Received January 16, 2019, accepted February 13, 2019, date of publication February 22, 2019, date of current version March 12, 2019.

Digital Object Identifier 10.1109/ACCESS.2019.2900473

Toward Formal Modeling and Verification of Resource Provisioning as a Service in Cloud

WENBO ZHOU[1,2], LEI LIU[1,3], SHUAI LÜ[1,2,4], AND PENG ZHANG[1,2,4]
[1]College of Computer Science and Technology, Jilin University, Changchun 130012, China
[2]Key Laboratory of Symbolic Computation and Knowledge Engineering of the Ministry of Education, Jilin University, Changchun 130012, China
[3]College of Software, Jilin University, Changchun 130012, China
[4]College of Mathematics, Jilin University, Changchun 130012, China

Corresponding author: Peng Zhang (zhangpengccst@jlu.edu.cn)

This work was supported in part by the National Natural Science Foundation of China under Grant 61300049, in part by the China Postdoctoral Science Foundation under Grant 2016M591482 and Grant 2016M590254, in part by the Natural Science Research Foundation of Jilin Province of China under Grant 20180101053JC and Grant 20190201193JC, and in part by the Graduate Innovation Fund of Jilin University of China under Grant 101832018C025.

ABSTRACT Cloud service provides a convenient pattern of delivery and management for the Internet-based resource sharing. Reliable resource provisioning is very important to the stability of cloud service systems. In order to guarantee the consistency of resource delivery in cloud service, we propose a method for modeling and verification of resource provisioning as a service in the cloud. First, the framework of resource provisioning as a service and the behaviors of its participants are presented. Then, client, service manager (including allocator, finish monitor, and time monitor), and resource service are modeled based on UPPAAL, respectively. Finally, we define some consistency properties that a service scenario needs to satisfy and formally verify whether our model satisfies these properties using the UPPAAL model checker. The results show that our model satisfies all the proposed properties, which demonstrates the rationality and trustworthiness of our model.

INDEX TERMS Clouds, resource provisioning, Web services, UPPAAL, formal verification.

I. INTRODUCTION

Cloud computing leads to innovative patterns of software development and deployment [1]. As a key infrastructure supporting various applications, cloud computing provides convenience for the rapid development of emerging fields, such as big data, Internet of things and artificial intelligence [1]. The essential characteristics of cloud computing include on-demand self-service, broad network access, resource pooling and measured service [2], which ensures cloud brings efficient and convenient patterns of computation. Cloud computing delivers its capacities through services, i.e., it supplies clients with available resources according to requirements. Therefore, reliable control of resource provisioning is one of important aspects of cloud services.

There are 3 typical models in cloud services, i.e., Infrastructure as a Service (IaaS), Platform as a Service (PaaS) and Software as a Service (SaaS). IaaS permits clients to rent CPU, storage, network, and other infrastructures. Clients can install or deploy all their software on these infrastructures. PaaS provides clients with development environments, which frees clients from managing or configuring basic infrastructures. Clients are capable of developing or performing other platform-related activities directly. SaaS takes functions of various software as services. It allows people to use software through client or browser online. No matter which service model is chosen, some kinds of resources are provisioned. These resources may be virtual machines, databases, routers, balancers, web applications, etc. Therefore, most cloud services can be abstracted as ‘‘Resource Provisioning as a Service (RPaaS)’’, i.e., they give clients access to resources in the form of services.

In cloud, RPaaS needs to meet basic control requirements, such as real-time and isolation, so as to deal with problems about multi-tenancy and multi-service. Moreover, to avoid resource conflicts or deadlock, RPaaS should also do well in managing and coordinating resources. The above-mentioned facts require that RPaaS meet consistency, i.e., states that service participants reach should be consistent. If client, service manager and resource service do not reach consistent

The associate editor coordinating the review of this manuscript and approving it for publication was Liqun Fu.

states, there may be resources misused or even abandoned. Many researchers have studied consistency problems in cloud services, such as data consistency [3], storage consistency [4], [5], queuing service consistency [6] and replication consistency [7], [8]. However, they do not consider resource provisioning from the perspective of service, and are rarely concerned about consistency in the process of resource delivery.

The aim of this paper is to propose a model of resource provisioning as a service, guaranteeing consistency and correctness during resource delivery. On the one hand, we believe that a good RPaaS model should provide basic functions that resource delivery needs, including: 1) multiple pairs of clients and services can exist at the same time without affecting each other; 2) clients can abort requests for services when they receive no response for a long time; 3) there will be timeout handling when a client's usage time is longer than its request time. On the other hand, a RPaaS model should comply with specifications about consistency to ensure resource delivery is correct. For example, a resource should be active when a client is using it. Also, there should be some allocated resource when scheduling successfully.

Formal methods are techniques with mathematical foundations for specifying, developing and verifying computer software and hardware systems [9]. As a mature formal tool, UPPAAL can express real time, synchronization and other properties well, which makes it suitable for modeling and verification of state-sensitive control systems. Applying this tool to cloud services makes it easier to specify behaviors of service participants, improve integrated design of service systems, and verify functional or non-functional requirements automatically. By this means, UPPAAL can be used to solve consistency problems of resource delivery and improve reliability of service systems [10].

By combining resource with service, this paper proposes a view of "Resource Provisioning as a Service (RPaaS)". Basing on UPPAAL, we design a RPaaS model in cloud and verify its consistency. Firstly, a RPaaS framework is presented and the behaviors in this framework are illustrated. Then, UPPAAL templates of RPaaS objects (i.e., client, service manager and resource service) are defined from the perspective of service delivery. Finally, consistency specifications that RPaaS should meet are analyzed, and these properties of RPaaS are verified in UPPAAL model checker.

The remainder of this paper is organized as follows: Section 2 presents the RPaaS framework and discusses some scenarios. Section 3 details UPPAAL-based models of participants in RPaaS. Section 4 describes consistency specifications and verifies these properties. Section 5 reviews related work. Finally, Section 6 concludes the paper and outlines the future work.

II. RPaaS FRAMEWORK

"Resource Provisioning as a Service" refers to providing clients with access to resources on demand in the form of services. Figure 1 shows the RPaaS framework, involving 3 participants, i.e., client, service manager and resource service. A client performs a series of actions to obtain the resource it needs. The service manager coordinates the interaction between clients and services, receives requests from clients, assigns suitable resource services to clients and monitors the process of resource services. Resource services provide corresponding resources to clients according to certain steps.

