Formal Modeling and Verification of Timed Connectors in IoT with Z3

Ziyun Xu and Meng Sun*
School of Mathematical Sciences, Peking University, Beijing, China
{xuziyun, sunm}@pku.edu.cn

***Abstract*—The Internet of Things (IoT) is rapidly advancing and reshaping the whole world. Coordination models and languages, like Reo and Orc, provide connectors that interconnect components in IoT applications and organize their interactions in distributed environments. In this paper, we propose a method for formally modeling and verifying the properties of timed connectors using Z3, an SMT (Satisfiability Modulo Theories) solver. We use Z3 Python-bindings to construct the models and carry out experiments. The formal model in Z3 clearly reflects the original structure of connectors. With the definition in Z3, we can automatically verify time-related properties of connectors, and automatically construct counter-examples when the properties do not hold.**

***Index Terms*—Coordination language, Reo, SMT, timed connector**

I. Introduction

The rapid advancement and ubiquitous penetration of the IoT paradigm is rapidly reshaping the industries and the whole world. Mobile applications, web-based information systems, and software defined networking systems are distributed over large networks of computing devices. The software components that make up such IoT systems often do not fit together perfectly, but leave significant interface gaps that should be filled with additional “glue code”. Coordination models and languages, such as Reo [2] and Orc [13], provide mechanisms to interconnect such constituent components with connectors / orchestrators and organize their interactions in a distributed processing environment.

Reo is a channel-based coordination language in which connectors are the core concept. Connectors are compositionally constructed from channels and capture the protocols for organizing and controlling cooperation, synchronization, communication and exclusion between their interconnected components. Recently, Reo has been successfully applied in multiple areas such as business processes [22], software-defined networks [10], bioinformatics [8], quantum internet software [5], and web services [16].

How to guarantee the correctness and trustworthiness of timed connectors is a challenging and critical problem due to the evolution of software systems and advancements in IoT technologies. The structures of connectors are becoming more and more complicated because of the rapid increase in size and complexity of IoT applications, which makes it more difficult to analyze and verify connector properties.

* Corresponding author.

In literature, there are some works for the formal modeling and verification of connectors, such as the operational semantics of Reo defined with constraint automata (CA) in [7], and the symbolic model checker Vereofy [6] which can be used to check CTL-like properties. Reo can also be coverted to other formal models, such as mCRL2 [14] and Alloy [12], making it possible to utilize already existing verification tools. Recently, an attractive approach is to encode Reo connectors in theorem provers like Coq [11], [21] and PVS [18], [19], based on the UTP semantics for Reo [1], [17], and make reasoning of connector properties in the theorem provers. The basic idea of this 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 verify properties of connectors by using proof assistant like Coq and PVS.

This work is an extension of the earlier results in [11], [21]. A family of timed channels and connectors has been provided in [3], [15], which can be used to measure the exact time elapsed between two events at input/output nodes and specify timed behavior happening in coordination. The theorem prover Coq is used in [11] to model and reason about the properties of timed connectors. In [21], the authors showed that the SMT solver Z3 [9] can be used to formally model and verify the properties of untimed connectors, and automatically construct bounded counterexamples when the properties do not hold. In this paper we extend the method used in [21]. We build the formal models in Z3 for timed channels and connectors and reconstruct those for untimed channels and connectors to make them consistent with the timed ones, and use the SMT solving approach to formally verify properties of timed connectors.

This paper is organized as follows. After the general introduction in Section I, we briefly summarize Reo and Z3 in Section II. The formal modeling of basic channels and compositional operators in Z3 are presented in Section III. Section IV shows how to reason about connector properties in our framework. Finally, Section V concludes the paper and discusses some future research directions. [20] provides the complete source code for further reference.

II. Preliminaries

We provide a brief introduction to the coordination language Reo and Z3 here.

979-8-3503-2759-5/23/$31.00 ©2023 IEEE
DOI 10.1109/CSCE60160.2023.00410

A. The coordination language Reo

Reo is a channel-based exogenous coordination language in which complex coordinators, called connectors, are composed of simpler ones [2]. More details about Reo can be found in [2], [4], [7].

