A Formal Design Model of Cloud Services

Meng Sun* and Guirong Fu†

*LMAM & DI, School of Mathematical Sciences, Peking University, Beijing, China
sunmeng@math.pku.edu.cn
†Yuanpei College, Peking University, Beijing, China
fgr079@126.com

***Abstract*—To support rigorous development of cloud applications, a formal model for understanding and reasoning about cloud services is needed. Unifying Theories of Programming (UTP) provide a formal semantic foundation for various expressive programming and specification languages. A key concept in UTP is design: the familiar pre / post-condition pair that describes a contract. In this paper we use UTP to provide a formal model for cloud computing, whereby cloud services are interpreted as designs in UTP. Refinement and equivalence relations between cloud services can be naturally established by implication between predicates. A family of composition operators that can be used to put different cloud services together to construct more complex services and applications are defined based on the design model for cloud services. On the other hand, dynamic reconfiguration of cloud applications can be dealt with in the context of the design model as well, by applying the reconfiguration rules on the design models for the corresponding applications.**

Keywords: Cloud service, design, composition, refinement, dynamic reconfiguration

I. Introduction

Cloud computing has been coined as an umbrella term to describe a family of sophisticated computing services and gained a significant amount of attention in the past decades. It denotes a model on which a computing infrastructure is viewed as a "cloud", from which businesses and individuals access applications from different locations. According to [3], Cloud is defined as a parallel and distributed computing system consisting of a collection of inter-connected and virtualized computers that are dynamically provisioned and presented as one or more unified computing resources based on service-level agreements (SLAs). Instead of running or storing applications locally, one can host their application in the cloud and access it from any location using a client such as a web browser.

To support rigorous development of cloud applications, we need to investigate the formal foundations for understanding and reasoning about clouds. Although researches on cloud computing are mainly focused on technical problems such as resource allocation [9], resource sharing [12], resource management [21], policy optimization [16] and task scheduling [22], there are some attempts to the formalization of fundamental notions in cloud computing. An abstract formal model of cloud workflows was proposed in [8] using the Z notation. In [7], the hierarchical colored Petri Net model was adopted to specify the security mechanism in cloud computing. The Petri Net 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 [19] to manage cloud resources and support cloud service discovery, negotiation and composition. A Bigraph model was proposed in [2], [18] to formally specify cloud services and customers and their interaction schemes. A formal model for aspects of performance, resource consumption, and deployment on the cloud is developed using the abstract behavioral specification language ABS in [5]. In [1], the model checker UPPAAL is used to synthesize an optimal infinite scheduler for a given specification of Mobile Cloud Computing systems.

The problem that we address in this paper is to develop a formal model for cloud services under the UTP (Unifying Theories of Programming) semantic framework. UTP was proposed by Hoare and He in [10], and aims to formalize the similar features of different languages in a similar style. UTP has been proved to be appropriate for formal semantics of various programming languages and specification languages like Circus [15], TCOZ [17], rCOS [11] and Reo [20]. We believe it is also well suited for developing a proper formal foundation of cloud computing. Furthermore, interpreting cloud services as UTP designs makes it easier to guarantee the consistency of cloud service specifications and the corresponding implementations in different concrete languages and platforms whose semantics can also be given in UTP, which is very important for Cloud computing.

Clouds provide applications / storages / services that can be used by local clients from different locations in the real world, and clients can link to the cloud via different instruments. Since our observation on the cloud can only be obtained by interactions between users and services provided by the cloud, for an arbitrary cloud service $\mathbf{C}$ the relevant observations usually come in pairs on its input ports and output ports respectively. When a user send a requirement to the cloud to acquire some service, the requirement will be distributed to some available resource and dealt with in the cloud. Afterwards, the result is sent out to the user and the resource is released and available again to be invoked later. Thus cloud services can be interpreted as *designs* in UTP, i.e., pairs of predicates $P \vdash Q$, where P is a predicate specifying the relationship among what happen as the inputs of the cloud service and Q is the predicate specifying the condition that should be satisfied by the outputs.

The design model for cloud services proposed in this paper provides a family of composition operators to compose different cloud services or resources to build complex applications. An advantage of representing cloud services as such designs in

DOI reference number: 10.18293/SEKE2017-055.

our model is that cloud applications can be decomposed into simpler services, and in suitable circumstances the behavior of the whole application can be captured by the composition of the predicates describing its component services. It is natural to verify cloud service properties by assume-guarantee reasoning based on the design model framework and the verification of separate services are becoming quite easy. Furthermore, dynamic reconfigurations of cloud applications can also be captured by the reconfiguration rules for transforming the design models.