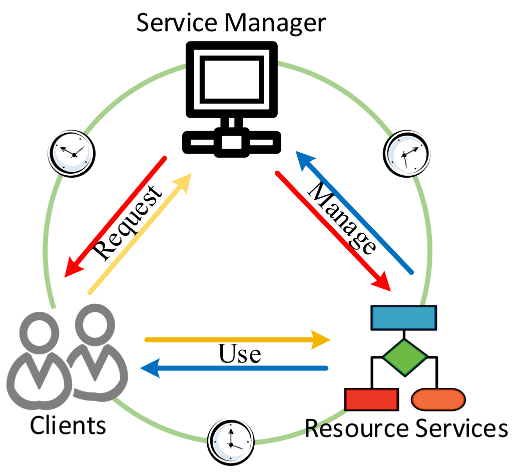


FIGURE 1. RPaaS framework.

Figure 2 illustrates a sample scenario that a client accesses to the resource it needs successfully. C1, A1 and S1 denote behaviors relevant to the 1st client, while C2, A2 and S2 denote behaviors relevant to the 2nd client. Here, we take the 1st client as an example to explain how a service is completed. Client 1 applies to Service Manager for the resource it needs (C1.1). Service Manager firstly chooses Client 1's request from the request queue. Then, it checks whether there are released resource services to be collected. If all the released resources have been collected, Service Manager schedules and informs the corresponding resource (A1.2). After that, Service Manager informs Client 1 that the resource service that it needs is allocated successfully (A1.3). Before using the resource, Client 1 should connect to its resource service first (C1.2). When connecting to a resource service successfully, Client 1 can start and stop it repeatedly (C1.3 and C1.4). Limited by space, Figure 2 only describes one start behavior and one stop behavior. Finally, Client 1 releases its resource service (C1.5) and Service Manager is informed that the service process has finished (S1.1). Similarly, the resource service relevant to Client 2 will be competed. In this case, multiple pairs of clients and services can interact with each other at the same time normally, which accords with the 1st basic function.

We further divide Service Manager into 3 components, i.e., Allocator, Finish Monitor and Time Monitor. Allocator is used to allocate resource services to clients according to their requests. The processing of a client's request is atomic, which means that Allocator can only processes clients' requests one

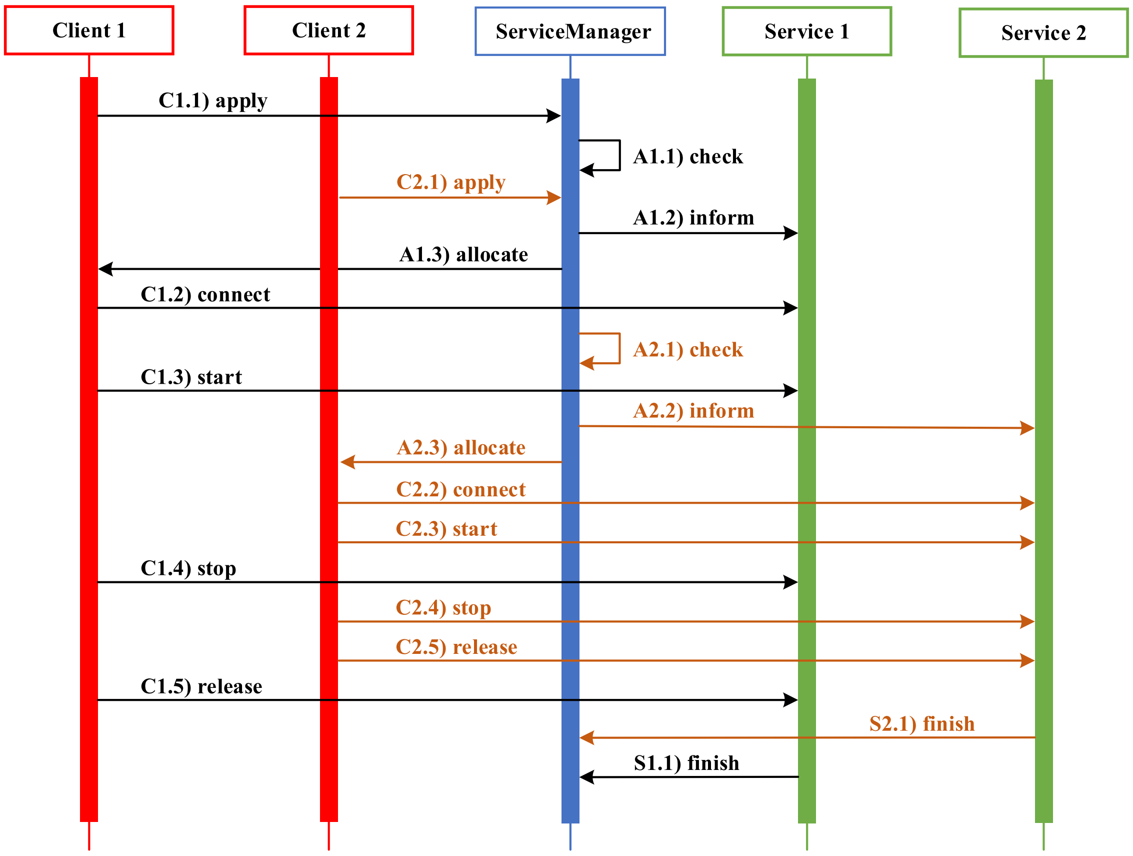


FIGURE 2. Normal service scenario.

by one. Finish Monitor is used to check whether there are some allocated resource services that have finished. Time Monitor is used to check whether there are timeout cases. The 3 components cooperate with each other to provide the functions of on-demand allocation, finishing control and timeout handling.

When there is no response to a client's request for a long time, the client can abort it. In this case, there may be no resource service allocated to the client, or there is one has been allocated to the client. If Service Manager has not allocated a resource service to the client, it is easy to directly abort the service process. Otherwise, the allocated resource service should be canceled. Figure 3 describes that a client aborts its request when a resource service has been allocated to the client. Similar to those in Figure 2, C.1, A.1 and A.2 represent *apply*, *check* and *inform*, respectively. When Client sends the *abort* command, Allocator cancels the resource service (i.e., Service) that is allocated to Client (A.3). This makes it possible that clients abort requests for services when they receive no response for a long time, which accords with the 2nd basic function. If there is no resource meets a client's requirement, the primitive *mismatch* will make Client and Allocator return to their initial states. This scenario is simpler than the client aborting scenario, so no more discussion about it will be presented here.

A client's usage time of a resource should not be longer than its request time. Otherwise timeout handling will occur. Here, timeout handling refers to terminating a client's use of its resource service. When a resource service is allocated to a client, a timer for recording usage time is started. Timeout handling may occur in the following cases: 1) the client is allowed to access to a resource service but has not connected to it; 2) the client has connected to its resource service but has not started it; 3) the client has started its resource service. If timeout handling occurs in any of the above cases, TimeMonitor will inform the corresponding client and resource service (T.1.1 and T.1.2) and terminate the service process forcibly (T.2.1 and T.2.2), which accords with the 3rd basic function.

