A Case Study on Algebraic Specification of Cloud Computing

Dongmei Liu†, Hong Zhu* and Ian Bayley*
* *School of Computer Science and Technology, Nanjing University of Science and Technology, Nanjing, 210094, P.R. China. Email:dmliukz@njust.edu.cn*
† *Department of Computing and Communication Technologies, Oxford Brookes University, Oxford OX33 1HX, UK. Email:hzhu@brookes.ac.uk, ibayley@brookes.ac.uk*

***Abstract*—A cloud often provides a RESTful interface with which to access its services. These are usually specified through an open but informal document in the IT industry. There is no agreed standard for the specification of RESTful web services. In this paper, we propose the application of an algebraic method to the formal specification of such services and report a case study with the GoGrid's RESTful API, an industrial real system that provides Infrastructure-as-a-Service. The case study demonstrates that the algebraic approach can provide formal unambiguous specifications that are easy to read and write. It also demonstrates that formalisation can identify and eliminate ambiguity and inconsistency in informal documents.**

***Keywords*-Cloud computing, RESTful Web Services, Formal specification, Algebraic specification.**

I. Introduction

The development of reliable and dependable software systems has long been regarded as a grand challenge. Formal methods have been advanced in the past several decades as a viable solution to this problem [1]. The advent of cloud computing, and other forms of service-oriented computing, has raised the demands further. Formal specifications must now be uniformly applicable to all levels of services including Infrastructure, Platform and Software as a Service (IaaS, PaaS, SaaS). It must also be flexible enough to support dynamic discovery and composition of services without revealing vendor-specific design and implementation details. This paper explores the applicability of the algebraic method to cloud computing.

Algebraic specification was proposed in the 1970s as an implementation-independent specification technique for abstract data types [2]. Since then, it has been extended to concurrent systems, state-based systems and software components, by applying theories of behavioural algebras [3] and co-algebras [4]–[6]. In [7], a specification language called CASOCC-WS was proposed and a supporting tool was reported for the specification of the so-called *big web services* (WS), i.e. SOAP/WSDL-based WS. A distinctive feature of CASOCC-WS is that it allows both the domain and co-domain of an operator to be multiply sorted, thus breaking the restrictions of algebraic and co-algebraic approaches. It was argued and demonstrated by a case study that this is necessary and useful to specify big WS [7].

However, the majority of existing WS, especially cloud services, are *RESTful* rather than based on SOAP and WSDL. In contrast to the big WS, no agreed standard exists for describing RESTful services either at the semantic or at the syntax level. Documents are in natural language, and thus leave space for ambiguity and misinterpretation. The current research on the description of RESTful WS focuses on developing formats for annotating the syntax and semantics of services. The most well known such efforts include WADL [8], hRESTS/MicroWSMO [9], and SA-REST [10]. They describe RESTful WS by defining the data types of the input and output and the operations in XML or HTML. The definition of semantics relies on ontology rather than the effects of the operations on the states of the resources. Recently, an extension of UML state machine diagrams was proposed to describe how the states of RESTful WS were changed [11].

In comparison with these approaches, algebraic specifications are at a very high level of abstraction and completely independent of any implementation detail. From such specifications, software properties, such as correctness, can be proven, and implementations can be derived [12]. A particularly attractive feature of algebraic specifications is that they can be used directly for automated software testing [13], [14]. This is particularly important when services bind dynamically and thus testing must be done on-the-fly. This paper proposes the application of algebraic methods to the specification of cloud's RESTful WS interfaces and reports a case study with GoGrid's API [15], a real industrial IaaS-level cloud. It demonstrates the applicability and merits of the algebraic approach to cloud computing and RESTful WS.

The remainder of this paper is organized as follows. Section II presents the specification of GoGrid to illustrate the applicability of algebraic approach to cloud services. Section III discusses the findings of the case study. Section IV concludes the paper with a discussion of future work.

II. Specification of GoGrid API

GoGrid is the world's largest pure-play Infrastructure-as-a-Service (IaaS) provider specializing in cloud infrastructure solutions. It provides an API with which its customers can

Table I
APPLICABLE OPERATORS ON OBJECTS

Object	**List**	**Get**	**Add**	**Delete**	**Edit**	**Other**
Load Balancer	Yes	Yes	Yes	Yes	Yes	
Server	Yes	Yes	Yes	Yes	Yes	Power
Server image	Yes	Yes		Yes	Yes	Save, Restore
Job	Yes	Yes				
IP	Yes					
Password	Yes	Yes				
Billing		Yes				
Option	Yes					

dynamically deploy and manage their resources through a programmatic interface.

