SOFIA: An Algebraic Specification Language for Developing Services

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

Hong Zhu and Ian Bayley
Dept of Comp. and Comm. Technologies
Oxford Brookes University
Oxford OX33 1HX, UK
hzhu@brookes.ac.uk, ibayley@brookes.ac.uk

***Abstract*—Describing the semantics of services accurately plays a crucial role in service discovery, execution, composition and interaction. Formal specification techniques, having evolved over the past 30 years, can define the semantics of software systems in a verifiable and testable manner. This paper presents a new algebraic specification language called SOFIA for describing the semantics of services. It unifies the approaches using algebras and co-algebras for software specifications. A case study with a real industry example, the GoGrid cloud's resource management services, demonstrates that the semantics of services can be specified in SOFIA.**

***Keywords*-formal specification; algebraic specification; service-oriented formalism in algebras (SOFIA); semantics of services**

I. Introduction

Service-oriented computing (SOC) is a computing paradigm that utilizes services as the fundamental elements for distributed computing. In this paradigm, the discovery and dynamic composition of services must be based on accurate computational understanding of their syntax and semantics. Most work in the description of service semantics, such as OWL-S [1] for the so-called *Big Web Services* and WADL [2] for the *RESTful web services*, has been based on ontology. In this approach, the semantics of a service is described by annotating its functions, and their input and output parameters, using a vocabulary defined in an ontology. Such descriptions have the advantages of being easy for human developers to understand and efficient for computers to process. However, most of these approaches are inadequate for providing a verifiable and testable definition of the functions of a service, because an ontology can do no more than define a vocabulary through the stereotypes of relationships between the concepts and their instances.

Algebraic specification was first proposed in the 1970s as an implementation-independent specification technique for abstract data types [3]. Over these years, it has been advanced to specify concurrent systems, state-based systems and software components based on the theories of behavioural algebras [4] and co-algebras [5]–[7].

In comparison with other formal approaches, algebraic specifications are at a very high level of abstraction and are thereby independent of implementation details. Another attractive feature they have is that they can be used directly in automated software testing [8]–[12]. This feature is particularly important for service engineering, because when services are composed together dynamically, testing must be performed automatically on-the-fly.

In our previous work, we extended and combined the behavioural algebra and co-algebra techniques, to apply them to service-oriented systems, and revised the algebraic specification language CASOCC, which was originally designed for traditional software entities such as abstract data types, classes and components [11], [12]. Its revised version CASOCC-WS was applied to the formal specification of Big Web Services [13]. A tool that can automatically generate the signatures of algebraic specifications from WSDL description of Big Web Services was also reported. More recently, we have also applied CASOCC-WS to the formal specification of RESTful Web Services and developed a tool to perform syntax-level consistency checking [14]. A case study with algebraic specification of a real industrial system GoGrid has been conducted [15]. Based on these works, we now propose a new algebraic specification language called SOFIA to improve the practical usability of algebraic specifications.

The remainder of this paper is organized as follows. In Section II, we present the algebraic specification language SOFIA. Section III reports a case study of a real industry example. Section IV concludes the paper with a discussion of future work.

II. Algebraic Specification Language SOFIA

The SOFIA language is designed for specifying the semantics of services in an accurate and machine-understandable manner. It is based on the theories of behavioural algebras and co-algebras. This section presents the syntax of the language, explains its semantics informally, discusses its design principles, and illustrates the style of specification with some examples. A formal definition of its semantics is beyond the scope of this paper and will be reported elsewhere.

A. Overall Structure of SOFIA Specifications

We regard a service-oriented system as consisting of a collection of software entities, which can be documents, XML document schemas, datatypes, classes, components, and most importantly, services. These software entities are linked to the real world through physical objects and equipment, data, abstract concepts, business processes, communication protocols, etc. All these varieties of objects, concepts, and processes are abstracted into various types of entities. Each type of entity is then specified by a specification unit.

Therefore, the overall structure of a SOFIA specification is a collection of specification units, reflecting the structure both of software systems and of the real world, conceptually. This also supports modular development of specifications, which is further enhanced by the splitting of each specification unit into two partial units: a *signature unit*, to define its syntax with a signature, and an *axiom unit*, to define the entity's semantics with a set of axioms that it must satisfy. The users can also define auxiliary functions and concepts in a Definition unit, which is particularly useful for defining concepts and functions that are common to many units.

```
<Specification> ::= <Unit>*
<Unit> ::= <Spec Unit> | <Signature Unit>
   | <Axiom Unit> | <Definition Unit>
```

Each specification unit contains two main parts: a *signature* to define the syntax and a set of *axioms* to define the semantics, as shown in the BNF rules below.

```
<Spec unit> ::= Spec <Sort Name> [<Observability>];
   [extends <Sort Names>] [uses <Sort Names>]
   <Signature>; [<Axioms>] End
```

where `<Sort Name>` is an identifier that names the unit. It corresponds to the type of software entities to be specified and is called the *main sort* of the unit.

