Applying Algebraic Specification To Cloud Computing

-- A Case Study of Infrastructure-as-a-Service GoGrid

Dongmei Liu
School of Computer Science and Technology
Nanjing University of Science and Technology
Nanjing, 210094, P.R. China
Email:dmliukz@njust.edu.cn

Hong Zhu and Ian Bayley
Dept of Computing and Communication Technologies
Oxford Brookes University
Oxford, OX33 1HX, UK
Email:hzhu@brookes.ac.uk, ibayley@brookes.ac.uk

Abstract— **Cloud Computing has attracted attention from both the research community and the industry. It is highly desirable to specify the syntax and semantics of the services precisely and accurately without giving away any design and implementation details. This challenge is even greater for cloud services based on RESTful web services techniques, where the invocation is through HTTP queries and there is no agreed standard exists for their specifications. In this paper, we propose an algebraic approach and apply it, as a case study, to the GoGrid Cloud Computing API. Not only does this give a formal unambiguous specification that is easy to write and understand, but it also identifies and eliminates errors in the existing documentation.**

Keywords-cloud computing; formal specification; algebraic specification; RESTful Web services, Infrastructure-as-a-Service

I. INTRODUCTION

Precise and accurate documentation of software systems has long been a challenge to the software engineering communities. The advent of cloud computing, and other forms of service-oriented computing, has raised the demands further, since software engineers, when developing their applications, have to depend solely on documents of the services provided by the cloud. Moreover, when services are dynamically discovered and composed at runtime, the specifications of the services must be machine readable, in the senses of both syntax and semantics.

To meet the challenges of software specifications, formal methods have been developed in the past forty years and have advanced significantly [1]. However, their application to services thus far has been limited, restricted to ontology definition languages and business process description languages, such as the Business Process Execution Language BPEL [2].

A formal specification technique for services must satisfy two requirements. First, it must be uniformly applicable to each of the various levels of services: Infrastructure, Platform and Software as a service. Secondly, it must 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 algebraic specification to RESTful web services [3], which is widely employed by cloud service providers.

A. Related works

Many cloud services provide an application programming interface (API) with which their customers can dynamically configure, manage and use their resources through a programmatic interface. For Infrastructure-as-a-Service (IaaS), the resources are hardware entities, such as servers and load balancers, etc. For Platform-as-a-Service (PaaS), and Software-as-a-Service (SaaS), the resources are platform and software entities respectively. Examples include virtual machines, network middleware, databases, software components, etc.

The current practice is to define the API informally with an open specification. The API is accessed through the RESTful web service protocol. In contrast to traditional SOAP-based web services [4], no agreed standards exist for describing RESTful services, either at the semantic level or the syntactic level. Documentation is often in natural language, leaving space for ambiguity and for errors in the definition of the services. The formal specification of RESTful web services is still an open problem. It is highly desirable that the API specification is formalized to reduce ambiguity, redundancy and inconsistency to the minimum, whilst still being easy to understand and requiring minimal training.

Current research on the description of RESTful web services is mostly focused on formats for annotating the syntax and semantics of RESTful web services. The most well-known such efforts include WADL [5], hRESTS/ MicroWSMO [6], and SA-REST [7]. They describe the syntax and data types of the input and output as well as the operations of the WS using a machine readable format in XML or HTML. The main problem of these approaches is that they rely heavily on the RPC-based operation model, which does not align well with the principles of RESTful web service. Moreover, the semantics are described at the level of ontology, rather than the effect on the states of the resources that the services operate on. In [8], Liskin et al. proposed an extension to UML state machine diagram to describe in graphic notation how RESTful web services changes the states of the resources. However, it is open to dispute whether graphic notations can capture the full complexity of the state changes involved in resources managed and manipulated by these services.

Algebraic specification was first proposed in the 1970s as an implementation-independent specification technique for

Copyright (c) IARIA, 2012. ISBN: 978-1-61208-230-1

abstract data types [9][10]. Since then, it has been extended to concurrent systems, state-based systems and software components, all by applying theories of behavioural algebras [11] and co-algebras [12][13][14][15]. The specifications produced are at a high level of abstraction, completely independent of any implementation detail. Properties can be proven for these specifications, they can be refined to implementations, and implementations can be proven correct with respect to the specifications [16], etc.

