International Journal of SERVICES COMPUTING

2014 VOL.02 NO.01

A Technical Publication of the Services Society

ISSN : 2330-4464 (Print) ISSN : 2330-4472 (Online)

TRANSFORMATION OF ALGEBRAIC SPECIFICATIONS INTO ONTOLOGICAL SEMANTIC DESCRIPTIONS OF WEB SERVICES

Dongmei Liu[1], Hong Zhu[2] and Ian Bayley[2]
[1]Nanjing University of Science and Technology, Nanjing, China; [2]Oxford Brookes University, Oxford, UK
dmliukz@njust.edu.cn, hzhu@brookes.ac.uk, ibayley@brookes.ac.uk

Abstract
The accurate description of service semantics plays a crucial role in service discovery, composition and interaction. Most work in this area has been focused on ontological descriptions, which are searchable and machine-understandable. However, they do not define service functionality in a verifiable and testable manner. In contrast, formal specification techniques, having evolved over the past 30 years, can define semantics in such a manner, but they have not yet been widely applied to service computing because the specifications produced are not searchable. There is a huge gap between these two methods of semantics description. This paper bridges the gap by advancing a transformation technique. It specifies services formally in an algebraic specification language, and then, extractsan ontological description of domain knowledge and service semantics as profiles in an ontology description language such as OWL-S. This brings the desired searchability benefits. The paper presents a prototype tool for performing this transformation and reports a case study to demonstrate the feasibility of our approach.
Keywords: Web services, Formal semantics, Algebraic specification, Ontology, OWL-S

1 INTRODUCTION

The advent of Web Services technology has greatly influenced the uptake and use of the paradigm of service-oriented computing. In this paradigm, services are autonomous, platform-independent and distributed computational entities (Papazoglou, 2012). Various techniques have been advanced to enable automated discovery, execution, composition and interoperation of services at runtime. Such techniques heavily depend on accurate descriptions of the semantics of services (Singh & Huhns, 2005). Ideally, such descriptions should be:

- *Comprehensible*as published documentation for developers of software that use the services.
- *Abstract*, hiding design and implementation detail to protect the vendor's intellectual property, and for other reasons.
- *Searchable*at run-time, since dynamic search and composition unlocks the full power of service-oriented computing. Services must be described with an interface syntax and specified with a functional semantics.Both must be machine understandable.
- *Testable* at run-time since dynamic composition delays integration testing until then, when service has already been deployed. Services must be highly reliable, and correct with respect to their semantic descriptions. Both providers and requesters must be able to verify this.

However, as we shall see in next subsection, no existing technique satisfies all of these requirements at once. This paper integrates existing techniques in an attempt to do so.

1.1 EXISTING WORK AND THE OPEN PROBLEM

Existing techniques for semantics descriptions of services are divided into two categories: *ontology-based approach* and *formal method based approach*. The former, comprising the majority of research, uses a vocabulary defined in application domain ontologies to annotate services; while the latter uses mathematical notations to formally define the functions of the software system.

Semantic Web Services have been proposed, and advanced,in the context of Big Web Services (i.e. those based on WSDL, SOAP and UDDI, etc.). They describe services using metadata based on domain ontologies (Mallraith, Son, & Zeng, 2001). OWL-S was the first major ontology definition language for this purpose (Martin & al., 2004). It provides a set of constructs for describing the properties and capabilities of Web Services in a machine-readable format. Formal methods were applied to provide a precise mathematical meaning in a formal ontology.

An alternative approach is the Web Service Modelling Ontology (WSMO) proposed by De Bruijn et al. (2005), a conceptual model that uses the Web Services Modelling Language (WSML) (Bruijn &et al., 2006).

As well as Big Web Services, work has also been carried out on how to specify the semantics of RESTful web services (Richardson & Ruby, 2007), such as, MicroWSMO/hRESTS (Kopecky, Gomadam, & Vitvar, 2008), WADL (Hadley, 2006) and SA-REST (Lathem, Gomadam, & Sheth, 2007).

This paper is an extended and revised version of the conference paper (Liu, Zhu, &Bayley, 2013b) presented at the IEEE 20th International Conference on Web Services (ICWS 2013).

The above mentioned works all take the same approach to specifying the semantics of services.A vocabulary is defined in an application domain ontology to give the meanings of the input and output parameters, as well as the functions of the services. Such descriptions are easy for human developers to understand and efficient for computers to process. However, they cannot provide a verifiable and testable definition of a service's function, because any ontology is limited to stereotypes formed from the relationship between the concepts and their instances.

Formal methods, which we consider as an alternative to the ontological approach, have been developed over the past 40 years to define the semantics of software systems in mathematical notations. One such formal method, algebraic specification was first proposed in the 1970s as an implementation-independent specification technique for defining the semantics of abstract data types (Ehrich, 1982; Goguen et al., 1977). Over these years, it has been advanced to specify concurrent systems, state-based systems and software components, all based on solid foundations of the mathematical theories of behavioural algebras (Goguen & Malcolm, 2000) and co-algebras (Bonchi & Montanari, 2008; Cirstea, 1997, 2002; Rutten, 2000).

Algebraic specifications are at a very high level of abstraction. They are independent of any implementation details. One attractive feature they have is that they can be used directly in automated software testing (Chen et al., 1998; Chen, Tse, & Chen, 2001; Gaudel & Gall, 2008; Kong, Zhu, & Zhou, 2007; Yu et al., 2008). This feature is particularly important for service engineering, because, when services compose together dynamically, testing must be performed automatically on-the-fly.

The algebraic method has been applied to service-oriented software by extending and combining the behavioural algebra and co-algebra techniques. Zhu and Yu (2010) originally applied the algebraic specification language CASOCC to define traditional software entities, such as abstract data types, classes and components (Kong, Zhu, & Zhou, 2007; Yu et al., 2008). They then extended the language to form CASSOC-WS and applied that to Big Web Services (Zhu & Yu, 2010). They developed a tool that can automatically generate the signatures of algebraic specifications from WSDL descriptions of Big Web Services. More recently, CASOCC-WS was also applied to RESTful web services. A tool was developed for it that performs syntax level consistency checking (Liu, Zhu, & Bayley, 2012), and a case study was conducted applying CASOCC-WS to a real industrial system, GoGrid (Liu, Zhu, & Bayley, 2013a). Based on these works, a new algebraic formal specification language called SOFIA was proposed to improve the practical usability of algebraic specification languages when applied to services (Zhu, Liu, & Bayley, 2013; Liu, Zhu& Bayley, 2014).

However, algebraic specifications, do not directly support efficient searching on services, and nor do other formal methods. This weakness has hampered their adoption for services because such searching is crucial for service-oriented computing. Service semantics must be specified in a testable and verifiable way and these specifications must be searchable.

In summary, with a vocabulary defined in an application domain ontology as annotation, we can create searchable and comprehensible descriptions. With the mathematical notations of formal methods, on the other hand, we can create descriptions that are testable and verifiable. Each approach has its strengths and weaknesses. The problem is how can we benefit from both strengths?

1.2 Proposed Approach and Main Contributions

To bridge the gap between algebraic specification and ontological descriptions, this paper proposes a transformational approach. Algebraic specifications are written for services and then transformed with the support of an automated tool into an ontology-based semantics description, thereby conferring onto formal specifications the machine-readability and human-understandability benefits of ontologies.

The main contributions of the paper are three-fold.

First, we propose a framework to solve the problem stated in the previous subsection. The semantics of a service and its domain knowledge are both described in a formal specification language. The domain knowledge is automatically transformed into adomain ontology, while the semantics is transformed into an ontology-based service description.