We choose *GoGrid* for the case study because it is a typical cloud service whose API is informally defined by an open specification [16] and accessed through the RESTful web service protocol. Moreover, the GoGrid, along with its rivals, supports multiple programming languages, such as Java, Ruby, Python, C#, as well as shell script languages such as Bash. It is a real industrial test case for the proposed formal method.

A. Overview of GoGrid API

The GoGrid API has a REST-like query interface; that is, it is based on the HTTP protocol and each GoGrid API call is an individual HTTP query. However, GoGrid API only uses GET and POST rather than all four HTTP operators. For HTTP GET calls, the input data are passed via the query string. For HTTP POST calls, the input data are passed in the request body, which is URL encoded. The server responds to each request by changing the internal state of the service if needed and returning a message to the service requester.

The current version of GoGrid API (version 1.8) has 11 types of objects and 5 common operators. Some of the operators are not applicable for all types of objects. There are 3 types of objects only used as parameters of the operators, thus they have no operators; while 2 types of objects have additional special operators. Table I gives the applicable operators for each type of objects.

B. Overall Structure of the Specification

The specification of GoGrid API consists of a number of units:

- For each type of objects, such as load balancer, there is a corresponding specification unit to define the applicable operators on the objects.
- For each applicable operator on each type of object, there are two more specification units that define the structures and constraints on the requests and responses of the operation, respectively. These units are imported to the unit that specify the type of objects.
- For each non-primitive data type that occurs in the requests and/or responses of multiple operators, there is also a unit to define its structure and constraints to reduce the redundancy. It is imported into the specification units of the requests and/or responses.

By dividing the specification of a system into a number of units, the specification is modular, making it easier to read and easier to revise as it evolves.

For the sake of space, here we only give the specification of the *Load Balancer* objects as an example.

C. Specification of Load Balancer Objects

The load balancer objects are structural, which consist of a number of attributes. Each attribute is defined by an *observer* operator, which is similar to the *getters* in object-oriented programs for getting the value of attributes.

The operators on the load balancer objects have a number of parameters, which include Option, IPPP, etc. Therefore, their corresponding sorts are imported into the specification unit for Load Balancer.

```
Spec LoadBalancer;
 Sort Option, IPPP, ListofIPPP;
  Operators:
   Observer:
    id: LoadBalancer -> Long;
    name, description: LoadBalancer -> String;
    virtualip: LoadBalancer -> IPPP;
    realiplist: LoadBalancer -> ListofIPPP;
    type, persistence, os, state, datacenter:
        LoadBalancer -> Option;
 Axiom:
  For all LBO: LoadBalancer that
   LBO.id <> NULL;
  End
End
```

where *NULL* is a value that represents no information.

An operation may also return a list of load balancer objects. Thus, we have a unit called `ListofLB`.

```
Spec ListofLB;
 Sort LoadBalancer;
  Operators:
   Observer:
    items: ListofLB,Int -> LoadBalancer;
    length: ListofLB -> Int;
End
```

D. Specification of Requests and Responses

1) Common query parameters: There are four query parameters that are common to all GoGrid API calls, and they are specified as follows.

```
Spec CommonParameter;
 Operators:
  Observer:
   api_key, sig, v, format:
     CommonParameter -> String;
 Axiom:
  For all CP: CommonParameter that
   CP.api_key <> NULL;
   CP.sig <> NULL;
   CP.v <> NULL;
  End
End
```

where *api_key* is a key generated by GoGrid for security in the access of resources. It is obtained before API calls can be made. *sig* is an MD5 signature of the API request data, *v* is the version id of the API, and *format* is an optional field to indicate the required response format. The signature is generated by an MD5 hash from the *api_key*, the user's *shared secret*, which is a string of characters set by the user and known only by the GoGrid server, and a *Unix timestamp*, which is the number of seconds since the Unix Epoch of the time when the request was made.

The *api_key* and *shared secret* act as an authentication mechanism. Because the signature is time-dependent, the relationship between query parameters cannot be specified without the context of the request. So, the axioms only state that these parameters cannot be omitted. The authentication mechanism is specified later.