The paper is organized as follows. Section II shows how cloud services are interpreted as designs in UTP based on observations on their input and output ports, and introduces the refinement and equivalence relations between cloud services. Then, in Section III we provide a family of composition operators that can be used to put different cloud services together to construct more complex services and applications. Section IV introduces dynamic reconfiguration for cloud services based on the design model. Finally, Section V summarizes the paper and comes up with some future work we are going to work on.

II. Cloud Services as Designs

Usually the computing and storage resources in the cloud are located far from the users. Users have no knowledge about its details and configuration, and can access the cloud applications regardless of their locations or what device being used. Thus the only possible way that users can know about a cloud application is via observations on the services provided by the application: A cloud service is interpreted as a relation between an initial observation on inputs to the cloud service and a subsequent observation of the behavior of the execution.

During observations it is usual to wait for some initial transient behavior to stabilize before making any further observation. To express this, two Boolean variables ok and ok' are introduced, where ok stands for a successful initialization of computation in the cloud service or communication with other services by external users or organizations, and ok' denotes the observation that the cloud service has either terminated or reached an intermediate stable state. When ok' is false, the cloud service becomes divergent.

A. Cloud Services as Designs

In this paper 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, 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 $\mathbb{R}_+$:

$$TS = \{a \in \mathbb{R}_+^\omega \mid a < a'\}$$

where a' is the derivative of a defined as

$$a' = (a(1), a(2), a(3), \ldots)$$

for $a = (a(0), a(1), a(2), \ldots)$, and 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 $\mathbf{C}$, then $in_{\mathbf{C}}$ and $out_{\mathbf{C}}$ are defined as the following mappings from the corresponding port sets to TDS.

$$\begin{aligned} in_{\mathbf{C}} : &\quad I_{\mathbf{C}} \to TDS \\ out_{\mathbf{C}} : &\quad O_{\mathbf{C}} \to TDS \end{aligned}$$

Definition 2: A design is a pair of predicates $P \vdash Q$, where neither predicate contains ok or ok', and P has only input variables. It has the following meaning:

$$P \vdash Q \;\equiv\; ok \wedge P \Rightarrow ok' \wedge Q$$

We use relations on timed data streams to model cloud services. Every cloud service $\mathbf{C}$ can be represented by the design $P(in_{\mathbf{C}}) \vdash Q(in_{\mathbf{C}}, out_{\mathbf{C}})$. In the following we write such a design for cloud service $\mathbf{C}$ as:

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

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 $\mathbf{C}$.

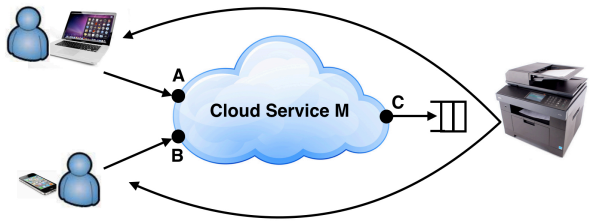

Fig. 1. Remote Printing Service

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 non-deterministic choice, we assume that requests from different clients never arrive simultaneously.

The cloud service $\mathbf{M}$ in Figure 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. The specification of such a service is given as follows:

$$\begin{aligned} &\mathbf{M}(in : (A \mapsto \langle \alpha, a \rangle, B \mapsto \langle \beta, b \rangle); out : C \mapsto \langle \gamma, c \rangle) \\ \mathbf{pre} &: \mathcal{D}(\langle \alpha, a \rangle) \wedge \mathcal{D}(\langle \beta, b \rangle) \wedge \forall i, j \geq 0.a(i) \neq b(j) \\ \mathbf{post} &: \mathcal{D}(\langle \gamma, c \rangle) \wedge M(\langle \alpha, a \rangle, \langle \beta, b \rangle, \langle \gamma, c \rangle) \end{aligned}$$

where $\mathcal{D}(\langle\alpha, a\rangle)$ is a predicate to judge whether $\langle\alpha, a\rangle$ is a well-defined TDS satisfying the requirements on the corresponding port, and $M(\langle\alpha, a\rangle, \langle\beta, b\rangle, \langle\gamma, c\rangle)$ is a predicate that captures the behavior of merging two timed data streams $\langle\alpha, a\rangle$ and $\langle\beta, b\rangle$ into $\langle\gamma, c\rangle$, and is defined as follows:

$$
\begin{aligned}
&M(\langle\alpha, a\rangle, \langle\beta, b\rangle, \langle\gamma, c\rangle) \\
&= \begin{cases}
\langle\gamma, c\rangle = \langle\alpha, a\rangle & \text{if } |\langle\beta, b\rangle| = 0 \\
\langle\gamma, c\rangle = \langle\beta, b\rangle & \text{if } |\langle\alpha, a\rangle| = 0 \\
\begin{cases}
\gamma(0) = \alpha(0) \wedge c(0) = a(0) \wedge \\
M(\langle\alpha', a'\rangle, \langle\beta, b\rangle, \langle\gamma', c'\rangle) \text{ if } a(0) < b(0) \\
\gamma(0) = \beta(0) \wedge c(0) = b(0) \wedge \\
M(\langle\alpha, a\rangle, \langle\beta', b'\rangle, \langle\gamma', c'\rangle) \text{ if } b(0) < a(0)
\end{cases} \\
\text{otherwise}
\end{cases}
\end{aligned}
$$

B. Refinement and Equivalence of Cloud Services

The notion of refinement has been widely used in different kinds of system descriptions. For example, in data refinement [6], the concrete model is required to have enough redundancy to completely represent the abstract model. Such a relationship is guaranteed by a surjective map from the concrete model to the abstract one. Implication of predicates establishes a proper refinement order over cloud services. Thus, more concrete implementations imply more abstract specifications. For two cloud services $\mathbf{C}_1$ and $\mathbf{C}_2$, if $in_{\mathbf{C}_1} = in_{\mathbf{C}_2}$ and $out_{\mathbf{C}_1} = out_{\mathbf{C}_2}$, then

$$\mathbf{C}_1 \sqsubseteq \mathbf{C}_2 \ =_{df} \ (P_1 \Rightarrow P_2) \wedge (P_1 \wedge Q_2 \Rightarrow Q_1) \qquad (1)$$

In other words, preconditions on inputs of cloud services are weakened under refinement, and postconditions on outputs of cloud services are strengthened. Taking equation (1) into consideration, $\mathbf{C}_2$ is stronger than $\mathbf{C}_1$ because it has a weaker assumption P_2, and so it can be used in more contexts. Furthermore, in all circumstances where $\mathbf{C}_1$ can be applied, $\mathbf{C}_2$ has a stronger commitment, so its behavior can be more precisely predicted and controlled comparing with $\mathbf{C}_1$.

Equivalence of cloud services is defined in the normal way by mutual refinement:

$$\mathbf{C}_1 \equiv \mathbf{C}_2 \quad \textbf{iff} \quad \mathbf{C}_1 \sqsubseteq \mathbf{C}_2 \wedge \mathbf{C}_2 \sqsubseteq \mathbf{C}_1$$

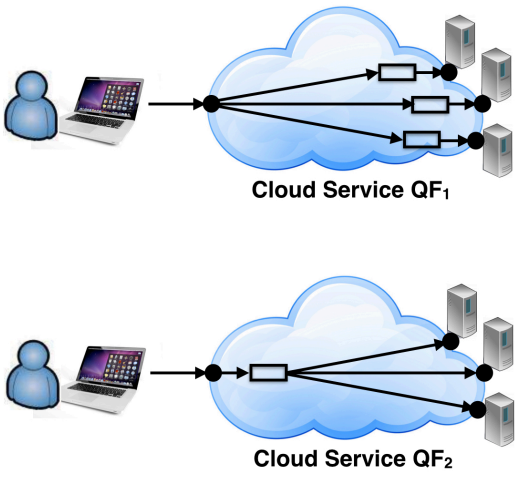

Fig. 2. Refinement of Query Flight Service

Example 2: Consider a cloud application where clients can check flight information and order tickets. If the client plan to make a trip between two places and make a query for the flight information, because there can be many flights between these two places provided by different companies, and the availability and price for each flight may change at any time, the client hope to collect the information for all available flights at the latest time. We first take the service $\mathbf{QF_1}$ into consideration. It has a buffer on the server side for each flight company, accepts query from the client and put a copy of the query into every buffer. We have its specification by design model as follows:

$$
\begin{aligned}
&\mathbf{QF}_1(in : A \mapsto \langle\alpha, a\rangle; out : (B_1 \mapsto \langle\beta_1, b_1\rangle, \\
&\qquad B_2 \mapsto \langle\beta_2, b_2\rangle, B_3 \mapsto \langle\beta_3, b_3\rangle)) \\
&\textbf{pre} : \mathcal{D}(\langle\alpha, a\rangle) \\
&\textbf{post} : \bigwedge_{1 \le i \le 3} (\mathcal{D}(\langle\beta_i, b_i\rangle) \wedge \beta_i = \alpha \wedge a < b_i < a')
\end{aligned}
$$

