Towards Formal Modeling and Verification of Probabilistic Connectors in Coq

Xiyue Zhang and Meng Sun
LMAM & DI, School of Mathematical Sciences, Peking University, Beijing, China
{zhangxiyue, sunm}@pku.edu.cn

***Abstract*—The coordination language Reo has played an important role in organizing the interactions among different components in large-scale distributed applications. A probabilistic extension on classical Reo is necessary to deal with the uncertainty of the real world. In this paper we developed a framework in Coq for formalizing probabilistic connectors and reasoning about their probabilistic properties. Different types of probabilistic channels are characterized by the relations on their input and output timed data distribution streams. More complex probabilistic connectors can be further constructed based on the probabilistic channels and composition operators. Within such a framework, properties under analysis and refinement / equivalence relations between probabilistic connectors can be naturally established as theorems and proved using tactics in Coq.**

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

I. Introduction

The coordination of interactions among large numbers of concurrent entities in large-scale distributed applications cannot be easily dealt with and has become a challenge for software technology. Coordination models and languages provide a mechanism to meet this challenge by introducing a formalization of connectors that integrate a number of heterogeneous components together and organize the mutual interaction among them in a distributed environment. There are many coordination models and languages that have been proposed in the past decades, such as Reo [2], Linda [21], BIP [10], [13] and Orc [12]. Almost all of these coordination models enhance modularity and reuse of existing components and portability. However, they still differ in many dimensions: the entities being coordinated, the mechanism of coordination, the coordination medium architecture, and so on.

Reo [2], [8], as a famous coordination model, forms a paradigm for coordination of software components based on the concept of *channel*. Such channel-based models have some inherent advantages over other coordination models, especially when it comes to concurrent systems that are distributed, mobile and whose communication topologies may dynamically evolve. Channels in Reo are in fact the simplest connectors and they can be composed to construct more complex connectors that are used as the glue code to organize the interaction and communication of components in distributed applications.

The reliability of large-scale distributed applications highly depends on the correctness of coordination models, which makes the formal analysis and verification of connectors much more crucial. There are several works that have been done in the formalization and verification of connectors in the past years: A coalgebraic semantics for Reo was developed in [5] in which connectors are interpreted as relations on timed data streams. And constraint automata (CA) was proposed as an operational model for connectors in [8]. A scheme to determine the behavior of connectors by resolving the synchronization and exclusion constraints based on connector coloring was introduced in [11]. The symbolic model checker Vereofy was developed in [7] which can be used to verify CTL-like properties for connectors. Kokash et al. presented a mapping from Reo to the specification language mCRL2 based on process algebra, where the models can be further verified conveniently using the model checker for mCRL2 in [15].

Complex distributed applications usually involve important features like real-time, probability, resource consumption, and so on. Various proposals on extending Reo to deal with such features have been reported, for example, in [3], [4], [6], [9], [17]. In this paper, we use Coq [19] to provide a formalization of Reo connectors with probabilistic behavior and show how the refinement / equivalence relations and properties of such probabilistic connectors can be further verified based on the formalization. This is a further extension to our previous work of the formalization of Reo and its timed extension in Coq [14], [22] on the probabilistic dimension, which is still based on the UTP (Unifying Theories of Programming) semantic framework for Reo that has been developed in [1], [18]. Probabilistic connectors are constituted by channels that can behave probabilistically, such as the probabilistic variant of *LossySync* channel or randomized *Sync* channel.

This is certainly not the first work to investigate probabilistic connectors. For example, probabilistic constraint automata (PCA) [6], which is a variant of CA, characterize the behavior of probabilistic connectors. However, the formalization by means of CA (and its extensions) is generally constrained by the memory limitation problem since infinite behavior is usually considered for Reo connectors. Modeling and verifying unbounded primitives or even bounded primitives with unbounded data domains, which always leads to the state space explosion problem, cannot be achieved with such finite CA-like models. But they can be specified and verified efficiently in theorem provers like Coq. The previous work in [22], [14] can certainly model a wide range of scenarios, but it is not good at dealing with the uncertainty of the real world. With this formalization for the probabilistic extension of Reo provided, more scenarios with uncertainty can be captured, and various properties under analysis or relations between such probabilistic connectors can be further verified in Coq.