2) Requests of the Get Operator: In addition to the parameters common to all types of requests, the requests of a specific operator on each type of object may also contain various other types of parameters. For the sake of space, here we only give the specification of the get operator, which, as shown in Table I, is the most common operator.

```
Spec LBGetRequest;
 Sort CommonParameter, ListofString;
  Operators:
   Observer:
    para: LBGetRequest -> CommonParameter;
    id, name, loadbalancer :
       LBGetRequest -> ListofString;
    timestamp: LBGetRequest -> Int;
 Axiom:
  For all x: LBGetRequest that
   x.name == NULL, If x.id <> NULL;
   x.loadbalancer == NULL, If x.id <> NULL;
   x.id == NULL, If x.name <> NULL;
   x.loadbalancer == NULL, If x.name <> NULL;
   x.id == NULL, If x.loadbalancer <> NULL;
   x.name == NULL, If x.loadbalancer <> NULL;
  End
End
```

where *para* is the common query parameters defined in the previous subsection. The parameters *id*, *name*, *loadbalancer* are used to filter load balancer objects, only one of them is required and they are exclusive and cannot be mixed in a request. The last parameter *timestamp* is used for authentication purposes.

3) Responses to the Get Operation: A *load balancer get* service call returns a list of load balancers in the GoGrid system that satisfy the query conditions. In addition to a list of returned objects, the response also contains the *response status*, *request method*, and a *summary* of the list. The summary can be specified as follows.

```
Spec GetSummary;
 Operators:
  Observer:
   total, start, returned, numpages:
      GetSummary -> Int;
 Axiom:
  For all GS: GetSummary that
   GS.total >= 0;
   GS.start == 0;
   GS.returned == GS.total;
  End
End
```

where *total* is the total number of objects in the system, *start* is the index of the first object in this returned list, *returned* is the number of objects returned, and *numpages* is the number of pages available.

The structure of the responses to Get Load Balancer requests is as follows. For the sake of space, we omit the axioms.

```
Spec LBGetResponse;
 Sort ListofLB, GetSummary;
  Operators:
   Observer:
    status, request_method:
       LBGetResponse -> String;
    summary: LBGetResponse -> GetSummary;
    objects: LBGetResponse -> ListofLB;
    statusCode: LBGetResponse -> Int;
 Axiom: ...
End
```

In addition to *status*, *request method*, *summary* of the list and a list of returned *objects*, each HTTP response will contain a status code: 200 means that the call is successful, 4xx means there is an error in the client's request, and 5XX means a server error occurred.

E. Semantics of the operations

An operator can be a transformer, such as Add, Delete and Edit, or an observer, such as List and Get. They all have the state of the cloud as the context. In particular, we need to know the clock time of the cloud and also the shared secret chosen by each user for checking the authentication of access. The following is the signature of the sort LBGoGrid that represents the state of the cloud and the operators.

```
Spec LBGoGrid;
 Sort CommonParameter, LoadBalancer, ListofLB,
      Option, IPPP, ListofIPPP, ListofString,
      LBListRequest, ListSummary, LBListResponse,
      LBGetRequest, GetSummary, LBGetResponse,
      LBAddRequest, AddSummary, LBAddResponse,
      LBDelRequest, DelSummary, LBDelResponse,
      LBEditRequest, EditSummary, LBEditResponse;
  Operators:
   Observer:
    clockTime: LBGoGrid  -> Int;
    sharedSecrets: LBGoGrid, String -> String;
    List: [LBGoGrid]
      LBListRequest -> LBListResponse;
    Get: [LBGoGrid]
      LBGetRequest -> LBGetResponse;
   Transformer:
    Add: [LBGoGrid]
      LBAddRequest -> LBAddResponse;
    Delete: [LBGoGrid]
      LBDelRequest -> LBDelResponse;
    Edit: [LBGoGrid]
      LBEditRequest -> LBEditResponse;
End
```

For each operator, its semantics can be characterised by a set of axioms. Here, we only illustrate the style of algebraic specification using the get operator as an example.

First of all, GoGrid checks the authentication of each get call by using the MD5 function to reconstruct the signature from the api-key, the user's shared secret and the time stamp. It then compares this to the signature contained in the request parameter. It also checks the time stamp with its server clock time, allows a discrepancy up to 10 minutes. This authentication rule can be specified as follows.

```
Axiom <Authentication>:
For all G:LBGoGrid, X:LBGetRequest that
 Let key = X.para.api_key,
     sig_Re = MD5(key, G.sharedSecret(key),
                  X.timeStamp)
 in  G.Get(X).statusCode == 403,
       If X.para.sig <> sig_Re
          or abs(X.timeStamp - G.clockTime) > 600;
 End
End
```

where MD5 and abs are auxiliary functions.

An important feature of the Get operator is that it is an observer. So, its application will not change the state of the context sort LBGoGrid.