In this section, the RPaaS framework is presented and its basic functions are discussed. In the next section, we model each participant of RPaaS using UPPAAL.

III. MODELING RPaaS IN UPPAAL

UPPAAL is an integrated tool environment developed in collaboration between Uppsala University and Aalborg University. It models systems as networks of timed automata and verifies them basing on computation tree logic [11]. In UPPAAL, a system is depicted as a series of templates, where each template defines a kind of timed automata that

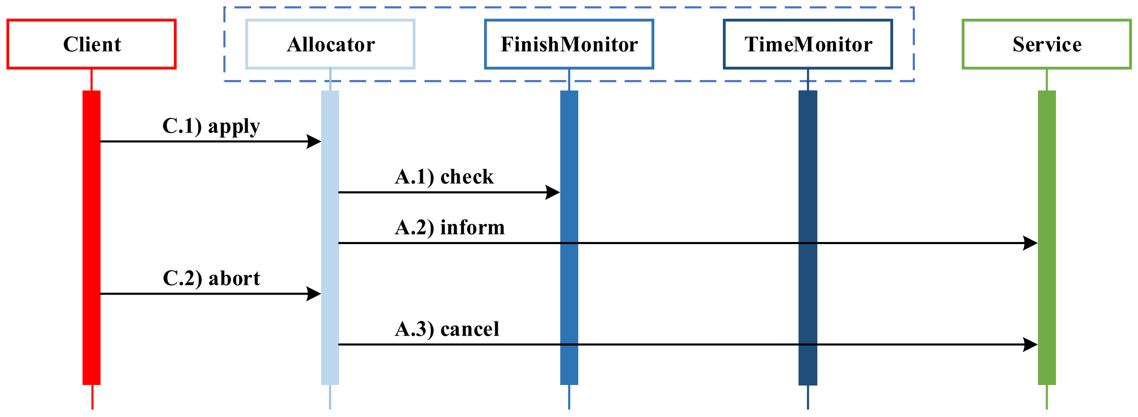


FIGURE 3. Client aborting scenario.

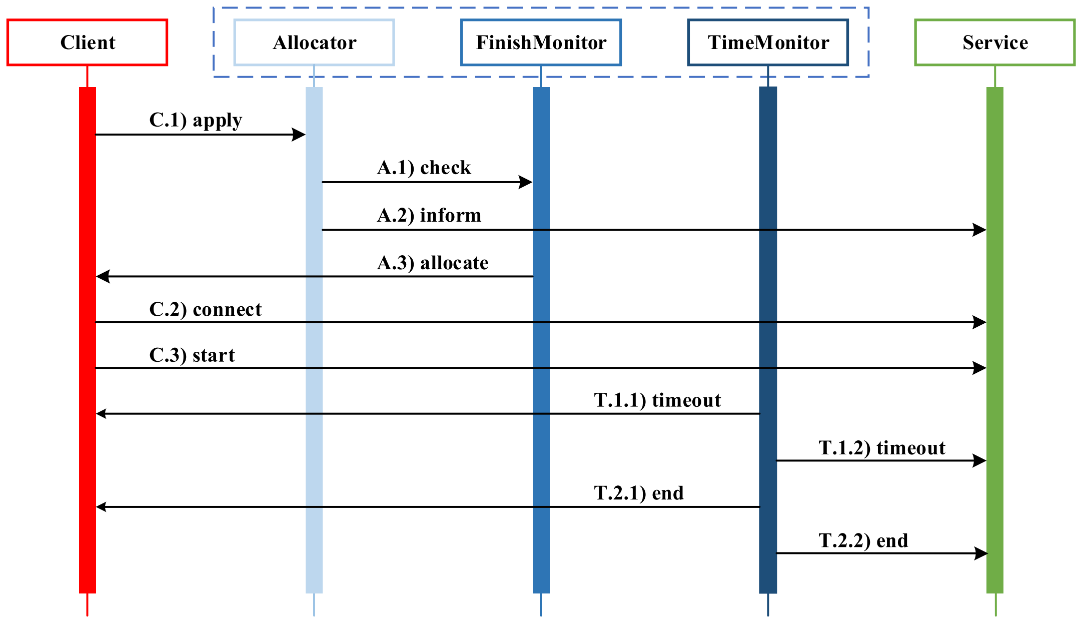


FIGURE 4. Timeout scenario.

describes behaviors of the system. UPPAAL can run the system model in its simulator and verify some properties using its model checker. We use UPPAAL to model each participant of RPaaS, where the client model simulates actions of clients, the service manager coordinates the interaction and management among clients and resource services, and the resource service model controls the service process. The 3 participants cooperate with each other to achieve resource delivery in RPaaS.

A. BASIC DEFINITIONS

We first give several definitions relevant to RPaaS. These definitions are implemented as some data structures when defining UPPAAL templates. The set of client identifiers is denoted as Φ, and the set of resource identifiers is denoted as Ψ. The definitions of resource, client request, resource state record and client-service bind record are as follows.

Definition 1 (Resource): A resource is a 4-tuple, i.e., $R = (rid, rtype, occupied, avltime)$, where:

(1) *rid* denotes a resource identifier;
(2) *rtype* denotes the type of resource;
(3) $occupied \in Bool$ denotes whether the resource is occupied;
(4) *avlTime* denotes how long the resource can be used.

A resource has 4 attributes. *rid* uniquely identifies a resource entity, while *rtype* indicates the type of resource entity. Since resources are allocated dynamically during a service process, we need to record whether a resource is occupied, i.e., the 3rd attribute of resource. In this paper, the usage time of a resource is limited and we use *avlTime* to denote the available time of a resource. We focus on the service process of

RPaaS and consider multiple relevant resources that a client requests as a whole. Technologies for resource composition and orchestration are out of the scope of this paper.

Definition 2 (Client Request): A client request is a 3-tuple, i.e., $C = (cid, ctype, dur)$, where:

(1) cid denotes a client identifier;

(2) $ctype$ denotes the type of the resource requested;

(3) dur denotes how long the client requires to use the resource.

Client request has 3 attributes. Similar to resource identifier, client identifier uniquely identifies a client. The 1st attribute indicates which of the clients proposes the request. The 2nd and 3rd attributes are type and usage time of the requested resource, respectively. Next, we give definitions about resource state record and client-service bind record.