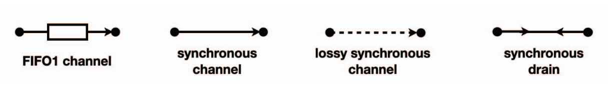

Figure 1. Some basic channels in Reo

The simplest connectors are *channels*, such as synchronous channels, FIFO1 channels, and so on. Each channel has two ends which are divided into two types: source ends and sink ends. The source end accepts data into the channel, while the sink end emits the data out of the channel. Figure 1 shows the graphical notations of some basic channels. Their behavior can be explained as follows:

- **Sync**: a synchronous channel with one source and one sink end. The pair of I/O operations on both ends can succeed only if the writing operation at source end is synchronized with the read operation at its sink end.
- **SyncDrain**: a synchronous channel with two source ends. The pair of input operations at both ends can only succeed at the same time, and the data items written to the channel are lost.
- **FIFOn**: an asynchronous channel with one source end and one sink end, and a bounded buffer cell of capacity n. It can accept n data items from its source end before its sink end emits data. The data items are kept in an internal buffer, and allocated to the sink end in FIFO order. In particular, the FIFO1 channel is an instance of FIFOn with a buffer size of 1.
- **LossySync**: a synchronous channel with one source end and one sink end. The source end always accepts all data items. The written data is lost immediately if no corresponding output operation is available at its sink end; otherwise, the channel transmits the data item as a Sync channel, and the output operation at the sink end succeeds.

Different from the primitive untimed channels mentioned above, a family of timed channels are defined as well to measure the time between two events and produce timeout signals.

- ***t*-Timer**: the basic *t-timer* channel which accepts any input value through its source end A and returns a timeout signal on its sink end B exactly after a delay of t time units.
- **OFF-*t*-Timer**: a *t-timer* with the *off*-option which allows the timer to be stopped before the expiration of its delay t is designed in case users require the timer to stop working as soon as possible. Under this circumstance, a special *off* value whose type is also assumed to be $Data$ can be consumed through the source end.
- **RST-*t*-Timer**: a *reset*-option allows the timer to be reset to 0 at once as the values in the stream remain unchanged. A *t-timer* channel with the *reset*-option is activated as soon as a special *reset* value is consumed through its source end.
- **EXP-*t*-Timer**: a *t-timer* channel with *expire*-option makes the timer produce its timeout signal through its sink end and reset itself instantaneously once it consumes a special *expire* value through its source end.

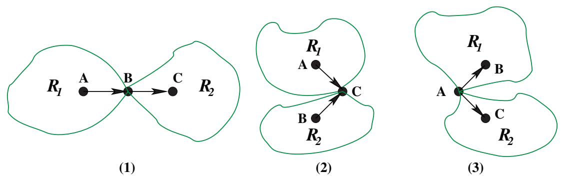

Figure 2. Operations of channel composition

In Reo, complex connectors are built by combining simpler connectors on the channel ends. The nodes that connect channels are divided into source nodes, sink nodes, and mixed nodes according to whether all the channel ends that overlap on the node are source ends, sink ends, or a combination of the two. Correspondingly, there are three types of channel composition operations: flow-through, merge and replicate. Figure 2 provides a graphical representation of these operations.

- *flow-through*: this operator which acts on mixed nodes simply allows data items to flow from one channel to the other through the junction node. Therefore, we do not need to give a specific definition of the flow-through operator because it can be implemented implicitly.
- *replicate*: this operator replicates the data items on a node and transfers them through channels which are connected to the node. Similar to the *flow-through* operator, the *replicate* operator can also be achieved implicitly.
- *merge*: when the *merge* operator acts on two channels, AB and CD, it causes the data items to be taken from either AB or CD. The two timed data sequences are merged together into one single timed data sequence where the order of elements in the sequence is decided by the time moments.

B. The Z3 SMT solver

Z3 [9] is an efficient SMT (Satisfiability Modulo Theories) solver for a variety of software verification and analysis applications. It extends to determining the satisfiability (or dually the validity) of first order formulas. Given the connectors' data and time constraints, Z3 can be used to verify the satisfiability of connector properties. Z3 provides bindings for several programming languages. In this work, we use Z3 python-bindings to build models and conduct experiments.

