SMT-Based Modeling and Verification of Cloud Applications

Xiyue Zhang and Meng Sun(✉)

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

Abstract. Cloud applications have been rapidly evolving and gained more and more attention in the past decade. Formal modeling and verification of cloud services are necessarily needed to guarantee their correctness and reliability of complex cloud applications. In this paper, we present a formal framework for modeling and verification of cloud applications based on the SMT solver Z3. Simple cloud services are specified as the basis for the modeling of composition and more complex cloud services. Three different classes *Service*, *Composition* and *Cloud* indicating simple cloud services, composition patterns and composed cloud services are defined, which facilitates the further development of attributes and methods. We also propose an approach to check the refinement and equivalence relations between cloud services, in which counter examples can be automatically generated when the relation is not valid.

Keywords: Cloud applications · Services · Z3 · Modeling · Verification

1 Introduction

With the rapid development of big data and cloud computing technologies, cloud applications spring up quickly in the past decade. The increasing complexity of modern cloud applications has changed the perspective of software designers who now have to consider large-scale applications consisting of a colossal number of services and featuring complex interaction mechanisms. Nowadays, cloud applications are usually distributed, heterogeneous, decentralized and safety-critical, having complex concurrent behavior and are operating in unpredictable environments, and it is notoriously difficult to guarantee their trustworthy. Therefore, formal verification of cloud applications becomes the focus of attention in both academic and industry.

Although current researches on cloud computing are mostly focused on technical problems such as resource allocation [8] and task scheduling [17], there are some attempts to the formalization of fundamental notions in cloud computing. For example, an abstract formal model of cloud workflows was proposed in [7] using the Z notation. In [6], the hierarchical colored Petri Net model was adopted to specify the security mechanism in cloud computing. The Petri Net

© Springer Nature Switzerland AG 2019
Y. Xia and L.-J. Zhang (Eds.): SERVICES 2019, LNCS 11517, pp. 1–15, 2019.
https://doi.org/10.1007/978-3-030-23381-5_1

model is also used in [4] to build the fault tolerant model of cloud computing, and as the basis for a dynamic fault tolerant strategy in cloud computing. The agent paradigm was adopted in [14] to manage cloud resources and support cloud service discovery, negotiation and composition. A Bigraph model was proposed in [3] to formally specify cloud services and customers and their interaction schemes. In [11], Event-B is integrated with discrete-event simulation to analyze the performance and reliability of resilience of data stored in the cloud.

To support rigorous development of cloud applications and enhance their trustworthy, only providing the formal specification is certainly not enough and we need to further investigate the formal verification techniques that help for understanding and reasoning about cloud applications and ensuring their trustworthy. Model checking and theorem proving are two most-widely used verification techniques. In [1], the model checker UPPAAL is used to synthesize an optimal infinite scheduler for a given specification of Mobile Cloud Computing systems. In [13] a holistic approach was proposed to verify the correctness of Hadoop cloud architectures using model checking techniques. Although the model checking approach is powerful and fully automatic, the state-space explosion is an inherent problem for all the model checking approaches which is serious for large-scale systems like cloud applications. On the other hand, theorem proving technique is used in [12] to verify properties of cloud services and their compositions in PVS, which is based on the relational UTP (Unifying Theories of Programming) [9] semantics for cloud services that has been proposed in [16]. However, automatic verification in interactive theorem provers like PVS is a hard problem due to the undecidable algorithms and proof methods. Furthermore, sometimes users may fail to prove that a property holds or not, and can not produce a counter example for it using theorem provers.

SMT-based techniques have been used extensively for program verification [2]. In this paper, we propose a formal framework for verification of cloud applications using the SMT solver Z3 [5], which is also based on the relational UTP semantics for cloud services [16]. UTP aims to formalize the similar features of different languages in a similar style. It has been proved to be appropriate for formal semantics of various programming languages and specification languages like rCOS [10] and Reo [15]. Z3 is a state-of-the-art SMT solver, which can be used to check the satisfiability of logical formulas over one or more theories. It provides bindings for various programming languages. In this paper, we use Z3 python-bindings to specify the models and develop the verification framework for cloud applications. Unlike other theorem provers, such as PVS, Coq, etc., Z3 can automatically generate counter examples or prove the validity of a specific goal.

The paper is organized as follows: Sect. 2 presents how simple cloud services/applications are specified in Z3 based on observations on their input and output ports. The models of a family of composition patterns, which are used to construct more complex cloud services are presented in Sect. 3. In Sect. 4, we use the refinement-relation-check function for complex cloud services as an example

to show how to verify properties of cloud services and applications, and provide some case studies. Finally, Sect. 5 summarizes the paper.