Second, we present the details of these two transformations in the form of transformation rules. We also report their implementation in an automated tool.

Finally, we demonstrate the feasibility of our solution with a case study of an actual industrial system called GoGrid. It is a RESTful web service interface toan Infrastructure-as-a-Service (IaaS).

To our knowledge, the only similar work that has ever been reported in the literature is (Doell & Dosch, 2005), where traditional algebraic specification signatures are transformed into object-oriented class signatures. However, such traditional signatures cannot be used for specifying services; we will see why in the next section. A further problem is that the language is not modularized enough to separate the definition of domain knowledge from the specification of service functional semantics. This makes the two transformations much more complicated, if not impossible. For example, when transforming an operation into a method, it is unclear which class to put it into. Our approach overcomes this difficulty by associating only one sort with each modular unit of specification.

1.3 Structure of the Paper

The remainder of the paper is organised as follows. Section 2 defines preliminary mathematical notions and the notations of algebraic/co-algebraic specification. It also

briefly introduces the specification language SOFIA. Section 3 presents the mapping rules that translate algebraic specifications into ontologies and the rules that extract the ontological descriptions of the service semantics. Section 4 describes the prototype tool TrS2O that implements both sets of rules for the SOFIA language. It represents the resulting ontology and service semantics in OWL and OWL-S profiles. Section 5 reports the case study of the GoGrid API. Section 6 concludes the paper with a discussion of future work.

2 PRELIMINARIES

In this section, we define preliminary mathematical notions and notations. We also briefly introduce the SOFIA language.

2.1 ALGEBRAIC STRUCTURES

We regard a service-oriented system as consisting of a collection of units. Each unit has a unique identifier, which is called the *sort* name. We recognise two different ways in which one unit can be constructed from another, *extension* and *usage*, as follows:

(1) A unit can be extended with additional elements, in a manner similar to the inheritance relation of object-orientation. The notation $s \rhd s'$ means that *s extends s'*, i.e. *s* inherits all the operations and axioms defined in *s'*.

(2) A unit can use another unit, e.g. as a component, operation parameter or operation result, just like the association relation of object-orientation. Such usage is denoted by the notation $s \succ s'$, which means that *s uses s'*.

As in (Zhu, 2003), we assume that the specification of a software system is well-structured in the following sense.

1) Each type of software entity has a corresponding specification unit with a unique sort name.
2) Each type of real-world entity involved in the software system is specified by a corresponding specification unit with a unique sort name.
3) The same is also true for each real-world concept.
4) Any extension or usage relationship between specification units has a corresponding relationship between real-world counterparts and vice versa.

Together, a set of specification units, extension relation and usage relation comprise a *system signature*, defined formally as follows.

Definition 1.(*System Signature*)A *system signature* is an ordered pair (Sp, Σ), where $Sp = \langle S, \succ, \rhd \rangle$ is a set *S* of sorts with two binary relations on S denoted by $\succ$ and $\rhd$, and $\Sigma = \{\Sigma^s | s \in S\}$ is a collection of unit signatures, with Σ^s denoting the unit signature for sort *s*.

Every kind of software entity, whether it is an abstract data type, a class, a component or, as here, a service, must define a set of typed operators. The syntactic aspect of an operator is determined by its domain, its co-domain and its identifier.This is specified in the following form.

$$op: (s_1, s_2, \dots, s_n) \to (s'_1, s'_2, \dots, s'_k)$$

where *op* is the identifier of the operator, $(s_1, s_2, \dots, s_n), n \geq 0$, are the domain sorts, and $(s'_1, s'_2, \dots, s'_k), k > 0$, are the co-domain sorts.

We allow an operator to have more than one domain sort and more than one co-domain sort at the same time. This is the main difference between our theory and that used for algebraic specifications, which require a single sort co-domain, and that used for co-algebraic specifications, which require a single sort domain. These restrictions are too tight to specify services so they are relaxed in our theory. This allows us, for example, to give a BookTicket operator for an online ticket booking service a signature like this:

BookTicket: DATE, NAT, BOOKING -> MESSAGE, BOOKING

Here, DATE is the date of the performance, NAT is the number of tickets wanted, MESSAGE is the response to the requester. BOOKING represents the state of the online booking services. It occurs in both the domain and the co-domain so that the original state can be taken as input and the modified state can be produced as output.

We now define the notion of *unit signature* to represent the structure of software units as follows.Let *X* be a finite set of symbols. We write X* to denote the set of finite sequences of the symbols in X. In the sequel, we use W_s to denote $\{x \in S | s \succ x \vee x = s\}^*$.

Definition 2.(*UnitSignature*)Given a system signature (Sp, Σ), the unit signature $\Sigma^s \in \Sigma$ for a sort $s \in S$ consists of a finite family of disjoint sets $\Sigma^s_{w,w'}$ indexed by pairs of units (w, w') with $w, w' \in W_s$. Each element φ in set $\Sigma^s_{w,w'}$ is an *operator symbol* of type $w \to w'$, where *w* is the *domain type* and *w'* the *co-domain type* of the operator.

Such operators can be classified as *constants*, *attributes*, and *general operations* as follows.

(1) φ is a *constant*, if $w = \emptyset, w' = (s)$,
(2) φ is an *attribute*, if $w = (s), w' = (s')$ and $s \succ s'$,
(3) Otherwise, φ is a *general operation*.

In the sequel, we will write Σ^s_C, Σ^s_A, and Σ^s_G for the subsets of Σ^s that contain the constants, the attributes and the general operations, respectively.

The semantics of the operators are defined by axioms that describe the properties that these functions must satisfy. An axiom consists of a number of universally quantified variables and a list of conditional equations.

Let (Sp, Σ) be a given system signature and $s \in S$ be any given sort. We define the notion of valid terms that can be used in the specification unit of sort *s* as *s*-terms. Each *s*-term is also typed. Each $w \in W_s$ is a *type* in unit *s*.Formally, we have the following definition.

Definition 3.(*Term*)For a unit $s \in S$, the set T^s of valid terms in *s*, called *s*-terms, is a family of disjoint sets $\{T^s_w | w \in W_s, s \in S\}$. Here, each T^s_w is the set of *s*-terms of type *w*, and is inductively defined as follows.

(1) *x* is an *s-term of* type *w*, if $x \in V_w^s$, where V_w^s is the set of variables in *s* of type *w*.
(2) For each$(op: \emptyset \to s) \in \Sigma_C^s$, *op* is an *s*-term of type *s*.
(3) For each $(op: s \to s') \in \Sigma_A^s$, $op(t)$ is a *s*-term of type *s'*, if *t* is an *s*-term of type *s*.
(4) For each $(op: w \to w') \in \Sigma_G^s$, $op(t)$ is an *s*-term of type *w'*, if *t* is an *s*-term of type *w*.
(5)$\langle \tau_1, \tau_2, \dots, \tau_n \rangle$is an s-term of type *w*, if τ_i is an *s*-term of types_i, for$i = 1,2, \dots, n$, where$w = (s_1, s_2, \dots, s_n)$.
(6) $\tau \# k$is an *s*-term of types_k, if $\tau$is an *s*-term of type $(s_1, s_2, \dots, s_n)$,and 0<*k*≤*n* is a natural number.

An equation in specification unit *s*has the form $\tau = \tau'$, where$\tau$and τ' are s-terms of the same type. A conditional equation in specification unit *s*has the form

$$\tau = \tau', \text{if } c_1 = d_1, \dots, c_n = d_n,$$

where $\tau$and $\tau'$are *s*-terms of the same type, c_i and d_i are *s*-terms of types$_i$such that $s \succ s_i \vee s_i = s$ for all $i = 1,2, \dots, n, c_1 = d_1, \dots, c_n = d_n$are the conditions.