Then we can consider another cloud service $\mathbf{QF_2}$ in Figure 2, whose buffer's location is on the client's side, which is different from $\mathbf{QF_1}$. It can accept the query from the client as well, puts it into the buffer and sends a copy of the query to each flight company simultaneously. The trick here is the query waits in the buffer and be delivered to each server end simultaneously. The design model description of it is as follows:

$$
\begin{aligned}
&\mathbf{QF}_2(in : A \mapsto \langle\alpha, a\rangle; out : (B_1 \mapsto \langle\beta_1, b_1\rangle, \\
&\qquad B_2 \mapsto \langle\beta_2, b_2\rangle, B_3 \mapsto \langle\beta_3, b_3\rangle)) \\
&\textbf{pre} : \mathcal{D}(\langle\alpha, a\rangle) \\
&\textbf{post} : \bigwedge_{1 \le i \le 3} (\mathcal{D}(\langle\beta_i, b_i\rangle) \wedge \beta_i = \alpha) \wedge a < b_1 = b_2 = b_3 < a')
\end{aligned}
$$

From the two design models we can easily derive that $\mathbf{QF_1} \sqsubseteq \mathbf{QF_2}$ since their preconditions are equal while the postcondition for $\mathbf{QF_2}$ is stronger.

III. Composition of Cloud Services

Different cloud services can be composed together to build more complex services / applications. Since cloud services are interpreted as designs, their composition can be naturally modeled by composition on designs, which leads to a new design capturing the behavior of the composed cloud service / application. In this section, we introduce a family of composition patterns for two cloud services $\mathbf{C}_i (i = 1, 2)$:

$$
\begin{aligned}
&\mathbf{C}_i(in : in_{\mathbf{C}_i}; out : out_{\mathbf{C}_i}) \\
&\textbf{pre} : P_i(in_{\mathbf{C}_i}) \\
&\textbf{post} : Q_i(in_{\mathbf{C}_i}, out_{\mathbf{C}_i})
\end{aligned}
$$

In the following, we use P_i and Q_i instead of $P_i(in_{\mathbf{C}_i})$ and $Q_i(in_{\mathbf{C}_i}, out_{\mathbf{C}_i})$ for simplicity of expression when it is clear from the context.

A. Sequential composition

Suppose one output port O of $\mathbf{C}_1$ and one input port I of $\mathbf{C}_2$ can be joined together and the timed data stream that happen on O thus can be taken as the input on I for $\mathbf{C}_2$. After joining these two ports, what happened on O (and I)

will be hidden from outside, which can be specified by using existential quantification on the corresponding predicates. Let the output on O in $\mathbf{C}_1$ and the input on I in $\mathbf{C}_2$ be $O \mapsto \langle \delta_1, d_1 \rangle \in out_{\mathbf{C}_1}$ and $I \mapsto \langle \delta_2, d_2 \rangle \in in_{\mathbf{C}_2}$, respectively. Then the results cloud service by sequentially composing $\mathbf{C}_1$ and $\mathbf{C}_2$ is:

$$\begin{aligned} &\mathbf{C}_1;_{O \to I} \mathbf{C}_2(in : \bigcup_{i=1,2} in_{\mathbf{C}_i} \setminus \{I \mapsto \langle \delta_2, d_2 \rangle\}; \\ &\qquad out : \bigcup_{i=1,2} out_{\mathbf{C}_i} \setminus \{O \mapsto \langle \delta_1, d_1 \rangle\}) \\ &\mathbf{pre} : P_1 \wedge \neg(Q_{1\,\langle \delta_1, d_1 \rangle};_{\langle \delta_2, d_2 \rangle} \neg P_2) \\ &\mathbf{post} : Q_{1\,\langle \delta_1, d_1 \rangle};_{\langle \delta_2, d_2 \rangle} Q_2 \end{aligned}$$

where the sequential composition of predicates is defined as follows:

$$\begin{aligned} &P_{\langle \delta_1, d_1 \rangle};_{\langle \delta_2, d_2 \rangle} Q \\ &\equiv \exists \langle \delta, d \rangle . P[\langle \delta, d \rangle / \langle \delta_1, d_1 \rangle] \wedge Q[\langle \delta, d \rangle / \langle \delta_2, d_2 \rangle] \end{aligned}$$