2 Formalization of Cloud Services in Z3

Usually the computing services in a cloud application are distributed over the internet and far from its clients. Clients have no knowledge about the implementation details of the services and configuration of the application, and can access the cloud application regardless of their locations or what device being used. The only possible way that clients can know about a cloud application is via observations on the services provided by the application at corresponding input/output ports.

2.1 Observations as Timed Data Streams

A cloud service $\mathbf{C}$ is interpreted as a relation between an initial observation on inputs to $\mathbf{C}$ and a subsequent observation of the behavior of $\mathbf{C}$. we use $in_{\mathbf{C}}$ and $out_{\mathbf{C}}$ to denote what happen as inputs and outputs of a cloud service $\mathbf{C}$, respectively.

For every port of a cloud service $\mathbf{C}$, the corresponding observation on it is given by a *timed data stream* (TDS), which is defined as follows:

Definition 1. *Let D be a set of data elements and $\mathbb{R}_+$ be the set of non-negative real numbers which is used to represent time moments. Let $DS = D^\omega$ be the set of data streams, that is, the set of all streams $\alpha = (\alpha(0), \alpha(1), \alpha(2), \ldots)$ over D, and $\mathbb{R}_+^\omega$ be the set of all streams $a = (a(0), a(1), a(2), \ldots)$ over $\mathbb{R}_+$. The set of time streams is defined by the following subset of $\mathbb{R}_+^\omega$:*

$$TS = \{a \in \mathbb{R}_+^\omega \mid \forall n \geq 0.a(n) < a(n+1) \wedge \forall t \in \mathbb{R}_+.\exists k \in \mathbb{N}.a(k) > t\}$$

For two time streams a and b, $a < b \equiv \forall n \geq 0.a(n) < b(n)$. A timed data stream *is defined as a pair $\langle \alpha, a \rangle$ consisting of a data stream $\alpha \in DS$ and a time stream $a \in TS$. We use TDS to denote the set of timed data streams.*

Let $I_{\mathbf{C}}$ and $O_{\mathbf{C}}$ be the set of input and output port names of $\mathbf{C}$, the specification focuses on the constraints on the timed data streams of the corresponding ports:

$$in_{\mathbf{C}} : I_{\mathbf{C}} \to TDS$$
$$out_{\mathbf{C}} : O_{\mathbf{C}} \to TDS$$

We use relations on timed data streams to model cloud services. Every cloud service $\mathbf{C}$ can be represented by a pair of predicates P and Q as follows:

$$\mathbf{C}(in : in_{\mathbf{C}}; out : out_{\mathbf{C}})$$
$$\mathbf{pre} : P(in_{\mathbf{C}})$$
$$\mathbf{post} : Q(in_{\mathbf{C}}, out_{\mathbf{C}})$$

where $\mathbf{C}$ is the name of the cloud service, $P(in_{\mathbf{C}})$ is the condition that should be satisfied by inputs $in_{\mathbf{C}}$ of the cloud service, and $Q(in_{\mathbf{C}}, out_{\mathbf{C}})$ is the condition that should be satisfied by outputs $out_{\mathbf{C}}$ of $\mathbf{C}$.

2.2 Modeling of Simple Cloud Services

We first present the formal definitions of simple cloud services, which are used as the basis for the specification of compositions and development of more complex cloud services. The simple cloud services are dealt with as object instances of *Class* Service. An object instance is attached with four attributes: *service type*, *pre-condition*, *nodes* (represent timed data streams), and its corresponding *method*. We specify the time and data constraints of relations between inputs and outputs in the service definitions (methods). Basically, the behavior constraints of cloud services consist of two parts: *data constraints* and *time constraints*.

SyncSer has an input port whose corresponding timed data stream is specified as `nodesParam[0]` and a set of output ports (at least one, guaranteed by the assertion of the method). The behavior pattern of this cloud service follows like this: every item in both data stream and time stream at the output ports should be exactly equal to the items in the input data stream and time stream. All the conditions are specified as constraints. Finally, we take the conjunction of every constraint in the constraint list if the pre-condition of the service is True.

```
1   def SyncSer(self, bound, nodesParam):
2       if self.preCondition == True:
3           assert len(self.nodes) >= 2
4           target_num = len(self.nodes) - 1
5           constraints = []
6           for i in range(target_num):
7               for j in range(bound):
8                   constraints += [ nodesParam[0]['data'][j] ==
                        nodesParam[i + 1]['data'][j] ]
9                   constraints += [ nodesParam[0]['time'][j] ==
                        nodesParam[i + 1]['time'][j] ]
10          return Conjunction(constraints)
11      else:
12          return True
```