An axiom in the specification unit *s* is a conditional or unconditional equation *E* with all variables in the equation universally quantified at the outermost.

A specification unit consists of a unit signature and a set of axioms.

Definition 4**.** (*Specification*) A *specification* is a triple(Sp, Σ, Ax) , where
(1) $Sp = \langle S, \succ, \rhd \rangle$, *S* is finite set of sorts,$\rhd$ and $\succ$ are the *extends* and *uses* relations on *S*, respectively;
(2) $\Sigma = \{\Sigma^s | s \in S\}$is a set of unit signatures indexed by *s*;
(3) $Ax = \{Ax^s | s \in S\}$is a finite collection of axiom sets indexed by *s*;
(4) for all *s* and $s' \in S$, $s \rhd s'$ implies that $\Sigma^{s'} \subseteq \Sigma^s$ and $Ax^{s'} \subseteq Ax^s$.

For each$s \in S$, (Σ^s, Ax^s) is called the *specification unit* for sort *s*.

Note that, by Definition 2, a specification consists of a system signature (Sp, Σ), and a collection *Ax* of axiom sets.

2.2 SEMANTICS OF ALGEBRAIC SPECIFICATION

We now define the semantics of algebraic specifications by defining what it means for an implementation to be correct with respect to a specification. In general, an implementation of a specification is a mathematical structure that realises the operators in the signature and satisfies the axioms.

Definition 5**.** (*Algebra*) Given a system signature(Sp, Σ) , a (Sp, Σ) *-algebra* Γ is a mathematical structure (*A*,*F*) that consists of a collection$A = \{A_s | s \in S\}$ of sets indexed by *s*, and a collection *F* of functions indexed by (w, w'), where $w, w' \in W_s, s \in S$ such that for each operator $\varphi: w \to w'$, the function $f_\varphi \in F$ has domain A_w and co-domain $A_{w'}$, where$A_u = A_{s_1} \times A_{s_2} \times \dots \times A_{s_n}$, when $u = (s_1, s_2, \dots, s_n)$.

The evaluation of a term in an algebra depends on the values assigned to the variables that occur in the term. Such an assignment α of variables $V_s, s \in S$, in an algebraΓ is a function fromV_s to A_s.

Definition 6**.** (*Evaluation of terms in an algebra*) Given an assignment α , the evaluation of a term τ in an (Sp, Σ) -*algebra*$\Gamma = (A, F)$, written$Eva_\alpha(\tau)$, is defined as follows.
(1) $Eva_\alpha(v) = \alpha(v)$;
(2)$Eva_\alpha(\varphi(\tau)) = f_{A,\varphi}(Eva_\alpha(\tau))$;
(3)$Eva_\alpha(\langle \tau_1, \tau_2, \dots, \tau_n \rangle) =$
$\langle Eva_\alpha(\tau_1), Eva_\alpha(\tau_2), \dots, Eva_\alpha(\tau_n) \rangle$;
(4) $Eva_\alpha(\tau \# k) = e_k$, if $Eva_\alpha(\tau) = \langle e_1, \dots, e_n \rangle$, and $1 \le k \le n$.

Definition 7**.** (*Satisfaction*) Let *e* be an equation in the following form.

$$\tau = \tau', \text{if } c_1 = d_1, \dots, c_n = d_n.$$

An (Sp, Σ)*-algebra*$\Gamma = (A, F)$ *satisfies e*, written$\Gamma \vDash e$, if for all assignments α, we have that$Eva_\alpha(\tau) = Eva_\alpha(\tau')$ whenever$Eva_\alpha(c_i) = Eva_\alpha(d_i)$ is true for all$i = 1,2 \dots, n$.

Let$\varepsilon = (Sp, \Sigma, Ax)$ be a specification. An(Sp, Σ)-*algebra* $\Gamma = (A, F)$satisfies specificationε, written$\Gamma \vDash \varepsilon$, if for all equations *e* in *Ax*, we have that $\Gamma \vDash e$.

2.3 THE SOFIA SPECIFICATION LANGUAGE

SOFIA is a new algebraic specification language designed for the formal specification of services. It is based on the algebraic structure described above. Here, we give a brief introduction to the language. The readers are referred to (Zhu, Liu, & Bayley, 2013) for the reference manual.

The overall structure of a SOFIA specification is a collection of specification units. A unit can be split into two partial units: a *Signature* unit, to define the signature, and an *Axiom* unit, to define the axioms that apply to the signature unit. The users can also define auxiliary functions and concepts in a *Definition* unit. More formally, in BNF notation we have:

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

The "uses" and "extends" relations between specification units are declared in clauses introduced with the keywords*uses*and *extends*, as shown below.

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

SOFIA also 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.

SOFIA explicitly declares three kinds of operators using keywords *Const* for constants, *Var* for attributes, and *Operation* for general operators. For example, the

following is the signature unit in the SOFIA specification of Stack.

```
Signature Stack;
    uses Int, Real, Bool;
    Const: nil;
    Var
    Length: Int;
    IsEmpty: Bool;
    Top: Real;
    Operation
    Push(Real);
    Pop;
    End;
```

Note that SOFIA assumes that the sort name of the unit occurs on both sides of the general operators. Thus, Push(Real)is syntactic sugar for Push: Stack, Real -> Stack.

An axiom in SOFIA is in the form of

```
for all x1: s1, x2: s2, ..., xn: sn that
    e1 = e2, if cond;
```

where x1, ..., xn are universally quantified variables that occur in the equation, and s1, ..., sn are their respective sorts. For example, the axioms for Stack are as follows.

```
for all x: Real, s: Stack that
    s.push(x).length = s.length+1;
    s.push(x).IsEmpty = False;
    s.push(x).top = x;
    s.push(x).pop = s;
    s.pop.length = s.length-1, if s.length>0;
    s.length=0, if s.IsEmpty= True;
    s.IsEmpty = True, if s.length=0;
    s.IsEmpty = False, if s.length>0;
    nil.IsEmpty = True;
```

SOFIA uses the *prefix-dot* notation for the application of an operator to the main sort.

To improve the readability of axioms, the language also allows the definition of local variables/identifiers for use in equations. The following is an example.

```
for all x: Real, s: Stack that
  let s' = s.push(x) in
    s'.length = s.length+1;
    s'.IsEmpty = False;
    s'.top = x;
    s'.pop = s;
end
```

3 TRANSFORMATION RULES

Anontology defines the concepts in a domain througha set of relations between them. Individual entities are the instances of these concepts. In ontology modeling languages, such as OWL, concepts are often modeled as *classes*. Relations are modeled as *properties* to describe the features and attributes of the concepts. Individuals are modeled as objects,which are instances of the classes that represent the corresponding concepts. Such an ontologyis a representation of domain knowledge (Uschold & Gruninger, 1996).

In this section, we present a set of mapping rules to derive ontological descriptions of services from algebraic specifications. We use general algebraic structures rather than the concrete syntax of SOFIA so that the rules are generally applicable.

3.1 EXTRACTION OF DOMAIN ONTOLOGY

Given an algebraic specification (Sp, Σ, Ax) , the following rules will extract classes, properties and individuals from algebraic specifications, and thus translate an algebraic specification into a domain ontology.

Rule 1: For each sort$s \in S$ of the specification, generate a formula *Class(s)*, wherepredicate *Class(x)* means that *x* is a class or, in other words, *x* is a concept.