A *Signature Unit* and an *Axiom Unit* with shared sort name, but supplied separately, is equivalent to a complete specification unit with corresponding signature and axiom parts.

```
<Signature unit> ::= Signature <Sort Name>
   [<Observability>];[extends <Sort Names>]
   [uses <Sort Names>] <Signature> End
<Axiom Unit> ::= Axioms <Sort Name>; <Axioms> End
```

In addition to specification units, a *Definition Unit* defines a set of auxiliary functions or concepts that are used in the specification.

```
<Definition unit> ::= Definition [uses <Sort Names>]
   <Signature>; [<Axioms>] End
```

We recognise two different ways in which a new unit can be constructed from existing ones, *extension* and *usage*. They are specified in the `<extends>` and `<uses>` clauses, respectively.

A unit can be extended with additional elements, in a manner similar to the inheritance relation of object-orientation. If a specification unit of sort A extends sort B, then it includes all constants, attributes, operations and axioms (explained later) in the specification unit of sort B together with the additional contents as specified.

A unit can use another unit, e.g. as a component, or as a parameter or a result from an operator, etc., just like the association relation of object-orientation. The list of sort names separated by a comma after the keyword *uses* gives the sorts that are used by the unit currently being specified.

SOFIA declares if a software entity is observable in the sense that its states or values can be directly tested for equality; otherwise, its states or values have to be checked by other means, e.g. through observers. The specification of observability has the following syntax:

```
<Observability> ::= is unobservable
   | is observable by <Op Id>
```

where the operator for an observable entity must be a binary function that returns a Boolean value. This function must be defined in the signature of the sort.

B. Signature

The signature specifies the syntactic elements of the software entity. SOFIA explicitly declares three kinds of operators: constants, attributes and operators. A constant is indicated by the keyword *Const*. Each constant identifier defines a specific constant of the sort.

```
<Signature> ::=
   {[<Constant>]|[<Attribute>]|[<Operator>]}*
<Constant> ::= Const : <ConstList>;
<ConstList> ::= <Const ID> [, <ConstList>]
<Const ID> ::= <Identifer>
```

An attribute is an element of the entity that represents its state. It is indicated by the keyword *Attr*.

```
<Attribute> ::= Attr <AttrList>;
<AttrList> ::= <AttributeType> [; <AttrList>]
<AttributeType> ::= <Attr IDs> : <Sort Name>
<Attr IDs> ::= <Attr ID> [, <Attr IDs>]
<Attr ID> ::= <Identifier> [ <index> ]
```

where when an attribute has an index, it actually specifies a set of attributes indexed by the index values. The Index set can be: (a) a consecutive set of natural numbers between an lower bound and an upper bound, which is either a specific natural number or "*", meaning indefinite, (b) an enumerated set of identifiers representing constants, and (c) a Cartesian product of the above two.

```
<Index> ::=  "[" <Index set> "]"
<Index set> ::= <Single index> [, <Index set>]
<Single index> ::= <Enumerated set>
   | <Lower bound>  ".." <Upper bound>
<Lower bound> ::= <Natural number>
<Upper bound> ::= <Natural number> | "*"
<Enumerated set> ::=
        <Enumerated ID> [, <Enumerated set>]
<Enumerated ID> ::= <Identifier>
```

An operator defines an operation on the entity. It changes the state of the entity when invoked, and may produce

outputs, too. The input parameters of an operator are given in the domain type and the outputs produced by an invocation are given in the co-domain. An operation always has access to the state of the entity and may change it.

```
<Operators> ::= Operation <OpList>;
<OpList> ::= <Operation> [; <OpList>]
<Operation> ::= <Operator ID>
   "(" <Domain Type> ")" ":" <Co-domain Type>
<Operator ID> ::= <Identifier>
<Domain Type> ::= <Type> | void
<Co-domain Type> ::= <Type> | void
<Type> ::= <Sort Name>[, <Type>]
```

Take `STACK` of natural numbers, for example, with the signature.

```
Spec STACK; uses BOOL, NAT;
 Const nilStack;
 Attr isNilStack: BOOL,
      top: NAT;
 Operation
  push(STACK, NAT): STACK;
  pop(STACK): STACK;
End
```

This means that `STACK` depends on `BOOL` for Boolean values and `NAT` for natural numbers. `nilStack` is a constant (a stack without any element), `isNilStack` and `top` are attributes, `push` and `pop` are operators.