BufferSer has one input port and one output port. The time constraints of this service type is more complex than *SyncSer*, while the data-related constraints are the same as *SyncSer*. The time when the current data item gets transferred into the service through the input port is strictly earlier than the time when the data item gets taken/read through the output port. Besides, the time of the next request (input data item) should be strictly later than the previous returned result (last output data item).

```
1   def BufferSer(self, bound, nodesParam):
2       if self.preCondition == True:
3           assert len(self.nodes) == 2
4           constraints = []
5           for i in range(bound):
6               constraints += [ nodesParam[0]['data'][i] ==
                    nodesParam[1]['data'][i] ]
7               constraints += [ nodesParam[0]['time'][i] <
                    nodesParam[1]['time'][i] ]
```

```
8           for i in range(bound - 1):
9               constraints += [ nodesParam[0]['time'][i + 1] >
                    nodesParam[1]['time'][i] ]
10          return Conjunction(constraints)
11      else:
12          return True
```

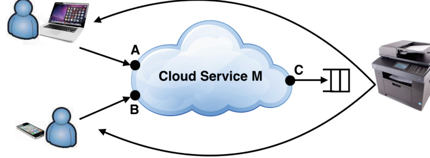

Fig. 1. Remote printing service

Example 1. Consider a simple example where a remote printer offers its printing service to two clients, which compete for the use of this shared resource. Each client can send out multiple printing requests to the printer and the requests from different clients are placed in a queue to be processed by the printer in a first-come first-served manner. After a file is printed out it can be collected by the client later. In order to keep the example simple to expose without considering priority for scheduling different printing tasks, we assume that requests from different clients never arrive simultaneously.

The cloud service **M** in Fig. 1 receives requests from different clients at ports A and B, and delivers a sequence of requests through port C to a queue on the printer side. Such a service is called *MergeSer*. It has two input ports and one output port. When there is only one timed data stream from one specifc input port, the *MergeSer* is reduced to the *SyncSer* with one input and one output ports. When there exists two different timed data streams from the input ports, the *MergeSer* is captured through the method *Merge*, which is defined recursively. Every time one of the timed data items from these two input ports is chosen to be fed into the service. The specification of this service is in the following:

```
1   def MergeSer(self, bound, nodesParam):
2       if self.preCondition == True:
3           assert 2 <= len(self.nodes) <=3
4           if len(self.nodes) == 2:
5               constraints = []
6               for i in range(bound):
7                   constraints += [ nodesParam[0]['data'][i] ==
                        nodesParam[1]['data'][i] ]
8                   constraints += [ nodesParam[0]['time'][i] ==
                        nodesParam[1]['time'][i] ]
9               return Conjunction(constraints)
```

```
10            elif len(self.nodes) == 3:
11                return self.Merge(bound, nodesParam)
12        else:
13            return True
```

where *Merge* is the method to deal with the case when the two input ports both have requests to provide for the *MergeSer*. It captures the behavior of merging two input timed data streams into one output stream and the detailed definition is omitted here due to the length limitation.

RouterSer has one input port and two output ports. The behavior of this service type is that the input timed data items are nondeterministically taken through the two output ports, which can be specified using the *Merge* method with the input timed data item handled as the output result in *Merge* and the two output timed data items dealt with as the two input requests in *Merge*.

```
1     def RouterSer(self, bound, nodesParam):
2         if self.preCondition == True:
3             assert len(self.nodes) == 3
4             new_nodes =
5             [nodesParam[1], nodesParam[2], nodesParam[0]]
6             return self.Merge(bound, new_nodes)
7         else:
8             return True
```

3 Composition of Cloud Services

Different cloud services can be composed together to build more complex services/applications. Simple cloud services are defined as object instances of class *Service*, therefore their composition can be naturally modeled by composition on *Service* instances, which leads to a new class *Composition* capturing the behavior of the composed cloud service/application. In this section, we introduce a family of composition operations for two cloud services $\mathbf{serv}_i (i = 1, 2)$:

Sequential Composition. Suppose one output port O of $\mathbf{serv}_1$ and one input port I of $\mathbf{serv}_2$ can be joined together and the timed data stream that happens on O thus can be taken as the input on I for $\mathbf{serv}_2$. After joining these two ports, the extra constraints restricted on O (and I) are about the equality between the output timed data stream of $\mathbf{serv}_1$ and the input timed data stream of $\mathbf{serv}_2$. Since there can exist more than one output ports in $\mathbf{serv}_1$, the attribute *index* is designed to indicate the correct output port being joined. The output timed data stream in $\mathbf{serv}_1$ is thus represented by `nodesParam1[index]` and the input timed data stream in $\mathbf{serv}_2$ is represented by `nodesParam2[0]`. Then the result specification of the cloud service by sequentially composing $\mathbf{serv}_1$ and $\mathbf{serv}_2$ is:

```
1     def SeqComp(self, bound, nodesParam1, nodesParam2):
2         if self.serv1.preCondition == self.serv2.preCondition ==
               True:
```

```
3            constraints = []
4            for i in range(bound):
5                constraints += [ nodesParam1[self.index]['data'][i]
                     == nodesParam2[0]['data'][i] ]
6                constraints += [ nodesParam1[self.index]['time'][i]
                     == nodesParam2[0]['time'][i] ]
7            return And(
8            self.serv1.valueFunctions[self.serv1.service](bound,
                 nodesParam1),
9            self.serv2.valueFunctions[self.serv2.service](bound,
                 nodesParam2),
10           Conjunction(constraints))
11       else:
12           return True
```