The paper is organized as follows. After this general

DOI reference number: 10.18293/SEKE2018-023.

introduction, we briefly review some main concepts of the coordination language Reo in Section II. In Section III, we present the basic specification for timed data distribution sequences and some auxiliary functions and predicates for more concise modeling. Section IV introduces the formal modeling of basic probabilistic channels and an adaptive deformation of the specification for other primitive untimed and timed channels, as well as the composition operators. Section V shows how to reason about refinement / equivalence relations between probabilistic connectors in our framework. Finally, Section VI concludes the paper and discusses some future work. The complete implementation in Coq can be found at [20].

II. Preliminary

In this section, we briefly introduce some basic concepts of the coordination language Reo. Complex coordinators, called connectors in Reo, are compositionally built out of simpler ones. We only review the primary concepts of Reo here. More details can be found in [2], [8].

Fig. 1. Basic channels in Reo

The focus in Reo is on connectors and their composition, not the different entities being connected by the connectors. It works very well in practice for controlling and organizing the communication, synchronization and cooperation among the distributed components. Each channel in Reo has exactly two channel ends with their own identifiers. There are two types of channel ends: *source end* and *sink end*. Data items are accepted into a channel through its source end and dispensed out of the channel through its sink end. It is not necessary for a channel to have both source end and sink end, i.e., a channel can have two source ends or two sink ends. Each channel end can be connected to at most one component instance at any given time. Reo allows an open-ended set of user-defined channel types as primitives for constructing connectors. Figure 1 shows some widely-used channel types in Reo which are interpreted as follows:

Sync: The *synchronous* channel has one source end and one sink end. A channel is called synchronous because it accepts a data item if and only if the dispensation of the data item through the sink end can simultaneously occur.

LossySync: The *lossy synchronous channel* is similar to the *Sync* channel except that it always accepts all data items through its source end, but it can only deliver some of the data items that it accepts, and lose the rest. Data items are transferred successfully only when the write operation on the source end and the take operation on the sink end occur simultaneously, otherwise the data items are lost.

FIFO1: A *FIFO1* channel is an asynchronous channel with a buffer whose capacity is bounded with 1. Initially, the buffer is empty. After accepting a data item through the source end, the data item will be kept in the buffer before being dispensed out of the channel. The next data item can only be accepted into the buffer after the data item in the buffer is dispensed.

SyncDrain: The *synchronous drain* has two source ends and no sink end. The pair of write operations on its two ends to accept data items can succeed only simultaneously, and all the data items written to this channel are lost.

t-Timer: The *t-Timer* channel is used to capture more time-related behavior. It accepts any data item through its source end and produces a $timeout$ signal after a delay of t time units on its sink end.

Apart from these channel types, more well-defined exotic channels can be found in [2], [3], [18]. Users can also define new channels according to their own demands and interaction policies. For example, several probabilistic and stochastic extensions of Reo have been proposed in [6], [9], [16].

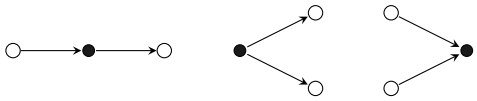

Fig. 2. Operators for channel composition

A connector is actually a set of channel ends together with the connecting channels, organized in a graph of nodes and edges, in which different channel ends coincide on each node and each edge captures the type of channel linking its two nodes. 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 types. According to the node types, we have three types of operators for channel / connector composition: *flow-through*, *replicate* and *merge*, as shown in Figure 2.

A source node corresponds to the replicate operator. Data items flowing through the source node are replicated and written into all the channels /connectors that are associated with the the source node. A sink node acts as the merge operator. A take / read operation on the sink node succeeds only when at least one of the channel ends coincident on the sink node offers a data item. If more than one channel ends offer data items, then one of them is selected non-deterministically. The flow-through operator simply allows the data item to flow through the mixed node without any change. When there are more than one sink ends and source ends coincident on the mixed node, the merge operator acts first, takes a data item offered by one of the sink ends, and then the selected data item flows through the mixed node, then the data item is replicated and the copies are written into all of its source ends.