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*. SOFIA 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.

C. Axiom

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

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

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

```
<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 uses clause. The variable identifiers must be unique: they must not clash with sort names, operator names nor with any such names in any used sorts nor with variables in this axiom.

The syntax rules for terms are as follows.

```
<Term> ::=  <Var ID> | <Constant ID>
  |<Op ID> ["("[<Parameters>]")"]
  |<Term> "." <Term> | "[" <term> "]"
  |<sort name> "<" <Term List> ">"
  |<Term> "#" <Number>
<Parameters> ::= <Term List>
<Term List> ::= <Term> ["," <Term List>]
```

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

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

Let $\varphi(w) : w'$ be an operator declared in a unit of sort s. The application of an operator φ to an entity e with parameters p is written in the form $e.\varphi(p)$. In particular, if w is VOID, we write $e.\varphi$.

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

```
<Equation> ::= <Condition> [, if <Conditions>]
  | Let <Var Definitions> in <Equation> End
<Conditions> ::=
  <Condition>[(","|"or")<Conditions>]
<Condition>  ::= <BOOL Term> |
  <Term> = <Term> | <Term> <> <Term>
<Var Definitions> ::=
  <Var Assignment> [, <Var Definitions>]
<Var Assignment> ::= <Var ID> = <Term>
```

The basic form of an equation is $t_1 = t_2$. Here is an example of sort `STACK`.

```
For all s: STACK, n: NAT That
  isNilStack(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, and these have the form:

$$\textbf{Let}\ \ x_1 = \tau_1, \cdots, x_n = \tau_n\ \ \textbf{in}\ \ equs\ \ \textbf{End}$$

where $x_1, \cdots x_n$ are local variables, limited in scope to $equs$, and $\tau_1, \cdots, \tau_n$ are terms denoting the values that are assigned to the variables. Local variables must have unique names, must not clash with other variables in this equation, nor with 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
    isNilStack(s1) = False;
    pop(s1) = s;
    top(s1) = n;
  End
End
```

III. Case Study

In this section, we report a case study of specifications written in SOFIA. We will specify a real industry RESTful web services provided by GoGrid.

GoGrid [16] 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 document, with which its customers can easily and dynamically 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 are based on the HTTP protocol, so each GoGrid API call is an individual HTTP query. The newest GoGrid API version 1.8 has 11 different types of objects. There are 5 types of common operators which can be applied to 8 objects. Some of the operators are not applicable to all types of objects, while some objects have their own special operators. Table I gives the applicable operators for each type of object.

Table I
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 in GoGrid have different meanings for different types of objects. So, in our specification of GoGrid, the definitions were grouped by object rather than by operator. For the sake of space, here we only give the specification of the server objects because they are one of the most important types of object and they also have the most operators.

For each type of object in GoGrid API, the formal specification in SOFIA consists of three types of specification units:

- Units that specify the valid requests, including their structures and constraints on how their components may be combined;
- Units that specify the responses with structures and constraints as above;
- Units that specify the objects of certain types, in terms of signatures and axioms, the latter to express semantics of operations

The specification of GoGrid API is based on the framework for specifying RESTful web services [17]. The framework consists of a collection of specification units that define the general structure of HTTP requests and responses so that a specific RESTful WS can be specified as extensions to these units. The details of the framework are omitted for the sake of space.

Table II
Number of Units in GoGrid Specification

Type of unit	No.
Framework of RESTful web service	10
Common features	37
Definition of Server operations	13
Definition of Server image operations	13
Definition of Load Balancer operations	11
Definition of Job operations	5
Definition of operations on other objects	14
Total	**103**

B. Objects and Collections

There are some objects that are related to server object including Option, IP, Server Image, Billing and Customer. Here we only give the specifications of server object and its collection ListofServer. From the specification of ListofServer, we can get each server object from the list and count the number of server objects in the list.