Definition 3 (Resource State Record): The resource state record is a set of pairs, i.e., $\rho \ =\{(rid,\ state) \mid rid{\in}\ \Psi{\wedge}state{\in}\{\text{idle, allocated, available, active, released}\}\}$ that satisfies $\forall(rid_1, state_1), (rid_2, state_2) \in \rho.(rid_1 \neq rid_2)$, where rid, rid_1 and rid_2 are all resource identifiers.

Resource state record is used to preserve the state of every resource during the service process. The pair $(rid,\ state)$ is called a state record of rid. The above constraint indicates that there is only one state record for every resource. Initially, the state of a resource is ''idle''. When the resource is allocated to a client, its state changes to ''allocated''. After the client connects to the resource, its state is updated to ''available''. If the client starts to use the resource, the state is set to ''active''. The client may release the resource, which makes the state change to ''released''. Finally, the state returns to ''idle'' when the released resource is collected.

Definition 4 (Client-Service Bind Record): The client-service bind record is a set of pairs, i.e., $\sigma \ = \{(cid,\ rid) \mid cid{\in}\ \Phi{\wedge}rid{\in}\ \Psi\}$ that satisfies $\forall(cid_1,\ rid_1),\ (cid_2,\ rid_2) \in \sigma.((cid_1 \neq cid_2){\wedge}(rid_1 \neq rid_2))$, where cid is a client identifier and rid is a resource identifier.

Client-service bind record identifies binding relationships between clients and resources. The pair $(cid,\ rid)$ is called a bind record. The assumed constraint indicates that each client can only request one resource every time (i.e., multiple resources can be composited into one compound resource) and each resource can only be allocated to one client (i.e., resource is not shareable).

We also need 3 auxiliary queues denoted as *qqueue*, *lqueue* and *tqueue*. 1) *qqueue* is used to record all the client requests. Each element of *qqueue* is a client request, i.e., $(cid,\ ctype,\ dur)$. 2) *lqueue* is used to record the identifiers of released resources that need to be collected. Each element of *lqueue* is a bind record i.e., $(cid,\ rid)$. 3) *tqueue* is used to record the identifiers of allocated resources. Each element of *tqueue* is also a bind record.

Basing on the above definitions, we can further model client, service manager and resource service using UPPAAL. In the next sections, UPPAAL templates of participants in RPaaS are presented.

B. MODELING CLIENT

Client model depicts expected behaviors of clients in Figure 5. When a client is in the ''initial'' state, it can apply to the service manager for a resource service. The detail actions include putting the client request into *qqueue*, synchronizing the client and the service manager, and setting the clock variable x to 0. If the request time is not longer than that the client can tolerate, the client synchronizes with the allocator by synchronous variable *allocate*[*cid*], i.e., allocator assigns a suitable resource service to the client. Otherwise, the client aborts its request and returns to the ''initial'' state. After a resource service is assigned to the client, the client connects to the resource service.

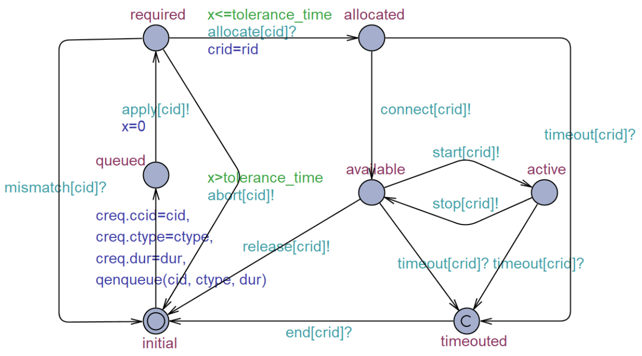


FIGURE 5. **UPPAAL template of client.**

If the connection is successful, the client reaches the ''available'' state. The client can switch between the ''available'' and ''active'' states by sending commands called *start* or *stop*. When the client's usage time is longer than its request time, the client is forced to leave its current state. In the meanwhile, timeout handling is completed by synchronous variables *timeout*[*crid*] and *end*[*crid*]. The client in the ''available'' state can also release its resource service. If there is no resource service that satisfies the client's request, the client returns to the ''initial'' state by synchronous variable *mismatch*[*cid*].

C. MODELING SERVICE MANAGER

Service manager model is the control center for the interaction between clients and resource services. Service manager model consists of allocator, finish monitor and time monitor, where allocator assigns suitable resource services to clients, finish monitor collects resource services released by clients, and time monitor manages timeout cases.

1) ALLOCATOR

Figure 6 presents the UPPAAL template of allocator. Before beginning all the activities, allocator needs to be initialized, i.e., variables are assigned initial values. Firstly, allocator selects a client request from *qqueue* and synchronizes with the client by *apply*[*cid*]. Before assigning a resource service, allocator should check whether there are released resources that have not been collected. If so, these resources will be

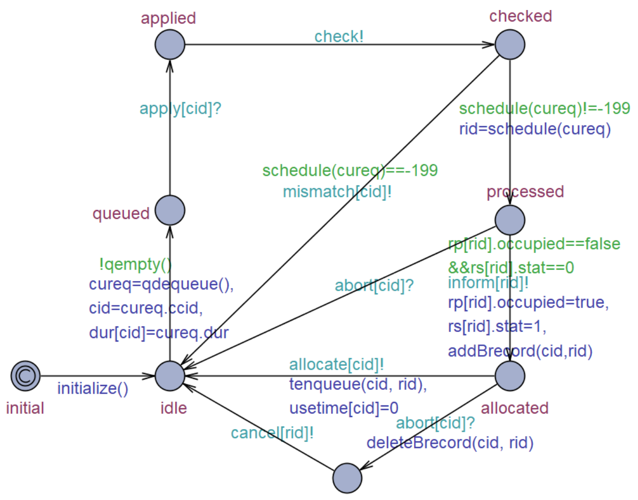


FIGURE 6. UPPAAL template of allocator.

collected. Then, allocator schedules a resource service satisfying this request and changes its state to ''processed''. Here, we use the function *schedule* to denote the scheduling process. Actually, this function can be implemented according to different scheduling strategies. If there is no resource service that satisfies the selected client request, allocator returns to the ''idle'' state by synchronous variable *mismatch*[*cid*]. Otherwise, allocator reaches the ''processed'' state as we just said.