III. Basic Definitions

An obvious way to represent a connector is to model it as a relation on its inputs and outputs, which take place through the source and sink nodes of the connector. Taking probabilistic behavior into consideration, sequences of data distributions in which the data pass through a node together with the moments in time when the data items are observed emerge as the key building blocks for describing connectors. Therefore, we can naturally specify the observations as timed data distribution sequences on nodes, which are used to characterize probabilistic connectors. Some auxiliary functions and predicates are introduced for concise modeling and reasoning about probabilistic connectors in Coq.

The main libraries of Coq being used here are the *Stream* library, the *Reals* library, and the *Utheory* library. The *Stream* library provides an appropriate co-inductive definition of infinite sequences on the input and output nodes of connectors. However, unlike the specification we used to describe the behavior of primitive channels in [14], [22], the observation sequences are adjusted here to timed data distribution streams instead of timed data streams. The *Reals* library is mainly used to support continuous time behavior; as a result, various operations and axioms on real numbers can be adopted directly for the modeling and reasoning. The *Utheory* library axiomatizes the properties required on the abstract type *U* representing the real interval $[0, 1]$, which facilitates the description of probabilistic behavior. The definition of the infinite data flow can be specified as follows, with the help of these libraries.

```
Definition Time := R.
Definition Data := nat.
Definition DataDist := Data * U.
Definition TDD := Time * DataDist.
```

In this framework, time is specified as real numbers, which is very natural and expressive enough for the modeling approach. Data, for simplicity, is defined as the set of natural numbers. Generally, probability functions that map from natural numbers to the corresponding probabilities are very common and other abstract sets of data can be processed first by mapping them to a set of natural numbers in an appropriate way. Moreover, the definition of data can be expanded easily in accordance with concrete application domains. The representation of the data distribution, denoted as *DataDist*, is the Cartesian product of the type *Data* and the abstract type *U*, namely the real interval $[0, 1]$. The timed data distribution *TDD* can be further defined as the Cartesian product of time and data distribution. *Stream TDD* represents the timed data distribution sequences efficiently via the *Stream* module.

Several auxiliary functions and predicates are used to facilitate the representation of channels and composition operators, and can be exploited and extended for further reasoning.

Functions *PrL* and *PrR* take an instance (a, b) of the Cartesian product type as the argument and return the first or second element of the pair, respectively. If b in this instance is still a Cartesian product (c, d), functions *LPrR* and *DPrR* can be applied on $(a, (c, d))$ and return the elements c and d, respectively.

The following axiom which specifies that the elements of all time streams should be in a strictly monotonous increasing order is a general requirement. More requirements that some streams should satisfy can be specified in a similar way, and some of them are provided in the definitions of specific channels.

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

Several predicates about time and data are defined for a more clear and concise formalization of the connectors. For example, *Tlt* (*Tgt*) represents that each element of the first time stream is strictly less (greater) than the second stream. For the modeling and verification of probabilistic connectors including timer channels, some predicates about time but with an extra parameter *t* are defined in a similar way. Each element of the first time stream is added by a delay of *t* time units which can be customized. Therefore, *Tltt* (*Tgtt*) represents that the time of the first stream with an addition of *t* is less (greater) than the second stream accordingly. Complete definitions of such auxiliary functions and predicates can be found at [20].

IV. Probabilistic Channels and Operators

Modeling of basic probabilistic channels and composition operators serves as the basis of the whole modeling and reasoning framework.

A. Probabilistic Channels

Constraints on input and output timed data distribution streams are used in Coq to specify primitive channels' behavior. The specification of channels with probabilistic behavior can be captured by the disjunction or conjunction of different predicates about time and data distributions as well. In this section we consider four types of probabilistic channels: *message-corrupting synchronous channel*, *randomized synchronous channel*, *probabilistic lossy synchronous channel* and *faulty FIFO1 channel*. Specifications of other primitive channels are omitted here and can be found at [20].