The following example gives an intuitive understanding of the Z3 solver. In this example, we define that both x and y are real numbers. The first two constraints shows that in addition to boolean operators like And, Or, Not, and

$Implies$, Z3 also supports operators such as $<$ and $>$ for comparison. The function `Solver` creates a solver instance in which constraints can be asserted via the `add` method. The command `check` evaluates the satisfiability of the asserted constraints. The solver returns sat (unsat) if the result is satisfiable (unsatisfiable) respectively. The solver may fail to check the satisfiability of the constraint system and return $unknown$. Each solver maintains a stack of assertions. The method `push` creates a new scope. The method `pop` removes all constraints asserted after the last `push` method.

```
from z3 import *
x, y = Real('x'), Real('y')

s = Solver()
s.add(x > 10, Or(x + y > 3, x - y < 2))
s.add(ForAll([y],y>0))
print(s.check())
print(s.model())

s.push()
s.add(y < 11)
print(s.check())
print(s.model())

s.pop()
print(s.check())
print(s.model())
```

III. Formal Modeling of Channels and Operators

In this section, we show how both primitive untimed and timed channels, as well as operators for connector composition, are specified in Z3 and used for modeling complex connectors. Then in Section IV we use Z3 to automate the verification of connectors' properties, or produce a bounded counterexample when the properties do not hold.

We implement the construction of connectors in a way that fits Z3. Typically, connector construction starts with the formal definitions of primitive channels, timed channels, and composition operators, which are used hereafter to build and assess more complex models. Such a framework must be carefully developed so that further reasoning and verification can be simplified as much as possible.

We first define an auxiliary function called $Conjunction$, which is used to take the conjunction of every constraint in the constraint lists parameterized by *constraints*.

```
def Conjunction(constraints):
  assert len(constraints) > 0
  result = None
  for c in constraints:
    if result is None:
      result = c
    else:
      result = And(result, c)

  return result
```

A. Formal Modeling of Primitive Channels

We use the Z3 implementation of the primitive untimed channels which has been developed in [21] and adapt it to the new framework.

As a starting example, we consider the *Sync* channel. The constraints for *Sync* channel are classified into two types: data constraints and time constraints. The definition of *Sync* provides the behavior pattern it needs to follow: each item in the timed data streams of output node specified as "node[1]" is equivalent to the timed data items of input node specified as "node[0]", which are exactly data and time constraints. Each constraint in the list will be combined finally.

```
def Sync(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] ]
  return Conjunction(constraints)
```

Similar to the *Sync* channel, the *SyncDrain* channel is defined such that only time related constraints are needed, i.e., equivalence of corresponding items in two time streams of the two inputs.

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

For *FIFO1* channel, data related constraints are same as for the *Sync* channel, while time related constraints are different. Since 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. The data item 'e' in the buffer should be the first one to transit and all the differences on the constraints result from it.

```
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
```

For the *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; otherwise it behaves exactly like a successful transit of *Sync* channel, and the data and time related constraints are identical with those in the definition of *Sync*. What is special is that *LossySync* channel is defined in a recursive way according to its own distinctive behavior. We can see that the constraints are added recursively which coincide 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)))
```

B. Formal Modeling of Timed Channels

While the primitive channels are already well defined as part of the basis of the whole framework, the timed channels also need to be specified properly in order to measure the time elapse between two events and produce timeout signals, so that we can refine the framework and address a wider range of coordination issues in IoT applications. In the following, we describe the modeling of timed channels in Z3. This approach also facilitates the analysis and proof of timed connector properties, since we can easily use Z3 to automatically verify the properties of a timed connector within certain bounds, and obtain counterexamples automatically when the properties do not hold.

We first define the signals that various types of timed channels will generate in Z3, specifically, the *timeout* signal, *off* signal, *reset* signal and *expire* signal.

```
timeout = Int('timeout')
off = Int('off')
reset = Int('reset')
expire = Int('expire')
```

Using these definitions, we can model timed channels in a way similar to primitive channels.

For the ***t*-Timer** channel, the first constraint in the code is about time and the second constraint is about data. In this specification, the first constraint describes the relation of the input and output streams in the time dimension. The second constraint presents that after a delay of t time units for every data item it receives at the source end, a *timeout* signal will be generated at the sink end. The third constraint is an inequality, requiring that an input data item cannot be accepted by the channel when there is no *timeout* signal for the previous data item it receives. This means that there should be no other input actions during the delay of t time units.