Rule 2: For an extension relation $s \rhd s'$ in the system signature (Sp, Σ) of the specification, generate a formula *subClassOf*(*s, s'*), where predicate *subClassOf*(*x, y*)means that class *x* is a subclass of *y*, or equally, *x* is a sub-concept of *y*.

Rule 3: For a uses relation$s \succ s'$ in the system signature (Sp, Σ) of the specification, generate a formula *uses*(*s, s'*), where predicate *uses*(*x, y*) means that concept *x* is defined by using the concept *y*, it is somewhat redundant because it can be deduced from other predicates later on.

Rule 4: For each constant$\varphi \in \Sigma_C^S$,

(1) Generate a formula *Individual*(φ), where predicate *Individual*(*y*) means that *y* is an individual, and

(2) Generate a formula *s*(φ), where *x*(*y*)means that *y* is an instance of class *x*.

Rule 5: For each operator: $s \to s', \varphi \in \Sigma_A^S$,

(1) Generate a formula *Property*(φ), where predicate *Property*(*z*) means that *z* is a property, and

(2) Generate a formula φ(*s, s'*), where *z*(*x, y*) means that *z* is a property of concept *x* (i.e. an attribute or an element of *x*), and its value is of type *y*.

Rule 6: For each general operation: $w \to w', \varphi \in \Sigma_G^S$,

(1) Generate a formula *Class*(φ), where predicate *Class*(*z*) means that *z* is a concept, and

(2) For each$s_i \in w$, generate a formula is*DomainOf*(φ,s_i), where *isDomainOf*(*z, x*) means that *x* is the domain of the relation *z, and*

(3) For each $s_i \in w'$, generate a formula *isCodomainOf*(φ, s_i) where the predicate *isCodomainOf*(*z, x*) means that *x* is the co-domain (or range or output) of the relation *z*..

To explain Rule 6, we regard an operation as a relation (i.e. a relational concept) that links the concepts of the domain to the concepts of the co-domain.

3.2 GENERATION OF SERVICE PROFILE

Having generated the ontology from a specification, the services can be described in an OWL-S profile based on the ontology. Such a profile can also be generated from the specification unit that defines the service's functionality.

Given a specification (Sp, Σ, Ax) of service S_v, the following rule will generate the service profile.

Rule 7: For each general operation $\varphi: w \to w', \varphi \in \Sigma_G^S$,

(1) Generate a service profile frame.

(2) Generate an element *serviceName* with value $s.\varphi$.

(3) For each $s_i \in w$, generate an element *hasInput* with resource "S_v.owl#s_i".

(4) For each $s_i \in w'$, generate an element *hasOutput* with resource "S_v.owl#s_i".

Figure 1 illustrates the above transformation rule.

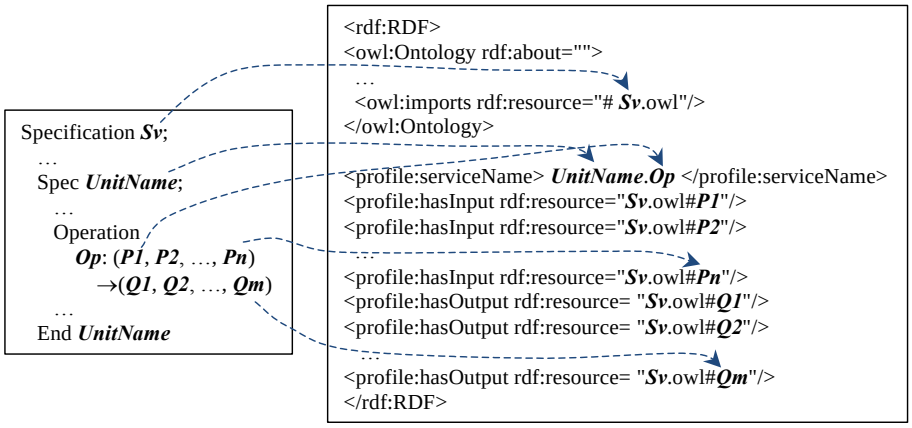

Figure 1. Illustration of Rule 7

For example, here is the specification unit in the SOFIA language that defines the operations on Servers in the GoGrid system. The axioms are omitted since they are not used in the translation.

```
Spec GServer;
 uses ServerListRequest, ServerListResponse,
       ServerGetRequest, ServerGetResponse,
       ServerAddRequest, ServerAddResponse,
       ServerEditRequest, ServerEditResponse,
       ServerDeleteRequest, ServerDeleteResponse,
       ServerPowerRequest, ServerPowerResponse;
 Var   clockTime: Int;
 Operation
   List(ServerListRequest) : ServerListResponse;
   Get(ServerGetRequest) : ServerGetResponse;
   Add(ServerAddRequest) : ServerAddResponse;
   Edit(ServerEditRequest) : ServerEditResponse;
   Delete(ServerDeleteRequest) : ServerDeleteResponse;
   Power(ServerPowerRequest) : ServerPowerResponse;
 Axiom
   …
End
```

The profile for the List operation is given as follows.

```
<rdf:RDF>
<owl:Ontology rdf:about="">
<owl:imports rdf:resource=
    "http://www.daml.org/services/owl-s/1.0/Profile.owl"/>
<owl:imports rdf:resource="#GServerOntology.owl"/>
</owl:Ontology>
<profile:serviceName> GServer.List</profile:serviceName>
<profile:hasInput rdf:resource="GServerOntology.owl#GServer"/>
<profile:hasInput rdf:resource=
 "GServerOntology.owl#ServerListRequest"/>
<profile:hasOutput rdf:resource=
 "GServerOntology.owl#GServer"/>
<profile:hasOutput rdf:resource=
 "GServerOntology.owl#ServerListResponse"/>
</rdf:RDF>
```

4 TRS2OTOOL

A prototype tool called TrS2O (*Tr*anslator from *S*pecification *toO*ntology) has been designed and implemented in Java. It translates formal specifications in SOFIA to ontological descriptions of services in OWL. Figure 2 shows the overall structure of the TrS2O Tool.

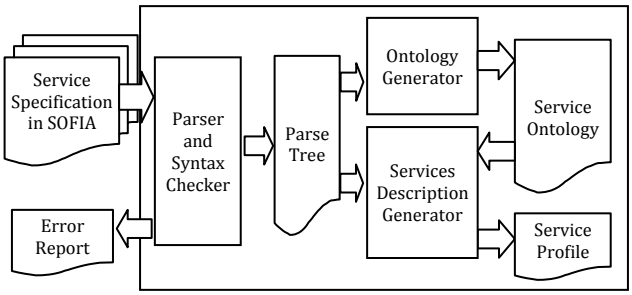

Figure 2.The Overall Structure of The TrS2O Tool

The tool TrS2O contains three main components.

(1) *Specification Parser and Syntax Checker*, which parses algebraic specifications written in SOFIA and generates a parse tree. It checks whether a specification is syntactically well-formed and whether the equations in the axioms are type correct.

(2) *Ontology Generator*, which takes the parse tree of the algebraic specification as input, and generates an ontology represented in the OWL language according to the rules defined in section 3.

(3) *Services Description Generator*, which takes as inputs the ontology and the parse tree of the algebraic specification and generates the descriptions of services in OWL-S profiles.