```
Spec Server; uses Option, IPO, ServerImage;
 Attr  id: Long;  name, description: String;
       ip: IPO; image: ServerImage;
       ram, state, type, os, datacenter: Option;
       isSandbox: Boolean;
 Axiom
  For all s: Server that s.id <> Null; End
End
Spec ListofServer; uses Server;
 Attr length: Integer;
 Operation
  items(Integer): Server;
  insert(Server);
End
```

C. Requests and Responses

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

```
Spec CommonParameter;
 Attr api_Key, sig, v, format: String;
 Axiom
  For all cp: CommonParameter that cp.v <> Null;
   cp.sig <> Null; cp.api_Key <> Null;
  End
End
```

where *api_key* is a key generated by GoGrid for security in the access of resources, *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 response format required.

Some parameters are common to all types of requests, but there are also further parameters depending on the type of request. So we first specify the structure of each type of request as one sort e.g. *ListRequest*, *GetRequest* and so on. Then the structure of each object's request is specified as

the extension of each type of request e.g. *ServerListRequest*, *LoadBalanceListRequest* etc. For the sake of space, here we give just the specification of list operation. A *server list* method call returns a list of server objects of a certain type in the GoGrid system. It is implemented using the HTTP request method GET. Note that such operations are the only way to determine the internal state of a service. We specify the list request and the list request of server object as follows.

```
Spec ListRequest;
 extends HTTPRequest;
 uses CommonParameter;
 Attr para: CommonParameter;
      num_items, page: Integer;
 Axiom
  For all lr: ListRequest that lr.num_items >= 0;
   lr.page >=0, if lr.num_items > 0;
  End
End
Spec ServerListRequest;
 extends ListRequest;
 uses ListofString;
 Attr server_type: String;
      isSandbox: Boolean;
      datacentre: ListofString;
End
```

where *para* denotes the common query parameters defined in last subsection, *num_items* is the number of items to return so that this value will effectively paginate the results into a number of pages with this number of items per page, and *page* is the page index to return for paginated results, indexed from 0. This parameter is ignored if *num_items* is not specified. The sort *ServerListRequest* extends *ListRequest* with an additional three parameters *server_type*, *isSandbox*, *datacenter* which are used to filter server objects.

2) List Responses: 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 a *summary*, which can be specified as follows:

```
Signature ResponseSummary;
 Attr total, start, returned, numpages: Integer;
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 current num-items value.

In addition to *summary* of the list, the response to a list call contains *status*, *request method*, *status code* and a *list of returned objects*. The meaning of the status code is as follows: 200 means that the call was successful, 4xx means there was 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. If the status code is 5XX, it means a server error occurred. We first specify the structure of the list response, then the structure of the list response of server object can be extended with a *list of returned server objects*.

```
Spec ListResponse;
 extends HTTPResponse;
 uses ResponseSummary;
 Attr summary: ResponseSummary;
      status, request_method: String;
      statusCode: Integer;
 Axiom
  For all lr: ListResponse that
   lr.summary.total >=0; lr.summary.start >=0;
   lr.summary.returned >=0; lr.summary.numpages >=0;
  End
End
Spec ServerListResponse;
 extends ListResponse;
 uses ListofServer;
 Attr objects: ListofServer;
End
```

D. Semantics of the operations

For each type of request, we define an operator that takes request as input and produces a response as the output. All such operators have GoGrid as the context. We also need to know the clock time on the grid, the shared secret chosen by each user and the timestamp for checking the access authentication. Thus, we have the following signature for the sort ServerGoGrid, which represents the Server web service of GoGrid cloud computing system.

```
Spec GServer;
  uses  ServerListRequest, ServerListResponse,
        ServerGetRequest, ServerGetResponse,
        ServerAddRequest, ServerAddResponse,
        ServerEditRequest, ServerEditResponse,
        ServerDeleteRequest, ServerDeleteResponse,
        ServerPowerRequest, ServerPowerResponse;
  Attr sharedSecrte: String;
       clockTime, timeStamp: Integer;
  Operation
   List(ServerListRequest): ServerListResponse;
   Get(ServerGetRequest): ServerGetResponse;
   Add(ServerAddRequest): ServerAddResponse;
   Edit(ServerEditRequest): ServerEditResponse;
   Delete(ServerDeleteRequest): ServerDeleteResponse;
   Power(ServerPowerRequest): ServerPowerResponse;
End
```

For each operator, the semantics can be characterised by a set of axioms, but for the sake of space, we shall give two axioms to illustrate the style of specification with the list operator.

An important feature of the List operator is that it is an observer. So applying it will not change the state of the context sort ServerGoGrid. This property can be expressed by axioms of the following form, though note that in this case it is unnecessary as we have already declared the operator as an observer.

```
For all G: ServerGoGrid, X: ServerListRequest,
```

```
        X1: ServerXOpRequest that
 [G.List(X)].XOp(X1) = G.XOp(X1);
End
```

where `XOp` is any of the operators `List`, `Get`, `Add`, `Edit`, `Delete`, etc.