For $in_{\mathbf{C}_i} : I_{\mathbf{C}_i} \to TDS, i = 1, \ldots, k$,

$$\bigcup_{i=1,\ldots,k} in_{\mathbf{C}_i} : \bigcup_{i=1,\ldots,k} I_{\mathbf{C}_i} \to TDS$$

is defined as:

$$\bigcup_{i=1,\ldots,k} in_{\mathbf{C}_i}(K) = in_{\mathbf{C}_j}(K) \text{ if } K \in I_{\mathbf{C}_j}.$$

And for an arbitrary port K,

$$\bigcup_{i=1,\ldots,k} in_{\mathbf{C}_i} \setminus \{K \mapsto \langle \delta, d \rangle\} = (\bigcup_{i=1,\ldots,k} in_{\mathbf{C}_i})_{\bigcup_{i=1,\ldots,k} I_{\mathbf{C}_i} \setminus K}$$

Definitions for union and subtraction on outputs are similar.

For a predicate P and a variable v in P, $P[u/v]$ is the predicate obtained by replacing all the occurrence of v in P by u. Note that when two cloud services $\mathbf{C}_1$ and $\mathbf{C}_2$ 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 $\mathbf{C}_1$ to all the input ports of $\mathbf{C}_2$. 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.

B. 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 $\mathbf{C}_1$ and $\mathbf{C}_2$, 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 $\mathbf{C}_1$, or on input ports of $\mathbf{C}_2$, which will trigger the corresponding cloud service $\mathbf{C}_1$ or $\mathbf{C}_2$, respectively, and produce the associated output on the corresponding output ports. Formally, the results cloud service as an external choice of $\mathbf{C}_1$ and $\mathbf{C}_2$ is defined as:

$$\begin{aligned} &\mathbf{C}_1 \,\square\, \mathbf{C}_2(in : \bigcup_{i=1,2} in_{\mathbf{C}_i}; out : \bigcup_{i=1,2} out_{\mathbf{C}_i}) \\ &\mathbf{pre} : P_1 \vee P_2 \\ &\mathbf{post} : (P_1 \Rightarrow Q_1) \wedge (P_2 \Rightarrow Q_2) \end{aligned}$$

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:

$$\begin{aligned} &\mathbf{C}_1 \sqcap \mathbf{C}_2(in : \bigcup_{i=1,2} in_{\mathbf{C}_i}; out : \bigcup_{i=1,2} out_{\mathbf{C}_i}) \\ &\mathbf{pre} : (in_{\mathbf{C}_1} \cap in_{\mathbf{C}_2} \neq \emptyset) \wedge P_1 \wedge P_2 \\ &\mathbf{post} : Q_1 \vee Q_2 \end{aligned}$$

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 design which means that if b is satisfied then the cloud service $\mathbf{C}_1$ is executed, and otherwise, $\mathbf{C}_2$ is executed:

$$\begin{aligned} &\mathbf{C}_1 \lhd b \rhd \mathbf{C}_2(in : \bigcup_{i=1,2} in_{\mathbf{C}_i}; out : \bigcup_{i=1,2} out_{\mathbf{C}_i}) \\ &\mathbf{pre} : P_1 \lhd b \rhd P_2 \\ &\mathbf{post} : Q_1 \lhd b \rhd Q_2 \end{aligned}$$

C. Parallel composition

After the study of the choice combinators we proceed to that of *parallel composition*. The simplest form of parallel combinator captures the case that both cloud services $\mathbf{C}_1$ and $\mathbf{C}_2$ are invoked and executed in parallel when triggered by a pair of inputs on the corresponding input ports of both $\mathbf{C}_1$ and $\mathbf{C}_2$. Therefore, to make it possible to execute the parallel combination of $\mathbf{C}_1$ and $\mathbf{C}_2$, both P_1 and P_2 should be satisfied and the execution will lead to the result that $Q_1 \wedge Q_2$.

$$\begin{aligned} &\mathbf{C}_1 \parallel \mathbf{C}_2(in : \bigcup_{i=1,2} in_{\mathbf{C}_i}; out : \bigcup_{i=1,2} out_{\mathbf{C}_i}) \\ &\mathbf{pre} : P_1 \wedge P_2 \\ &\mathbf{post} : Q_1 \wedge Q_2 \end{aligned}$$