CptSync: The message-corrupting synchronous channel $-\not{p}\rightarrow$ is a synchronous channel which has an extra parameter p compared with the primitive synchronous channel. The delivered message can be corrupted with probability p. Hence, if a data item flows into the channel through the source end, then the correct data value will be obtained at the sink end with probability $1-p$ and a corrupted data value $\perp$ will be obtained with probability p. The corrupted data value is represented by the initial letter c in the Coq specification.

```
Parameter CptSync:
  Stream TDD -> Stream TDD -> U -> Prop.
Axiom CptSync_coind:
  forall (Input Output:Stream TDD) (p:U),
  CptSync Input Output p ->
  ( PrL(hd Output) = PrL (hd Input)
  /\
    (
     PrR(hd Output) =
     (c, p*(DPrR(hd Input)))
    \/
     PrR(hd Output) =
     (LPrR(hd Input),([1-]p)*(DPrR(hd Input)))
    )
  /\
    CptSync (tl Input) (tl Output) p
  ).
```

The *CptSync* channel is defined recursively here. The first predicate of the conjunction is the constraint on the equality of time reflecting the synchronous behavior. The disjunction formula in the middle captures the probabilistic behavior. The data of the output can be the exact value of the input with the updated probability, i.e., the original companied probability multiplied by $1 - p$ or the corrupted value with probability p.

RdmSync: The randomized synchronous channel $\xrightarrow{rand(0,1)}$ can generate a random number $b \in \{0, 1\}$ with equal probability

when it is activated through an arbitrary write operation on its source end, and this random number will be taken on the sink end synchronously.

```
Definition RdmSync(Input Output:Stream TDD)
:Prop :=
(forall n:nat,
PrR (Str_nth n Output)=(1%nat, [1/]1+1)
\/
PrR (Str_nth n Output)=(O, [1/]1+1))
/\ Teq Input Output.
```

The formula in the first disjunction branch with the universal quantifier properly describes the probabilistic behavior observed on output streams. Each element of the output data distribution stream can be 1 or 0 both with the probability $\frac{1}{2}$. As for the constraint about time, the predicate *Teq* is used here to indicate that the time dimension of input and output streams are equal, conforming to the synchronous behavior.

ProbLossy: The message transmitted by the probabilistic lossy synchronous channel $-\overset{q}{-}\rightarrow$ can get lost with a certain probability q. It can also act like a *Sync* channel and the message will be delivered successfully with probability $1-q$.

```
Parameter ProbLossy:
  Stream TDD -> Stream TDD -> U -> Prop.
Axiom ProbLossy_coind:
  forall (Input Output:Stream TDD) (q:U),
  ProbLossy Input Output q ->
  (
    (PrL(hd Output) = PrL(hd Input)
    /\
    PrR(hd Output) =
    (LPrR(hd Input), DPrR(hd Input)*([1-]q))
    /\
    ProbLossy (tl Input) (tl Output) q)
  \/
    ProbLossy (tl Input) Output q
  ).
```

The *ProbLossy* channel is defined recursively but may take two different courses in each step. The data can be totally lost when going through the channel, which leads to the recursive behavior that the last formula reflects in the specification. If the data item is successfully delivered, then there are three constraints that need to be satisfied. These constraints are represented by the conjunction of three formulas. The first formula is specified for the equality of time in accordance with the synchronous delivery. The second formula reflects that the current data item is transmitted successfully with an extra multiplication $1-q$ to the original probability of the input. The third formula is the second course that the recursion takes when last data item has a successful transmission.

FtyFIFO1: The messages flowing into a faulty FIFO1 channel $\cdots\overset{r}{\square}\rightarrow$ can get lost with probability r when it is inserted into the buffer. In this case, the buffer remains empty. It can also behave as a normal *FIFO1* channel when the insertion of data into the buffer is successful with probability $1-r$.

```
Parameter FtyFIFO1:
  Stream TDD -> Stream TDD -> U -> Prop.
Axiom FtyFIFO1_coind:
forall (Input Output:Stream TDD) (r:U),
FtyFIFO1 Input Output r ->
(
  (PrL(hd Output) > PrL(hd Input)
   /\
   PrL(hd Output) < PrL(hd (tl Input))
   /\
   PrR(hd Output) =
   (LPrR(hd Input), DPrR(hd Input)*([1-]r))
   /\
   FtyFIFO1 (tl Input) (tl Output) r)
 \/
   FtyFIFO1 (tl Input) Output r
).
```