A particularly attractive feature of algebraic specifications is that they can be used directly by automated software testing tools [17][18]. This is particularly important when services bind dynamically since testing must be done on-the-fly.

B. Main Contributions of the paper

This paper reports a case study of an algebraic specification of *GoGrid* [19], a real industrial-strength system that provides infrastructure-as-a-service. GoGrid provides an API, defined by an open specification [21] and accessed through a RESTful interface. The specification language used in the case study is CASOCC-WS, which we proposed in [20] for the specification of SOAP-based web services. By successfully specifying every operation of the GoGrid API, we demonstrate that CASOCC-WS can be used for RESTful Web services too.

During the formalization, we detected non-trivial errors in GoGrid's documentation. These errors included ambiguity, inconsistency and also incompleteness. This case study shows, therefore, that formal specification can improve the precision and accuracy of service documentation.

It shows too that algebraic specifications can be abstract and implementation independent, since the GoGrid API, like many other RESTful web services, supports multiple programming languages, such as Java, Ruby, Python, C#, as well as shell script languages such as Bash.

Finally, our case study also demonstrates that algebraic specifications can be easy to understand with minimal training, confirming the findings in [22].

To our knowledge, there is no similar work in the literature on the algebraic specification of web services, nor on RESTful web services in particular.

C. Organisation of the paper

The remainder of this paper is organized as follows. Section II briefly outlines the algebraic specification language CASOCC-WS. In Section III, we use GoGrid API as a case study to demonstrate how the algebraic approach can be used to specify cloud computing interface. Section IV discusses some of the benefits of applying algebraic approach on cloud computing interface. Section V concludes the paper and discusses possible future work.

II. ALGEBRAIC SPECIFICATION LANGUAGE CASOCC-WS

The CASOCC-WS language is an extension of the CASOCC language [17][18]. The specifications in CASOCC-WS are modular. A specification is built from a number of units, one for each software entity in the system. A software entity can be an abstract data type, a class, a component, a web service, and so on. A specification has the following syntactic form, in a variant of BNF:

```
<Specification> ::= {<Spec Unit>}
<Spec Unit> ::=
    Spec <Sort Name> [<Observability>];  <Signature> [<Axioms>]
    End
<Sort Name> ::= <Identifier>
<Observability> ::=
     is observable by <Operator ID> | is unobservable
<Operator ID> ::= <Identifier>
```

Each specification unit contains two main parts: a *signature* and a set of *axioms*. The <Sort Name> is an identifier that names the main sort of the unit. Observability is an important property of software entities. A software entity is directly observable if its state or value can be tested for equality; otherwise, its state or value has to be checked by other means, e.g. through observers. The operator for an observable entity must be a Boolean function.

A. Signature

The signature specifies the syntactic aspect of the software entity. A signature has the following syntactic form:

```
<Signature> ::=  [<Imported Sorts>;] <Operations>
<Imported Sorts> ::= Sort <Imported Sort List>
<Imported Sort List> ::=  <Sort Name>[, <Imported Sort List>]
<Operations> ::=
    Operators:  [<Creators>;][<Transformers>;][<Observers>;]
<Creators> ::= Creator: <OpList>
<Transformers> ::= Transformer: <OpList>
<Observers> ::= Observer: <OpList>
<OpList> ::= <Operation> [; <OpList>]
<Operation> ::= <Operator ID> :['['<Context Sort>']']
        [<Domain Type>] -> <Co-domain Type>
<Context Sort> ::= <Sort Name>
<Domain Type> ::= <Type> | VOID
<Co-domain Type> ::= <Type> | VOID
<Type> ::= <Sort Name> [, <Type>]
```

The *Imported Sorts* clause is a comma-separated list of the sorts upon which the sort currently being specified depends. The operators defined for a sort are classified into creator, transformer and observer. Take STACK for example, its signature is as follows.

```
Spec STACK;
    Sort BOOL, NAT;
        Operators:
            Creator:        newStack:  -> STACK;
            Transformer:    push: STACK, NAT -> STACK;
                            pop: STACK -> STACK;
            Observer:       isNewStack: STACK -> BOOL;
                            top: STACK -> NAT;
End
```

This means that STACK depends on BOOL for Boolean values and NAT for natural numbers. newStack is a creator, push and pop are transformers, isNewStack and top are

observers.

