Reasoning about Connectors Using Coq and Z3

Xiyue Zhang, Weijiang Hong, Yi Li, Meng Sun

LMAM & DI, School of Mathematical Science, Peking University, Beijing, China

Abstract

Reo is a channel-based exogenous coordination language in which complex coordinators, called connectors, are compositionally built out of simpler ones. In this paper, we present an approach to model and reason about connectors using Coq and Z3. Both models reflect the original structure of connectors as closely as possible. In our framework, both basic connectors (channels) and composition operations are modeled as axioms. Furthermore, complex connectors are modeled as the combination of logical predicates which correspond to simpler connectors. With such definitions provided, connector properties, as well as equivalence and refinement relations between different connectors, can be formalized as *goals* in Coq and proved using pre-defined *tactics*, if satisfied by connectors. When failing to prove whether a property is satisfiable or not with Coq, we use Z3, an SMT solver, to search for possible bounded counter examples automatically.

Keywords: Coordination language, Reo, Coq, Z3, Reasoning

1. Introduction

Modern software systems are typically distributed over large networks of computing devices. Usually the components that comprise a system do not exactly fit together as pieces of a jigsaw puzzle, but leave significant interfacing gaps that must somehow be filled with additional code. Compositional coordination models and languages provide a formalization of the "glue code" that interconnects the constituent components, organizes the mutual interactions between them in a distributed processing environment, and plays a crucial role for the success of component-based systems in the past decades.

As an example, Reo [3] is a channel-based exogenous coordination language, that offers a powerful framework for implementation of coordinating component connectors. Connectors provide the protocols that control and organize the communication, synchronization and cooperation among the components that they interconnect. Primitive connectors, called *channels* in Reo, can be composed to build complex connectors. Reo has been successfully applied in different application domains, such as service-oriented computing and bioinformatics [7, 19]. In recent years, verifying the correctness of connectors is becoming a critical challenge, especially due to the advent of Cloud computing technologies. The rapid growth in size and complexity of the computing infrastructures leads to more complex structure of connectors, makes it more difficult to model and verify connector properties, and thus decreases the confidence on the correctness of connectors.

Several works have been done for formal modeling and verification of connectors. An operational semantics for Reo using Constraint Automata (CA) was provided by Baier et al. [6], and later the symbolic model checker Vereofy [5] was developed, which can be used to check CTL-like properties. Besides, one attractive approach is to translate Reo to other formal models such as Alloy [13], mCRL2 [15], UTP [2, 20], etc., which makes it possible to take advantage of existing verification tools. A comparison of existing semantic models for Reo can be found in [12].

Email addresses: zhangxiyue@pku.edu.cn (Xiyue Zhang), wj.hong@pku.edu.cn (Weijiang Hong), liyi_math@pku.edu.cn (Yi Li), sunmeng@math.pku.edu.cn (Meng Sun)

In a previous paper [21] we showed how to formally model and reason about connectors' properties in Coq. The basic idea of our approach is to model the behavior of a connector as a logical predicate which describes the relation among the timed data streams on the input and output nodes, and to reason about connectors' properties, as well as the equivalence and refinement relations between connectors, by using proof principles and tactics in Coq. Almost all existing approaches for modeling and verification of Reo connectors are state-based or automata-based, in which the automata for complex connectors are constructed through product operation on simpler ones. As a result, the number of states of connectors may grow exponentially, which makes it inefficient and even impossible to verify connector properties in model checkers. Compared with existing approaches for verifying connectors' properties [5, 14, 15], reasoning about connectors using theorem provers like Coq is especially helpful when we take infinite behavior into consideration. The coinductive proof principle makes it possible to prove connectors' properties easily while it is difficult (sometimes impossible) for other approaches (like model checking) because of the huge (or maybe infinite) number of states.

The approach in this paper is not a brand new idea, as we have already provided a solution for modeling Reo in Coq in [16], where connectors are represented in a constructive way, and verification is essentially based on simulations. We do believe that the approach in this paper is reasonably different from its predecessor [16] where Coq seldom shows its real power. To be more specific, this work has its certain advantages comparing with [16] in the following aspects:

- **Modeling Method:** We use axioms to describe basic channels and their composition operations, which is more natural on a proof-assistant platform than the simulation-based approach in [16].
- **Expressive power:** Any valid Coq expression can be used to depict properties, which is obviously more powerful than just using LTL formulas in [16]. Then, support for continuous time behavior is also possible in our approach in this paper as we make use of real numbers to represent time values.

This paper is an extension of out earlier work [21] where equivalence and refinement relations can be proved among different connectors, while the approach in [16] is not capable of either equivalence or refinement checking. In the current paper, we aim to provide a more complete approach to formally model and reason about Reo connectors compared with the approach in [21]. The main drawback in [21] is that sometimes the user fails to prove that the complementary refinement relation holds or not using Coq. This problem is partially made up by using Z3 [8] in this paper. While Coq provides a platform to prove properties with a series of tactics and strategies to be considered, Z3 can be used as a component that can build automatically counter-examples when the properties do not hold. The logical formulas involved in the Reo models should reflect, in the same way as the Coq model, the constraints of connectors and be consistent with their behavior.

The paper is organized as follows. After this general introduction, we briefly summarize Reo, Coq and Z3 in Section 2. Section 3 introduces the formal model of basic channels, operators and complex connectors in Coq. Section 4 shows how to reason about connector properties and equivalence (or refinement) relations under our Coq formalization. Section 5 provides a Reo model in Z3 and shows the usage of Z3 in refinement checking. In Section 6, we conclude with some further research directions. Full source codes are provided at [1] for further reference.

2. Preliminaries

In this section, we provide a brief introduction to the coordination language Reo, the Coq proof assistant and the Z3 SMT solver.

2.1. The Coordination Language Reo

Reo is a channel-based exogenous coordination language where complex coordinators, called connectors, are compositionally built out of simpler ones [3]. The simplest connectors are channels with well-defined behavior such as synchronous channels, FIFO channels, etc. There are two types of channel ends: *source*

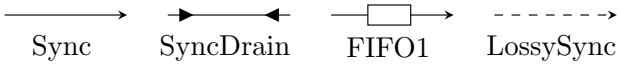


Figure 1: Four types of basic channels.

ends and *sink* ends. A source channel end accepts data into the channel, and a sink channel end dispenses data out of the channel.

The graphical notations of some basic channels are presented in Figure 1, and their behavior can be interpreted as follows:

- **Sync:** a synchronous channel with one source end and one sink end. The pair of I/O operations on its two ends can only succeed simultaneously.
- **SyncDrain:** a synchronous channel which has two source ends. The pair of input operations on its two ends can only succeed simultaneously. All data items written to this channel are lost.
- **FIFOn:** an asynchronous channel with one source end and one sink end, and a bounded buffer with capacity n. It can accept n data items from its source end before emitting data on its sink end. The accepted data items are kept in the internal buffer, and dispensed to the sink end in FIFO order. Especially, the FIFO1 channel is an instace of FIFOn where the buffer capacity is 1.
- **LossySync:** a synchronous channel with one source end and one sink end. The source end always accepts all data items. If there is no matching output operation on the sink end of the channel at the time that a data item is accepted, then the data item is lost; otherwise, the channel transfers the data item as a Sync channel, and the output operation at the sink end succeeds.

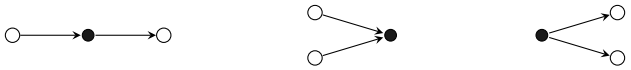


Figure 2: Operations of channel composition.

Complex connectors are constructed by composing simpler ones via the *join* and *hiding* operations. Channels are joined together in nodes. The set of channel ends coincident on a node is disjointly partitioned into the sets of source and sink channel ends that coincide on the node, respectively. Nodes are categorized into source, sink and mixed nodes, depending on whether all channel ends that coincide on a node are source ends, sink ends or a combination of the two. The hiding operation is used to hide the internal topology of a connector. The hidden nodes can no longer be accessed or observed from outside. There are three types of operations for channel composition: *flow-through*, *merge* and *replicate*. Figure 2 provides the graphical representation of these operations. Further details about Reo and its semantics can be found in [3, 4, 6].