Note that when two cloud services $\mathbf{serv}_1$ and $\mathbf{serv}_2$ are sequentially composed, we can certainly join more than one pair of ports together and the definition of the resulting service is similar, but it is not necessary to join all the output ports of $\mathbf{serv}_1$ to all the input ports of $\mathbf{serv}_2$. Some ports in the services can be left as the input/output ports for the resulting service. The definition for the general situation is similar and can be easily obtained.

External, Internal and Conditional Choices. Cloud services can be aggregated in a number of different ways, besides the sequential composition. In the following we consider a few such combinators. A typical composition pattern being widely used is *external choice*. For the two cloud services $\mathbf{serv}_1$ and $\mathbf{serv}_2$, when they are put together and interacting with the environment, clients from the environment are allowed to choose either to input on the input ports of $\mathbf{serv}_1$, or on input ports of $\mathbf{serv}_2$, which will trigger the corresponding cloud service $\mathbf{serv}_1$ or $\mathbf{serv}_2$, respectively, and produce the associated output on the corresponding output ports. Formally, the result specification of the cloud service as an external choice of $\mathbf{serv}_1$ and $\mathbf{serv}_2$ is defined as:

```
1    def ExChoice(self, bound, nodesParam1, nodesParam2):
2        if self.serv1.preCondition == self.serv2.preCondition ==
              True:
3            return And( self.serv1.valueFunctions[self.serv1.service
                 ](bound, nodesParam1),
4            self.serv2.valueFunctions[self.serv2.service](bound,
                 nodesParam2))
5        elif self.serv1.preCondition == True and self.serv2.
              preCondition == False:
6            return self.serv1.valueFunctions[self.serv1.service](
                 bound, nodesParam1)
7        elif self.serv1.preCondition == False and self.serv2.
              preCondition == True:
8            return self.serv2.valueFunctions[self.serv2.service](
                 bound, nodesParam2)
9        else:
10           return True
```

Sometimes it is possible that both cloud services might have input ports in common so that there is no clear prescription as to which route is followed when one of these common ports is chosen. In the implementation, either service can be chosen to be executed. This case is captured by the *internal choice* pattern, which is formally defined as follows:

```
1  def InChoice(self, bound, nodesParam1, nodesParam2):
2      if self.serv1.preCondition == self.serv2.preCondition ==
           True:
3          return Or( self.serv1.valueFunctions[self.ser1.service](
               bound, nodesParam1),
4          self.serv2.valueFunctions[self.serv2.service](bound,
               nodesParam2))
5      else:
6          return True
```

Besides the external and internal choices, a further form of choice, the *conditional choice* which is based on the value of a boolean expression, is also needed for combination of cloud services. This case is formally defined by the following definition which means that if the boolean expression is satisfied then the cloud service $\mathbf{serv}_1$ is executed, and otherwise, $\mathbf{serv}_2$ is executed:

```
1  def ConChoice(self, bound, nodesParam1, nodesParam2):
2      if self.bool_con == True:
3          return self.serv1.valueFunctions[self.serv1.service](
               bound, nodesParam1)
4      else:
5          return self.serv2.valueFunctions[self.serv2.service](
               bound, nodesParam2)
```

Parallel Composition. The simplest form of parallel combinator captures the case that both cloud services $\mathbf{serv}_1$ and $\mathbf{serv}_2$ are invoked and executed in parallel when triggered by a pair of inputs on the corresponding input ports of both $\mathbf{serv}_1$ and $\mathbf{serv}_2$. Therefore, to make it possible to execute the parallel combination of $\mathbf{serv}_1$ and $\mathbf{serv}_2$, both pre-conditions of the two services should be satisfied and the execution will lead to the result that the post-conditions of both $\mathbf{serv}_1$ and $\mathbf{serv}_2$ should be satisfied.

```
1  def ParallelComp(self, bound, nodesParam1, nodesParam2):
2      if self.serv1.preCondition == self.serv2.preCondition ==
           True:
3          return And(
4          self.serv1.valueFunctions[self.serv1.service]
5                  (bound, nodesParam1),
6          self.serv2.valueFunctions[self.serv2.service]
7                   (bound, nodesParam2))
8      else:
9          return True
```