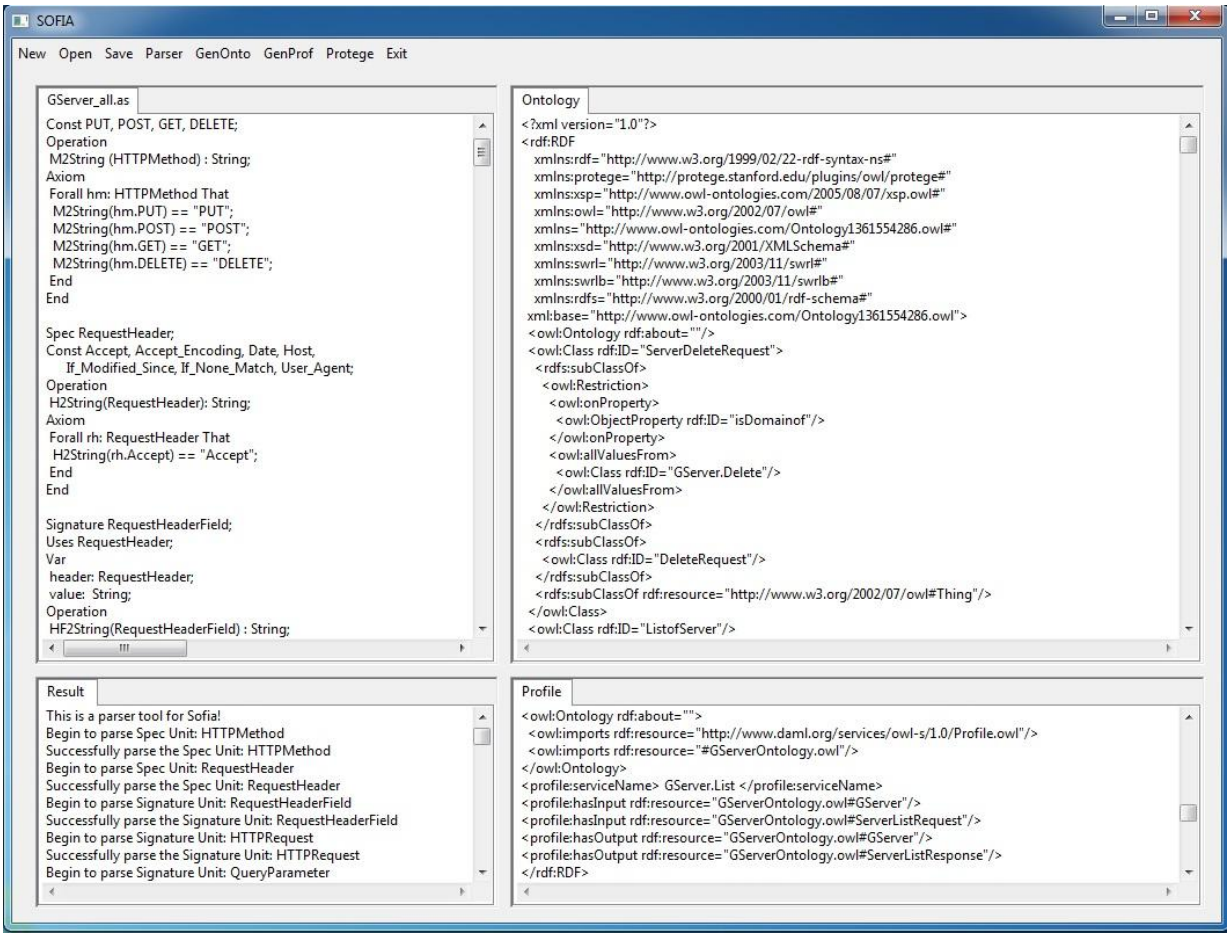

Figure 3. The Interface of TrS2O Tool

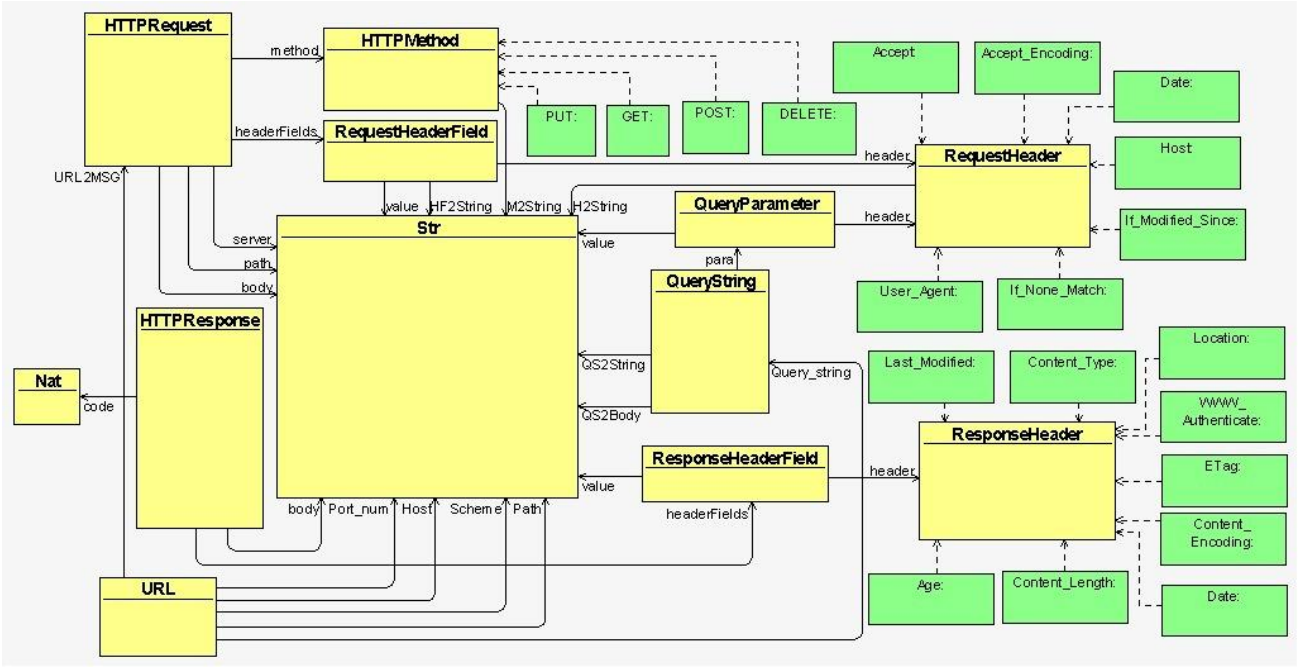

Figure 4. Visualization of Ontology Generated byTrS2O

Figure 3 shows the user interface of TrS2O. The upper-left pane displays the specification in SOFIA, while the lower-left displays the parsing report for it. The panes on the right are generated from the specification. The upper-right andshows the ontology and the lower-right showsprofile of services.

It is worth noting that the ontology generated by TrS2O can be processed by any OWL tool.Figure 4 illustrates the visualization of the ontology for the GoGrid specification; the tool used was Protege. Reasoning and searching on domain knowledge can also be performed.

5 Case Study

In this section, we report a case study with a real industrial RESTful web services GoGrid.

5.1 GoGrid API

GoGrid[1] is an infrastructure-as-a-service (IaaS) provider. It provides an easy-to-use API for developers, system administrators and end-users to access its functions. Its services can be accessed through a RESTful web service interface in a number of different programming and scripting languages. RESTful web services, unlike SOAP/WSDL, are based on the HTTP protocol, so each GoGrid API call is an individual HTTP query.

The latest version of the GoGrid API has 11 different types of objects and 5 types of common operators. Not all operators can be applied to all types of objects, however. There are three types of objects that are only used as parameters of the operators, so no operators are applicable on them, and there are some objects that have special operators. *Table 1*gives the applicable operators for each type of object.

It is worth noting that some operators in GoGrid have different meanings for different types of objects. In order to achieve well-structuredness, in our specification of GoGrid, the definitions were grouped by object rather than by operator. For the sake of space, we give here just the applicable operators for the load balancer object and its systematic specification, because it is one of the most important objects with the most operators.