When allocator is in the ''processed'' state, it will inform the scheduled resource service and change its state to ''allocated''. At the same time, a bind record is added to client-service bind record. This bind record is a pair of identifiers, where one identifier denotes the client making this request, and the other identifier denotes the resource scheduled to satisfy the request. This action indicates that the service manager has authorized the client to use the scheduled resource. Finally, allocator informs the client by synchronous variable *allocate*[*cid*]. In the meantime, the above bind record is added to *tqueue* and the clock variable *usetime*[*cid*] for this bind is set to 0 (i.e., timing begins). When the request time is longer than that the client can tolerate, allocator may be in the ''processed'' or ''allocated'' state: if allocator is in the ''processed'' state, it directly aborts this request and synchronizes with the client by synchronous variable ''*abort*[*cid*]''; otherwise, besides aborting the request, allocator also needs to delete the corresponding bind record and cancel the scheduled resource to guarantee consistency.

2) FINISH MONITOR

As shown in Figure 7, finish monitor is used to control check synchronization and finish synchronization. If *lqueue* is empty, all the released resource services have been collected. Otherwise, finish monitor collects released resource services by synchronous variable *finish*[*frid*] and deletes its corresponding bind record. Here, ''collect'' means that the state of a resource service is reset to ''idle'' by the

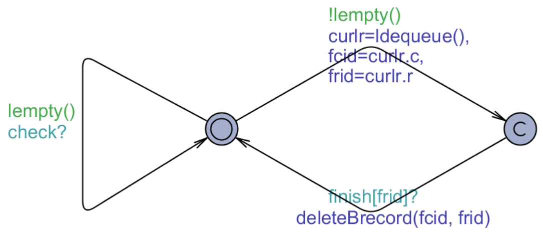


FIGURE 7. UPPAAL template of finish monitor.

synchronization between service manager and resource service, which permits this resource service to be assigned to other clients.

3) TIME MONITOR

As shown in Figure 8, time monitor handles timeout cases by checking usage time of each resource service. Once a resource service is allocated, its corresponding bind record will be added to *tqueue* and the timer for recording usage time is started. If *tqueue* is not empty, time monitor judges whether there is a case that a client's usage time is longer than its request time. If so, time monitor informs the relevant client and resource service by synchronous variable *timeout*[*trid*] and deletes the bind record. The bind records that clients' usage time is not longer than their request time are still in *tqueue*.

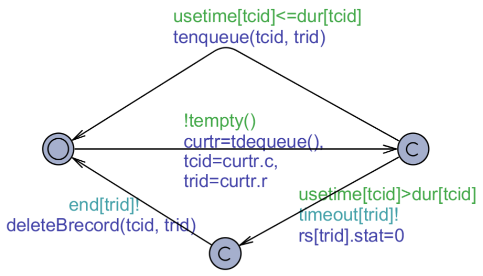


FIGURE 8. UPPAAL template of time monitor.

D. MODELING RESOURCE SERVICE

Figure 9 presents the internal control process of a resource service. When a resource service is informed by the service manager, it changes to the ''allocated'' state. Then, if the connection between this resource service and a client is successful, this resource service changes to the ''available'' state. Controlled by *start* and *stop* commands, resource service switches between the ''available'' and ''active'' state. ''available'' means that a client has connected to but not run this resource service. ''active'' means that a client is running this resource service. When the resource service is in the ''available'' state, its client can release it by synchronous variable *release*[*rid*], changing the 3rd attribute of the resource from true to false (i.e., occupied to unoccupied) and adding the corresponding bind record to *lqueue*.

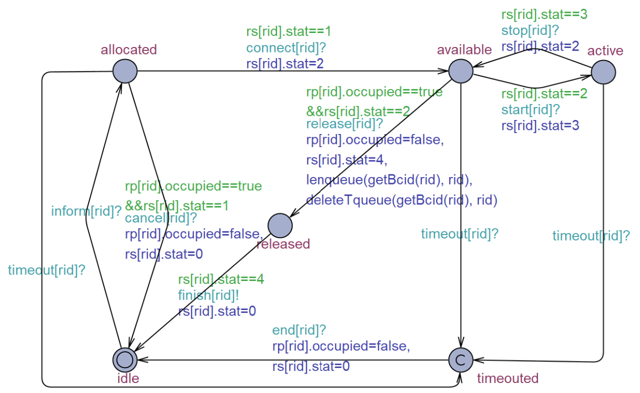


FIGURE 9. UPPAAL template of resource service.

Releasing synchronization indicates that a client ends the use of a resource service. Therefore, the corresponding bind record should be removed from *tqueue*. After released, the resource service should also inform the service manager, so as to be collected by synchronous variable *finish*[*rid*].

If the resource service is allocated but its client aborts it, the service manager cancels this resource service and resets it to "idle". The corresponding resource also becomes unoccupied. When timeout happens, the resource service will be forced to reach the "idle" state and its corresponding resource will also become unoccupied.

We model client, service manager (including allocator, finish monitor and time monitor) and resource service by defining a UPPAAL template for each of these 5 objects. During the state transitions, the records are maintained, providing useful information for further property analysis and process control. By instantiating UPPAAL templates of clients and resource services, we can manage and control multi-client and multi-service RPaaS.

IV. VERIFYING CONSISTENCY OF RPaaS

The consistency of behaviors is very important to a service system. In order to demonstrate the correctness of our model, we verify some properties of RPaaS model basing on UPPAAL tools. Firstly, we describe these properties using a specification language. Then, we verify whether RPaaS model satisfies them in UPPAAL model checker. The experiment is carried out in a PC with Intel(R) Core(TM) i7-4790 CPU 3.60GHz, Java 1.8.0_191 and UPPAAL 4.1.19.

Our verification is based on the following instance of RPaaS model, which involves 3 clients and 3 resource services. The attributes of these client requests and resources are shown in Table 1 and Table 2. We just discuss the properties and verification of this instance, since instances with more clients and resource services are similar to it.

A. FORMAL SPECIFICATIONS

RPaaS model should satisfy consistency. We analyzed some researches about cloud service models [12]–[15] and extracted 10 properties about consistency of services.