```
def Timert(t):
  def TimertInstance(nodes,bound):
    assert len(nodes) == 2
    constraints = []
    for i in range(bound):
      constraints += [nodes[0]['time'][i] + t
          == nodes[1]['time'][i]]
      constraints += [nodes[1]['data'][i] ==
          timeout]
      if i != 0:
        constraints += [nodes[1]['time'][i-1] <=
            nodes[0]['time'][i]]
    return Conjunction(constraints)
  return TimertInstance
```

The **OFF-*t*-Timer** channel needs to be defined in a recursive way similar to the modelling of *LossySync* channel. First, we deal with some edge cases, such as when the bound is very small or when the constraints under consideration are close to the bound. The general situation is the constraints defined in the 'else' branch. *constraints_0* represents one situation where the *off* signal is the data element to be accepted. In this case, the timer is stopped and we remove the data *off* and the current data without any output. Then a new input stream is resumed. *constraints_1* represents another situation that complements *constraints_0*, where the data element to be accepted is not *off*, and a *timeout* signal is produced as output after a delay of t time units. Then a new input stream is resumed. The general idea is that there are some requirements on the inter-arrival time of the inputs that depends on the value of incoming data, while different actions are taken for the output in order to deal with the data element to be accepted. The definitions of the **RST-*t*-Timer** and the **EXP-*t*-Timer** in Z3 are similar to the **OFF-*t*-Timer**. The input signals change from the *off* signal to *reset* and *expire*, and trigger subtly different channel behaviors respectively. In the following we only give the detailed code of the **OFF-*t*-Timer** in Z3 while the code for other timed channels are omitted due to the length limitation and can be found in [20].

```
def OFFTimert(t):
  def OFFTimertInstance(nodes, bound, idx = 0,
       num = 0):
    assert len(nodes) == 2
    if idx == bound:
      return True
    if num == bound:
      return True
    if idx == bound-1:
      constraints_last = []
      constraints_last += [nodes[0]['time'][idx
         ] + t == nodes[1]['time'][num]]
      constraints_last += [nodes[1]['data'][num
         ] == timeout]
      return Conjunction(constraints_last)
    else:
      constraints_0 = []
      constraints_1 = []
      constraints_0 += [nodes[0]['data'][idx+1]
         == off]
      constraints_0 += [nodes[0]['time'][idx+1]
         < nodes[0]['time'][idx] + t]
      constraints_1 += [nodes[0]['data'][idx+1]
         != off]
      constraints_1 += [nodes[0]['time'][idx] +
         t == nodes[1]['time'][num]]
      constraints_1 += [nodes[1]['data'][num]
         == timeout]
      if idx == 0 and num == 0:
        constraints_all = []
        for i in range(bound-1):
          constraints_all += [ Or(nodes[0]['data
             '][i+1] == off ,
          nodes[0]['time'][i+1] >= nodes[0]['
             time'][i] + t )]
        return And(Or(And(Conjunction(
            constraints_0),
          Channel.OFFTimert(t)(nodes, bound, idx
             + 2, num)),
          And(Conjunction(constraints_1),
          Channel.OFFTimert(t)(nodes, bound, idx
             + 1, num + 1))),
          Conjunction(constraints_all))
      else:
        return Or(And(Conjunction(constraints_0)
           ,
          Channel.OFFTimert(t)(nodes, bound, idx
             + 2, num)),
          And(Conjunction(constraints_1),
          Channel.OFFTimert(t)(nodes, bound, idx
             + 1, num + 1)))
  return OFFTimertInstance
```

Specifying timed channels by Z3 makes the model intuitive and concise as each constraint describes a simple relation on time or data. Furthermore, we can easily extend the existing model to new user-defined channels.

C. Formal Modeling of Operators

When constructing more complex connectors, it is necessary to apply composition operators on channels and simpler ones according to the topological structures of the connectors. As mentioned earlier, both *flow-through* and *replicate* can be achieved implicitly using same node names when we compose channels to construct connectors. Therefore no additional Z3 code is needed for them.

However, the implementation of the *merge* operator in Z3 is more complicated. In [21], it is specially dealt with as a special channel *Merger* with three channel ends. Together with the other two kinds of composition operators: *flow-through* and *replicate*, they provide a complete basis for the construction of any kinds of connectors, coinciding with the original set of composition operations [2], [17].

The *Merger* channel also has a recursive definition like the *LossySync* channel. The two input nodes are specified as “node[0]” and “node[1]” respectively, while the output node is specified as "node[2]". For each item in the output node, there is a non-deterministic choice between the two input nodes. If the time corresponding to the input data of "node[0]" is earlier than time corresponding to the input data of "node[1]", which is the case in *constraints_1*, then the first data and time elements of "node[2]" are equal to the elements of "node[0]" and vice versa. We use recursive calls to deal with these two different cases.