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 $\mathbf{C}_1$ and $\mathbf{C}_2$ arrive only at the same time. 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 $\mathbf{C}_1$ and $\mathbf{C}_2$ respectively. And the data arriving at all the other input ports except I_1 and I_2 are independent. Furthermore, we assume that $I_i \mapsto \langle \delta_i, d_i \rangle \in in_{\mathbf{C}_i}$ for $i = 1, 2$. Then we have

$$\begin{aligned} &\mathbf{C}_1 \;{}_{I_1}\!\parallel_{I_2} \mathbf{C}_2(in : \bigcup_{i=1,2} in_{\mathbf{C}_i}; out : \bigcup_{i=1,2} out_{\mathbf{C}_i}) \\ &\mathbf{pre} : P_1 \wedge P_2 \wedge d_1 = d_2 \\ &\mathbf{post} : Q_1 \wedge Q_2 \end{aligned}$$

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 both parallel combinators $\parallel$ and ${}_{I_1}\!\parallel_{I_2}$ are very helpful. The definition of $\parallel$ and ${}_{I_1}\!\parallel_{I_2}$ can be easily generalized

to the case for composing multiple services and we will omit the technical details here.

A similar situation we consider is the case of merging two input ports of cloud services $\mathbf{C}_1$ and $\mathbf{C}_2$ into one port. Let $\langle \delta_i, d_i \rangle$ for $i = 1, 2$ be the timed data streams on the input port I_i in $\mathbf{C}_i$, 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 $\mathbf{C}_1$ and $\mathbf{C}_2$ to trigger their execution simultaneously. The definition of this operation is as follows:

$$\mathbf{C}_1 \|_{\{I_1, I_2\} \gg I} \mathbf{C}_2(in : (\bigcup_{i=1,2} in_{\mathbf{C}_i} \setminus \{I_i \mapsto \langle \delta_i, d_i \rangle\}) \cup \{I \mapsto \langle \delta, d \rangle\}; out : \bigcup_{i=1,2} out_{\mathbf{C}_i})$$

$$\mathbf{pre} : \bigwedge_{i=1,2} P_i[\langle \delta, d \rangle / \langle \delta_i, d_i \rangle]$$

$$\mathbf{post} : \bigwedge_{i=1,2} Q_i[\langle \delta, d \rangle / \langle \delta_i, d_i \rangle]$$

Based on the design model of cloud services, we can develop various (equivalence and refinement) laws for cloud services, especially for those composed by applying the combinators defined previously, and encode them as theorems to support a reasoning system in theorem provers like Coq or PVS. However, instead of proceeding further with such laws, we shall discuss about dynamic reconfiguration of cloud applications in the following section, which is a rather important topic for Cloud computing.

IV. Dynamic Reconfiguration

Dynamic reconfiguration is necessary in Cloud computing. For example, when we consider a cloud application that uses cloud services provided by different providers, where the services are usually not under our control, and thus cannot be reset in general. During the execution of such an application, it is not surprising that some service provider becomes unavailable or the QoS properties of some service becomes much worse, which might be unacceptable for the clients of the application. In such cases, a dynamic reconfiguration is certainly necessary and can be either very simple like switching to an alternative cloud service, or very complex like modifying the whole application architecture.

In our approach, we model dynamic reconfigurations of a cloud application using a family of reconfiguration rules on the corresponding design model. A reconfiguration rule captures a possible pattern that should be matched by the reconfiguration, and specifies how the application should be changed during the reconfiguration. It is obvious that we cannot provide a complete set of rules for all possible reconfiguration situations here. Instead, we only consider the basic reconfiguration situation for adding new services during the execution, which is rather simple but quite common in Cloud computing. Details about the operation are given in the following of this section.

Suppose we have a cloud application $\mathbf{S}$, and $\mathbf{C}_1$ is a service being used in this application. Another service $\mathbf{C}_2$ which behaves like $\mathbf{C}_1$ might be offered as another option for use when $\mathbf{C}_1$ is invoked during the execution of the application, while $\mathbf{C}_1$ is still available there. When the request to execute service $\mathbf{C}_1$ is coming, either $\mathbf{C}_1$ or $\mathbf{C}_2$ will be invoked and return the corresponding results. Such a situation is captured by the reconfiguration rule for adding service as another option to choose:

Rule 1: $\mathbf{S}[\![\mathbf{C}_1]\!] \rightsquigarrow \mathbf{S}[\![\mathbf{C}_1 \between \mathbf{C}_2]\!]$.