2.2. Coq and Z3

Coq [10] is a widely-used proof assistant, where denotational formalizations (e.g. theorem and hypothesis) and operational ones (e.g. functions and algorithms) are naturally integrated. Moreover, it allows the interactive construction of formal proofs. The formal language used in Coq, *Gallina*, provides a convenient way to define both programming statements and mathematical propositions, for example:

```
1 (* a variable definition *)
2 Variables a b: nat.
3 (* a simple non-recursive function *)
4 Definition inc(a:nat) := a + 1.
5 (* axioms don't have to be proved *)
6 Axiom inc_ax: forall c:nat, inc(c) > c.
```

```
(* theorems rely on proving *)
Theorem inc_eq: forall c:nat, inc(c) = c + 1.
Proof.
(* interactive proving based on tactics *)
auto.
Qed.
```

As shown in this example, there are two different modes in Coq's interactive shell. When we start Coq, we can write declarations and definitions in a functional-programming mode. Then, when we start a *Theorem*, or *Lemma*, Coq jumps into the proving mode. We need to write *tactics* to reduce the proving goal and finally finish the formal proof.

Coq is equipped with a set of well-written standard libraries. For example, as used in this paper, *Stream* provides a co-inductive definition of infinite lists, and *Reals* defines various operations and theorems on real numbers. Usually, quite a few lemmas and theorems are pre-defined in such libraries, making it substantially easier to prove our goals.

Z3 [8] is an efficient SMT (Satisfiability Modulo Theories) solver freely available from Microsoft. It has been used in various software verification and analysis applications. Z3 expands to deciding the satisfiability (or dually the validity) of first order formulas with respect to combinations of theories such as: arithmetic, bit-vectors, arrays, and uninterpreted functions. Given the data and time constraints of connectors, it allows us to verify the satisfiability of properties or refinement relations. Z3 provides bindings for several programming languages. In this paper, we use *Z3 python-bindings* to construct the models and carry out experiments. The following example provides an intuitive understanding of the Z3 solver.

```
x, y = Int('x'), Int('y')

s = Solver()
s.add(x > 10, Or(x + y > 3, x - y < 2))
# Checking satisfiability of constraints in the solver s
print s.check()

# Create a new scope
s.push()
s.add(y < 11)
# Checking satisfiability of updated set of constraints
print s.check()

# Restoring state
s.pop()
# Checking satisfiability of restored set of constraints
print s.check()
```

This example contains two constraints at first. From these two constraints, we can see that Z3 supports operators like $<$, $>$ for comparison, apart from boolean operators, such as *And*, *Or*, *Not* and *Implies*. The function *Solver* creates a general purpose solver instance where constraints can be asserted through its *add* method. The method *check* assesses the satisfiability of the asserted constraints. Finally, the result is satisfiable (unsatisfiable) if the solver returns *sat* (*unsat*) respectively. A solver may fail to check the satisfiability of a system of constraints and *unknown* is returned. Each solver maintains a stack of assertions. The command *push* creates a new scope by saving the current stack size. The command *pop* removes all constraints that have been asserted after the last *push* command.

3. Formal Modeling in Coq

In this section, we show how primitive connectors, i.e., channels, and operators for connector composition are specified in Coq and used for modeling complex connectors. Basic channels, which can be regarded as axioms of the whole framework, are specified as logical predicates illustrating the relation between the

timed data streams of input and output. When we need to construct a more complex connector, appropriate composition operators are applied depending on the topological structure of the connector.

3.1. Basic Definitions

The behavior of a connector can be formalized by means of data-flows at its sink and source nodes which are essentially infinite sequences. With the help of the stream library in Coq, such infinite data-flows can be defined as *timed data streams*:

```
1  Definition Time := R.
2  Definition Data := nat.
3  Definition TD := Time * Data.
4  Variable Input : Stream TD.
5  Variable Output : Stream TD.
```

In our framework, time is represented by real numbers. The continuity of the set of real numbers is sufficiently enough for our modeling approach. Also the continuous time model is more appropriate than the discrete model since it is very expressive and closer to the nature of time in the real world. Thus, the time sequence consists of increasing and diverging time moments. For simplicity, here we take the natural numbers as the definition of data, which can be easily expanded according to different application domains. The Cartesian product of time and data defines a TD object. We use the stream module in Coq to produce streams of TD objects.

Some auxiliary functions and predicates are defined to ease the representation of axioms for basic channels in Reo. This part can be extended for further use in different problems. The terms *PrL* and *PrR* take a pair of values (a, b) that has Cartesian product type A × B as the argument and return the first or second value of the pair, respectively. The following functions provide some judgment of time, which can make the description of axioms and theorems for connectors more concise and clear: *Teq* means that values of time in the two streams taken as parameters are equal and *Tneq* has the opposite meaning. *Tle* (resp. *Tgt*) represents that time of the first stream is strictly lesser (greater) than the second stream. The judgement about equality of data is analogous to the judgement of time. *Deq* represents that the data of two streams are equal.The complete definition of these functions can be found at [1].

3.2. Formal Modeling of Basic Channels

We use the previous predicates to describe the constraints on time and data, respectively, and their conjunction to provide the complete specification of basic channels. This approach provides a relational semantics for Reo using predicates that relates the inputs and outputs of connectors. This model offers convenience for the analysis and proof of connector properties. In the following, we present a few examples of the formal model of basic channels.

The simplest form of a synchronous channel is denoted by the Sync channel type. For a channel of the Sync type, a read operation on its sink end succeeds only if there is a write operation pending on its source end. Thus, the time and data of a stream flowing into the channel are exactly the same as the stream that flows out of the channel [1]. The Sync channel can be defined as follows in the Coq system:

```
1  Definition Sync (Input Output:Stream TD) : Prop :=
2      Teq Input Output /\ Deq Input Output.
```

The channel of type SyncDrain is a synchronous channel that allows pairs of write operations pending on its two ends to succeed simultaneously. All written data items are lost. Thus, the SyncDrain channel is used for synchronising two timed data streams on its two source ends. This channel type is an important basic synchronization building block for the construction of more complex connectors. The SyncDrain channel can be defined as follows:

[1] If we use α, β to denote the data streams that flow through the channel ends of a channel and a, b to denote the time stream corresponding to the data streams, i.e., the i-th element $a(i)$ in a denotes exactly the time moment of the occurrence of $\alpha(i)$, then we can easily obtain the specifications for different channels, as discussed in [18, 20]. For example, a synchronous channel can be expressed as $\alpha = \beta \wedge a = b$.

```
Definition SyncDrain (Input Output:Stream TD) : Prop :=
    Teq Input Output.
```

The channel types FIFO and FIFOn where n is an integer strictly greater than 0 represent the typical unbounded and bounded asynchronous FIFO channels. A write to a FIFO channel always succeeds, and a write to a FIFOn channel succeeds only if the number of data items in its buffer is less than its bounded capacity n. A read or take operation on a FIFO or FIFOn channel suspends until the first data item in the channel buffer can be obtained and then the operation succeeds. For simplicity, we take the FIFO1 channel as an example. This channel type requires that the time when it consumes a data item through its source end is strictly earlier than the time when the data item is delivered through its sink end. Besides, as the buffer has the capacity 1, time of the next data item that flows in should be strictly later than the time when the data in the buffer is delivered. We use the conjunction of predicates in its definition as follows:

```
Definition FIFO1(Input Output:Stream TD) : Prop :=
    Tle Input Output /\ Tle Output (tl Input) /\ Deq Input Output.
```

For a FIFO1 channel whose buffer already contains a data element e, the communication can be initiated only if the data element e can be taken via the sink end. In this case, the data stream that flows out of the channel should get an extra element e settled at the beginning of the stream. And time of the stream that flows into the channel should be strictly earlier than time of the tail of the stream that flows out. But as the buffer contains the data element e, new data can be written into the channel only after the element e has been taken. Therefore, time of the stream that flows out is strictly earlier than time of the stream that flows in. The channel can be represented as:

```
Definition FIFO1e(Input Output:Stream TD)(e:Data) : Prop :=
    Tgt Input Output /\ Tle Input (tl Output)
    /\ PrR (hd Output) = e /\ Deq Input (tl Output).
```

In the following we define LossySync as an Axiom because it is easier to use than the coinductive expression that specifies its behavior.

A LossySync channel behaves the same as a Sync channel, except that a write operation on its source end always succeeds immediately. If a compatible read or take operation is already pending on its sink end, the written data item is transferred to the pending operation and both succeed. Otherwise, the write operation succeeds and the data item is lost. The LossySync channel can be defined as follows:

```
Parameter LossySync: Stream TD -> Stream TD -> Prop.
Axiom LossySync_coind:
    forall Input Output: Stream TD,
    LossySync Input Output ->
    (( hd Output = hd Input /\ LossySync (tl Input)(tl Output)) \/
       LossySync(tl Input) Output).
```

Defining basic channels by conjunction and disjunction of predicates provides the following benefits:

- Firstly, this makes the structure of the definition in Coq identical to the usual implementation.
- Secondly, we can easily split predicates for proofs of different properties which can make the proving process simpler.