```
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)))
```

IV. Reasoning about Timed Connectors

With the formal modeling of primitive untimed and timed channels as well as composition operators in Z3, we can easily model a timed connector and verify its properties. In this section, we first show how to build a connector on the existing basis, and how to define functions and methods about its time-related properties, followed by two examples to show how to verify timed connector properties in our framework.

Users build the connectors to be verified with the *connect* method in the class **Connector**, then other methods are applied to automatically verify the properties in concern. Note that the attribute *channels* is a list whose items are tuples with the first element as the type of the channel and the second and third elements being the names of the source and sink ends respectively. In particular, if the first element of a tuple is *Merger*, then the other three elements are the names of the three channel ends.

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

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

In the following, we introduce methods to verify time-related properties of connectors, such as *Deq*, *Teq*, *Tneq*, *Tlt*, *Tgt*, *Teqt*, *Tltt*, *Tgtt*, *Tlet*, and *Tget*. These methods use two functions as auxiliary functions: *AllNodes* and *AllConstraints*. Calling *AllNodes* on an instance of **Connector** class returns a list of all names of the nodes in the connector. As for *AllConstraints*, it traverses each channel in the connector and combines all time constraints and data constraints to return a conjunctive total constraint. Note that not only constraints for channels, but also temporal order constraints on each node need to be considered.

With the two auxiliary functions, it is very easy to specify the methods for verification of time-related properties. For example, we consider the *Deq* function which can be used to express that the datas of the two nodes are exactly the same within the given bound. For robustness, we first verify that the two nodes in the parameters are indeed nodes in this connector by using function *AllNodes*. The parameter *constr* combines the constraints for the equality between the datas on the two nodes within the bound. It turns out that, this property of the connector is correct if and only if *constr* can be derived from the return value of *AllConstraints*. To verify that a formula is valid, we can dually verify that the negation of the formula is unsatisfiable, in which case the call to Z3 produces *unsat*.

The judgment about equality of time being defined as *Teq* is analogous to the judgment of data. *Teq* means that the time components of two streams taken as parameters are equal. It behaves almost same as *Deq*, only to change *constr* from data constraints to time constraints. And *Tneq* has the opposite meaning. *Tlt* means that each time dimension of the first stream is strictly less than the other stream while *Tgt* means that every time dimension of the first stream is greater than the other stream. The design of all these functions are similar to that of *Deq*. Due to the length limitation we omit the Z3 code for these functions here, which can be found at [20].

There are three more definitions that serve to facilitate the modeling of timed channels. All of them are plain and easy to understand with one of the time streams is added by a t time delay. An extra t is appended to the names of these new functions about judgment of time to distinguish them from the original ones.

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

We give the modeling of *Teqt* here as an example.