The *FtyFIFO1* channel is also defined recursively here but can take two different ways of recursion in each step. The data written into the channel on the source end can be lost before being inserted into the buffer, which leads to the first way of recursion represented by the last formula. If the data item is successfully written into the buffer, then there are four constraints that need to be satisfied, represented by the conjunction of four formulas. The first and the second formula are used to constrain the behavior on time dimension. The time delay from input to output is captured by the first formula. Since the buffer capacity is 1, next data item cannot be written into the buffer unless the data item currently in the buffer is taken on the sink end first, which is reflected by the second formula. The third formula indicates that the current data item is transmitted successfully through the buffer with an extra $1-r$ being multiplied to the original probability of the input. The fourth formula reflects the second way of recursion when last data item is written into the buffer successfully.

Another kind of faulty FIFO1 channel $-\square\overset{r}{-}\rightarrow$ works perfectly on the insertion of data item into its buffer but may loose messages from the buffer before being taken on the sink end. The difference between this channel and *FtyFIFO1* is that the loss of data items happens in different steps. Loss behavior in this channel arises in the process of being taken from the buffer, while loss behavior in *FtyFIFO1* arises in the process of storage into the buffer. But in our modeling framework, channels are specified only by the relations between observations on input and output channel ends. As a result, the specifications of these two faulty FIFO1 channels in Coq are exactly same.

The specification of primitive untimed and timed channels in [22], [14] are properly adjusted in this modeling framework. Moreover, this new formalization is still consistent with the untimed / timed version by means of assigning the value 1 to the companied probability of the data in the definitions of channels with no probabilistic behavior. Hence, the observations on input and output are all specified by timed data distribution streams, and connectors composed by primitive untimed / timed channels and probabilistic channels can be constructed without a hitch. Once there is a probabilistic channel in a connector, it will be taken as a probabilistic connector.

B. Composition Operators

Composition operators are the other essential factor for the construction of complex connectors. As described in Section II, there are three kinds of composition operators: *flow-through*, *replicate* and *merge*.

The *flow-through* and *replicate* operators do not need to be adjusted. The specification for these operators in [22], [14] can be adopted here without any change, since the behavior of these two operators are independent of the form or content of the data flow. Both of them can still be specified implicitly by means of renaming. For example, for two channels *ProbLossy(A, B)* and *FIFO1(C, D)*, the *replicate* operator has been implemented directly by renaming *C* with *A* for the *FIFO1* channel. The *flow-through* operator can be implemented in a similar way. For example, when we illustrate channels *ProbLossy(A, B)* and *FIFO1(B, C)*, the *flow-through* operator that acts on node *B* has already been implemented.

The *merge* operator seems to depend on the content of the data flow. However, it is easy to understand that the comparison of time in the original specification of the operator does not need any change, since the time dimension is exactly same between the timed data streams and timed data distribution streams. As for the data dimension, the equality also does not need to change with the aid of the *Utheory* library. But in this framework, the equality relation for data is changed to the equality relation on data distribution. Therefore, both data items and their companied probabilities should be equal.

```
Parameter merge:
Stream TDD->Stream TDD->Stream TDD->Prop.
Axiom merge_coind:
  forall s1 s2 s3:Stream TDD,
  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)).
```

V. Reasoning about Relations

The formalization of all types of basic channels and composition operators completes the ground modeling framework, which serves well to construct different probabilistic connectors that users are interested in. Complex connectors can be constructed according to their topological orders. As soon as the construction is done, connector properties and refinement / equivalence relations between connectors can be further specified and reasoned about in Coq.

The concept of refinement has been widely used in different system descriptions. The refinement relation for connectors is defined in [18], in which the refinement order over connectors is established based on the implication relation of predicates. Connector C_2 is a refinement of connector C_1, both represented by a set of predicates, if and only if $C_2 \rightarrow C_1$, i.e., the behavioral properties of C_1 can be derived from the properties of C_2. In this case, the properties of connector C_2 are regarded as the hypothesis and the properties of connector C_1 as the conclusion. The refinement relation between C_1 and C_2 is denoted as $C_1 \sqsubseteq C_2$ and the equivalence relation is defined typically by mutual refinement, i.e., $C_1 \equiv C_2$ **iff** $C_1 \sqsubseteq C_2 \wedge C_2 \sqsubseteq C_1$. Thus the equivalence relation can be represented by the implications in both directions $C_2 \leftrightarrow C_1$.