In the parallel composition defined above, when two cloud services are put into parallel, they may evolve completely autonomously, i.e., we have no restriction on the inputs for the two services and they can arrive at any time. Sometimes we may hope to have some inputs for $\mathbf{serv}_1$ and $\mathbf{serv}_2$ arrive only simultaneously. For simplicity, we assume that the data can only arrive at the input ports I_1 and I_2 simultaneously, where I_1 and I_2 belong to the input ports of $\mathbf{serv}_1$ and $\mathbf{serv}_2$ respectively. And the data arriving at all the other input ports except I_1 and I_2 are independent. In this case, extra constraints on the input timed data streams of the two services should be captured. Then we have the following specification:

```
1   def SyncParallel(self, bound, nodesParam1, nodesParam2):
2       if self.serv1.preCondition == self.serv2.preCondition ==
             True:
3           constraints = []
4           for i in range(bound):
5               constraints += [ nodesParam1[0]['data'][i] ==
                     nodesParam2[0]['data'][i] ]
6               constraints += [ nodesParam1[0]['time'][i] ==
                     nodesParam2[0]['time'][i] ]
7           return And(
8           self.serv1.valueFunctions[self.serv1.service](bound,
                nodesParam1),
9           self.serv2.valueFunctions[self.serv2.service](bound,
                nodesParam2),
10          Conjunction(constraints))
11      else:
12          return True
```

In many cases, a family of cloud services may exist and behave in parallel in a pairwise fashion. To model this, the n-ary version of these two parallel combinators are very helpful. The corresponding specification can be easily generalized to the case for composing multiple services.

A similar situation we consider is the case of merging two input ports of cloud services $\mathbf{serv}_1$ and $\mathbf{serv}_2$ into one port. Let `nodesParami` for $i = 1, 2$ be the timed data streams on the input port I_i in $\mathbf{serv}_i$, respectively. By merging I_1 and I_2 into one port I, when the resulting service receives a request on I, it will behave in a broadcasting way. In other words, the request will be replicated on I and sent to both $\mathbf{serv}_1$ and $\mathbf{serv}_2$ to trigger their execution simultaneously. This operation can be realized by renaming without defining a specific method.

4 Verification of Cloud Applications in Z3

Based on the specification of simple cloud services and the family of composition patterns, we can develop more complex cloud services for different needs, which are defined as a new class *Cloud*. In this class, we provide two methods *config* and *compose* to develop complex cloud services out of the simple cloud services and the composition patterns. Meanwhile, we encode the refinement and equivalence

relations between cloud services as propositions under analysis. Based on the Z3 SMT solver, we defined the main method *Refine* to check the validity of the refinement and equivalence relations or to generate counter examples for the dissatisfaction of relations.

```
1  class Cloud:
2      def __init__(self):
3          self.services = []
4          self.compositions = []
5
6      def config(self, service, pre_condition, *nodes):
7          self.services += [Service(service, pre_condition, nodes)]
8          return self
9
10     def compose(self, composition, serv1, serv2, index, bool_con =
            True):
11         self.compositions +=
12         [Composition(composition, serv1, serv2, index, bool_con)]
13         return self
```

The predicate capturing the refinement relation between two cloud services is an implication statement. Assume we have two complex cloud services $cServ_1$ and $cServ_2$, $cServ_1$ is a refinement of $cServ_2$ (or $cServ_1$ refines $cServ_2$) if and only if the specification of $cServ_2$ is further restricted by $cServ_1$, i.e., $cServ_1 \rightarrow cServ_2$. Furthermore, two cloud services are in equivalence relation, i.e. $cServ_1 \leftrightarrow cServ_2$, if and only if they refine each other. Such proposition is valid if it is always true whatever the assignment of variables is. It is satisfiable if there exists an evidence (an assignment to the variables) under which this proposition evaluates true. If the proposition $cServ_1 \rightarrow cServ_2$ is valid, i.e. always true under any assignment of values, then its negation will not have any witness (any satisfying assignment). In other words, the negation of the implication is unsatisfiable. On the other hand, if a solution is found for the negation of the proposition, then this solution is actually the counter example for the satisfaction of the relation.

Algorithm 1 provides the pseudocode for the definition of the *Refine* method, in which a solver is created and the constraints of the negation of the target implication are added into the solver. If the solver returns `unsat` (corresponding to True in Algorithm 1), then it acts as a proof for the validity of the refinement relation between the two cloud services under analysis. If the solver returns `sat` (corresponding to False in Algorithm 1), the model (witness) of the negation is the counter example we need for proving the dissatisfaction of the relation.

In the beginning, the command `Solver()` is used to create a general purpose Z3 Solver. The dictionary `nodes` stores the uninterpreted variables, which can be further used to generate node parameters for invocation of simple cloud services. The first double `for loop` generates the time constraints (for well-definedness) and constraints for each specific service in $cServ_1$ and the next double `for loop` generates the time constraints and constraints for each specific composition in

$cServ_1$. The procedures up to now are used to handle $cServ_1$. The subsequent procedures are for the refined cloud service $cServ_2$. The generation of time constraints and constraints for each simple cloud service and composition is similar. Besides, the uninterpreted symbols that need to be under universal quantifier are stored in `UniVar`.