TABLE 1. Attributes of client requests.

cid	ctype	dur
c1	t1	90
c2	t1	75
c3	t2	150

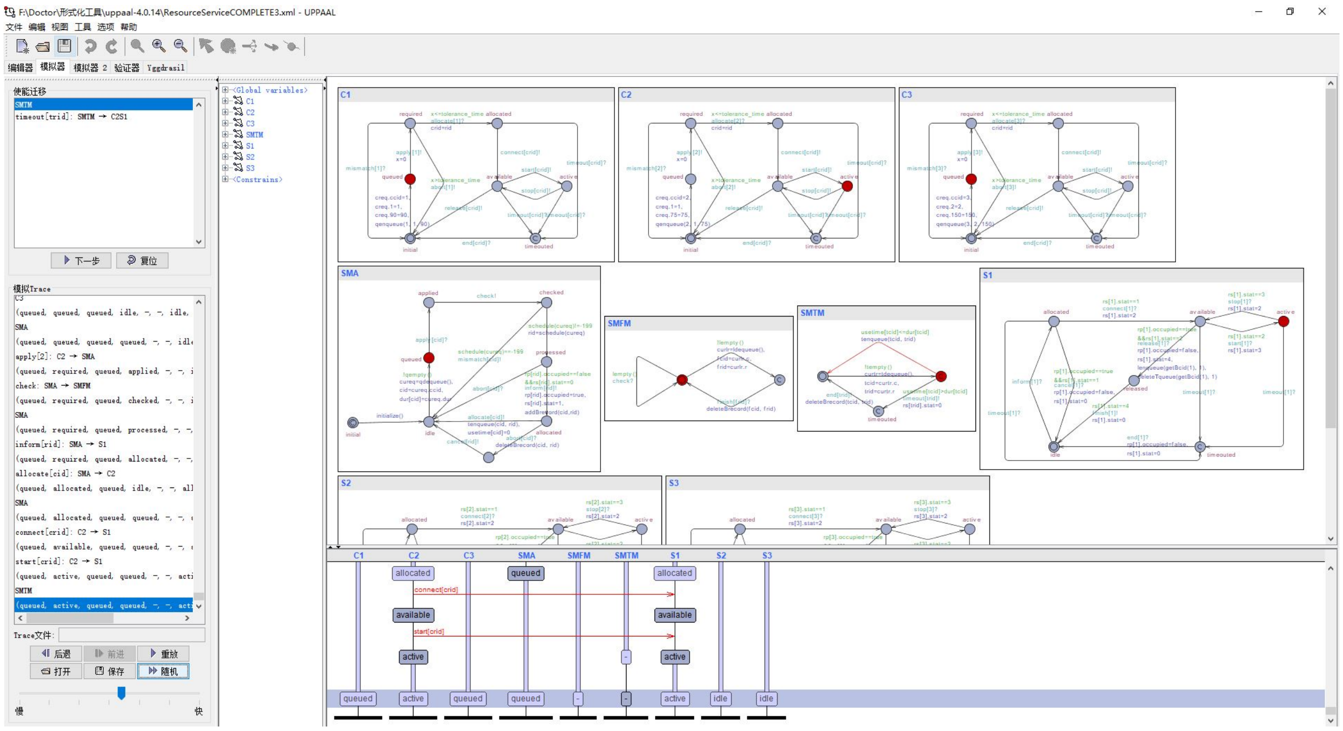
TABLE 2. Attributes of resources.

rid	rtype	occupied	avlTime
r1	t1	false	100
r2	t2	false	200
r3	t1	false	160

We analyze consistency of our model by checking the satisfaction of these properties. These 10 consistency properties are as follows.

1) A[] not (C1.active and not S1.active and not S2.active and not S3.active), the same thing is true with respect to C2 and C3
2) A[] not (C1.available and not (S1.available or S2.available or S3.available)), the same thing is true with respect to C2 and C3
3) A[] not (C1.initial and C2.initial and C3.initial and (S1.available or S2.available or S3.available))
4) A[] not (C1.initial and C2.initial and C3.initial and (S1.active or S2.active or S3.active))
5) A[] S1.idle imply (forall (i: int[0, RLEN-1]) cr[i].r != 1), the same thing is true with respect to S2 and S3
6) A[] SMA.allocated imply (S1.allocated or S2.allocated or S3.allocated)
7) A[] SMTM.timeouted imply (C1.timeouted or C2.timeouted or C3.timeouted) and (S1.timeouted or S2.timeouted or S3.timeouted)
8) A<> C1.allocated imply exists (i: int[0, RLEN-1]) (cr[i].c==1) and (C1.creq.ctype==rp[cr[i].r].type) and (C1.creq.dur<=rp[cr[i].r].avltime), the same thing is true with respect to C2 and C3
9) A[] not deadlock
10) A[] not (SMA.checked and S1.released and S2.released and S3.released)

Property 1) indicates that there must be some resource service in the "active" state when a client is using a resource service. Property 2) indicates that there must be some resource service in the "available" state when a client has been allocated a resource service. Property 3) indicates that no resource service is in the "available" state when all the clients are in the "initial" state. Property 4) indicates that no resource service is in the "active" state when all the clients are in the "initial" state. Property 5) indicates that if a resource service is in the "idle" state, there is no bind record about it in client-service bind record, i.e., the state of a resource service is consistent with client-service bind record. Property 6) indicates that there must be some resource service has been allocated after allocator successfully schedules one and reaches to the


F:\Doctor\形式化工具\uppaal-4.0.14\ResourceServiceCOMPLETE3.xml - UPPAAL
文件 编辑 视图 工具 选项 帮助
编辑器 模拟器 模拟器 2 验证器 Yggdrasil
使能迁移
下一步
复位
模拟Trace
Trace文件:
后退
前进
重放
打开
保存
随机
慢
快
C1
C2
C3
SMA
SMFM
SMTM
S1
S2
S3

```
SMTM
timeout[trid]: SMTM → C2S1
```

```
C3
(queued, queued, queued, idle, –, –, idle,
SMA
(queued, queued, queued, queued, –, –, idl
apply[2]: C2 → SMA
(queued, required, queued, applied, –, –, i
check: SMA → SMFM
(queued, required, queued, checked, –, –, i
SMA
(queued, required, queued, processed, –, –,
inform[rid]: SMA → S1
(queued, required, queued, allocated, –, –,
allocate[cid]: SMA → C2
(queued, allocated, queued, idle, –, –, all
SMA
(queued, allocated, queued, queued, –, –, 
connect[crid]: C2 → S1
(queued, available, queued, queued, –, –, 
start[crid]: C2 → S1
(queued, active, queued, queued, –, –, acti
SMTM
(queued, active, queued, queued, –, –, acti
```

FIGURE 10. UPPAAL simulation.

TABLE 3. Results of the verification.

No.	Verification time/Kernel time/Total time (s)	Resident memory/Virtual memory peak (KB)	Results
1	56.969 / 0.171 / 57.178	495,304 / 1,001,284	Satisfied
2	24.687 / 0 / 24.716	495,316 / 1,001,296	Satisfied
3	24.61 / 0 / 24.614	495,316 / 1,001,296	Satisfied
4	24.625 / 0 / 24.64	495,320 / 1,001,300	Satisfied
5	25 / 0 / 25.032	495,324 / 1,001,304	Satisfied
6	24.62 / 0 / 24.66	495,324 / 1,001,304	Satisfied
7	24.797 / 0 / 24.816	495,328 / 1,001,308	Satisfied
8	0.36 / 0.063 / 0.422	495,332 / 1,001,316	Satisfied
9	105.5 / 0.25 / 105.784	497,100 / 1,006,324	Satisfied
10	56.094 / 0.188 / 56.311	496,008 / 1,002,744	Satisfied

''allocated'' state. Property 7) indicates that when there is a timeout case, the relevant client and resource service will change to the ''timeouted'' state. Property 8) indicates that once a client is allocated a resource, there is a bind record that conforms to the client request. Property 9) indicates that there is no deadlock. Finally, property 10) indicates that all the released resource services have been collected before allocator begins to schedule a new resource service.