Fig.3 shows two probabilistic connectors both are built from the same set of five channels *RdmSync*, *FIFO1*, *t-Timer*, *SyncDrain* and *Sync*, but in different topological orders. In fact, the four channels *FIFO1*, *t-Timer*, *SyncDrain* and *Sync* make up a timed connector *tFIFO1* that we have studied in [14]. Different from the primitive *FIFO1* channel whose output timed data distribution streams will be of the same data distribution as the input, but with an arbitrary time delay, the time delay of *tFIFO1* channel is fixed by the parameter *t*, apart from the same data distribution between input and output streams. Therefore, connectors R_1 and R_2 are actually constituted by the same two subconnectors *RdmSync* and *tFIFO1* but with interchanged positions. In general, two connectors composed with same set of subconnectors as we call, in commutative orders are not equivalent, i.e., the construction of connectors does not satisfy the commutative law. But in this case R_1 and R_2 are equal. The equivalence relation between these two connectors has been proved in Coq.

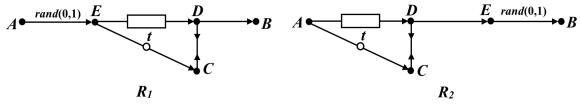

Fig. 3. Equivalence between connectors

The configurations of both R_1 and R_2 can be reduced to the constitution of a *RdmSync* channel and a *tFIFO1* subconnector with interchanged topological orders. Therefore, the equivalence relations between the construction from basic channels and the reduced method of construction from a *RdmSync* channel and a *tFIFO1* subconnector are proved first as lemmas in Coq, to make the subsequent proof process simplified and easier to understand.

```
Lemma RSync_tFIFO_eq:
  forall (A B:Stream TDD) (t:Time),
  exists E: Stream TDD,
  (RdmSync A E) /\ (t_FIFO1 E B t)
  <->
  (RdmSync A E) /\ (exists (D C:Stream TDD),
  (FIFO1 E D)  /\ (SyncDrain D C)
  /\ (Timert E C t) /\ (Sync D B)).
```

The above lemma shows the equivalence relation between the reduced construction and the construction from basic channels for R_1. Another lemma is defined for R_2 in Coq similarly. As a result, the goal of equivalence relation between connectors R_1 and R_2 boils down to the following theorem:

```
Theorem equivalence:
  forall (A B:Stream TDD) (t:Time),
  (exists E,(RdmSync A E)/\(t_FIFO1 E B t))
  <->
  (exists R,(t_FIFO1 A R t)/\(RdmSync R B)).
```

Intuitively, the core of the proof for the goal is to find the corresponding mediated timed data distribution stream to complete the construction, given the construction method of the other connector. However, a matched timed data distribution stream cannot be found directly. Therefore, we need to construct two timed data distribution streams first and then prove that they serve well as precise matches for the refinement relations in both directions, respectively. The two streams *A_t* and *B_t* are constructed to satisfy the following properties which act as hypotheses in Coq:

```
Hypothesis A_R_t: Deq A A_t/\Teqt A A_t t.
Hypothesis B_R_t: Deq B_t B/\Teqt B_t B t.
```

There are other hypotheses being used directly in the proof, such as *transfer_eqt* which has been proved in [14]. Some new properties that aid in proving the current goal but not proved before are formalized as lemmas and get proved first. For example, the following lemma demonstrates a property that two streams which are both greater than a same stream by t are equal in the time dimension. It is simple to formalize and prove this property using some commonly-used tactics like *intro*, *destruct* and *rewrite*.

```
Lemma trans_t_eq:
  forall (s1 s2 s3:Stream TDD)(t:Time),
  (Teqt s1 s2 t) /\ (Teqt s1 s3 t) ->
  (Teq s2 s3).
```