Example 2. Consider the cloud applications shown in Fig. 2, where users can check the information about flights and can order flight tickets. Different companies offer flights to the same location and the ticket availability and price for each flight varies and may change at any time. If a user wants to make a trip between two places and make a query for the flight information, he/she hopes to collect the information for all the available flights at the latest time.

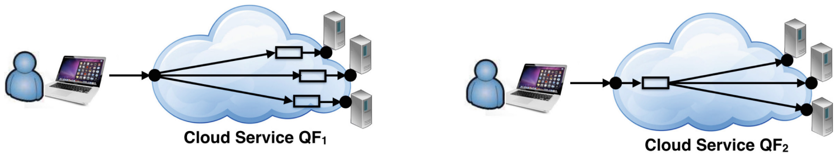

Fig. 2. Flight query services

The cloud service QF1 in Fig. 2 has a buffer on the server side for each flight company. It accepts queries from users and store queries in every buffer. Based on the simple cloud services, the model of QF1 can be developed in Z3 in the following way:

```
1  QF1 = Cloud()
2  QF1.config('Buffer', 'True', 'A', 'B')
3  QF1.config('Buffer', 'True', 'A', 'C')
4  QF1.config('Buffer', 'True', 'A', 'D')
```

In cloud service QF2, the location of the buffer is on the user's side. QF2 can accept the queries from the user, place the queries in the buffer and simultaneously send a copy of the queries to each flight company. The model of QF2 is constructed like this:

```
1  QF2 = Cloud()
2  QF2.config('Buffer', 'True', 'A', 'E')
3  QF2.config('Sync', 'True', 'E', 'B')
4  QF2.config('Sync', 'True', 'E', 'C')
5  QF2.config('Sync', 'True', 'E', 'D')
```

Next we invoke the *Refine* function in two directions to check the refinement and equivalence relation between QF1 and QF2. The result of `QF2.Refine(QF1, 10)` is *True* and *None*, which means that QF2 is indeed a refinement of QF1 and therefore no counter example is provided.

Algorithm 1. $cServ_1$.Refine ($cServ_2$, bound)

Require: $cServ_1$ and $cServ_2$ are both instances of cloud services.
Ensure: the function returns *True* or *False with a counter example*

```
1: solver ← Solver( ) ; nodes ← {}
2: for ser ← services in cServ_1 do
3:    for n ← ports in ser do
4:       if n ∉ nodes then
5:          nodes[n] ←
            {time : [n_t_0, ··· , n_t_(bound − 1)], data : [n_d_0, ··· , n_d_(bound − 1)]}
6:          add time constraints n_t_0 ≥ 0 ∧ n_t_i < n_t_i + 1 to the solver
7:       end if
8:    end for
9:    add service constraints to the solver according to definitions in Section 2.2
10: end for
11: for comp ← compositions in cServ_1 do
12:    for n ← ports in two services in comp do
13:       if n ∉ nodes for two services then
14:          nodes[n] ←
            {time : [n_t_0, ··· , n_t_(bound − 1)], data : [n_d_0, ··· , n_d_(bound − 1)]}
15:          add time constraints n_t_0 ≥ 0 ∧ n_t_i < n_t_i + 1 to the solver
16:       end if
17:    end for
18:    add composition constraints to the solver according to definitions in Section 3
19: end for
20: UniVar ← {}; ReSerConstr ← {}; RecompConstr ← {}; ReConstr ← {}
21: for ser ← services in cServ_2 do
22:    for n ← ports in ser do
23:       if n ∉ nodes then
24:          generate uninterpreted variables
25:          add the variables to UniVar (variables under the universal quantifier)
26:          add time constraints n_t_0 ≥ 0 ∧ n_t_i < n_t_i + 1 to ReSerConstr
27:       end if
28:       add service constraints to ReSerConstr according to definitions in Section 2.2
29:    end for
30: end for
31: for comp ← compositions in cServ_2 do
32:    for n ← ports in two services in comp do
33:       if n ∉ nodes then
34:          generate uninterpreted variables
35:          add the variables to UniVar (variables under the universal quantifier)
36:          add time constraints n_t_0 ≥ 0 ∧ n_t_i < n_t_i + 1 to ReCompConstr
37:       end if
38:    end for
39:    add composition constraints to ReCompConstr according to definitions in
       Section 3
40: end for
41: ReConstr = ¬ (ReSerConstr ∧ ReCompConstr)
42: let UniVar be {n_1, ··· , n_m}, add the following constraints to solver
43:       (∀n_1) ··· (∀n_m).ReConstr

44: RefineResult ← solver.check()
```