The following axiom states that when an operation changes the state of the cloud, for example, by adding a load balancer, the Get operator should be able to observe the difference accordingly. In fact, such an axiom also defines the semantics of the transformer operator.

```
Axiom <Add-Get>:
For all G: LBGoGrid, X1: LBAddRequest,
        X2, X3: LBGetRequest, i: Int that
 [G.Add(X1)].Get(X2).objects == G.Add(X1).objects,
   If X2.name.length == 1,
      X1.name == X2.name.items(0),
      G.Add(X1).statusCode == 200,
      G.Get(X2).statusCode == 200;
 [G.Add(X1)].Get(X2).objects == G.Get(X2).objects,
   If search(X2.name, X1.name) == False,
      G.Add(X1).statusCode == 200,
      G.Get(X2).statusCode == 200;
 [G.Add(X1)].Get(X2).objects ==
     insert(G.Get(X3).objects, G.Add(X1).objects)
   If search(X2.name, X1.name) == True,
      search(X3.name, X1.name) == False,
      search(X3.name, X2.name.items(i)) == True,
      X2.name.items(i) <> X1.name,
      0 =< i, i < X2.name.length,
      G.Add(X1).statusCode == 200,
      G.Get(X2).statusCode == 200;
      G.Get(X3).statusCode == 200;
End
```

where *insert* and *search* are auxiliary functions, the former inserts a list of load balancer objects into another list, and the later searches for a string in a list of strings.

The following axiom states that when an operation changes the state of the cloud by deleting a load balancer, the Get operator should also be able to observe the difference.

```
Axiom <Delete-Get>:
For all G: LBGoGrid, X1: LBDeleteRequest,
        X2: LBGetRequest that
 [G.Delete(X1)].Get(X2).statusCode == 500,
   If search(X2.name, X1.name) == True,
      G.Delete(X1).statusCode == 200,
 [G.Delete(X1)].Get(X2).objects
  == G.Get(X2).objects,
   If search(X2.name, X1.name) == False,
      G.Delete(X1).statusCode == 200,
      G.Get(X2).statusCode == 200;
End
```

III. Discussion

The original document of the GoGrid API [16] specifies the data types of the API request parameters and their corresponding responses, and describes the meaning of each in normative text one by one. Sample requests and responses are also given to explain the semantics and usage of the API.

The algebraic specification of GoGrid can be beneficial to cloud computing community in many different ways. In this section, we will discuss these advantages with examples from the case study.

A. *Removing Ambiguity and Improving preciseness*

As one may expect, in the process of formalisation, we found that in some places the original documentation is ambiguous, thus the narrative texts could be interpreted differently by different users.

A typical example of such ambiguity is caused by incompleteness. For example, the GoGrid document is unclear about the range of values for a parameter and what will happen when the value is out of the range. For instance, should the `num_items` parameter (the number of items in a page) in the list request be greater than 0? What will happen if it is 0? These questions remain open in our formal specification due to the lack of clear documentation.

Another example of ambiguity in the GoGrid document is about the relationship between the request and response of some service calls. Take *get job objects* operation as an example. In the sample given in the GoGrid document, the id of the request is not the same as in the response. There is no way to work out from the document what id should be in the response.

A similar ambiguity due to incompleteness occurs in the description of the *list job objects* operation, which states that '*This call will list all the jobs in the system for a specified date range*'. However, requests can be also made to list all the job objects of a specified type, state or owner.

B. *Checking consistency and support evolution*

As all software artefacts, the API of a cloud service is subject to frequent changes. The support to service evolution is of particular importance. Algebraic specifications in CASOCC-WS are modular. The consistency within a specification unit and between specification units can be checked at both syntactic and semantic levels. In our case study, we have employed a parser of the CASOCC-WS language to perform consistency checking at the syntactic level. This detected a number of inconsistencies.

For example, from version 1.5 and later, GoGrid adds the attribute named datacenter in the request query parameter, but the description of objects does not add the attribute correspondingly. This results in inconsistency between the description of job objects and request query parameters. Similarly, there is no attribute named numpages in the Get, Edit, Delete requests, but it exists in the sample responses. Moreover, the port in parameters of Edit operator on LoadBalancer objects is required to be more than or equal to 0, while it is required to more than 0 in other operators of LoadBalancer objects.

In an unstructured document, redundancy is also a problem. In the GoGrid document, it is common for the same information to be given in several different places with different presentations. Take *error code* for example. There is a detailed description in the document entitled *Anatomy of a GoGrid API Call*. However, in each document on a particular type of API calls, there is also a description. *Response format* is another example of such redundancy.

C. Easiness to understand and write algebraic specification

Formal methods have been widely regarded as difficult to learn and expensive to apply. However, our case study demonstrates that it is easy to learn how to write algebraic specifications. This confirms the findings reported in [17].