3.3. Formal modeling of Operators

We have shown the formalization of channel types in Coq. Now we start defining the composition operators for connector construction. There are three types of composition operators, which are *flow-through*, *replicate* and *merge*, respectively.

The flow-through operator simply allows data items to flow through the junction node, from one channel to the other. We do not need to give the flow-through operator a specific definition in Coq as this operator can be achieved directly through renaming of nodes. For example, while we specify two channels *Sync(A,B)*

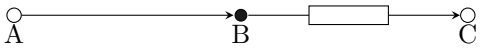


Figure 3: A connector consisting of a Sync channel and a FIFO1 channel.

and *FIFO1(B,C)*, a flow-through operator that acts on node B for these two channels has been achieved implicitly by using the same identifier.

The replicate operator puts the source ends of different channels together into one common node, and a write operation on this node succeeds only if all the channels are able to consume a copy of the written data. Similar to the flow-through operator, the replicate operator can be directly achieved through renaming of nodes according to the structure of connectors, i.e. making all the nodes connected by the replicate operator having the same name. For example, for two channels *Sync(A,B)* and *FIFO1(C,D)*, we can illustrate *Sync(A,B)* and *FIFO1(A,D)* in Coq 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 merge operator is more complicated. We consider merging two channels AB and CD. When the merge operator acts on these two channels, it leads to a choice of taking from the common node that delivers a data item out of AB or CD. The first predicate is related to the time constraint. We require that data items from the two source ends cannot come at the same time. Similar to the definition of basic channels, we define merge as the conjunction of predicates and use recursive definition here:

```
1  Parameter merge: Stream TD -> Stream TD ->Stream TD -> Prop.
2  Axiom merge_coind:
3      forall s1 s2 s3:Stream TD,
4      merge s1 s2 s3-> (
5      ~ (PrL(hd s1) = PrL(hd s2)) /\
6      ( (PrL(hd s1) < PrL(hd s2)) -> ((hd s3 = hd s1) /\ merge (tl s1) s2 (tl s3)) ) /\
7      ( (PrL(hd s1) > PrL(hd s2)) -> ((hd s3 = hd s2) /\ merge s1 (tl s2) (tl s3)) )
8      ).
```

Based on the definition of basic channels and operators, more complex connectors can be constructed structurally. To show how a composite connector is constructed, we consider a simple example as shown in Figure 3, where a FIFO1 channel is attached to the sink end of a Sync channel. Assume AB is of type Sync and BC is of type FIFO1, then we can construct the required connector by illustrating *Sync(A,B)* and *FIFO1(B,C)*. The configuration and the functionality of the required connector can be specified using this concise method. Note that the composition operations can be easily generalized to the case of multiple nodes, where the modeling of connectors is similar. More examples can be found in Section 4.

4. Reasoning about Connectors in Coq

After modeling a connector in Coq, we can analyse and prove important properties of the connector. In this section, we give some examples to elucidate how to reason about connector properties and prove refinement/equivalence relations between different connectors with the help of Coq.

4.1. Derivation of Connector Properties

The proof process of a property is as follows: the user states the proposition that needs to be proved, called a *goal*, and then he/she applies commands called *tactics* to decompose this goal into simpler subgoals or prove it directly. This decomposition process ends when all subgoals are completely proved. In the following, we use four examples to illustrate our approach instead of giving all the complex technical details.

Example 4.1. *We first consider the connector given in Figure 3, which consists of two channels* AB *and* BC *with types Sync and FIFO1, respectively. We use a and b to denote the time streams when the corresponding data streams flow in and out of the Sync channel* AB*, and c to denote the time stream for the data stream that flows out of the FIFO1 channel* BC*. Here we can see that flow-through behavior is performed by the mixed node* B*. The time when the stream flows into the FIFO1 channel* BC *is equal to the time when the*

Step	Feedback

`Theorem thm_4_1:`
`forall A B C,`
`Sync A B → FIFO1 B C → Tle A C.`
	1 subgoal:
	forall A B C,
	Sync A B → FIFO1 B C → Tle A C

`intros`	1 subgoal: *Tle A C* *H : Sync A B; H0 : FIFO1 B C*
`destruct H`	1 subgoal: *Tle A C* *H : Teq A B; H1 : Deq A B;* *H0 : FIFO1 B C*
`destruct H0`	1 subgoal: *Tle A C* *H : Teq A B; H1 : Deq A B;* *H0 : Tle B C; H2 : Tle C (tl B) ∧ Deq B C*
`intro n`	1 subgoal: *PrL (Str_nth n A) < PrL (Str_nth n C)* *H : Teq A B; H1 : Deq A B; H0 : Tle B C;* *H2 : Tle C (tl B) ∧ Deq B C*
`rewrite H`	1 subgoal: *PrL (Str_nth n B) < PrL (Str_nth n C)* *H : Teq A B; H1 : Deq A B; H0 : Tle B C;* *H2 : Tle C (tl B) ∧ Deq B C; n : nat*
`apply H0`	No more subgoals

Table 1: Steps and feedbacks for proving Theorem 4.1

stream flows out of the Sync channel AB. *The following theorem states the property $a < c$ for this connector. The connector is based on the axioms Sync and FIFO1, which can be used as hypotheses for the proof of the following theorem.*

Theorem 4.1 (Transfer_le)**.** $\forall A, B, C.\ Sync(A, B) \wedge FIFO1(B, C) \rightarrow Tle(A, C)$.

In Coq, the theorem can be proved as follows:

```
1 Theorem thm_4_1: forall A B C,
2   Sync A B /\ FIFO1 B C -> Tle A C.
3 Proof.
4     intros. destruct H. destruct H0.
5     intro n. rewrite H. apply H0.
6 Qed.
```

First we give the Coq system a proposition `thm_4_1` which needs to be proved. The proposition is represented by a logical expression. Table 1 shows the detailed proof steps and the feedback that the Coq system provides during the proof.

The aforementioned benefits of using conjunction and disjunction of logical predicates to describe basic channels instead of formalizing them based on automata or states have emerged while proving this example. After constructing the new connector, we use `intros` to split conditions and conclusions. Then we can use `destruct` to obtain the conditions for time and data separately, and make the proving procedure much more convenient. Once we obtain the concrete conditions, we can use `intro` to reduce the subgoal to the comparison between each time point in these two sequences. Then by using `rewrite H`, we can make the proof a step forward with known conditions of the comparison of time a and b, and finally by `apply H0` we can prove the goal. This is the implementation for reasoning about the constructed connector. Note that proper selection of strategies and tactics is essential for the proof of connector properties.

Example 4.2. *In this example, we show a more interesting connector named* alternator *which consists of three channels* AB, AC *and* BC *of type SyncDrain, FIFO1 and Sync, respectively. With the help of this*

(a) Before merge

(b) After merge

Figure 4: Alternator

connector, we can get data from node B *and* A *alternatively at node* C. *By using the axioms for the basic channels and operators of composition, we can get the connector as shown in Figure 4(b). The two channels* AC *and* BC *are merged together at node* C. *Before the merge operation, the connector's structure is as shown in Figure 4(a), which is useful in the reasoning about the alternator.*

Here the replicate operation has been applied twice for the alternator: node A *becomes the common source node of* SyncDrain (A,B) *and* FIFO1(A,C1), *and node* B *becomes the common source node of* SyncDrain(A,B) *and* Sync(B,C2). *Let the time streams when the data streams flow in the two source nodes* A *and* B *be denoted by* a *and* b, *and the time streams when the data streams flow out of the channels* FIFO1(A,C1) *and* Sync(B,C2) *be denoted by* $c1$ *and* $c2$, *respectively. Theorem 4.2 specifies the property* $c2 < c1 \wedge c1 < tl(c2)$ *of the connector in Figure 4(a). The connector is based on the axioms Sync, SyncDrain and FIFO1. These three corresponding axioms are used as hypotheses for the proof of this theorem.*

Theorem 4.2 (Unmerge)**.** $\forall A, B, C1, C2\,.\,SyncDrain(A, B) \wedge FIFO1(A, C1) \wedge Sync(B, C2) \rightarrow Tle(C2, C1) \wedge Tle(C1, tl(C2))$