B. FORMAL VERIFICATION

UPPAAL model checker uses on-the-fly searching technique and symbolic technique to search state space and checks whether a property is satisfied by a system description [11]. Figure 10 presents a scenario of our instance produced by UPPAAL simulator.

As shown in Table 3, we verified all the properties given in the last section. Our model satisfies all the 10 consistency properties, which indicates that the proposed model is reliable to some extent. The 2nd and 3rd columns of Table 3 are about time and space overhead, respectively. The most time overhead is 105.5s and the average space overhead is about 495,567/1,001, 947KB, which is acceptable in the normal case.

Actually, instances with more clients and resource services are similar to this instance. Therefore, our model can be applied to control of multiple-client and multiple-service RPaaS. Moreover, the scheduling process in our model is regarded as a separate module, providing possibility for further verification of different scheduling strategies.

V. RELATED WORK

Formal modeling and verification of cloud services is an interesting topic, which attracts attention in academics recently. We review researches about formal models of cloud service, formal verification of cloud resource management and applications of UPPAAL tool as follows.

Formal models of cloud services provide good basis for further analysis and verification, contributing to the improvement of security and quality of service. Sun and Fu [16] used Unifying Theories of Programming (UTP) to provide a formal model for cloud computing. They interpreted cloud services as designs in UTP and discussed composition and dynamic reconfiguration of cloud services [16]. Newcombe *et al.* [17] introduced how Amazon applies formal methods (particularly, TLA+) to the design of complex real-world software, including public cloud services. They emphasized that formal methods can find bugs that cannot be found through other technique and bring good return on investment [17]. Klai and Ochi [18] used Symbolic Observation Graphs (SOG) to abstract cloud services and check the correction of their composition with respect to event- and state-based LTL formulae (Hybrid LTL). Rezaee *et al.* [19] proposed the Fuzzy Inference Cloud Service (FICS) modeled using the CSP process algebra and introduced four formal verification tests to allow strict analysis of certain behavioral disciplines in the FICS. The above work provides good solutions for modeling, design and verification of cloud services. However, they do not consider related effects of resources.

The reliable management of cloud resources is crucial to a trustworthy cloud computing system. Weerasiri *et al.* [20] provided a unified and comprehensive analysis framework to accelerate fundamental understanding of cloud resource orchestration in terms of concepts, paradigms, languages, models, and tools. Graiet *et al.* [21] proposed a formalism based on the Event-B language for specifying cloud resource allocation policies in business process models, which can be used to check the consistency and correctness of cloud resource allocation according to user requirements and resource capabilities. Jlassi *et al.* [22] developed a Cloud Resources Allocation Model (CRAM4FOSS) for FOSS applications using the Event-B method, and formally verified the resource allocation behaviors deployed in a cloud environment. These researches contribute to the understanding and correctness of cloud resource management. Compared with them, our model provides the point of studying resource provisioning and delivery from the perspective of service.

UPPAAL is a useful tool for analysis and verification of real-time and distributed systems. Zhou *et al.* [23] studied two kinds of evolution scenarios and proposed a novel verification approach based on hierarchical timed automata to model check dynamically evolvable service oriented systems. Meng *et al.* [24] used UPPAAL to model and verify a robot joint bus communication system, and they also proposed a dynamic priority strategy for reducing the worst arbitration delay. Balasubramaniyan *et al.* [25] presented a methodology to design and verify CPS using multi-objective evolutionary optimization, model checking and supporting software tools. They verified properties such as safety using UPPAAL model checker [25]. Kartal *et al.* [26] developed the algorithm TUConvert for converting distributed Timed Input/Output Automata (TIOA) models to UPPAAL behavioral models, formally proved its correctness and demonstrated its applicability by the formal verification of a distributed real-time industrial communication protocol. Sultana and Arif [27] provided translation rules that automatically transforms C++ codes into UPPAAL's automata and explored a case study of traffic light system. The objects of these researches are general real-time or service-oriented systems. We focus on the application of UPPAAL to cloud services, particularly the verification of RPaaS.

VI. CONCLUSION

Our work is one of the attempts that apply the techniques of formal verification to cloud services. We proposed a UPPAAL-based method for modeling and verifying RPaaS. The RPaaS framework mainly includes client, service manager and resource service, where service manager can be further divided into allocator, finish monitor and time monitor. The control of multiple-client resource delivery is implemented by state transitions and synchronous actions of the above 5 objects. We modeled these objects using UPPAAL, defined consistency properties and verified whether our model satisfies these properties. The results of the verification show that our RPaaS model satisfies all the proposed properties, demonstrating the rationality and reliability of this model.

In our current model, we only focus on basic requirements of clients, i.e., types and available time of resources. More complex attributes, such as priority and quality of service, should be taken into consideration. For example, we can set priorities by adjusting the order of client requests in a queue. Moreover, we only implemented a basic resource scheduling strategy as a simple function now. In our future work, we intend to study how to verify more complicated resource scheduling strategies basing on our model.

REFERENCES

[1] *Republic of China, A Three-year Action Plan for the Development of Cloud Computing (2017-2019)*. Accessed: Dec. 28, 2018. [Online]. Available: http://www.miit.gov.cn/n1146290/n4388791/c5570594/content.html

[2] P. Mell and T. Grance. (2011). *The NIST Definition of Cloud Computing*. Accessed: Dec. 28, 2018. [Online]. Available: https://www.nist.gov/publications/nist-definition-cloud-computing?pub_id=909616

[3] I. Fetai and H. Schuldt, "Cost-based data consistency in a data-as-a-service cloud environment," in *Proc. IEEE 5th Int. Conf. Cloud Comput.*, Jun. 2012, pp. 526–533.

[4] J. Sun, C. Hu, T. Wo, L. Du, and S. Yang, "HCFS2: A file storage service with weak consistency in the hybrid cloud," in *Proc. IEEE Symp. Service-Oriented Syst. Eng.*, Mar. 2018, pp. 228–233.