Note that, in a traditional algebraic specification language, the co-domain of an operator must be a singleton. Such a signature is called *algebraic*; STACK has such a signature. More recent languages, based on co-algebras, require instead the domain to be singleton; such signatures are called *co-algebraic*, and can be used.

CASOCC-WS extends the algebraic and co-algebraic approaches by allowing both the domain and the co-domain of an operator to be non-singleton at the same time. This makes it possible to specify stateful services naturally. For example, infinite streams of natural numbers are specified as follows. Each application of the operator next to a stream will give a natural number and change the state of the stream.

```
Spec STREAM is unobservable;
    Sort NAT;
        Operators:
            Transformer:    next: STREAM -> STREAM, NAT;
End
```

In general, when the main sort of the unit occurs in both the domain and the co-domain of an operator, we call it the *context sort* of the operator. In such a case, CASOCC-WS use the following format to indicate the context sort, while omitting it from the domain and co-domain.

$$Op : [s]\ s_1, \ldots, s_n \rightarrow s'_1, \ldots, s'_k,$$

where s is the context sort.

If the main sort is the only sort in an operator's domain or co-domain, we write VOID for the type of the latter. For example, the signature of the operator push of STACK and the operator next of STREAM can now be specified respectively as follows.

```
push: [STACK] NAT -> VOID;
next: [STREAM] VOID -> NAT;
```

B. Axiom

Each specification unit consists of logical axioms describing the properties that functions are required to satisfy. An axiom consists of a variable declarations block and a list of conditional equations.

```
<Axioms> ::= Axiom: <Axiom List>
<Axiom List> ::= <Axiom> [<Axiom List>]
<Axiom> ::= <Var Declarations> <Equations> End
<Equations> ::= <Equation> [<Equations>]
```

1) Variable declarations

Variable declarations declare a list of variables and their types. Variables are declared "globally" to all equations in the axiom using "For all" keywords.

```
<Var Declarations> ::= For all <Var-Sort Pairs> that
<Var-Sort Pairs> ::=  <Var IDs> : <Sort Name> [, <Var-Sort Pairs>]
<Var IDs> ::= <Var ID> [, <Var IDs>]
<Var ID> ::= <Identifier>
```

where the sort name can only be the main sort or a sort listed in the imported sorts clause. The variable identifiers must be unique: they must not clash with sort names, operator names nor with any such names in any sorts imported and other variables in this axiom.

2) Equation

Equations declare a list of conditional equations. The syntax rule of an equation is as follows.

```
<Equation> ::=  [<Label ID>:] <Condition> [, if <Conditions>];
                | Let <Var Definitions> in <Equations> End
<Label ID> ::= <Identifier>
<Conditions> ::= <Condition> [(, | or) <Conditions>]
<Condition> ::=  <Bool Term> | <Term> <Relation OP> <Term>
<Bool Term> ::= True | False
<Relation OP> ::=  "==" | "<>" | ">" | "<" | ">=" | "<="
<Var Definitions> ::=  <Var Assignment> [, <Var Definitions>]
<Var Assignment> ::= <Var ID> = <Term>
```

The most basic form of an equation is thus $t_1 == t_2$. Here is an example of sort STACK, assuming that sort BOOL is predefined.

```
For all s: STACK, n: NAT that
     isNewStack(push(s,n)) == False;
    pop(push(s, n)) == s;
    top(push(s, n)) == n;
End
```

The second syntax rule for equations is designed to allow *local variable* definitions in the form

Let $x_1 = \tau_1, \ldots, x_n = \tau_n$ ***in*** *equs* ***End***

where $x_1, \ldots, x_n$ are local variables, limited in scope to *equs*, and $\tau_1, \ldots, \tau_n$ are terms denoting the values that are assigned to the variables. Local variables must have unique names, not clashing with other variables in this equation and any other names, just as with global variables. The above example can be specified as follows.

```
For all s: STACK, n: NAT that
    Let s1 = push(s,n)  in
        isNewStack(s1) == False;
        pop(s1) == s;
        top(s1) == n;
    End
End
```