We first introduce some lemmas to ease the proof.

```
1  Lemma transfer_eq : forall s1 s2 s3 : Stream TD
2    ((Teq s1 s2) /\ (Teq s2 s3)) -> (Teq s1 s3).
3  Lemma transfer_eqtl : forall s1 s2 : Stream TD,
4    (Teq s1 s2) -> (Teq tl s1)) (tl s2)).
5  Lemma transfer_leeq : forall s1 s2 s3 : Stream TD,
6    ((Tle s1 s2) /\ (Teq s2 s3)) -> (Tle s1 s3).
7  Lemma transfer_hdle : forall s1 s2 : Stream TD,
8    (Tle s2 s1) -> (PrL (hd s1) > PrL (hd s2)).
```

In Coq, the theorem can be proved as follows. Note that the formalism is slightly different from the previous one. The *section* environment encapsulate hypotheses as assumptions of the theorem. So the two definitions are exactly equivalent.

```
1  Section Alt.
2  Hypothesis D1: SyncDrain A B.
3  Hypothesis D2: FIFO1 A C1.
4  Hypothesis D3: Sync B C2.
5  Theorem unmerge:
6     (Tle C2 C1) /\ (Tle C1 (tl C2)).
```

After constructing the connector in Figure 4(a), we use `destruct` to obtain the conditions for time and data, respectively. Because the goal we are going to prove is the conjunction of logical predicates, we use `split` to obtain the single subgoals represented by logical predicates. Further, `intros` is used to reduce the goal into the element-wise comparison between each data in these two sequences. Then `rewrite` and `apply` are used similarly multiple times until the goal is proved finally. Concrete proof steps and feedbacks are specified in Table 2.

Step	Feedback
`Theorem unmerge`	1 subgoal: *Tle C2 C1 ∧ Tle C1 (tl C2)*
`destruct D2`	1 subgoal: *Tle C2 C1 ∧ Tle C1 (tl C2)* *H : Tle A C1; H0 : Tle C1 (tl A) ∧ Deq A C1*
`destruct D3`	1 subgoal: *Tle C2 C1 ∧ Tle C1 (tl C2)* *H : Tle A C1; H0 : Tle C1 (tl A) ∧ Deq A C1;* *H1 : Teq B C2; H2 : Deq B C2*
`destruct H0`	1 subgoal: *Tle C2 C1 ∧ Tle C1 (tl C2)* *H : Tle A C1; H0 : Tle C1 (tl A);* *H3 : Deq A C1; H1 : Teq B C2; H2 : Deq B C2*
`split`	2 subgoals: *Tle C2 C1; Tle C1 (tl C2)* *H : Tle A C1; H0 : Tle C1 (tl A);* *H3 : Deq A C1; H1 : Teq B C2; H2 : Deq B C2*
`intros` n	2 subgoals: *PrL (Str_nth n C2) < PrL (Str_nth n C1); Tle C1 (tl C2)* *H : Tle A C1; H0 : Tle C1 (tl A); H3 : Deq A C1;* *H1 : Teq B C2; H2 : Deq B C2; n : nat*
`rewrite ←` *H1*	2 subgoals: *PrL (Str_nth n B) < PrL (Str_nth n C1); Tle C1 (tl C2)* *H : Tle A C1; H0 : Tle C1 (tl A); H3 : Deq A C1;* *H1 : Teq B C2; H2 : Deq B C2; n : nat*
`rewrite ←` *D1*	2 subgoals: *PrL (Str_nth n A) < PrL (Str_nth n C1); Tle C1 (tl C2)* *H : Tle A C1; H0 : Tle C1 (tl A); H3 : Deq A C1;* *H1 : Teq B C2; H2 : Deq B C2; n : nat*
`apply` *H*	1 subgoal: *Tle C1 (tl C2)* *H : Tle A C1; H0 : Tle C1 (tl A); H3 : Deq A C1;* *H1 : Teq B C2; H2 : Deq B C2*
`intros` n	1 subgoal: *Tle C1 (tl C2)* *H : Tle A C1; H0 : Tle C1 (tl A); H3 : Deq A C1;* *H1 : Teq B C2; H2 : Deq B C2*
`rewrite ←` *D4*	2 subgoals: *PrL (Str_nth n C1) < PrL (Str_nth n (tl B)); Teq B C2* *H : Tle A C1; H0 : Tle C1 (tl A); H3 : Deq A C1;* *H1 : Teq B C2; H2 : Deq B C2; n : nat*
`rewrite ←` *D5*	3 subgoals: *PrL (Str_nth n C1) < PrL (Str_nth n (tl A)); Teq A B; Teq B C2* *H : Tle A C1; H0 : Tle C1 (tl A); H3 : Deq A C1;* *H1 : Teq B C2; H2 : Deq B C2; n : nat*
`apply` *H0*	2 subgoals: *Teq A B; Teq B C2* *H : Tle A C1; H0 : Tle C1 (tl A); H3 : Deq A C1;* *H1 : Teq B C2; H2 : Deq B C2; n : nat*
`apply` *D1*	1 subgoal: *Teq B C2* *H : Tle A C1; H0 : Tle C1 (tl A); H3 : Deq A C1;* *H1 : Teq B C2; H2 : Deq B C2; n : nat*
`apply` *D3*	No more subgoals.

Table 2: Steps and feedback

An additional hypothesis is needed for the proof of alternator which merges *C1* and *C2* into a common node *C*. Based on the three hypotheses for channels and the additional one. The alternation theorem is presented as the following proposition which needs to be proved:

Figure 5: Example of connector refinement

```
1  Hypothesis D4: merge C1 C2 C.
2  Theorem alternator:
3    hd C = hd C2 /\ merge C1 (tl C2) (tl C).
4  Proof.
5    destruct unmerge. (* ... *)
```

Here we only present the first step which shows how a proven theorem can be applied in another proof and omit the full details. Note that the property of the unmerged alternator (i.e. Theorem 4.2) mentioned at the beginning of the example turns out to be very useful in the proof of the property of alternator. The unmerged alternator consists of three channels. *AB*, *AC1* and *BC2* are of type SyncDrain, FIFO1 and Sync shown on the left side of Figure 4. We construct the connector *Alternator* through the unmerged one instead of building it from basic channels. As for verification, it also simplifies the process of proving the property of alternator as we can take advantage of the property of the unmerged one directly.

4.2. Refinement and Equivalence

A refinement relation between connectors which allows developing connectors systematically in a stepwise fashion, may help to bridge the gap between requirements and the final implementations. The notion of refinement has been widely used in different system descriptions. For example, in data refinement [9], the 'concrete' model is required to have *enough redundancy* to represent all the elements of the 'abstract' one. This is captured by the definition of a surjection from the former into the latter (the *retrieve map*). If models are specified in terms of pre and post-conditions, the former are weakened and the latter strengthened under refinement [11]. In process algebra, refinement is usually discussed in terms of several 'observation' preorders, and most of them justify transformations entailing *reduction of nondeterminism* (see, for example, [17]). For, the refinement relation can be defined as in [20], where a proper refinement order over connectors has been established based on the implication relation on predicates.

Connector Q is a refinement of another connector P if the behavior property of P can be derived from *hypothesis* Q, i.e., the behavior property of connector Q (denoted by $P \sqsubseteq Q$). Two connectors are equivalent if each one of them is a refinement of the other. In the following, we show two examples of such refinement and equivalence relations for connectors.

Example 4.3 (Refinement)**.** *Taking the two connectors in Figure 5 into consideration, connector Q is a refinement of connector P.*

We have mentioned that newly constructed connectors can be specified as theorems. Given arbitrary input timed data stream at node A *and output timed data streams at nodes* C, D*, connector P enables the data written to the source node* A *to be asynchronously taken out via the two sink nodes* C *and* D*, but it has no constraints on the relationship between the time of the two output events. On the other hand, connector Q refines this behavior by synchronizing the two sink nodes, which means that the two output events must happen simultaneously. To be more precise, we use c, d to denote the time streams of the two outputs and a to denote the time stream of the input. Connector P satisfies condition $a < c \wedge a < d$ and connector Q satisfies $a < c \wedge a < d \wedge c = d$.*

The refinement relation can be formally defined in Coq as:

```
1  Theorem refinement : forall A C D,
2      (exists B, (FIFO1 A B) /\ (Sync B C) /\ (Sync B D)) ->
3      (exists E, (Sync A E) /\ (FIFO1 E C) /\ (FIFO1 E D)).
```

To prove this refinement relation, we first introduce a lemma which is frequently used in the proof.