On the other hand, the result of QF1.Refine(QF2, 10) is *False*, which indicates the equivalence relation doesn't hold between QF1 and QF2. The counter example actually presents a solution which satisfies the constraints of QF1 but doesn't satisfy the constraints of QF2. "_d_" indicates the data item on the corresponding port indicated by the capitalized letter while "_t_" indicates the time moment. Time-related constraints for well-definedness are clearly satisfied. The data streams in this specific counter example satisfy the data-related constraints in both QF1 and QF2. However, the time stream on the three output ports of QF2 should be exactly equal, which are not satisfied. Time constraints in QF1 are relaxed a bit. Only delay of the data item transfer in three *BufferSer* needs to be satisfied and this counter example provides a feasible and correct solution.

```
1  True, None
2  False
3  A_d_0 = 0, A_d_1 = 0, A_d_2 = 0, A_d_3 = 0, A_d_4 = 0,
4  A_d_5 = 0, A_d_6 = 0, A_d_7 = 0, A_d_8 = 0, A_d_9 = 0;
5  A_t_0 = 0, A_t_1 = 3, A_t_2 = 6, A_t_3 = 9, A_t_4 = 12,
6  A_t_5 = 15, A_t_6 = 18, A_t_7 = 21, A_t_8 = 24, A_t_9 = 27;
7  B_d_0 = 0, B_d_1 = 0, B_d_2 = 0, B_d_3 = 0, B_d_4 = 0,
8  B_d_5 = 0, B_d_6 = 0, B_d_7 = 0, B_d_8 = 0, B_d_9 = 0;
9  B_t_0 = 1, B_t_1 = 4, B_t_2 = 7, B_t_3 = 10, B_t_4 = 13,
10 B_t_5 = 16, B_t_6 = 19, B_t_7 = 22, B_t_8 = 25, B_t_9 = 28;
11 C_d_0 = 0, C_d_1 = 0, C_d_2 = 0, C_d_3 = 0, C_d_4 = 0,
12 C_d_5 = 0, C_d_6 = 0, C_d_7 = 0, C_d_8 = 0, C_d_9 = 0;
13 C_t_0 = 2, C_t_1 = 5, C_t_2 = 8, C_t_3 = 11, C_t_4 = 14,
14 C_t_5 = 17, C_t_6 = 20, C_t_7 = 23, C_t_8 = 26, C_t_9 = 29;
15 D_d_0 = 0, D_d_1 = 0, D_d_2 = 0, D_d_3 = 0, D_d_4 = 0,
16 D_d_5 = 0, D_d_6 = 0, D_d_7 = 0, D_d_8 = 0, D_d_9 = 0;
17 D_t_0 = 2, D_t_1 = 5, D_t_2 = 8, D_t_3 = 11, D_t_4 = 14,
18 D_t_5 = 17, D_t_6 = 20, D_t_7 = 23, D_t_8 = 26, D_t_9 = 29,
```

Example 3. Consider a travel service scenario which involves reserving hotel and booking transportation (flight or train). The service package processes clients' requests in the following way: The service is initiated through a request from some client, then the request is first handled by the hotel service through a *SyncSer*. Next the request is further transferred to the transportation service, where the *RouterSer* operates and sends it to the flight service and train service. Meanwhile, flight and train booking are both monitored by a government service, which is aggregated through a *MergeSer*. The transportation reservation succeeds only if the government service accepts the reservation. Besides, the hotel service and the transportation service are composed through a sequential composition. This service package can be modeled in Z3 as follows.

```
1  Serv1 = Service('Sync', 'True', ('A', 'B'))
2  Serv2 = Service('Router', 'True', ('C', 'D', 'E'))
3  TravelPack1 = Cloud()
4  TravelPack1.compose('SeqComp', Serv1, Serv2, 1)
5  TravelPack1.config('Merger', 'True', 'D', 'E', 'F')
```

Another service package can be simpler, which involves a hotel service and a government service in charge of transportation reservation. Moreover, these two services are also composed through a sequential composition. The model of this simplified service package is presented in the following.

```
1  Serv1 = Service('Sync', 'True', ('A', 'B'))
2  Serv3 = Service('Sync', 'True', ('C', 'F'))
3  TravelPack2 = Cloud()
4  TravelPack2.compose('SeqComp', Serv1, Serv3, 1)
```

Intuitively, these two service packages provide the clients with the same result: hotel and transportation reservation. Therefore, next we check if the equivalence relation exists between them. After invoking the `Refine` function in two directions, we get the returned results shown as follows:

```
1  True, None
2  True, None
```

The result shows that the *TravelPack1* is indeed a refinement of *TravelPack2* while the refinement relation holds also in the other direction. Finally, the equivalence relation between them gets proved.