```
def Teqt(self, bound, time, *nodes):
  assert len(nodes) == 2
  nd_0 = nodes[0]
  nd_1 = nodes[1]
  allnodes = self.AllNodes()
  assert nd_0 in allnodes
  assert nd_1 in allnodes
  constr = True
  for i in range(bound):
    constr = And(constr,Real(nd_0 + '_t_' + str
        (i)) + time == Real(nd_1 + '_t_' + str(
        i)))

  solver = Solver()
  solver.add(Not(Implies(self.AllConstraints(
      bound), constr)))
  result = solver.check()
```

In the following, we use some examples to illustrate our approach instead of giving all the complex technical details.

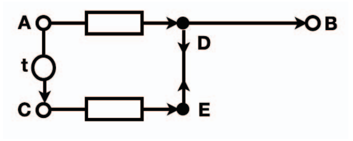

Figure 3. Lower bounded FIFO1 channel

Example 4.1: Take the connector in Figure 3 into consideration. In this connector, node A is a source node, whereas C, D and E are mixed nodes and B is a sink node. This connector consists of five channels AC, AD, CE, DE and DB with channel types ***t*-Timer**, **FIFO1**, **FIFO1**, **SyncDrain** and **Sync** respectively. We can automatically prove that $Deq(100,'A','B')$ and $Tltt(100, 20,'A','B')$, where 100 is the bound and 20 can be changed into any given real number.

The code for the lower bounded FIFO1 and the corresponding proof is as follows.

```
c1 = Connector()

c1.connect('Timert(20)','A','C')
c1.connect('Fifo1','A','D')
c1.connect('Fifo1','C','E')
c1.connect('SyncDrain','D','E')
c1.connect('Sync','D','B')

result1, counterexample1, smt1 = c1.Deq(100,'A
    ','B')
print(result1)
print(counterexample1)
result2, counterexample2, smt2 = c1.Tltt
    (100,20,'A','B')
print(result2)
print(counterexample2)
```

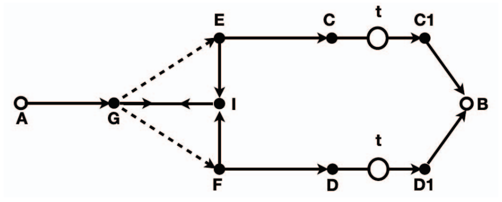

Figure 4. $2 \times t$ timed connector

Example 4.2: Figure 4 shows the topology structure of the **2 × t Timed Channel** defined in [11], where A is a source node, B is a sink node and all other nodes are mixed nodes.

The code for the $2 \times t$ timed connector in Figure 4 and the corresponding proof are as follows. Here we try to automatically prove the theorem stated in [11] within bound 10 with the help of Z3. The advantage of our approach is that we do not need to semi-automatically disassemble the theorem and prove it step by step as in Coq [11], but can directly verify its correctness with a machine automatically and produce a counterexample when the property does not hold.

```
c1 = Connector()

c1.connect('Sync','A','G')
c1.connect('LossySync','G','E')
c1.connect('LossySync','G','F')
c1.connect('Sync','E','I')
c1.connect('Sync','F','I')
c1.connect('SyncDrain','G','I')
c1.connect('Merger','E','F','I')
c1.connect('Sync','E','C')
c1.connect('Sync','F','D')
c1.connect('Timert(20)','C','C1')
c1.connect('Timert(20)','D','D1')
c1.connect('Merger','C1','D1','B')

result, counterexample, smt = c1.Teqt(10,20,'A
    ','B')
print(result)
print(counterexample)
```

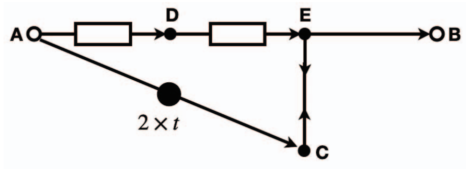

Figure 5. Timed FIFO2 Connector

Example 4.3: Figure 5 presents a useful timed connector, called **Timed FIFOn Connector**, which is defined in [15] and is widely used in modeling real-time networks. The function of this connector is to delay every input for t time units, even if the inter-arrival time of the inputs is less than t (for up to n such inputs). The number of FIFO1 channels between nodes A and E should be equal to n in the $n \times t$ Timed Channel. We take Timed FIFO2 Connector as an example here. It contains a $2 \times t$ Timed Channel and 2 FIFO1 channels. We try to automatically prove that $Deq(10,'A','B')$ and $Teqt(10,20,'A','B')$ with the bound being 10. Here 20 can be changed into any given real number.

The code is as follows.