Lemma 4.1 (Eq)**.** $\forall A, B : Stream\ TD.\ Sync(A, B) \Leftrightarrow A = B.$

The lemma means that $Sync(A, B)$ and $A = B$ can be derived from each other. Although this lemma seems to make the presence of Sync channels in connectors redundant, it is not the case for most connectors. For example, if we consider the alternator in Example 2, it cannot accept any input data if we remove the synchronous channel BC and use one node for it.

By using the axioms for the basic channels and the operators of composition, we can obtain the two connectors easily. In the process of constructing the connectors, the flow-through and replicate operations act once for each connector, respectively.

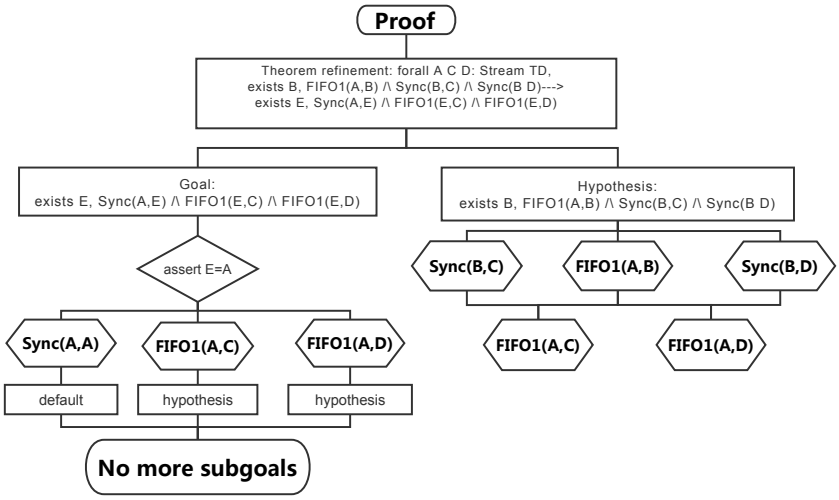


Figure 6: Proof Steps Flow Chart

Figure 6 shows the flow chart for the proof steps of connector refinement in this example.

We now show the specific tactics used in the proof of refinement $P \sqsubseteq Q$ for connectors P and Q in this example. We need to find a timed data stream which specifies the data-flow through node E of connector P, i.e., an appropriate E that satisfies $Sync(A, E) \wedge FIFO1(E, C) \wedge FIFO1(E, D)$.

First, we employ `intros` to acquire a simpler subgoal $\exists E_0.Sync(A, E_0) \wedge FIFO1(E_0, C) \wedge FIFO1(E_0, D)$. Then we assert that $E = A$. After, using `split`, we split the goal into two subgoals $Sync(A, E)$ and $FIFO1(E, C) \wedge FIFO1(E, D)$. And, by `rewrite` H_0 $(H_0 : E = A)$ we replace the two subgoals with $Sync(A, A)$ and $FIFO1(E, C) \wedge FIFO1(E, D)$, respectively.

Through `apply` Lemma 4.1 (Eq), we have $A = A$ in place of $Sync(A, A)$. Next the tactic `reflexivity` makes the subgoal $A = A$ proved directly. Up to now, the initial subgoal $Sync(A, E)$ has been achieved.

Using `split` again, the remaining unproven subgoal is split into two subgoals *FIFO1 (E, C)* and *FIFO1 (E, D)*. After destructing the precondition three times, we succeed in obtaining three hypotheses: *H: FIFO1 (A, x); H1: Sync (x, C); H2: Sync (x, D)*. Assuming $x = C$ and then using tactics `apply Eq` and `assumption`, assertion $x = C$ is proved easily. Meanwhile, we get hypothesis *H3:* $x = C$. Via *Rewrite* $\leftarrow$*H3*, we bring left in place of the right side of the equation *H3:* $x = C$ into *FIFO1 (E, C)* and have *FIFO1 (E, x)*. Similarly, `rewrite H0` and further we get the result *FIFO1 (A, x)* which is exactly hypothesis *H*. By using `assumption`, the second subgoal is proved already. Using substantially the same tactic steps, *FIFO1 (E, D)* can be proved. Finally, we have no more subgoals.

Example 4.4 (Equivalence)**.** *For the connector P in Example 4.3, we can add three more basic channels to build a new connector* R *which is equivalent to Q.* R *can be interpreted similarly based on basic channels and operators. We will omit the details for its construction here and prove the equivalence between the two connectors* R *and Q directly.*

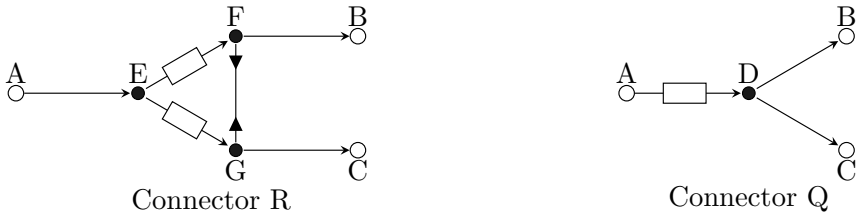


Figure 7: Example of connector equivalence

Equivalence relationship between the two connectors can be formalized as:

```
1  Theorem equivalence: forall A B C,
2    (exists E F G,
3      (Sync A E) /\ (FIFO1 E F) /\ (Sync F B) /\
4      (FIFO1 E G) /\ (Sync G C) /\ (SyncDrain F G)
5    ) <->
6    (exists D,
7      (FIFO1 A D) /\ (Sync D B) /\ (Sync D C)
8    ).
```

The proof of this theorem has two steps. Firstly, we prove that the new connector R is a refinement of connector Q. We aim at finding an appropriate D that satisfies

$$FIFO1(A, D) \wedge Sync(D, B) \wedge Sync(D, C)$$

Similar to Example 4.3, we first assert $D = F$, which leads to

$$FIFO1(A, F) \wedge Sync(F, B) \wedge Sync(F, C)$$

From Lemma 4.1, we have $Sync(A, E)$, or $A = E$. Therefore, *FIFO1(E,F)* can be replaced by *FIFO1(A,F)*. By adopting *FIFO1(E,F)* and *FIFO1(E,G)*, we can prove that the data sequences at F and G are equal. Similarly, data sequences at C, G and F are also equal, wrt. $Sync(G, C)$.

Further according to *Sync(G,C)* and *SyncDrain(F,G)*, the time sequences at F and C are proved equal. With the combination of relations on time and data between F and C, we can draw the conclusion *Sync(F,C)*.

Up to now, we present a proof for *Sync(F,C)* and *FIFO1(A,F)* by the derivation. Besides, *Sync(F,B)* is already declared in the assumptions. Consequently, the refinement relation has been proved.

Secondly, we prove that connector Q is a refinement of connector R. We hope to find appropriate timed data streams at *E,F,G* which satisfy

$$\begin{aligned} Sync(A,E) \quad &\wedge \quad Sync(G,C) \wedge FIFO1(E,G) \wedge Sync(F,B) \\ &\wedge \quad FIFO1(E,F) \wedge SyncDrain(F,G). \end{aligned}$$

We can directly assume $E = A$, $F = D$ and $G = D$. Now we only need to prove $Sync(A, A) \wedge Sync(D, C) \wedge FIFO1(A, D) \wedge Sync(D, B) \wedge FIFO1(A, D) \wedge SyncDrain(D, D)$, which can be easily derived from the assumptions.

5. Refinement Checking in Z3

As shown previously, modeling and reasoning of Reo connectors in Coq allows us to reason about a relatively wide range of properties. However, when dealing with verification of refinement/equivalence relations between connectors, it is only capable of proving that one connector is indeed a refinement of the other one but cannot automatically provide a counter example when one connector is not a refinement of the other connector. Moreover, the refinement relation is a fairly important part of properties, which has been investigated in many applications. When we fail to find a proof of the existence of refinement relations between connectors using Coq, we propose to use Z3, an SMT solver, to search for bounded counter examples. If a counter example is successfully located, it means that there is no refinement relation between the connectors.

5.1. Formalization of Basic Channels in Z3

We first need to implement the construction of connectors and also define the refinement checking function in a way that fits Z3. Typically, connector construction starts with the formal definition of basic channels and composition operators, which will be used hereafter to build and assess more complex models. Such a framework must be carefully developed so that the further modeling can be simplified as much as possible.