3) Term

A term is constructed from constants and variables by the application of operators. All names used in terms may be qualified with the intended type and the intended sort of the term may be specified. In particular, a term is called *ground* term if it contains no variable. The syntax rules for term are as follows.

```
<Term> ::=   <Var ID> | "(" <Term> ")" | "<" <Term List> ">"
    | <Operator ID> ["(" [<Parameters>] ")"]
    | "[" <Term>  "]" | <Term> "." <Term>  | NULL
    | <Term> "#" <Term> | <numeric_expression>
```

Copyright (c) IARIA, 2012. ISBN: 978-1-61208-230-1

```
    | <string_expression> | <literal_expression>
<Parameters> ::= <Term List>
<Term List> ::= <Term> [, <Term List>]
<numeric_expression> ::= <Term> <Algorithm OP> <Term>
<string_expression> ::= <Term> ("+"|"+=") <Term>
<literal_expression> ::= <integer_literal>  | <float_literal>
    | <string_literal> | <character_literal>
<Algorithm OP> ::= "+" | "-" | "*" |  "/"
```

Any operator in a term must either be declared in the signature part of the sort being specified or in the signature of an imported sort. For example, if *s* is a variable of the STACK sort, and *m* and *n* are variables of the NAT sort, then the following are STACK-terms of the STACK sort.

```
push(s, n)
push(push(s,n),m)
pop(push(push(s,n),m))
pop(push(pop(push(s,n)),m))
```

Note that when an operator φ is declared in the form φ: $[s]s_1 \rightarrow s_2$ using a sort *s* as the context, the type of a term like φ(τ) is s_2, rather than (s, s_2). The new context state in the sort *s* after applying the operator φ to τ is given by the expression [φ(τ)]. For example, let NatSt: STREAM be an infinite stream of natural numbers. Then NatSt.next is the natural number at the front of the stream and [NatSt.next] is state of the stream after the next operation; i.e., the stream after taken the front number away.

III. CASE STUDY

In this section, we specify GoGrid API in CASOCC-WS as a case study.

GoGrid is the world's largest pure-play Infrastructure-as-a-Service (IaaS) provider specializing in Cloud infrastructure solutions. It provides an API, defined by an open specification, with which its customers can deploy and manage their applications and workloads through a programmatic interface.

A. GoGrid API

The GoGrid API is a REST-like query interface. RESTful web services, unlike SOAP/WSDL, are based on the HTTP protocol, so each GoGrid API call is an individual HTTP query. 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. Only GET and POST are used in GoGrid API. The server responds to each request by changing the internal state of the service if need be and by returning a message to the service requester.

The latest version of GoGrid API (version 1.8) has 11 different types of objects and 5 types of common operators. Some of the operators are not applicable to some types of objects. There are 3 types of objects that are only used as parameters of the operators, so no operators are applicable on them, while some objects have special operators. **TABLE 1** gives the applicable operators for each type of object.

TABLE 1. APPLICABLE OPERATORS ON OBJECTS

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

It is worth noting that some operators have different meanings for different types of object, so in our specification of GoGrid, the definitions were grouped by object rather than by operator. For each object, we start by specifying the requests and responses of the operations, defining their structures and the constraints on the values of the elements.

The requests (or responses) for one operator are specified in one specification unit. But, there may be a number of other specification units that specify the elements in the structure of the requests (or responses). Then, we specify the semantics of the operators on the type of objects by defining the relationships between the requests and the responses. Note that, the internal states of an object that the operator changes cannot be observed directly. They can only be observed by applying observers, which are API requests, too. The set of operators for one type of object is specified in one specification unit, but there may be auxiliary specification units, such as for lists of objects, as found in the responses to some operators.

Here, we only give the details of the specification of the operators applicable to the object type Server. This is the most important object of the system and also the most complicated to specify. Other operators are similar but less complex.

B. Requests and Responses

1) Common query parameters in requests

There are four query parameters common to all GoGrid API calls, and they are specified as follows:

```
Spec CommonQueryParameter ;
    Operators:
        Observer:
            api_key, sig, v, format:
                CommonQueryParameter -> string;
    Axiom:
        For all CQP: CommonQueryParameter that
            CQP.api_key <> NULL;
            CQP.sig <> NULL;
            CQP.v <> NULL;
        End
End
```

where api_key is a key generated by GoGrid for security in the access of resources, sig is an MD5 [23] signature of the API request data, v is the version id of the API, and format is

an optional field to indicate the response format required. NULL is a value that represents no information. The signature can be generated by an MD5 hash from the api_key, which is obtained before API calls can be made, 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.