```
c1 = Connector()
c1.connect('Fifo1','A','D')
c1.connect('Fifo1','D','E')
c1.connect('Sync','E','B')
c1.connect('SyncDrain','E','C')
c1.connect('Sync','A','G0')
c1.connect('LossySync','G0','E0')
c1.connect('LossySync','G0','F0')
c1.connect('Sync','E0','I0')
c1.connect('Sync','F0','I0')
c1.connect('SyncDrain','G0','I0')
c1.connect('Merger','E0','F0','I0')
c1.connect('Sync','E0','C0')
c1.connect('Sync','F0','D0')
c1.connect('Timert(20)','C0','C10')
c1.connect('Timert(20)','D0','D10')
c1.connect('Merger','C10','D10','C')

result1, counterexample1, smt1 = c1.Teqt
    (10,20,'A','B')
result2, counterexample2, smt2 = c1.Deq(10,'A'
    ,'B')
print(result1)
print(counterexample1)
print(result2)
print(counterexample2)
```

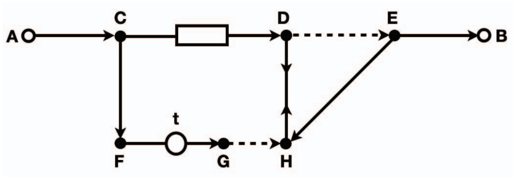

Figure 6. Expiring FIFO1 Channel

Example 4.4: **Expiring FIFO1 channel** [15] is an extension of FIFO1 which behaves just like FIFO1 except that the data item is lost if it is not taken out of the buffer through the sink end of the channel within t time units after it enters through the source end. Figure 6 illustrates how to construct expiring FIFO1 from FIFO1 channel and a ***t*-Timer**.

According to the construction of expiring FIFO1 channel, it is easy to find that the time of the stream at node A with an addition of t is less than or equal to the time of the stream at node B. We can automatically prove this using $Tlet(15, 20,'A','B')$ with the bound 15. Here 20 can be changed into any real number.

Meanwhile, we can prove that $Deq(15,'A','B')$ with the bound 15, which is in fact guaranteed by the $D - E - H - D$ loop. Since the *SyncDrain* channel only allows the data streams at both ends to be equal in time, when the *LossySync* channel DE loses data, the constraints of DH channel cannot be satisfied. Thus it is only possible for DE channel to work like *Sync*.

The code is as follows.

```
c1 = Connector()
c1.connect('Sync','A','C')
c1.connect('Fifo1','C','D')
c1.connect('Sync','C','F')
c1.connect('Timert(20)','F','G')
c1.connect('LossySync','G','H')
c1.connect('LossySync','D','E')
c1.connect('SyncDrain','D','H')
c1.connect('Sync','E','H')
c1.connect('Sync','E','B')