Coinciding with the definition of channels in Coq, we dealt with both time and data relation constraints in the basic definition. An auxiliary function `Conjunction` is used to take the conjunction of every constraint in the constraint lists parameterized by *constraints*. For example, the specific behavior constraints for *Sync* channel are classified into two types: data constraints and time constraints. The definition of *Sync* in Coq provides the behavior pattern it needs to follow: each item in timed data streams of output specified as "node[1]" is equivalent to the timed data items of input specified as "node[0]", which are exactly data and time constraints. As they are both requirements that we need to satisfy no matter data or time is related, each constraint in the list will all be combined finally.

```
1  def Sync(nodes, bound):
2     assert len(nodes) == 2
3     constraints = []
4     for i in range(bound):
5       constraints += [ nodes[0]['data'][i] == nodes[1]['data'][i] ]
6       constraints += [ nodes[0]['time'][i] == nodes[1]['time'][i] ]
7     return Conjunction(constraints)
```

SyncDrain channel is defined in a similar way. On the basis of the behavior of *SyncDrain* channel, only time related constraints are needed, i.e., equivalence of corresponding items in two time streams of the two inputs.

```
1  def SyncDrain(nodes, bound):
2     assert len(nodes) == 2
3     constraints = []
4     for i in range(bound):
5        constraints += [nodes[0]['time'][i] == nodes[1]['time'][i]]
6     return Conjunction(constraints)
```

Regarding *FIFO1* channel, data related constraints are the same as the *Sync* channel. But there are different time related constraints for the *FIFO1* channel. As the buffer capacity is one, the relations of each item in the input time stream and output time stream need to be carefully dealt with, especially as the next input should be strictly later than the present output. If the buffer contains a data item at the beginning, i.e. the variant version *FIFO1e* channel, then the constraints related to data and time are just a little different. Note that the data item "e" in the buffer should be the first one to transit and all the differences made to the constraints result from it. At last, their corresponding constraint lists consist of constraint statements similar to the statements above.

```
def Fifo1(nodes, bound):
   assert len(nodes) == 2
   constraints = []
   for i in range(bound):
      constraints += [ nodes[0]['data'][i] == nodes[1]['data'][i] ]
      constraints += [ nodes[0]['time'][i] < nodes[1]['time'][i] ]
      if i != 0:
          constraints += [ nodes[0]['time'][i] > nodes[1]['time'][i-1] ]
   return Conjunction(constraints)
```

```
def Fifo1e(e):

  def Fifo1eInstance(nodes, bound):
    assert len(nodes) == 2
    constraints = []
    constraints += [nodes[1]['data'][0] == e]
    for i in range(bound-1):
        constraints += [nodes[0]['data'][i] == nodes[1]['data'][i + 1]]
        constraints += [nodes[0]['time'][i] < nodes[1]['time'][i + 1]]
    for i in range(bound):
        constraints += [nodes[0]['time'][i] > nodes[1]['time'][i]]
    return Conjunction(constraints)

  return Fifo1eInstance
```

In accordance with the behavior of *LossySync* channel, each item of the input stream may or may not be lost. If a timed data item is lost, then the corresponding output gets nothing. If not, it behaves exactly like a successful transit of *Sync* channel, so in this case the data and time related constraints are identical with those in the definition of *Sync* channel. The biggest difference is that *LossySync* channel should be defined in a recursive way according to its own distinctive behavior. Note that the constraints are added recursively which coincides with the original definition.

```
def LossySync(nodes, bound, idx = 0, num = 0):
   assert len(nodes) == 2
   if bound == num:
     return True
   if bound == idx:
     return True
   constraints_0 , constraints_1 = [], []
   constraints_0 += [ nodes[0]['time'][idx] != nodes[1]['time'][num]]
   constraints_1 += [ nodes[0]['data'][idx] == nodes[1]['data'][num]]
   constraints_1 += [ nodes[0]['time'][idx] == nodes[1]['time'][num]]
   return Or(
     And(Conjunction(constraints_0), Channel.LossySync(nodes, bound, idx + 1, num)),
     And(Conjunction(constraints_1), Channel.LossySync(nodes, bound, idx + 1, num + 1))
   )
```

5.2. Composition Operators

While the basic channels are already well defined as the basis of the whole construction framework, the composition operators also need to be specified for large-scale connectors' construction just like cement for bricks. There are three kinds of composition operators: *replicate*, *flow-through* and *merge*.

Both *replicate* and *flow-through* operators can be implicitly achieved when we compose channels to construct connectors using same node names. A simple example for the implementation of *replicate* is `c1.connect('Sync', 'A', 'E'), c1.connect('Fifo1', 'E', 'F'), c1.connect('Fifo1', 'E', 'G')` where the message that node `E` receives from `A` is duplicated and sent to `F` and `G` simultaneously. As

for *Flow-through* operator, a simple example is `c2.connect('Fifo1', 'A', 'D'), c2.connect('Sync', 'D', 'B')` where the timed data stream flows from node `A` to node `B` through the mixed node `D`. In these two examples, `c1` and `c2` both belong to the `Connector` class which will be elaborated in detail later.

Among the three types of composition operators, the most difficult one is *Merge*, which is specially dealt with as a channel *Merger* in our model. Together with the other two kinds of composition operators: *replicate* and *flow-through*, they provide a complete basis for the construction of any connector, which serve as the original set of composition operations [3], [20]. For each item in the output stream of the *Merger* channel, there is a non-deterministic choice between the two input nodes. Hence, the *Merger* channel also needs to be defined in a recursive way like the *LossySync* channel. Note that the constraints also conform well to the original semantics of the *merge* operator. Derived from the two recursive definitions, an additional advantage of defining *LossySync* and *Merger* channel in this way is that it can preserve the behavioral pattern to a great extent without any assumption such as priorities of reading or taking data from the two input streams.

```
def Merger(nodes, bound, idx_1 = 0, idx_2 = 0):
   assert len(nodes) == 3
   if bound == idx_1 + idx_2:
     return True
   constraints_1 , constraints_2 = [] , []
   constraints_1 += [ nodes[0]['data'][idx_1] == nodes[2]['data'][idx_1 + idx_2]]
   constraints_1 += [ nodes[0]['time'][idx_1] == nodes[2]['time'][idx_1 + idx_2]]
   constraints_1 += [ nodes[0]['time'][idx_1] < nodes[1]['time'][idx_2]]
   constraints_2 += [ nodes[1]['data'][idx_2] == nodes[2]['data'][idx_1 + idx_2]]
   constraints_2 += [ nodes[1]['time'][idx_2] == nodes[2]['time'][idx_1 + idx_2]]
   constraints_2 += [ nodes[1]['time'][idx_2] < nodes[0]['time'][idx_1]]
   return Or(
     And(Conjunction(constraints_1), Channel.Merger(nodes, bound, idx_1 + 1, idx_2)),
     And(Conjunction(constraints_2), Channel.Merger(nodes, bound, idx_1, idx_2 + 1))
   )
```

5.3. Refinement Checking

With the above basic channels and composition operators, the next step is to construct Reo connectors in this framework. In our model, `Connector` is defined as a class, which provides a method for constructing complex connectors out of the basic channels and composition operators. The statements inside the class definition are function definitions. We have seen the function `connect` used in the two simple examples of *replicate* and *flow-through* in Section 5.2.

```
class Connector:
    def __init__(self):
        self.channels = []

    def connect(self, channel, *nodes):
        self.channels += [(channel, nodes)]
        return self

    def isRefinementOf(self, abstraction, bound):
      ...
```

The `isRefinementOf` is used to express refinement checking. The result type of the function is Boolean, i.e., a connector is either or not a refinement of another connector. According to the definition of refinement relation in Section 4, we have a formula that represent it properly: $P \sqsubseteq Q$ if and only if the behavior property of connector P can be derived from the property of connector Q, i.e., $Q \rightarrow P$. If Q is indeed a refinement of P, it can be rephrased as $\neg Q \vee P$. If Z3 solver presents a model satisfying the negation of this formula, then the refinement relation is falsified by a counter example. On the contrary, if there doesn't exist a counter example within a given bound, we say that the refinement relation, $P \sqsubseteq Q$, holds within the

bound. The formal definition of `isRefinementOf` is given in Algorithm 1. The complete definition of the function can be found at [1].