5 Conclusion and Future Work

This paper extends our previous work on the design model for cloud services and proposes a framework on formal specification of cloud services and compositions in SMT solver Z3. The composition of cloud services is given by a family of composition operators which are specified in a class capturing the behavior of the composition. The framework naturally preserves the original choreography of cloud applications, and thus makes the description of cloud services and applications reasonably readable. This work also provides a complement of the verification by theorem proving approach in our previous work. In fact, sometimes users of theorem provers like Coq or PVS need to construct proofs or even build counter examples manually first to show that a property is not satisfiable. For such cases, using Z3 makes it possible to automatically search for possible bounded counter examples.

However, the proposed framework focuses on addressing the data- and time-related properties of the top level cloud services while failing to address the availability and failure rate of online services. The case studies are also set in a high level conceptual setting. In future work, we plan to incorporate the QoS aspects on cloud services into this model and will investigate the formalization and quantitative reasoning about low level programs of cloud applications in SMT solvers. On the other hand, we hope to develop the formal model for dynamic reconfiguration and adaptation of cloud services as well, which is quite useful in real world scenarios.

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

References

1. Aceto, L., Larsen, K.G., Morichetta, A., Tiezzi, F.: A cost/reward method for optimal infinite scheduling in mobile cloud computing. In: Braga, C., Ölveczky, P.C. (eds.) FACS 2015. LNCS, vol. 9539, pp. 66–85. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-28934-2_4
2. Alt, L., et al.: HiFrog: SMT-based function summarization for software verification. In: Legay, A., Margaria, T. (eds.) TACAS 2017. LNCS, vol. 10206, pp. 207–213. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54580-5_12
3. Benzadri, Z., Belala, F., Bouanaka, C.: Towards a formal model for cloud computing. In: Lomuscio, A.R., Nepal, S., Patrizi, F., Benatallah, B., Brandić, I. (eds.) ICSOC 2013. LNCS, vol. 8377, pp. 381–393. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-06859-6_34
4. Chen, L., Fan, G., Liu, Y.: Modeling and analyzing cost-aware fault tolerant strategy for cloud application. In: Proceedings of SEKE 2016, pp. 439–442. KSI Research Inc. and Knowledge Systems Institute Graduate School (2016)
5. de Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24
6. Fitch, D., Xu, H.: A raid-based secure and fault-tolerant model for cloud information storage. Int. J. Softw. Eng. Knowl. Eng. **23**(05), 627–654 (2013)
7. Freitas, L., Watson, P.: Formalizing workflows partitioning over federated clouds: multi-level security and costs. Int. J. Comput. Math. **91**(5), 881–906 (2014)
8. Graiet, M., Mammar, A., Boubaker, S., Gaaloul, W.: Towards correct cloud resource allocation in business processes. IEEE Trans. Serv. Comput. **10**(1), 23–36 (2017)
9. Hoare, C.A.R., He, J.: Unifying Theories of Programming. Prentice Hall International, Upper Saddle River (1998)
10. Jifeng, H., Li, X., Liu, Z.: rCOS: a refinement calculus of object systems. Theor. Comput. Sci. **365**(1–2), 109–142 (2006)
11. Laibinis, L., Byholm, B., Pereverzeva, I., Troubitsyna, E., Eeik Tan, K., Porres, I.: Integrating Event-B modelling and discrete-event simulation to analyse resilience of data stores in the cloud. In: Albert, E., Sekerinski, E. (eds.) IFM 2014. LNCS, vol. 8739, pp. 103–119. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-10181-1_7
12. Nawaz, M.S., Sun, M.: Using PVS for modeling and verifying cloud services and their composition. In: Proceedings of CBD 2018, pp. 42–47. IEEE (2018)
13. Reddy, G.S., Feng, Y., Liu, Y., Dong, J.S., Sun, J., Kanagasabai, R.: Towards formal modeling and verification of cloud architectures: a case study on Hadoop. In: Proceedings of SERVICES 2013, pp. 306–311. IEEE Computer Society (2013)
14. Sim, K.M.: Agent-based cloud computing. IEEE Trans. Serv. Comput. **5**(4), 564–577 (2012)
15. Sun, M., Arbab, F., Aichernig, B.K., Astefanoaei, L., de Boer, F.S., Rutten, J.J.M.M.: Connectors as designs: modeling, refinement and test case generation. Sci. Comput. Program. **77**(7–8), 799–822 (2012)
16. Sun, M., Fu, G.: A formal design model of cloud services. In: Proceedings of SEKE 2017, pp. 173–178. KSI Research Inc. and Knowledge Systems Institute (2017)
17. Zhang, P., Lin, C., Ma, X., Ren, F., Li, W.: Monitoring-based task scheduling in large-scale SaaS cloud. In: Sheng, Q.Z., Stroulia, E., Tata, S., Bhiri, S. (eds.) ICSOC 2016. LNCS, vol. 9936, pp. 140–156. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-46295-0_9