result1, counterexample1, smt1 = c1.Tlet
    (15,20,'A','B')
result2, counterexample2, smt2 = c1.Deq(15,'A'
    ,'B')
print(result1)
print(counterexample1)
print(result2)
print(counterexample2)
```

V. Conclusion

In this paper we present an approach for formal modeling of timed connectors and reasoning about timed connector properties in Z3. The model naturally preserves the original structure of timed connectors, which also makes the connector description reasonably readable. All the analysis and verification work are based on the definitions of primitive untimed and timed channels. By the various functions defined in class **Connector**, we can easily reason about temporal properties for connectors. Some of the benefits of this approach are inherited from Z3, especially that the verification work and the search for possible bounded counterexamples can be done automatically. Compared with other techniques (like theorem proving in Coq or PVS), such automatic verification methods may help us avoid tons of hands-written proofs. Moreover, counterexamples can be provided as additional diagnostic feedback while the property is not satisfied. Although here we only focus on temporal properties checking, this approach can be applied to verification of various properties.

Besides its benefits, this approach has some drawbacks as well. The main limitation of Z3 solver is the failure of providing ideally infinite timed data streams as witnesses for connector properties. It is only possible to prove a property within a given finite bound. A finite prefix of timed data stream is not enough to ensure certain properties. Another flaw is that when the given connector gets more complex and the given bound becomes larger (such as the connector in Example 2 with the bound being 15), the running time increases rapidly. In the future, we plan to extend our approach to a wider range of properties which can improve the existing connector properties verification framework, and try to speed up the verification process within a given bound.

Acknowledgement

This research was sponsored by the National Natural Science Foundation of China under Grant No. 62172019 and CCF-Huawei Populus Grove Fund.

References

[1] B. K. Aichernig, F. Arbab, L. Astefanoaei, F. S. de Boer, S. Meng, and J. J. M. M. 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. *Math. Struct. Comput. Sci.*, 14(3):329–366, 2004.

[3] F. Arbab, C. Baier, F. S. de Boer, and J. J. M. M. Rutten. Models and temporal logics for timed component connectors. In *Proceedings of SEFM 2004*, pages 198–207. IEEE Computer Society, 2004.

[4] F. Arbab and J. J. M. M. Rutten. A coinductive calculus of component connectors. In *WADT 2002, Revised Selected Papers*, volume 2755 of *LNCS*, pages 34–55. Springer, 2002.

[5] E. Ardeshir-Larijani and F. Arbab. Reo coordination model for simulation of quantum internet software. In *STAF 2018 Collocated Workshops, Revised Selected Papers*, volume 11176 of *LNCS*, pages 311–319. Springer, 2018.

[6] 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.

[7] C. Baier, M. Sirjani, F. Arbab, and J. J. M. M. Rutten. Modeling component connectors in reo by constraint automata. *Sci. Comput. Program.*, 61(2):75–113, 2006.

[8] D. Clarke, D. Costa, and F. Arbab. Modelling coordination in biological systems. In *ISoLA 2004, Revised Selected Papers*, volume 4313 of *LNCS*, pages 9–25. Springer, 2004.

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

[10] H. Feng, F. Arbab, and M. M. Bonsangue. A reo model of software defined networks. In *Proceedings of ICFEM 2019*, volume 11852 of *LNCS*, pages 69–85. Springer, 2019.

[11] W. Hong, M. S. Nawaz, X. Zhang, Y. Li, and M. Sun. Using coq for formal modeling and verification of timed connectors. In *SEFM 2017 Collocated Workshops, Revised Selected Papers*, volume 10729 of *LNCS*, pages 558–573. Springer, 2017.

[12] 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.

[13] D. Kitchin, A. Quark, W. R. Cook, and J. Misra. The Orc programming language. In *Proceedings of FMOODS/FORTE 2009*, volume 5522 of *LNCS*, pages 1–25. Springer, 2009.

[14] N. Kokash, C. Krause, and E. P. de Vink. Reo + mcrl2: A framework for model-checking dataflow in service compositions. *Formal Aspects Comput.*, 24(2):187–216, 2012.

[15] S. Meng. Connectors as designs: The time dimension. In *Proceedings of TASE 2012*, pages 201–208. IEEE Computer Society, 2012.

[16] S. Meng and F. Arbab. Web services choreography and orchestration in reo and constraint automata. In *Proceedings of SAC 2007*, pages 346–353. ACM, 2007.

[17] S. Meng, F. Arbab, B. K. Aichernig, L. Astefanoaei, F. S. de Boer, and J. J. M. M. Rutten. Connectors as designs: Modeling, refinement and test case generation. *Sci. Comput. Program.*, 77(7-8):799–822, 2012.

[18] M. S. Nawaz and M. Sun. Reo2PVS: Formal Specification and Verification of Component Connectors. In *Proceedings of SEKE 2018*, pages 391–390. KSI Research Inc. and Knowledge Systems Institute Graduate School, 2018.

[19] M. S. Nawaz and M. Sun. Using PVS for modeling and verification of probabilistic connectors. In *FSEN 2019, Revised Selected Papers*, volume 11761 of *LNCS*, pages 61–76. Springer, 2019.

[20] Package of source files. https://github.com/Ziyun-Xu/reo-z3.

[21] X. Zhang, W. Hong, Y. Li, and M. Sun. Reasoning about connectors using coq and Z3. *Sci. Comput. Program.*, 170:27–44, 2019.

[22] Z. Zlatev, N. K. Diakov, and S. Pokraev. Construction of negotiation protocols for e-commerce applications. *SIGecom Exch.*, 5(2):12–22, 2004.