In particular, we find in the case study that following the guidelines given in [7], the GoGrid API's semantics and syntactic structures can be defined fairly straightforwardly. The cost of writing the specification is not expensive, as shown in Table II, where column *Sorts* is the number of the sorts used for specifying the type of objects, and column *Lines* gives the number of lines that the algebraic specification of each type of objects contains.

Table II
RESULTS OF CASE STUDY

Object	Sorts	Lines	Object	Sorts	Lines
Common	24	272			
Load Balancer	16	455	Billing	4	70
Server	19	557	IP	4	170
Server image	19	546	Password	7	212
Job	7	242	Option	4	174
Total				104	2698

IV. CONCLUSION

In this paper, we reported a case study on the algebraic specification of RESTful cloud services in the CASOCC-WS language. It clearly demonstrated the value of algebraic approach in the specification of cloud services.

We are currently studying its theoretical foundation of the algebraic specification language that combines algebraic and co-algebraic features. We are also extending the language by including an extension mechanism so that specifications can be written more concisely. Meanwhile, we are developing a tool that uses the language as input to support automated testing of cloud services. The case study reported in this paper fully specifies the functions of resource management what GoGrid API original document specifies. Further case studies on the formal specification of PaaS and SaaS will also be interesting.

ACKNOWLEDGEMENT

The work reported in this paper is partially supported by EU FP7 project MONICA on Mobile Cloud Computing (Grant No.: PIRSES-GA-2011-295222) and National Natural Science Foundation of Jiangsu (Grant No. BK2011022).

REFERENCES

[1] A. van Lamsweerde, “Formal specification: a roadmap,” in Proc. of *ICSE 2000 - Future of SE Track*, 2000, pp. 147–159.

[2] J. A. Goguen, J. W. Thatcher, E. G. Wagner, and J. B. Wright, “Initial algebra semantics and continuous algebras,” *J. ACM*, vol. 24, no. 1, pp. 68–95, 1977.

[3] J. A. Goguen and G. Malcolm, “A hidden agenda,” *Theor. Comput. Sci.*, vol. 245, no. 1, pp. 55–101, 2000.

[4] J. M. Rutten, “Universal coalgebra: a theory of systems,” *Theor. Comput. Sci.*, vol. 249, no. 1, pp. 3–80, 2000.

[5] C. Cîrstea, “A coalgebraic equational approach to specifying observational structures,” *Theor. Comput. Sci.*, vol. 280, no. 1-2, pp. 35–68, 2002.

[6] F. Bonchi and U. Montanari, “A coalgebraic theory of reactive systems,” *Electr. Notes Theor. Comput. Sci.*, vol. 209, pp. 201–215, 2008.

[7] H. Zhu and B. Yu, “Algebraic specification of web services,” in Proc. of *QSIC'10*, 2010, pp. 457–464.

[8] M. J. Hadley, “Web application description language WADL,” Sun Microsystems Inc., CA, USA, SMLI TR-2006-153, March 2006.

[9] J. Kopecky, K. Gomadam, and T. Vitvar, “hRESTS: An HTML microformat for describing RESTful web services,” in Proc. of *WI-IAT'08*, Dec. 2008, pp. 619–625.

[10] J. Lathem, K. Gomadam, and A. P. Sheth, “SA-REST and (S)mashups: Adding semantics to RESTful services,” in Proc. of *ICSC'07*, 2007, pp. 469–476.

[11] O. Liskin, L. Singer, and K. Schneider, “Welcome to the real world: A notation for modeling REST services,” *IEEE Internet Computing*, pp. 36–44, July-Aug. 2012.

[12] D. Sannella and A. Tarlecki, “Algebraic methods for specification and formal development of programs,” *ACM Computing Surveys*, vol. 31, no. 3es, p. 10, 1999.

[13] H. Y. Chen, T. H. Tse, and T. Y. Chen, “Taccle: a methodology for object-oriented software testing at the class and cluster levels,” *ACM TOSEM*, vol. 10, no. 1, pp. 56–109, 2001.

[14] B. Yu, L. Kong, Y. Zhang, and H. Zhu, “Testing java components based on algebraic specifications,” in Proc. of *ICST'08*, 2008, pp. 190–199.

[15] GoGrid.com, “Gogrid website,” http://www.gogrid.com/. Last Access: Feb. 20, 2012.

[16] ——, “Gogrid wiki,” https://wiki.gogrid.com/wiki/index.php/Main_Page. Last access: Feb. 20, 2012.

[17] H. Zhu and B. Yu, “An experiment with algebraic specifications of software components,” in Proc. of *QSIC'10*, 2010, pp. 190–199.