This rule means that the service $\mathbf{C}_1$ is replaced by $\mathbf{C}_1 \between \mathbf{C}_2$ in the application $\mathbf{S}$, while all the other parts in $\mathbf{S}$ are kept unchanged in the reconfiguration. Let $\mathbf{C}_i(in : (I_i \mapsto \langle \alpha_i, a_i \rangle); out : (O_i \mapsto \langle \beta_i, b_i \rangle))$, $i = 1, 2$, be two cloud services, then we have *:

$$\begin{aligned} &\mathbf{C}_1 \between \mathbf{C}_2 \\ =\ &\mathbf{Router};_{\{K_i \to I_i\}} (\mathbf{C}_1 \square \mathbf{C}_2);_{\{O_i \to J_i\}} \mathbf{Merger} \end{aligned}$$

where $\mathbf{Router}$ and $\mathbf{Merger}$ are defined as follows:

$$\mathbf{Router}(in : (I_1 \mapsto \langle \alpha, a \rangle); out : (K_1 \mapsto \langle \alpha_1, a_1 \rangle, K_2 \mapsto \langle \alpha_2, a_2 \rangle))$$

$$\mathbf{pre} : \mathcal{D}(\langle \alpha, a \rangle)$$

$$\mathbf{post} : (\bigwedge_{i=1,2} \mathcal{D}(\langle \alpha_i, a_i \rangle)) \wedge M(\langle \alpha_1, a_1 \rangle, \langle \alpha_2, a_2 \rangle, \langle \alpha, a \rangle)$$

and

$$\mathbf{Merger}(in : (J_1 \mapsto \langle \beta_1, b_1 \rangle, J_2 \mapsto \langle \beta_2, b_2 \rangle); out : (O_1 \mapsto \langle \beta, b \rangle))$$

$$\mathbf{pre} : \bigwedge_{i=1,2} \mathcal{D}(\langle \beta_i, b_i \rangle)$$

$$\mathbf{post} : \mathcal{D}(\langle \beta, b \rangle) \wedge M(\langle \beta_1, b_1 \rangle, \langle \beta_2, b_2 \rangle, \langle \beta, b \rangle)$$

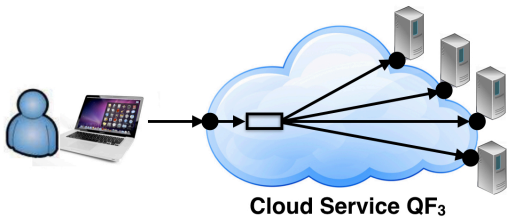

Fig. 3. Dynamic Reconfiguration of Query Flight Service

We can also add a service in parallel with another existing service. Just for a brief illustration, we can consider the flight query application depicted in Example 2. When the application is running, it is possible that some new flight company provide the flight query service for flights provided by this company, which is not available for the application at the beginning. In this case, the application should still work properly when the new service is added to the application and return the flight information for possible flights of all the companies, including the new one. Such a situation is captured by Figure 3, which specifies the result after reconfiguration on $\mathbf{QF}_2$.

*For simplicity of representation here we assume that both $\mathbf{C}_1$ and $\mathbf{C}_2$ have only one input port and one output port, while the definition can be easily generalized to cloud services with multiple input and output ports.

It is also possible that some services are becoming unavailable during the execution of a cloud application. In this case, we can have similar rules to remove a service from the application, which is either running in parallel with other services, or just as an option for choice.

In retrospect, the design model illustrated here seems promising either to capture known reconfiguration patterns, or to identify new ones, for a variety of cloud applications. A lot of work, however, remains to be done.

V. Conclusion

In this paper we define a comprehensive formal model for cloud services, their composition and dynamic reconfiguration under the UTP framework. Cloud services are interpreted as designs in UTP, which is a pair of predicates specifying the relationship between inputs and outputs. A number of combinators are defined corresponding to different ways of combining cloud services, and dynamic reconfiguration of cloud applications can be specified as rules for transforming between different designs.

Prospects for future work include the investigation of other features present in cloud computing, such as service discovery, coordination and negotiation, as well as the planning of significative case studies to assess empirically the merits of the approach proposed here by linking the model to concrete implementations. We are also interested in simulation of application behavior. In addition, another natural follow up of this paper concerns reasoning about cloud applications in theorem provers like PVS or Coq, extending our previous work on formal reasoning about component connectors (e.g., in [13]) to take into account behavior of services. Developing rules for composition and dynamic reconfiguration of cloud services is in our plan as well. In particular, we would like to tackle the consistency problem among different applications / services, and to explore the relationship between reconfiguration and other kinds of model transformation, such as architectural refinement [14].

Acknowledgements

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

References

[1] L. Aceto, K. G. Larsen, A. Morichetta, and F. Tiezzi. A Cost/Reward Method for Optimal Infinite Scheduling in Mobile Cloud Computing. In *Proceedings of FACS 2015*, volume 9539 of *LNCS*, pages 66–85. Springer, 2016.

