size n, that are harder or infeasible to be verified using other verification tools like model checkers.

Acknowledgement

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

References

1. F. Arbab. Reo: A Channel-based Coordination Model for Component Composition. *Mathematical Structures in Computer Science*, 14(3):329–366, 2004.
2. F. Arbab, C. Baier, F. de Boer, and J. Rutten. Models and Temporal Logics for Timed Component Connectors. In *Proceedings of SEFM2004*, pages 198–207. IEEE Computer Society, 2004.
3. F. Arbab and J. Rutten. A coinductive calculus of component connectors. In *WADT 2002*, volume 2755 of *LNCS*, pages 34–55. Springer-Verlag, 2003.

4. 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.
5. 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.
6. Y. Bertot and P. Casteran. *Interactive Theorem Proving and Program Development: Coq'Art: The Calculus of Inductive Construction.* Springer-Verlag, 2003.
7. 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.
8. D. Clarke, D. Costa, and F. Arbab. Connector Coloring I: Synchronization and Context Dependency. *Science of Computer Programming*, 66(3):205–225, 2007.
9. E. M. Clarke, W. Kibler, M. Novacek, and P. Zuilani. Model checking and the state space explosion problem. In *Proceedings of LASER 2011*, pages 1–30. LNCS, 2011.
10. Coq Implementation of Connectors. https://github.com/saqibdola/TimedReoinCoq.
11. N. Diakov and F. Arbab. Compositional construction of web services using Reo . In *Proceedings of International Workshop on Web Services: Modeling, Architecture and Infrastructure (ICEIS 2004)*, pages 13–14. INSTIC Press, 2004.
12. D. Gelernter and N. Carriero. Coordination Languages and their Significance. *Coomunication of the ACM*, 35(2):96, 1992.
13. G. Huet, G. Kahn, and C. Paulin-Mohring. The Coq Proof Assistant A Tutorial. *Rapport Technique*, 178, 1997.
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. T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic.* Springer Science & Business Media, 2002.
18. S. Owre, J. M. Rushby, and N. Shankar. PVS: A Prototype Verification System. In *Proceedings of International Conference on Automated Deduction*, pages 748–752. Springer, 1992.
19. M. Sun. Connectors as designs: The time dimension. In *Proceedings of TASE 2012*, pages 201–208. IEEE Computer Society, 2012.
20. 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.
21. 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.
22. 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, 2017.
23. Z. Zlatev, N. Diakov, and S. Porkaev. Construction of Negotiation Protocols for E-Commerce Applications. *ACM SIGecom Exchanges*, 5(2):12–22, 2004.