All these lemmas and hypotheses make the main proof more concise. The proof of the theorem actually has two steps. The original goal is *split* first into two subgoals that represent refinement relations in both directions. For the first subgoal, the antecedent or the precondition of the implication serves as another hypothesis. After asserting that the matched timed data distribution stream R is A_t, the current subgoal is reduced to $t\_FIFO1\ A\ A\_t\ t \wedge RdmSync\ A\_t\ B$, which can be *split* again and proved using specific tactics based on the lemmas and hypotheses, especially the fact that *exists E:Stream TDD*, $RdmSync\ A\ E \wedge t\_FIFO1\ E\ B\ t$. The subgoal of the refinement relation in the other direction can be proved similarly. A complete proof process can be found at [20].

VI. Conclusion

This paper proposes a method of modeling and reasoning about probabilistic connectors in Coq, which is also compatible with the formalization for the primitive untimed / timed connectors. Basic probabilistic channels and composition operators are formalized as the ground framework. After adjusting timed data streams to timed data distribution streams, all the channels can be specified by a set of predicates capturing the relations between inputs and outputs. The formalization of composition operators makes it possible to construct more complex connectors. Properties related to probability distributions and refinement / equivalence relations between probabilistic connectors can be specified easily and further presented with machine-checked proofs. Compared to LTL or CTL formulas, Coq expressions are more powerful to depict properties and we can be free from worrying about the state space explosion problems in other verification approaches like model checking. Moreover, this architecture is promising to capture the uncertainty of different applications in real world.

In the future, we plan to investigate some more scenarios related to coordination in real world based on this architecture. In particular, we would like to deal with more probabilistic properties users care about among different applications or services. The modeling and verification of hybrid behavior of connectors in Coq is in our scope as well.

Acknowledgements

The work is partially supported by NSFC under grant no. 61772038, 61532019, 61202069 and 61272160.

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. S. de Boer, and J. J. M. M. Rutten. Models and temporal logical specifications for timed component connectors. *Software and System Modeling*, 6(1):59–82, 2007.

[4] 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.

[5] 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.

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

[7] 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.

[8] 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.

[9] C. Baier and V. Wolf. Stochastic Reasoning About Channel-Based Component Connectors. In P. Ciancarini and H. Wiklicky, editor, *COORDINATION 2006*, volume 4038 of *LNCS*, pages 1–15. Springer-Verlag, 2006.

[10] S. Bliudze and J. Sifakis. The algebra of connectors - structuring interaction in BIP. *IEEE Trans. Computers*, 57(10):1315–1330, 2008.

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

[12] W. R. Cook, S. Patwardhan, and J. Misra. Workflow Patterns in Orc. In *Proceedings of COORDINATION 2006*, volume 4038 of *LNCS*, pages 82–96. Springer, 2006.

[13] R. Edelmann, S. Bliudze, and J. Sifakis. Functional BIP: embedding connectors in functional programming languages. *Journal of Logical and Algebraic Methods in Programming*, 92:19–44, 2017.

[14] W. Hong, S. Nawaz, X. Zhang, Y. Li, and M. Sun. Using Coq for Formal Modeling and Verification of Timed Connectors. In *Software Engineering and Formal Methods: SEFM 2017 Collocated Workshops, Revised Selected Papers*, volume 10729 of *LNCS*, pages 558–573. Springer, 2018.

[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, X. Zhang, Y. Ji, and M. Sun. Capturing Stochastic and Real-time Behavior in Reo Connectors. In *Proceedings of SBMF 2017*, volume 10623 of *LNCS*, pages 287–304. Springer, 2017.

[17] M. Sun and F. Arbab. On resource-sensitive timed component connectors. In *Proceedings of FMOODS 2007*, volume 4468 of *LNCS*, pages 301–316. Springer, 2007.

[18] 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.

[19] The Coq Proof Assiatant. https://coq.inria.fr/.

[20] The source code. https://github.com/Xiyue-Selina/Prob-Reo.

[21] M. Viroli, D. Pianini, and J. Beal. Linda in space-time: An adaptive coordination model for mobile ad-hoc environments. In *Proceedings of COORDINATION 2012*, volume 7274 of *LNCS*, pages 212–229. Springer, 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.