However, because the signature is time-dependent, and therefore, also dependent on the context, the relationship between these query parameters cannot be specified without the context of the request. So, the axiom part of the specification states only that these parts cannot be omitted. We specify the authentication mechanism later in the systematic specification.

2) Request of the List operator

In addition to the parameters common to all requests, each type of request also contains variable parts. Below, we only give the specification of the requests of the List operation as an example. A *server list* call returns a list of server objects of a certain type in the cloud.

```
Spec ServerListRequest;
    Sort CommonQueryParameter, ListofString;
    Operators:
        Observer:
            para: ServerListRequest -> CommonQueryParameter;
            num_items, page, timestamp: ServerListRequest -> int;
            server_type, datacenter: ServerListRequest -> string;
            isSandbox: ServerListRequest -> boolean;
    Axiom:
        For all SLR: ServerListRequest that
            SLR.num_items >=0;
            SLR.page >=0, if SLR.num_items > 0;
        End
End
```

where para is the common query parameters defined above. num_items is the number of items to return. Its value is used to paginate the results into a number of pages so that each page contains num_items number of items. page is the index of the page to be returned when the results are paginated. The index starts from 0. This parameter is ignored if num_items is not specified. server_type, isSandbox, and datacenter are used to filter server objects. timestamp is used in authentication.

3) Responses to the List Operation

The GoGrid API responses can be in three different formats: *JSON* (JavaScript Object Notation), *XML*, and *CSV* (Comma Separated Values). The default format, used when the optional *format* parameter is omitted, is JSON. However, one benefit of using algebraic specification is that we need only one formal specification for all output formats.

The response to a list call contains the response status, request method, summary of the list and a list of returned objects. The summary part of the responses can be specified as follows:

```
Spec ListResSummary;
    Operators:
        Observer:
            Total, start,
returned, numpages:
                ListResSummary -> int;
    Axiom:
        For all LRS: ListResSummary that
            LRS.total >= 0;
            LRS.start >= 0;
            LRS.returned >= 0;
        LRS.numpages >= 0;
    End
End
```

where total is the total number of objects in the list; start is the current start index for this list of objects; returned is the number of objects returned in this list; and numpages is the total number of pages available given the num-items value in the request.

The structure of the responses of the list operator when applied to server object can be specified as follows.

```
Spec ServerListResponse;
    Sort ListofServer, ListResSummary, ListofString;
    Operators:
        Observer:
            Status, request_method: ServerListResponse -> string;
            summary: ServerListResponse -> ListResSummary;
            objects: ServerListResponse -> ListofServer;
            statusCode: ServerListResponse -> int;
    Axiom:
        For all SLR: ServerListResponse that
            SLR.request_method == "/grid/server/list";
        End
        For all SLR: ServerListResponse, i, j: int that
            SLR.objects.items(i).id <> SLR.objects.items(j).id,
                if  status == "success", i <> j,
                    0<= i, i <= SLR.summary.returned,
                    0<= j, j <= SLR.summary.returned;
        End
        For all SLR: ServerListResponse, i: int,
                X: ServerListRequest that
    search(X.datacenter,SLR.objects.items(i).datacenter.name)
        == True,
                if  status == "success",
                    0<= i, i <= SLR.summary.returned,
                    X.datacenter.length > 0;
            SLR.objects.items(i).type.name == X.server_type,
                if  status == "success",
                    0<= i, i <= SLR.summary.returned,
                    X.server_type <> NULL;
            SLR.objects.items(i).isSandbox == X.isSandbox,
                if  status == "success",
                    0<= i, i <= SLR.summary.returned,
                    X.isSandbox <> NULL;
        End
End
```

where search is an auxiliary function of the type ListofString, string -> Boolean.

Copyright (c) IARIA, 2012. ISBN: 978-1-61208-230-1

In addition to status, request method, summary of the list and a list of returned objects, each response will contain a status code: 200 means that the call is successful, and 4xx means there is an error in the client's request, of which 400 means the argument is illegal, 401 means unauthorised, 403 means authentication failed, and 404 means not found. A status code of 5xx means that a server error occurred.