The following axiom states that when an operation changes the state of the server, e.g. by adding, deleting, editing and powering a server, the List operator should be able to observe the difference accordingly. In fact, these axioms also define the semantics of the operators that change the state of the system.

```
For all G: GServer, X1: ServerDeleteRequest,
        X2: ServerListRequest that
 [G.Delete(X1)].List(X2).statusCode = 500,
 if search(X2.name, X1.name) = True,
    G.Delete(X1).statusCode = 200;
 [G.Delete(X1)].List(X2).objects = G.List(X2).objects,
 if search(X2.name, X1.name) = False,
    G.Delete(X1).statusCode = 200,
    G.List(X2).statusCode = 200;
End
```

IV. Conclusion and Future Work

In this paper, we presented the design of the algebraic specification language SOFIA and illustrated its style of formal specification with examples. We also reported a case study of a real industry cloud system GoGrid to demonstrate the value of algebraic approach in the development of services.

We have already implemented the SOFIA language for checking the syntax correctness and type consistency of specification units. We have also developed a tool to translate algebraic specifications in SOFIA into ontological descriptions of service semantics [17].

We are currently developing a tool that uses specifications in SOFIA as input to perform automated testing and verification of web services. Another future work is to check the consistency of specification based on ontological reasoning as well as equational logic inferences.

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), National Natural Science Foundation of China (Grant No. 61272420), National Natural Science Foundation of Jiangsu Province (Grant No. BK2011022), and the Jiangsu Qinglan Project.

References

[1] D. Martin, et al., *OWL-S: Semantic Markup for Web Services*, member submission 22 ed., W3C, http://www.w3.org/Submission/OWL-S/, November 2004, last access: May 25, 2012.

[2] M. J. Hadley, "Web application description language (WADL)," Sun Microsystems Inc., CA, USA, Tech. Rep. SMLI TR-2006-153, March 2006.

[3] 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.

[4] J.A. Goguen and G. Malcolm, "A hidden agenda," *Theorectical Computer Science*, vol. 245, no.1, pp.55–101, 2000.

[5] C. Cîrstea, "Coalgebra semantics for hidden algebra: Parameterised objects and inheritance," in *Recent Trends in Algebraic Development Techniques, 12th International Workshop (WADT'97)*, 1997, pp.174–189.

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

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

[8] M.-C. Gaudel and P. L. Gall, "Testing data types implementations from algebraic specifications," *CoRR*, vol. abs/0804.0970, 2008.

[9] H. Y. Chen, T. H. Tse, F. T. Chan, and T. Y. Chen, "In black and white: An integrated approach to class-level testing of object-oriented programs," *ACM Trans. Softw. Eng. Methodol.*, vol. 7, no. 3, pp. 250–295, 1998.

[10] 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 Trans. Softw. Eng. Methodol.*, vol. 10, no. 1, pp. 56–109, 2001.

[11] L. Kong, H. Zhu, and B. Zhou, "Automated testing ejb components based on algebraic specifications," in *Proceedings of the 31th IEEE International Conference on Computer Software and Applications (COMPSAC'07)*, vol. 2. Beijing, China: IEEE CS Press, July 2007, pp. 717–722.

[12] B. Yu, L. Kong, Y. Zhang, and H. Zhu, "Testing java components based on algebraic specifications," in *Proceedings of the First International Conference on Software Testing, Verification, and Validation (ICST 2008)*, Lillehammer, Norway, April 9-11 2008, pp. 190–199.

[13] H. Zhu and B. Yu, "Algebraic specification of web services," in *Proc. of the 10th International Conference on Quality Software (QSIC 2010)*. IEEE CS Press, 2010, pp.457–464.

[14] D. Liu, H. Zhu, and I. Bayley, "Applying algebraic specification to cloud computing–a case study of infrastructure-as-a-service gogrid," in *Proceeding of The Seventh International Conference on Software Engineering Advances (ICSEA 2012)*, 2012, pp. 407–414.

[15] ——, "A case study on algebraic specification of cloud computing," in *Proc. of the 21st Enuromicro International Conference on Parallel, Distributed and network-Based Processing (PDP 2013)*, Queens University Belfast, Northern Ireland, Feb. 2013, pp. 269-273.

[16] GoGrid.com, "Gogrid website," http://www.gogrid.com/, last Access: Nov, 2013.

[17] D. Liu, H. Zhu, and I. Bayley, "From Algebraic Specification to Ontological Description of Service Semantics, *Proceedings of the 20th International Conference on Web Services (ICWS 2013)*, Santa Clara, USA, Jun. 2013, pp.579-586.