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					

[1] http://www.gogrid.com/

5.2 Specification of GoGrid in SOFIA

For each type of objects in the GoGrid system, we write several specification units to define various aspects of the object and its operators, including

(1) Valid requests, for which we define their structures and constraints on how their components may be combined;
(2) Responses,with structures and constraints as above;
(3) Objects of certain types, with signatures and semantics, including signatures and axioms that characterize the relationships between the valid requests and the responses.

Other specification units define features and concepts common to many types of objects. Examples include the four query parameters common to all GoGrid API calls. Some properties are common to all objects too.

The specification of the GoGrid API is based on a framework for specifying RESTful web services (Liu, Zhu, & Bayley, 2013b). The framework consists of a collection of specification units that define the general structure of HTTP requests and responses so that a specific RESTful web services can be specified as extensions to these units. In particular, the following sorts in the framework are used in the GoGrid specification: *URL*, *HTTPMethod*, *RequestHeader*, *RequestHeaderField*, *HTTPRequest*, *QueryParameter*, *QueryString*, *ResponseHeader*, *ResponseHeaderField* and *HTTPResponse*. Details are omitted for the sake of space.

5.2.1 Objects and Collections

Here we give the specifications of the load balance object and its collection, *ListofLB*. The latter has an operation itemsto get an individual load balancer object, an operation insert to add on object to the list, and an attribute length to give the number of load balancer objects in the list.The specifications of*Option*, *IPPP* (which stands for IP Port Pair), and *ListofIPPP*(its collection) are omitted here.

```
Spec LoadBalancer;
  uses Option, IPPP, ListofIPPP;
  Var
    id: Long;
    name, description: String;
    virtualip: IPPP;
    realiplist: ListofIPPP;
    type, persistence, os, state, datacenter: Option;
  Axiom
    For all lb: LoadBalancer that
      lb.id <> Null;
    End
End
Spec ListofLB;
  uses LoadBalancer;
  Var
    length: Int;
  Operation
    items(Int) : LoadBalancer;
    insert(LoadBalancer);
End
```

Note that, when an object is structural (i.e. it consists of a number of elements), each element of the object can be specified using an *attribute*in the SOFIA language. Traditionallyin algebraic specifications, an attribute is an observer, i.e. an operation from the sort being defined to another sort. It is similar to the getters in object-oriented programs for getting the value of attributes. Here, SOFIA provides attribute as a language facility to specify the object's structure directly.

5.2.2 Requests

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

```
Spec CommonParameter;
 Var
  api_key, sig, v, format: String;
 Axiom
  Forall cp: CommonParameter That
      cp.api_key <> Null;
      cp.sig <> Null;
      cp.v <> Null;
  End
End
```

Here api_key is a key generated by GoGrid for security when accessing 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. NULL is a value that represents no information.

The signature can be generated by an MD5 hash from three parts:

- the api_key, obtained before API calls can be made,
- the user's sharedsecret, a string of characters set by the user and known only by the GoGrid server, and
- a Unixtimestamp, the number of seconds since the Unix Epoch of when the request was made.

Together, the api_key and sharedsecret act as an authentication mechanism. Their uses in authentication depend on system context such as time, becausesig is time-dependent.Therefore, the axioms for specifying the authentication mechanism are given in the specification of the whole system. Here, we can only say that both are required.

In addition to the parameters common to all service requests, each specific type of service request may also contain various specific parameters. So,for each type of request, we first specify the common structure as one sort: *ListRequest*, *GetRequest*, and so on. These are then extended for the different types of objects, giving *ServerListRequest*, *LBListRequest*, and so on. Here we only have space for the *get* operation on *load balancer*, but it is the most common operation, and complex enough to be representative. It is implemented using the HTTP request method GET and is the only way to determine the internal state of a service.

```
Spec GetRequest;
 extends HTTPRequest;
 uses CommonParameter, ListofString;
 Var
  para: CommonParameter;
  id, name : ListofString;
 Axiom
  For all gr: GetRequest that
   gr.id = Null, if gr.name <> Null;
   gr.name = Null, if gr.id <> Null;
  End
End
```

As you can see, the sort *GetRequest*adds to *HTTPRequest*some extra attributes:para,the common query parameters defined before, and both idandname; these are used to select the object; only one is required and it is an error to use both. Now *GetRequest* can be extended to load balancers as *LBGetRequest* as follows.

```
Spec LBGetRequest;
 extends GetRequest;
 uses ListofString;
 Var
  loadbalancer: ListofString;
 Axiom
  For all lbgr: LBGetRequest that
   lbgr.id = Null, if lbgr.loadbalancer <> Null;
   lbgr.name = Null, if lbgr.loadbalancer <> Null;
 End
End
```

5.2.3 Responses

The GoGrid API responses can be in any of 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, algebraic specification is abstract enough to specify all three at once.

The response to a get call starts with a summary, defined below,containing the total number of objects available, start index, number of objects returned in a page, and number of pages.

```
Signature ResponseSummary;
 Var
  total, start, returned, numpages: Int;
End
```

As well as this summary, the response contains *status*, *request method*, *status code* and *a list of returned objects*.

```
Spec GetResponse;
 extends HTTPResponse;
 uses ResponseSummary;
 Var
  summary: ResponseSummary;
  status, request_method: String;
  statusCode: Int;
 Axiom
  For all gr: GetResponse that
   gr.summary.total >=0;
   gr.summary.start = 0;
   gr.summary.returned = gr. summary.total;
  End
End
```

For load balancers, this is extended with an attribute for the *list of returned load balancer objects*.

```
Spec LBGetResponse;
  extends GetResponse;
  uses ListofLB;
  Var
  objects: ListofLB;
End
```

5.2.4 Semantics of the operations

For each type of request, we define an operator that takes a request as the 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 and also the shared secret chosen by each user and timestamp for checking the authentication of access. Thus, we have the following signature for the sort *GLB*, which represents the load balancer web services of the GoGrid cloud computing system.

```
Spec GLB;
  uses
    LBListRequest, LBListResponse,
    LBGetRequest, LBGetResponse,
    LBAddRequest, LBAddResponse,
    LBEditRequest, LBEditResponse,
    LBDeleteRequest, LBDeleteResponse,
  Var
    clockTime, timeStamp: Int;
    sharedSecret: String;
  Operation
    List(LBListRequest): LBListResponse;
    Get(LBGetRequest): LBGetResponse;
    Add(LBAddRequest): LBAddResponse;
    Edit(LBEditRequest): LBEditResponse;
    Delete(LBDeleteRequest): LBDeleteResponse;
  Axiom
  ...
End
```

Axioms are used to characterize the semantics of each operator, but here, as illustration, we give just the get operator.

First of all, GoGrid authenticates 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 it 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.

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

An important feature of the Get operator is that it is an observer. So, applying it will not change the state of the context sort GLB. This property can be expressed by axioms in the following form.

```
Axiom <Get-XOp>:
   For all G: GLB, X: LBGetRequest, X1: LBXOpRequest that
[G.Get(X)].XOp(X1) = G.XOp(X1);
End
```

where *XOp* is any of the operators *List*, *Get*, *Add*, *Edit* or *Delete*.

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

```
For all G: GLB, 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, defined in a definition unit, that insert a list of load balancer objects into another list, and search for a string in a list of strings.

The final axiom listed here 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 accordingly.