Algorithm 1 cannot produce any spurious counter examples, i.e. all the counter-examples of the bounded model located by the algorithm are also counter-examples of the original model. To illustrate this conclusion, first we assume the bound we set is b, for any channel, the constraints added to Z3 solver are bounded, i.e., only constraints on the prefixes of timed data streams are fed to Z3. Let Σ be the set of all constraints of connectors and σ be the set of bounded ones. If there exists a counter example π (π is a set of timed data streams' prefixes corresponding to the nodes) that fails to satisfy the weaker constraints: $\pi \not\models \sigma$, then we can deduce that $\pi \not\models \Sigma$. Let π be a prefix of the full timed data stream Π and $\Pi \setminus \pi$ satisfies all the left constraints $\Sigma \setminus \sigma$. A valid prefix leads to a valid timed data stream, hence we have $\Pi \not\models \Sigma$. Thus, the counter examples generated by Algorithm 1 are always sound and genuine.

Algorithm 1 Q.isRefinementOf (P, bound)

Require: Both P and Q are connectors.
Ensure: A boolean result: *True* or *False with a counter example*

solver ← create a Z3 Solver instance
nodes ← {}
for *ch* ← channels in Q **do**
  for n ← channel ends in *ch* **do**
    if $n \notin nodes$ **then**
      $tds_n \leftarrow \{time : [n\_t\_0, \cdots, n\_t\_(bound-1)],\ data : [n\_d\_0, \cdots, n\_d\_(bound-1)]\}$
      $nodes[n] \leftarrow tds_n$
      add time constraints $n\_t\_0 \geq 0 \wedge n\_t\_i < n\_t\_i + 1$ to *solver*
    end if
  end for
  add channel-specific constraints to *solver* according to the definitions in Section 5.1
end for

$foralls \leftarrow \{\}$
$absGlobalConstr \leftarrow \{\}$
$absTimeConstr \leftarrow \{\}$
for *ch* ← channels in P **do**
  for n ← channel ends in *ch* **do**
    if $n \notin nodes \cup foralls$ **then**
      $tds_n \leftarrow \{time : [n\_t\_0, \cdots, n\_t\_(bound-1)],\ data : [n\_d\_0, \cdots, n\_d\_(bound-1)]\}$
      $foralls[n] \leftarrow tds_n$
      add time constraints $n\_t\_0 \geq 0 \wedge n\_t\_i < n\_t\_i + 1$ to $absTimeConstr$
    end if
    add channel-specific constraints to $absGlobalConstr$ according to the definitions in Section 5.1
  end for
end for

$absGlobalConstr = \neg\ (absTimeConstr \cap absGlobalConstr)$
let $foralls$ **be** $\{n_1, \cdots, n_m\}$, add the following constraint to *solver*
  $(\forall n_1\_t\_0) \cdots (\forall n_1\_t\_(bound-1))(\forall n_1\_d\_0) \cdots (\forall n_1\_d\_(bound-1)) \cdots$
  $(\forall n_m\_t\_0) \cdots (\forall n_m\_t\_(bound-1))(\forall n_m\_d\_0) \cdots (\forall n_m\_d\_(bound-1))\ absGlobalConstr$

solver.check()

Example 5.1. *We consider two simple connectors* Sync(A,B) *and* Fifo1(A,B)*, where the* Fifo1 *channel is not a refinement of the* Sync *channel. These channels can be constructed with the following python codes.*

```
sync = Connector(); sync.connect('Sync', 'A', 'B')
fifo1 = Connector(); fifo1.connect('Fifo1', 'A', 'B')}
```

After invoking the function `isRefinementOf`*, we can obtain the result of the refinement relation checking* `fifo1.isRefinementOf(sync, 10)` *and the counter example that Z3 solver yields, which are presented as follows. Here we use* `A_d_i` *to denote the* i^{th} *element in the data stream of A and* `A_t_i` *to denote the* i^{th} *element in its time stream.*

```
False
A_d_0 = 0, A_d_1 = 0, A_d_2 = 0, A_d_3 = 0, A_d_4 = 0,
A_d_5 = 0, A_d_6 = 0, A_d_7 = 0, A_d_8 = 0, A_d_9 = 0,
A_t_0 = 0, A_t_1 = 2, A_t_2 = 4, A_t_3 = 6, A_t_4 = 8,
A_t_5 = 10,A_t_6 = 12,A_t_7 = 14,A_t_8 = 16,A_t_9 = 18,
B_d_0 = 0, B_d_1 = 0, B_d_2 = 0, B_d_3 = 0, B_d_4 = 0,
B_d_5 = 0, B_d_6 = 0, B_d_7 = 0, B_d_8 = 0, B_d_9 = 0,
B_t_0 = 1, B_t_1 = 3, B_t_2 = 5, B_t_3 = 7, B_t_4 = 9,
B_t_5 = 11,B_t_6 = 13,B_t_7 = 15,B_t_8 = 17,B_t_9 = 19.
```

Note that the counter example is easy to understand. There exist two corresponding timed data streams whose time satisfies the constraints of FIFO1 *channel, but doesn't satisfy the constraints of* Sync *channel.*

Example 5.2. *Another simple example is also to seek the refinement relation between two basic channels* Sync(A,B) *and* LossySync(A,B)*. The construction process is similar to the above one.*

```
sync = Connector(); sync.connect('Sync', 'A', 'B')
lossy = Connector(); lossy.connect('LossySync', 'A', 'B')
```

We know that Sync *channel is a refinement of* LossySync *channel. When* LossySync *channel behaves well enough to lose no data items, it behaves exactly like* Sync *channel. On the contrary,* LossySync *channel is not a refinement of* Sync *channel, which is consistent with the following result from execution of* `lossy.isRefinementOf(sync, 10)`*. The counter example is also very intelligible. Note that the last timed data item was lost with the previous nine timed data items being transferred successfully, which is consistent with the behavior constraints of* LossySync *channel and obviously not in concordance with the constraints of* Sync *channel.*

```
False
A_d_0 = 0, A_d_1 = 0, A_d_2 = 1, A_d_3 = 2, A_d_4 = 3,
A_d_5 = 4, A_d_6 = 5, A_d_7 = 6, A_d_8 = 7, A_d_9 = 0,
A_t_0 = 0, A_t_1 = 1, A_t_2 = 2, A_t_3 = 3, A_t_4 = 4,
A_t_5 = 5, A_t_6 = 6, A_t_7 = 7, A_t_8 = 8, A_t_9 = 10,
B_d_0 = 0, B_d_1 = 0, B_d_2 = 1, B_d_3 = 2, B_d_4 = 3,
B_d_5 = 4, B_d_6 = 5, B_d_7 = 6, B_d_8 = 7,
B_t_0 = 0, B_t_1 = 1, B_t_2 = 2, B_t_3 = 3, B_t_4 = 4,
B_t_5 = 5, B_t_6 = 6, B_t_7 = 7, B_t_8 = 8, B_t_9 = 9,
```

Before demonstrating complementary refinement checking between large-scale connectors, we will present some experiment results for bounded refinement and equivalence relation checking for the next two examples.

Example 5.3. *In this example, we consider the two connectors in Figure 5 again. In Example 4.3, we have seen the refinement checking between the two connectors in Coq. However, tactics and strategies can be too complex to comprehend or to tackle with. Although Z3 can only provide a bounded refinement relation checking for connectors, it will be much easier to check the refinement relation as we do not need to master concrete theorem proving tactics. All we need to do is to construct the connectors in a way accepted by Z3 solver and use the function* `isRefinementOf` *to check the relation between them.*

```
c1 = Connector()
c1.connect('Sync', 'A', 'E').connect('Fifo1', 'E', 'C').connect('Fifo1', 'E', 'D')

c2 = Connector()
c2.connect('Fifo1', 'A', 'B').connect('Sync', 'B', 'C').connect('Sync', 'B', 'D')
```

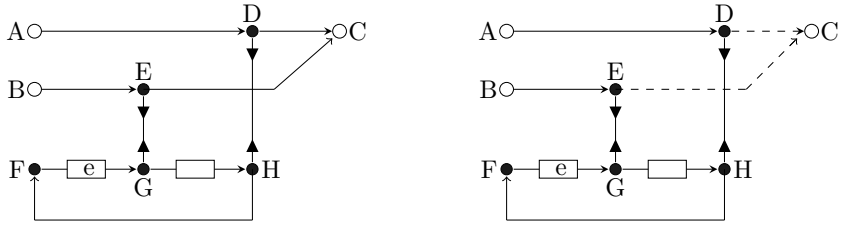
Assume the bound is 10 and then carry out the function `c2.isRefinementOf(c1, 10)`*. Finally, we can get the result* True *which means that* Connector Q *in Figure 5 is indeed a refinement of* Connector P *within bound 10. We can further extend the bound and the time complexity increases linearly as no recursively defined channels are involved.*