C. Semantics of the operations

For each type of request, we define an operator that takes common query parameters and various typed parts as input and produces a response as the output. All such operators have GoGrid as the context. Some are transformers, such as Add, Delete and Edit; some are observers, such as List and Get. We also need to know the clock time on the grid and also the shared secret chosen by each user for checking the authentication of access. Also we define some auxiliary functions. Thus, we have the following signature for the sort ServerGoGrid, which represents the Server web service of GoGrid cloud computing system.

```
Spec ServerGoGrid;
    Sort     Server, ListofServer,
             ServerListRequest, ServerListResponse,
             ServerAddRequest, ServerAddResponse, …
    Operators:
        Observer:
            clockTime: -> int;
            sharedSecret: string -> string;
            List: [ServerGoGrid]
                ServerListRequest -> ServerListResponse;
            Get: [ServerGoGrid]
                ServerGetRequest -> ServerGetResponse;
        Transformer:
            Add: [ServerGoGrid]
                ServerAddRequest -> ServerAddResponse;
            Delete: [ServerGoGrid]
                ServerDeleteRequest -> ServerDeleteResponse;
            Edit: [ServerGoGrid]
                ServerEditRequest -> ServerEditResponse;
            Power: [ServerGrid]
                ServerPowerRequest -> ServerPowerResponse;

    Axiom
    …
End
```

where the following auxiliary functions are used.

```
MD5: string, string, int -> string ;
abs: int -> int;
insert: [ListofServer]ListofServer -> VOID;
remove: [ListofServer]ListofServer -> VOID;
update: [ListofServer]ServerPowerRequest -> VOID;
```

For each operator, its semantics can be characterised by a set of axioms. For the sake of space, here we only give the axioms that define the semantics of the list operator.

First of all, GoGrid checks the authentication of each API call 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, allowing a discrepancy of up to 10 minutes. This authentication rule can be specified as follows.

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

The second axiom is about the semantics of the List operation. It states that if the number of items per page required by the request is greater than 0 (i.e., $N_{per\ page} > 0$) and the call is successful, then, in the response summary, the number N_{pages} of pages, total number N_{items} of items and the number $N_{per\ page}$ of items on each page has the following relationship:

$$N_{pages} = \lceil N_{items} / N_{per\ page} \rceil,$$

```
Axiom <list.pagination1>:
    For all G: ServerGoGrid, X: ServerListRequest that
        Let res = G.List(X),
            sCode = G.List(X).statusCode,
            itemsPerPage = X.num_items
        in
    res.summary.numpages == res.summary.total/itemsPerPage,
        if  sCode == 200, itemsPerPage > 0;
        End
    End
```

The third axiom of the List operator states that, when each page contains N items, the i'th item on page k must be the same as the j'th item when there is no pagination, where

$$j = k \times N + i$$

```
Axiom <list.pagination2>:
For all  G: ServerGoGrid, i, j: int,
         X, X1: ServerListRequest that
    Let
        res  = G.List(X),       sCode  = G.List(X).statusCode,
        res1 =  G.List(X1),  sCode1 = G.List(X1).statusCode,
        n = X.num_items,   n1= X1.num_items
        k = X.page,
    in
        res.objects.items(i) == res1.objects.items(j),
            if  sCode == 200, sCode1 == 200,
                n > 0,   n1== 0,
                j == k*n + i,
                0 <= i, i < res.summary.returned;
    End
End
```

The fourth axiom states that when the number of items

per page is specified, the list of objects in a page either contains exactly the number of objects if the page is not the last, or at most that number if the page is the last. Moreover, the number of items in the result must equal the value of parameter returned in the result summary.

```
Axiom <list.pagination3>:
    For all  G: ServerGoGrid, i, j: int,
             X: ServerListRequest that
        Let
             result = G.List(X).objects,
             sCode = G.List(X).statusCode,
             n = X.num_items,
             nr = G.List(X).summary.returned,
             numPages = G.List(X).summary.numpages,
             lpage = X.page
        in
             result.length == nr,  if  sCode == 200, n > 0;
             nr == n,  if  sCode == 200, n > 0, lpage < numPages;
             nr <= n,  if sCode == 200, n > 0, lpage == numPages;
    End
End
```

An important property of the List operator is that, being an observer, it will not change the state of the system to which it is applied. This can be stated in the following axiom, though this need not be included in the specification because we have already declared the operator as an observer.

```
Axiom <List-Op>:
    For all  G: ServerGoGrid, X: ServerListRequest,
             X1: ServerXOpRequest that
        [G.List(X)].XOp(X1) == G.XOp(X1);
End
```

where XOp can be any of the operators List, Get, Add, Edit, Delete, etc.