```
For all G: GLB, 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
```

5.2.5 Summary of GoGrid Specification

The complete GoGrid API has been specified in SOFIA. The numbers of different types of specification units in the specification are shown in Table 2.

Table 2. 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

5.3 GoGrid Ontology

Using the TrS2O tool, we have extracted an ontology from the GoGrid specification. Take specification *GLB*for example. Table 3 gives the numbers of classes, properties and individuals in the GoGrid Ontology in OWL.

Table 3. Basic Data of GoGrid Ontology

Ontology Concept	Specification Concept	No.
Class	Sort	39
	General Operator	9
Property	extends	9
	uses	36
	Domain	16
	Codomain	12
	Variable Operator	97
Individual	Constant Operator	20

For example, here is a fragment of the ontology profile for the *GetRequest* sort. It has, in order,one class for the sort, one property for the *extends* relation, two properties for the *uses* relations and three properties for attributes, each defined as *ObjectProperty*.

```
<owl:Class rdf:ID="GetRequest">
<rdfs:subClassOf rdf:resource=
    "http://www.w3.org/2002/07/owl#Thing"/>
<rdfs:subClassOf>
<owl:Class rdf:ID="HTTPRequest"/>
</rdfs:subClassOf>
<rdfs:subClassOf>
<owl:Restriction>
<owl:onProperty rdf:resource="#uses"/>
<owl:allValuesFrom>
<owl:Class rdf:ID="CommonParameter"/>
</owl:allValuesFrom>
</owl:Restriction>
</rdfs:subClassOf>
<rdfs:subClassOf>
<owl:Restriction>
<owl:onProperty rdf:resource="#uses"/>
<owl:allValuesFrom>
<owl:Class rdf:ID="ListofString"/>
</owl:allValuesFrom>
</owl:Restriction>
</rdfs:subClassOf>
</owl:Class>
<owl:ObjectProperty rdf:about="#GetRequest.para">
<rdfs:domain rdf:resource="#GetRequest"/>
<rdfs:range rdf:resource="#CommonParameter"/>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:about="#HTTPRequest.id">
<rdfs:domain rdf:resource="#GetRequest"/>
<rdfs:range rdf:resource="#ListofString"/>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:about="#HTTPRequest.name">
<rdfs:domain rdf:resource="#GetRequest"/>
<rdfs:range rdf:resource="#ListofString"/>
</owl:ObjectProperty>
```

Similarly, here is a fragment of the ontology profile for the *GLB* sort. It has, in order, one class for the sort, ten properties for the uses relations, five properties for *isDomainOf* and fiveproperties for *isCodomainOf*,.five classes for general operators, and three properties for the attributes, defined as *ObjectProperty*.

```
<owl:Class rdf:ID="GLB">
<rdfs:subClassOf rdf:resource=
    "http://www.w3.org/2002/07/owl#Thing"/>
<rdfs:subClassOf>
<owl:Restriction>
<owl:onProperty rdf:resource="#uses"/>
<owl:allValuesFrom>
<owl:Class rdf:ID="LBListRequest"/>
</owl:allValuesFrom>
</owl:Restriction>
</rdfs:subClassOf>
<rdfs:subClassOf>
<owl:Restriction>
<owl:onProperty rdf:resource="#uses"/>
<owl:allValuesFrom>
<owl:Class rdf:ID="LBListResponse"/>
</owl:allValuesFrom>
</owl:Restriction>
</rdfs:subClassOf>
… //the other 8 properties for the uses relation
<rdfs:subClassOf>
<owl:Restriction>
<owl:onProperty rdf:resource="#isDomainOf"/>
<owl:allValuesFrom rdf:resource="#GLB.List "/>
</owl:Restriction>
</rdfs:subClassOf>
…//the other 4 properties for isDomainOf
<rdfs:subClassOf>
<owl:Restriction>
<owl:onProperty rdf:resource="#isCodomainOf"/>
<owl:allValuesFrom rdf:resource="#GLB.List "/>
</owl:Restriction>
```

```
</rdfs:subClassOf>
  …//the other 4 properties for isCodomainOf
  </owl:Class>
  <owl:Class rdf:ID="GLB.List">
  <rdfs:subClassOf rdf:resource=
        "http://www.w3.org/2002/07/owl#Thing"/>
  </owl:Class>
  …//the other 4 classes for general operators
  <owl:ObjectProperty rdf:about="#GLB.clockTime">
  <rdfs:domain rdf:resource="#GLB"/>
  <rdfs:range rdf:resource="#Integer"/>
  </owl:ObjectProperty>
 …//the other two properties for variable operators
```

5.4 GoGrid Server Profile.

With the TrS2O tool, we have also generateda service profile. Here it is for the example of *GLB*.