[2] Z. Benzadri, F. Belala, and C. Bouanaka. Towards a Formal Model for Cloud Computing. In *ICSOC 2013 Workshops*, volume 8377 of *LNCS*, pages 381–393. Springer, 2014.

[3] R. Buyya, C. S. Yeo, S. Venugopal, J. Broberg, and I. Brandic. Cloud computing and emerging it platforms: Vision, hype, and reality for delivering computing as the 5th utility. *Future Generation Computer Systems*, 25:599–616, 2009.

[4] L. Chen, G. Fan, and Y. Liu. Modeling and analyzing cost-aware fault tolerant strategy for cloud application. In *Proceedings of SEKE 2016*, pages 439–442. KSI Research Inc. and Knowledge Systems Institute Graduate School, 2016.

[5] F. S. de Boer, R. Hähnle, E. B. Johnsen, R. Schlatte, and P. Y. H. Wong. Formal Modeling of Resource Management for Cloud Architectures: An Industrial Case Study. In *Proceedings of ESOCC 2012*, volume 7592 of *LNCS*, pages 91–106. Springer-Verlag, 2012.

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

[7] D. Fitch and H. Xu. A raid-based secure and fault-tolerant model for cloud information storage. *International Journal of Software Engineering and Knowledge Engineering*, 23(05):627–654, 2013.

[8] L. Freitas and P. Watson. Formalizing workflows partitioning over federated clouds: multi-level security and costs. *International Journal of Computer Mathematics*, 91(5):881–906, 2014.

[9] M. Graiet, A. Mammar, S. Boubaker, and W. Gaaloul. Towards Correct Cloud Resource Allocation in Business Processes. *IEEE Transactions on Services Computing*, 10(1):23–36, 2017.

[10] C. A. R. Hoare and J. He. *Unifying Theories of Programming*. Prentice Hall International, 1998.

[11] H. Jifeng, X. Li, and Z. Liu. rCOS: a Refinement Calculus of Object Systems. *Theoretical Computer Science*, 365(1-2):109–142, 2006.

[12] A. Jin, W. Song, P. Wang, D. Niyato, and P. Ju. Auction Mechanisms Toward Efficient Resource Sharing for Cloudlets in Mobile Cloud Computing. *IEEE Transactions on Services Computing*, 9(6):895–909, 2016.

[13] Y. Li and M. Sun. Modeling and Verification of Component Connectors in Coq. *Science of Computer Programming*, 113(3):285–301, 2015.

[14] S. Meng, L. S. Barbosa, and Z. Naixiao. On Refinement of Software Architectures. In *Proceedings of ICTAC'05*, volume 3722 of *LNCS*, pages 469–484. Springer, 2005.

[15] M. Oliveira, A. Cavalcanti, and J. Woodcock. A Denotational Semantics for Circus. *Electronic Notes in Theoretical Computer Science*, 187:107–123, 2007.

[16] X. Pei, H. Yu, and G. Fan. Achieving Efficient Access Control via XACML Policy in Cloud Computing. In *Proceedings of SEKE 2015*, pages 110–115. KSI Research Inc. and Knowledge Systems Institute Graduate School, 2015.

[17] S. Qin, J. S. Dong, and W. Chin. A Semantic Foundation for TCOZ in Unifying Theories of Programming. In *Proceedings of FME'03*, volume 2805 of *LNCS*, pages 321–340. Springer, 2003.

[18] H. Sahli, C. Bouanaka, and A. T. E. Dib. Towards a formal model for cloud computing elasticity. In *Proceedings of 2014 IEEE 23rd International WETICE Conference*, pages 359–364. IEEE Computer Society, 2014.

[19] K. M. Sim. Agent-based cloud computing. *IEEE Transactions on Services Computing*, 5(4):564–577, 2012.

[20] M. Sun, 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. *Science of Computer Programming*, 77(7-8):799–822, 2012.

[21] D. Weerasiri, M. C. Barukh, B. Benatallah, and J. Cao. A Model-Driven Framework for Interoperable Cloud Resources Management. In *Proceedings of ICSOC 2016*, volume 9936 of *LNCS*, pages 186–201. Springer, 2016.

[22] P. Zhang, C. Lin, X. Ma, F. Ren, and W. Li. Monitoring-Based Task Scheduling in Large-Scale SaaS Cloud. In *Proceedings of ICSOC 2016*, volume 9936 of *LNCS*, pages 140–156. Springer, 2016.