[5] B. H. Kim and D. Lie, "Caelus: Verifying the consistency of cloud services with battery-powered devices," in *Proc. IEEE Symp. Secur. Privacy*, Aug. 2015, pp. 880–896.

[6] Z. Zhang, Y. Wang, H. Chen, M. Kim, J. M. Xu, and H. Lei, "A cloud queuing service with strong consistency and high availability," *IBM J. Res. Dev.*, vol. 55, no. 6, p. 10:1, Oct. 2011.

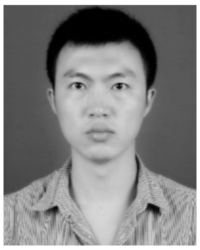
[7] T. Chen, R. Bahsoon, and A.-R. H. Tawil, "Scalable service-oriented replication with flexible consistency guarantee in the cloud," *Inf. Sci.*, vol. 264, pp. 349–370, Apr. 2014.

[8] Q. Liu, G. Wang, and J. Wu, "Consistency as a service: Auditing cloud consistency," *IEEE Trans. Netw. Service Manage.*, vol. 11, no. 1, pp. 25–35, Mar. 2014.

[9] J. Wang, N. Zhan, X. Feng, and Z. Liu, "Overview of formal methods," *Ruan Jian Xue Bao/J. Softw.*, vol. 30, no. 1, pp. 41–63, Dec. 2017.
[10] A. Souri, N. J. Navimipour, and A. M. Rahmani, "Formal verification approaches and standards in the cloud computing: A comprehensive and systematic review," *Comput. Standards Interfaces*, vol. 58, pp. 1–22, May 2018.
[11] *UPPAAL*. Accessed: Dec. 28, 2018. [Online]. Available: http://www.uppaal.org/
[12] R. D. Cosmo, J. Mauro, S. Zacchiroli, and G. Zavattaro, "Aeolus: A component model for the cloud," *Inf. Comput.*, vol. 239, pp. 100–121, Dec. 2014.
[13] A. Brogi and J. Soldani, "Finding available services in TOSCA-compliant clouds," *Sci. Comput. Program.*, vols. 115–116, pp. 177–198, Jan./Feb. 2016.
[14] M. Dorigatti, A. Guarnaschelli, O. Chiotti, and H. E. Salomone, "A service-oriented framework for agent-based simulations of collaborative supply chains," *Comput. Ind.*, vol. 83, pp. 92–107, Dec. 2016.
[15] S. Huang, X. Gu, H. Zhou, and Y. Chen, "Two-dimensional optimization mechanism and method for on-demand supply of manufacturing cloud service," *Comput. Ind. Eng.*, vol. 117, pp. 47–59, Mar. 2018.
[16] M. Sun and G. Fu, "A formal design model for cloud services," in *Proc. Int. Conf. Softw. Eng. Knowl. Eng.*, Jul. 2017, pp. 173–178.
[17] C. Newcombe, T. Rath, F. Zhang, B. Munteanu, M. Brooker, and M. Deardeuff, "How amazon Web services uses formal methods," *Commun. Acm*, vol. 58, no. 4, pp. 66–73, Mar. 2015.
[18] K. Klai and H. Ochi, "Model checking of composite cloud services," in *Proc. IEEE Int. Conf. Web Services*, Jul. 2016, pp. 356–363.
[19] A. Rezaee, A. M. Rahmani, A. Movaghar, and M. Teshnehlab, "Formal process algebraic modeling, verification, and analysis of an abstract fuzzy inference cloud service," *J. Supercomput.*, vol. 67, no. 2, pp. 345–383, Feb. 2014.
[20] D. Weerasiri, M. C. Barukh, B. Benatallah, Q. Z. Sheng, and R. Ranjan, "A taxonomy and survey of cloud resource orchestration techniques," *ACM Comput. Surv.*, vol. 50, no. 2, pp. 1–41, May 2017.
[21] M. Graiet, A. Mammar, S. Boubaker, and W. Gaaloul, "Towards correct cloud resource allocation in business processes," *IEEE Trans. Serv. Comput.*, vol. 10, no. 1, pp. 23–36, Jan./Feb. 2017.
[22] S. Jlassi, A. Mammar, I. Abbassi, and M. Graiet, "Towards correct cloud resource allocation in FOSS applications," *Futur. Gener. Comput. Syst.*, vol. 91, pp. 392–406, Feb. 2019.
[23] Y. Zhou, J. Ge, P. Zhang, and W. Wu, "Model based verification of dynamically evolvable service oriented systems," *Sci. China Inf. Sci.*, vol. 59, no. 3, pp. 032101:1–032101:17, 2016.
[24] Y. Meng, X. Li, Y. Guan, R. Wang, and J. Zhang, "Modeling and verification for robot joint bus communication system," *Ruan Jian Xue Bao/J. Softw.*, vol. 29, no. 6, pp. 1699–1715, Aug. 2018.
[25] S. Balasubramaniyan, S. Srinivasan, F. Buonopane, B. Subathra, J. Vain, and S. Ramaswamy, "Design and verification of cyber-physical systems using truetime, evolutionary optimization and UPPAAL," *Microprocess. Microsyst.*, vol. 42, pp. 37–48, Jul. 2016.
[26] Y. B. Kartal, E. G. Schmidt, and K. W. Schmidt, "Modeling distributed real-time systems in TIOA and UPPAAL," *ACM Trans. Embed. Comput. Syst.*, vol. 16, no. 1, pp. 1–26, Sep. 2016.
[27] S. Sultana and F. Arif, "Computational conversion via translation rules for transforming C++ code into UPPAAL's automata," *IEEE Access*, vol. 5, pp. 14455–14467, 2017.

WENBO ZHOU received the B.S. and M.S. degrees in computer software and theory from Jilin University, Changchun, China, in 2014 and 2017, respectively, where he is currently pursuing the Ph.D. degree in computer software and theory. His research interests include formal methods and cloud computing.

LEI LIU received the B.S. and M.S. degrees in computer software from Jilin University, China, in 1982 and 1985, respectively, where he is currently a Professor and a Doctoral Supervisor with the College of Computer Science and Technology. His research interests include program theory, semantic web, formal methods, and compiler theory.



SHUAI LÜ received the M.S. and Ph.D. degrees in computer software and theory from Jilin University, China, in 2007 and 2010, respectively, where he is currently an Associate Professor with the College of Computer Science and Technology. His research interests include artificial intelligence, machine learning, and automated reasoning.

PENG ZHANG received the B.S. and Ph.D. degrees in computer software and theory from Jilin University, China, in 2009 and 2014, respectively, where he is currently a Lecturer with the College of Computer Science and Technology. His research interests include software testing and cloud computing.