Finally, when an operation does changes the state of the system, the List operator should be able to observe the difference accordingly. For example, the following axioms state the effect of Add when observed by the List operator.

```
Axiom <Add-List>:
    For all  G: ServerGoGrid, X1: ServerAddRequest,
             X2: ServerListRequest that
         [G.Add(X1)].List(X2).objects ==
              insert(G.List(X2).objects, G.Add(X1).objects),
                   If  X2.num_items == 0, X2.server_type == NULL,
                       X2.isSandbox == NULL, X2.datacenter == NULL,
                       G.Add(X1).statusCode == 200,
                       G.List(X2).statusCode == 200;
    End
```

The corresponding axioms for other operators are similar and are omitted to save space.

IV. DISCUSSION

In this section, we report the main findings of the case study.

A. Improving Document Preciseness

As one may expect, the ambiguity in the original documentation of the GoGrid API [21] was detected in the process of formalization. This documentation [21] specifies the data types of the API request parameters and their corresponding responses, and describes the meaning of each in normative text. Sample requests and responses are also given to explain the semantics and usage of the API.

In most cases, the meanings of the operations are left to the reader to interpret, according to his understanding. For example, in the description of the results of the List operator, the meaning of pagination according to num-items is not formally defined. We, however, specified it exactly to ensure that there is only one interpretation.

B. Detecting Incompleteness

Ambiguity in natural language documents is often caused when the specification is incomplete, meaning that some information is missing. The GoGrid documentation has this problem too. For example, in some cases, it is unclear about the range of values for a parameter and what will happen when the value is out of the range. An example of this is the num_items parameter, for the number of items in a page in a List request, which must be greater than 0, with no alternative behaviour specified for when it is not. Error codes are another example. It is unclear when each code will be returned. Both this and the num_items issue have been left unresolved because we do not have the relevant information.

There are two more serious cases of incompleteness, however. The List operation can list all the jobs in the system for a specified range of dates, but it can also list all the jobs for a specified object type, of a certain state or belonging to a certain owner. There is no documentation of these additional features. Another example concerns the relationship between requests and responses. The *id* parameter occurs in both the request and the response for the Get operator on Job objects. There is no statement explaining what the *id* parameter is for and the two occurrences are different in the samples given in the document, again with no explanation. Writing the formal specification has forced us to be precise and complete, making this incompleteness immediately apparent.

C. Checking Consistency

Cloud Computing is a relatively young field; so, some evolution in cloud software is inevitable. New API versions may emerge frequently. This often causes a mismatch between the software and its documentation. We detected many such cases for GoGrid. Here are three examples:

1. from version 1.5, GoGrid added a new general attribute called *datacenter* as a request query parameter but in the documentation of the Job object, there is no mention of this attribute.
2. similarly, there is no description of the attribute *numpages* in the documentation of the Get, Edit, and Delete operators even though it appears in the sample responses for these operators.
3. moreover, the parameter *port* in the Edit operation on

Copyright (c) IARIA, 2012. ISBN: 978-1-61208-230-1

LoadBalancer objects must satisfy *port >=0* but the condition is *port >0* for the other operations.

D. Reducing Redundancy

In an unstructured document, redundancy is also a common problem. The same information may arise in several different places with different descriptions although the meanings are the same. This often causes confusion. Take error code for example. There is a detailed description of error code in the chapter *Anatomy of a GoGrid API Call* but this is duplicated in the documentation for every API call. Another example is the three different response formats associated with each operation. These descriptions are duplications and occupy most of the space. The algebraic specification that we presented in the paper uses specification units to organize the document structure. Consequently, the redundancy is reduced.