Example 5.4. *This example checks the equivalence relation between the two connectors in Figure 7. First we construct the two connectors in the way similar to Example 5.3 and invoke the function* `isRefinementOf` *in both directions with bound 10. The results for both directions that we get are* `True`*. With the yielded result and the soundness theorem, we can conclude that the two connectors are equivalent with each other within the bound.*

(a) Router (b) Variant Router

Figure 8: Example of connector complementary refinement

Example 5.5. *In this example, we will take the well-known* Exclusive Router *into consideration. This connector and one possible variant are shown in Figure 8. The* variant router *is defined with only one major difference: one of the two* LossySync *channels in the Exclusive Router is replaced with a* Sync *channel. Therefore, the behavior of the variant router is actually not exclusive and all data items will be transported to node* E *rather than being transmitted to node* E *or* F *exclusively in the original router. Notice that the variant router is a refinement of the original router and conversely not. We construct these two connectors first and set a concrete bound 10. We invoke the function* `isRefinementOf` *in both directions, i.e.,* `c2.isRefinementOf(c1, 10), c1.isRefinementOf(c2, 10)` *where* `c2` *is the variant router. The results Z3 yields are* `True` *and* `False` *up to bound* 10 *(together with a counter example) respectively, which are consistent with our intuition. The counter example is omitted here and the complete results including the counter example can be found at [1].*



(a) Connector C1 (b)Variant Connector C2

Figure 9: Example of connector complementary refinement

Example 5.6. *Figure 9 shows an example of the application of a two-node sequencer, with the addition of a pair of* Sync *channels and a* SyncDrain *channel connecting each of the nodes of the sequencer to the nodes* A *and* C*, and* B *and* C*, respectively. The leftmost channel of the sequencer is initialized to have a data item in its buffer, as indicated by the presence of the symbol* e *in the box, which is actually a* FIFO1e

channel. As the sequencer ensures that the take operations on nodes G *and* H *can succeed only in the strict left-to-right order, the behavior of the connector* $C1$ *can be seen as imposing an order on the flow of the data items written to* A *and* B*. The timed data stream obtained by successive take operations on* C *consists of the first data item written to* B*, followed by the first data item written to* A*, followed by the second data item written to* B*, and so on. The variant connector* $C2$ *is obtained by replacing the pair of* Sync *channels with a pair of one* Sync *channel and one* LossySync *channel. On account of the behavior of* LossySync *channel, the first data item written to* B *can be lost, which leads to the failure of the sequence order. Thus, the variant connector* $C2$ *is not a refinement of the original connector* $C1$*, but the original connector is indeed a refinement of the variant connector. The details of the concrete construction and the counter example are left out. The results Z3 solver presented are* `False` *for* `c2.isRefinementOf(c1, 10)` *and* `True` *for* `c1.isRefinementOf(c2, 10)`*, which are consistent with the results of theoretical deduction.*

Comparing the behavior of connector $C1$ *described in this example with the* alternator *in Section 4, the data stream obtained on* C *is made up of the data items written to* B *and* A *alternately for both connectors. Intuitively, the two connectors have the same behavior and thus are equivalent. Nevertheless, by constructing the connectors and executing the function* `isRefinementOf` *in both directions, the returned results are both* `False`*, which is contrary to our expectation. In fact, the reason for this result is that the constraints on the input nodes of the two connectors are different.* SyncDrain *channel of* alternator *ensures that the input time on nodes* A *and* B *are equal. However, the application of* Sequencer *guarantees that the time for the two input nodes must be different. We can see that Z3 helps to avoid possible artificial mistakes by this example, which makes the advantage of Z3 more evident.*

5.4. Z3 as complement of Coq

In the previous assessments of the refinement and equivalence relations or other properties in Coq, the main constraint is that users need to be able to build a proof. Furthermore, various behavioral properties or relations on infinite timed data streams can be proven, which makes Coq sufficiently powerful. Besides, the connectors whose properties have been proved can be regarded as theorems or lemmas to assist in proving properties of much larger ones. Recall that the main reason of introducing Z3 solver is that Coq cannot provide the proof of complementary refinement checking, i.e., one connector is not a refinement of the other one, or provide counter examples automatically. Coq users need to build the counter example manually to show that the property is not satisfiable.

As we have seen, Z3 solver has shone light on complementary refinement checking. Besides, Z3 solver is much easier to use after the definition of the function `isRefinementOf` and the construction framework of Reo connectors, compared with using specific tactics and strategies in Coq. Moreover, counter examples can be provided as additional diagnostic feedback while the refinement checking process returns `False`. The main drawback of Z3 solver is the failure of providing ideally infinite timed data streams as witnesses for refinement relations. A finite prefix of timed data stream is not enough to ensure the refinement relation between two Reo connectors for certain. As a result, the equivalence relation is also not ensured completely. Nevertheless, it has no influence on the complementary refinement checking. Once Z3 presents a counter example, the complementary refinement checking is finished, which is ensured by the aforementioned soundness of counter example generation using Algorithm 1. However, we have shown that a bounded counter example is sufficient to prove the inexistence of refinement/equivalence relations and the bound in our experiments is usually less than 10. When checking complementary refinement relation, the experiments show that a bound limit around 5 can guarantee a proper counter example to be given in a reasonable time for many cases.

6. Conclusion and Future Work

In this paper, we present a new approach to model and reason about connectors in Coq and Z3. The model naturally preserves the original structure of connectors. This also makes the connector description reasonably readable. We implement the proof of properties for connectors using identified techniques and tactics provided by Coq. Properties are defined in terms of predicates which provide an appropriate description of the relation among different timed data streams on the nodes of a connector. All the analysis

and verification work are based on the logical framework where basic channels are viewed as axioms and composition operations are viewed as operators. As we can address the relation among different timed data streams, we can reason about various properties including equivalence and refinement relations for connectors. Furthermore, since Coq cannot decide if a proposition is provable or not, we can also use Z3 to search for possible bounded counter examples automatically. Although in this paper we focus on refinement checking as an example, this approach can be applied to verification of various properties.

As some of the benefits of this approach are inherited from Coq, our approach has got some of its drawbacks as well. The main limitation is that the analysis needs much more tactics and techniques when the constructor becomes large. In the future work, we plan to enhance our framework by two different approaches. Firstly, we are working on automating the proofs of Reo connector properties. In [23] (which is an extended version of [22]), we developed an approach to generate Coq tactics automatically based on neural networks, which also works on all the proved theorems in this paper. We are looking for more efficient network structures and algorithms to improve the performance of this approach. Secondly, more attention is needed to precisely evaluate how expressive this approach is for modeling temporal properties.

Acknowledgement

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

References

[1] Package of source files. `https://github.com/liyi-david/reo-z3`.

[2] B. K. Aichernig, F. Arbab, L. Astefanoaei, F. S. de Boer, M. Sun, and J. Rutten. Fault-based Test Case Generation for Component Connectors. In *Proceedings of TASE 2009*, pages 147–154. IEEE Computer Society, 2009.

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

[4] 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, 2003.

[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] 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] L. M. de Moura and N. Bjørner. Z3: an Efficient SMT Solver. In *Proceedings of TACAS 2008*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008.

[9] W.-P. de Roever and K. Engelhardt. *Data Refinement: Model-Oriented Proof Methods and their Comparison*. Cambridge University Press, 1998.

[10] G. Huet, G. Kahn, and C. Paulin-Mohring. The Coq Proof Assistant A Tutorial. *Rapport Technique*, 178, 1997.

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

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

[13] 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.

[14] S. Klüppelholz and C. Baier. Symbolic Model Checking for Channel-based Component Connectors. *Science of Computer Programming*, 74(9):688–701, 2009.

[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] A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice Hall, 1998.

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

[19] 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.

[20] 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.

[21] 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.

[22] 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, volumn 10729 of *LNCS*, pages 558-573. Springer, 2018.

[23] Y. Li, X. Zhang, W. Hong, S. Nawaz and M. Sun. Using Coq and Recurrent Neural Network to Model and Verify Timed Connectors. submitted to *Science of Computer Programming*.