```
<rdf:RDF>
<owl:Ontology rdf:about="">
<owl:imports rdf:resource=
       "http://www.daml.org/services/owl-s/1.0/Profile.owl"/>
<owl:imports rdf:resource="#GLBOntology.owl"/>
</owl:Ontology>
<profile:serviceName>GLB.List</profile:serviceName>
<profile:hasInput rdf:resource= "GLBOntology.owl#GLB"/>
<profile:hasInput rdf:resource=
      "GLBOntology.owl#GLBListRequest"/>
<profile:hasOutput rdf:resource="GLBOntology.owl#GLB"/>
<profile:hasOutput rdf:resource=
      "GLBOntology.owl#GLBListResponse"/>
<profile:serviceName>GLB.Get</profile:serviceName>
<profile:hasInput rdf:resource= "GLBOntology.owl#GLB"/>
<profile:hasInput rdf:resource=
      "GLBOntology.owl#GLBGetRequest"/>
<profile:hasOutput rdf:resource="GLBOntology.owl#GLB"/>
<profile:hasOutput rdf:resource=
      "GLBOntology.owl#GLBGetResponse"/>
<profile:serviceName>GLB.Add</profile:serviceName>
<profile:hasInput rdf:resource= "GLBOntology.owl#GLB"/>
<profile:hasInput rdf:resource=
      "GLBOntology.owl#GLBAddRequest"/>
<profile:hasOutput rdf:resource="GLBOntology.owl#GLB"/>
<profile:hasOutput rdf:resource=
      "GLBOntology.owl#GLBAddResponse"/>
<profile:serviceName>GLB.Edit</profile:serviceName>
<profile:hasInput rdf:resource= "GLBOntology.owl#GLB"/>
<profile:hasInput rdf:resource=
      "GLBOntology.owl#GLBEditRequest"/>
<profile:hasOutput rdf:resource="GLBOntology.owl#GLB"/>
<profile:hasOutput rdf:resource=
      "GLBOntology.owl#GLBEditResponse"/>
<profile:serviceName>GLB.Delete</profile:serviceName>
<profile:hasInput rdf:resource= "GLBOntology.owl#GLB"/>
<profile:hasInput rdf:resource=
     "GLBOntology.owl#GLBDeleteRequest"/>
<profile:hasOutput rdf:resource="GLBOntology.owl#GLB"/>
<profile:hasOutput rdf:resource=
    "GLBOntology.owl#GLBDeleteResponse"/>
</rdf:RDF>
```

6 Conclusions and Future Work

In this paper, we propose an approach that bridges the gap between formal specification and ontological description of service semantics. We do this by transforming formal specifications into domain ontology and ontological descriptions of services. The former is capable of providing verifiable and testable specifications of service semantics, whilst the latter has the advantage of being practically usable and easy for software developers to understand. The prototype tool is built for the specification language SOFIA, and the output is in OWL. A case study with the tool demonstrates the feasibility of the proposed approach.

We are pursuing a formal approach for specifying and testing service-oriented systems. Currently, we are developing a tool that uses specifications in SOFIA as input to perform automated testing and verification of web services. Another possible avenue for future work is to check the consistency of specification using both ontological reasoning and equational logic inferences.

Acknowledgment

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 Jiangsu Qinglan Project.

References

Bonchi, F., & Montanari, U. (2008). A coalgebraic theory of reactive systems. *Electronic Notes in Theoretical Computer Science, 209*, 201-215.

Bruijn, J., et al. (2006). The web service modelling language WSML: An overview, *Proceedings of the 3rd European Semantic Web Conference* (pp. 590-604): Springer-Verlag.

Bruijn, J. d., et al. (2005). *Web service modeling ontology (WSMO),* (W3C member submission): W3C.

Chen, H. Y., Tse, T. H., Chan, F. T., & Chen, T. Y. (1998). In black and white: An integrated approach to class-level testing of object-oriented programs. *ACM Transactions on Software Engineering and Methodology, 7*(3), 250-295.

Chen, H. Y., Tse, T. H., & Chen, T. Y. (2001). TACCLE: A methodology for object-oriented software testing at the class and cluster levels. *ACM Transactions on Software Engineering and Methodology, 10*(4), 56-109.

Cirstea, C. (1997). Coalgebra semantics for hidden algebra: Parameterised objects and inheritance, *Proceedings of the 12th International Workshop on Recent Trends in Algebraic Development Techniques* (pp. 174-189).

Cirstea, C. (2002). A coalgebraic equational approach to specifying observational structures. *Theoretical Computer Science, 280*(1-2), 35-68.

Doell, B., & Dosch, W. (2005). Transforming functional signatures of algebraic specifications into object-oriented class signatures, *Proceedings of the 12th Asia-Pacific Software Engineering Conference* (pp. 323-332): IEEE CS Press.

Ehrich, H.-D. (1982). On the theory of specification, implementation, and parametrization of abstract data types. *Journal of ACM, 29*(1), 206-227.

Gaudel, M.-C.,& Le Gall, P. (2007). Testing data types implementations from algebraic specifications. In *Formal Methods and Testing*, R. Hierons, J. Bowen, and M. Harman,eds, Lecture Notes in Computer Science, Vol. 4949, (209-239) Springer-Verlag.

Goguen, J. A., & Malcolm, G. (2000). A hidden agenda. *Theoretical Computer Science, 245*(1), 55-101.

Goguen, J. A., Thatcher, J. W., Wagner, E. G., & Wright, J. B. (1977). Initial algebra semantics and continuous algebras. *Journal of ACM, 24*(1), 68 - 95

Hadley, M. J. (2006). *Web application description language (WADL)* (SMLI TR-2006-153). CA, USA: Sun Microsystems Inc.,.

Kong, L., Zhu, H., & Zhou, B. (2007). Automated testing components based on algebraic specifications, *Proceedings of the 31th IEEE International Conference on Computer Software and Applications (COMPSAC 2007)* (pp. 717-722).

Kopecky, J., Gomadam, K., & Vitvar, T. (2008). hRESTS: An HTML microformat for describing RESTful web services, *Proceedings of theIEEE/WIC/ACM 2008 International Conference on Web Intelligence and Intelligent Agent Technology (WI-IAT'08)* (pp. 619-625). Sydney, Australia.

Lathem, J., Gomadam, K., & Sheth, A. P. (2007). SA-REST and (S)mashups: Adding semantics to RESTful services, *Proceedings of ICSC* (pp. 469-476).

Liu, D., Zhu, H., & Bayley, I. (2012). Applying algebraic specification to cloud computing -- a case study of Infrastructure-as-a-Service GoGrid, *Proceedings of The Seventh International Conference on Software Engineering Advances* (pp. 407-414).

Liu, D., Zhu, H., & Bayley, I. (2013a). A case study on algebraic specification of cloud computing, *Proceedings of the 21st Enuromicro International Conference on Parallel, Distributed and Network-Based Processing* (pp.269-273). Queen's University, Belfast, Northern Ireland.

Liu, D., Zhu, H., & Bayley, I. (2013b). From algebraic specification to ontological description of service semantics, *Proceedings of the 20th International Conference on Web Services (ICWS 2013)*. Santa Clara, CA.

Mallraith, S. A., Son, T. C., & Zeng, H. (2001). Semantic web services. *IEEE Intelligent Systems*(March/April), 46-53.

Martin, D., al., e. (2004). *Semantic Markup for Web Services* (W3C member submission): W3C.

Papazoglou, M. P. (2012). *Web Services and SOA: Principles and Technology* (2nd ed.): Pearson.

Richardson, L., & Ruby, S. (2007). *RESTful Web Services*: O'Reily.

Rutten, J. M. (2000). Universal coalgebra: a theory of systems. *Theoretical Computer Science, 249*(1), 3-80.

Singh, M. P., & Huhns, M. N. (2005). *Service-Oriented Computing: Semantics, Processes, Agents*: John Wiley & Sons.

Uschold, M., & Gruninger, M. (1996). Ontologies: Principles, methods, and applications. *Knowledge Engineering Review, 11*(2), 93-155.

Yu, B., Kong, L., Zhang, Y., & Zhu, H. (2008). Testing java components based on algebraic specifications, *Proceedings of the First International Conference on Software Testing, Verification, and Validation (ICST 2008)* (pp.190-199). Lillehammer, Norway: IEEE CS Press.

Zhu, H. (2003). A note on test oracles and semantics of algebraic specifications, *Proceedings of the 3rd International Conference on Quality Software (QSIC 2003)* (pp. 91-98). Dallas, TX.

Zhu, H., Liu, D., & Bayley, I. (2013). *Reference manual of the SOFIA algebraic specification language* (TR-CCT-AFM-01-2013). Oxford, UK: Department of Computing and Communication Technologies, Oxford Brookes University.

Liu, D., Zhu, H. & Bayley, I. (2014). SOFIA: An Algebraic Specification Language for Developing Services, In Proc. of The 8th IEEE International Symposium on Service-Oriented Systems Engineering (SOSE 2014). (pp.70-75) Oxford, UK.

Zhu, H., & Yu, B. (2010). Algebraic specification of web services, *Proceedings of the 10th International Conference on Quality Software (QSIC 2010)* (pp. 457-464): IEEE CS Press.

Authors

Dongmei Liu received the BS degree in Computer Applied Technology, the MS degree in Computer Architecture, and the PhD degree in Computer Software and Theory from Wuhan University, China, in 1999, 2001 and 2004, respectively. Currently, she is an associate professor at Nanjing University of Science and Technology, China. Her research interests include software testing, software reliability modeling, formal specification methods and intelligent computation.

Hong Zhu is a full professor of computer science at the Oxford Brookes University, UK, where he chairs the Applied Formal Methods research group. He received a BSc, MSc and PhD degree in Computer Science from Nanjing University, China, in 1982, 1984 and

1987, respectively. He was with Nanjing University from August 1987 to November 1998 and joined Oxford Brookes University in Nov. 1998. His main research interests are in software engineering, including software testing, modeling, design and development methodologies. He has published two books and more than 170 research papers in journals and international conferences and chapters in peer reviewed edited books. He is a senior member of IEEE and a member of ACM and BCS.

Ian Bayley was born in Liverpool, studied for an MEng Computing (Mathematical Foundations and Formal Methods) at Imperial College London and earned a DPhil Computation at Balliol College Oxford with research into the semantics of functional programming languages. Since 2005, he has been a lecturer at Oxford Brookes University. His research interests include software engineering, formal methods, and programming paradigms.

SERVICES COMPUTING

hipore.com

A Technical Publication of the Services Society

ISSN: 2330-4464 (Print) ISSN: 2330-4472 (Online)