E. Understandability of Document

An advantage of natural language documentation is its understandability. It is widely perceived by industry that formal methods are difficult to learn and expensive to apply. However, our case study demonstrates that without much training ordinary software developers can write algebraic specification, even for real industrial strength software systems like GoGrid. This confirms the discovery reported in [20]; that algebraic specification is easy to write and easy to understand. It can be part of the job of any ordinary software developer.

V. Conclusion and Future Works

In this paper, we applied the CASOCC-WS specification language to cloud computing interface with a case study on the GoGrid system. This demonstrated the value of algebraic specification for RESTful web services.

We are currently extending the algebraic specification language and studying its theoretical foundation. We are also developing a tool that uses the language as input to support automated testing of a cloud computing interface. The case study reported in this paper specifies only the functions of resource management that the GoGrid API original document specifies. However, the specification of the properties and dynamic behaviours of the resources are left as an open problem. Further case studies of the formal specification of PaaS and SaaS will also be conducted.

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 Province, China (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] M. Juric, B. Mathew and P. Sarang, Business Process Execution Language for Web Services, 2nd ed., Packt Publishing, Jan 2006.

[3] L. Richardson and S. Ruby, RESTful web services, O'Reilly, 2007.

[4] M. Papazoglou, Web services & SOA: principles and technology, Prentice Hall, 2012

[5] M. J. Hadley, Web Application Description Language (WADL), Sun Microsystems Inc., CA, USA, SMLI TR-2006-153, March 2006.

[6] J. Kopecky, K. Gomadam, and T. Vitvar, hRESTS: An HTML microformat for describing RESTful web services. In: Proc. of WI-IAT'08, Dec 2008, Sydney, Australia, IEEE/WIC/ACM.

[7] J. Lathem, K. Gomadam and A. P. Sheth, SA-REST and (S)mashups: Adding Semantics to RESTful Services, in Proc. of ICSC'07, 2007, pp469-476.

[8] O. Liskin, L. Singer, K. Schneider, Welcome to the Real World: A Notation for Modeling REST Services, IEEE Internet Computing, pp. 36-44, July-Aug., 2012.

[9] J. A. Goguen, J. W. Thatcher, E. G. Wagner, and J. B. Wright, Initial algebra semantics and continuous algebras, Journal of ACM, vol. 24, no. 1, pp. 68-95, 1977.

[10] H.-D. Ehrich, On the theory of specification, implementation, and parametrization of abstract data types, Journal of ACM, vol. 29, no. 1, pp. 206-227, 1982.

[11] J. A. Goguen and G. Malcolm, A hidden agenda, Theoretical Computer Science, vol. 245, no. 1, pp. 55-101, 2000.

[12] C. Cirstea, Coalgebra semantics for hidden algebra: Parameterised objects an inheritance, in Proc. of WADT'97, 1997, pp. 174-189.

[13] J. M. Rutten, Universal coalgebra: a theory of systems, Theoretical Computer Science, vol. 249, no. 1, pp. 3-80, 2000.

[14] C. Cirstea, A coalgebraic equational approach to specifying observational structures, Theoretical Computer Science, vol. 280, no. 1-2, pp. 35–-68, 2002.

[15] F. Bonchi and U. Montanari, A coalgebraic theory of reactive systems, Electr. Notes Theor. Comput. Sci., vol.209, pp.201-215, 2008.

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

[17] L. Kong, H. Zhu, and B. Zhou, Automated testing EJB components based on algebraic specifications, in Proc. of COMPSAC'07, vol.2, 2007, pp. 717-722.

[18] 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.

[19] GoGrid.com, http://www.gogrid.com, last access: July 10, 2012.

[20] H. Zhu and B. Yu, Algebraic specification of web services, in Proc. of QSIC'10, 2010, pp. 457-464.

[21] GoGrid.com, GoGrid wiki,https://wiki.gogrid.com/wiki/index.php, last access: July 10, 2012.

[22] H. Zhu and B. Yu, An experiment with algebraic specifications of software components, in Proc. of QSIC'10, 2010, pp. 190-199.

[23] T. A., Berson, Differential Cryptanalysis Mod 2^{32} with Applications to MD5. Proc. of *EUROCRYPT'92*. pp. 71–80, 1992.

Copyright (c) IARIA, 2012. ISBN: 978-1-61208-230-1