2010 10th International Conference on Quality Software

Algebraic Specification of Web Services

Hong Zhu
Department of Computing and Electronics,
Oxford Brookes University,
Oxford OX33 1HX, UK, Email: hzhu@brookes.ac.uk

Bo Yu
Department of Computer Science,
National University of Defense and Technology,
Changsha 410074, China, Email: hnaxdsjk@163.com

***Abstract*—This paper presents an algebraic specification language for the formal specification of the semantics of web services. A set of rules for transforming WSDL into algebraic structures is proposed. Its practical usability is also demonstrated by an example.**

***Keywords*-Web Services, Algebraic specification, WSDL, Formal methods, Specification language.**

I. Introduction

Formal specification of software systems has been a significant challenge to both communities of formal methods and software engineering for at least the last three decades [1]. More recently, the advent of service oriented computing raises the stakes: can we specify services with high flexibility to support the dynamic discovery and composition of services? In particular, we are concerned with specifications of services in a modular and composable manner without releasing internal design and implementation details because services are often owned and operated by different vendors.

Algebraic specification was first proposed in the 1970s as an implementation independent specification technique for abstract data types [2], [3]. In the past three decades, it has developed into a systematic formal methodology that can be applied to various types of software systems. In particular, by applying the theories of behavioral algebra [4] and co-algebra [5], [6], [7], [8] developed recently, concurrent systems, state-based systems and software components can be specified in a modular, composable and implementation independent manner at a very high level of abstraction. Thus, it is very suitable for the specification of web services. Moreover, as a formal method, techniques and tools have been developed to support many formal software development activities, such as formal refinement from specifications to implementations, proving correctness of implementations against formal specifications, and proving properties of software systems based on formal specifications [9], etc.

One of its most attractive features of algebraic specification is that it supports automatic testing of software systems. This is particularly important for testing web services because test on-the-fly must be fully automated. There are software tools for testing implementations of abstract data types [10], [11], [12], classes [13], [14], [15], [16] and software components [17], [18] base on algebraic specifications. In our previous work, we have developed the CASOCC algebraic specification language that specifies abstract data types, classes and software components in a unified formalism [17]. We have also developed an automated software testing tool called CASCAT, which tests Java EJB components automatically based on specifications in written CASOCC. Our experiments with CASOCC/CASCAT have shown that automated testing of Java EJB components based on algebraic specifications can detect about 85% faults in mutation analysis [18]. Moerover, specification of software components can be learnable and efficient [19]. However, the algebraic approach has not been applied to the specification of web services as far as we know. This is mostly due to the restrictions imposed by algebraic specification languages.

In this paper, we will extend CASOCC language to enable the specification of services with automated testing of services as our ultimate goal. The extended specification language is called CASOCC-WS. It facilitates the specification of web services in a unified syntax and semantics in which software components, object-oriented systems and traditional data types are specified. We will demonstrate by an example that web services can be specified by CASOCC-WS with the algebraic structures automatically derived from WSDL.

The remainder of this paper is organized as follows. Section II gives a brief introduction to CASOCC-WS. Section III presents a set of rules for transforming WSDL descriptions of Web Services into algebraic structures, and some heuristic rules for writing axioms to define the semantics of web services. Section IV gives an example of algebraic specification of a web service. Section V concludes the paper with a brief discussion of related works and future work.

II. The CASOCC-WS Specification Language

CASOCC-WS specifications are modular. A specification consists of a number of modular units, each for one software entity type in the software system. A type of software entity can be an abstract data type, a class, a component, a web service, a message type defined by an XML schema and passed between services, and so on. We will not distinguish them in a specification. Instead, we will abstract out such implementation details in our specifications of their functions and behavioral properties.

As shown in the following BNF syntax rules, each specification unit contains three parts: (a) a sort name of the entity and its observability, (b) a signature and (c) a set of axioms. They are presented in the subsections below.

```
<Specification> ::= {<Spec unit> }
<Spec unit> ::=
  Spec <Sort Name> is <Observability>;
      <Signature>; <Axioms>  End
```

A. Signature

Signature specifies the syntax aspect of a software entity. Each specification unit has a unique identifier, which corresponds to a *sort* in the traditional terminology of algebraic specification.

Let S be a finite non-empty set of sorts. We will also define a binary relation $\prec$, called *importation* relation, on its elements. Informally, the relation $\prec$ represents the dependency between sorts. If sorts $s_1 \prec s_2$, it means that the computational entity s_2 is constructed based on the computational entity s_1. We can use the operators and axioms defined in s_1 to construct s_2.

In the CASOCC-WS language, the relation $\prec$ is defined by the *Import* clause in the specification of a sort. It lists the sorts that the specified sort depends on. For example, suppose that the following Import clause is written in the specification of Stack.

```
Import BOOL, NAT;
```

It means that `STACK` depends on `BOOL` and `NAT`. Thus, `BOOL` $\prec$ `STACK` and `NAT` $\prec$ `STACK`.

A common feature of software entities, like abstract data type, class, component and services, is that each entity defines a set of operations. The syntax aspect of an operator is specified by giving its identifier, its domain and co-domain types. It is written in the following form.

$$Op : s_1, s_2, \cdots, s_n \rightarrow s'_1, s'_2, \cdots, s'_k$$

where $n, k \geq 0$, $(s_1, s_2, \cdots, s_n)$ are the domain sorts, and $(s'_1, s'_2, \cdots, s'_k)$ are the codomain sorts.

For example, the `Push` operator of the `STACK` abstract data type has two input parameters: the stack that stores data and a nature number to be push into the stack. The result of the operation is a new state of the stack. The signature of the operator can be defined as follows in CASOCC-WS.

```
Push: STACK, NAT -> STACK
```

Note that, in a traditional algebraic specification language, the co-domain of an operator must be a singleton. Such a signature is called *algebraic*.

Recently, specifications based on co-algebras allows the co-domains of operators to be non-singleton. It can be any sequence of sorts. However, the domain must be a singleton. Such a signature is called *co-algebraic* [20]. For example, the following signature for infinite streams of natural numbers is co-algebraic.

```
Spec STREAM Is Unobservable
  Import NAT;
  Operators:
    Transformer:
      NEXT: STREAM -> NAT,STREAM;
  Axioms:
 ...
End
```

In the above specification, there is only one operator, i.e. `NEXT`, applicable to an infinite stream of natural numbers. Each time the operator is applied to a stream will give a natural number and change the state of STREAM.

However, we will show in the example in section IV, both algebraic and co-algebraic signatures are too restrictive for the specification of web services. Thus, CASOCC-WS language allows both the domain and codomain of an operator to be non-singleton sequence of sorts.

Moreover, when the main sort occurs in both domain and co-domain of an operator, we consider the main sort as the *context sort* of the operator and specify the operator in the following format, where the occurrences of the context sort in the domain and co-domain are removed.

$$Op : [s_c]s_1, \cdots, s_n - > s'_1, \cdots, s'_k,$$

where s_c is the context sort. When the main sort is the only sort in the domain (or co-domain) of an operator, we write `VOID` as the type of domain (or co-domain) in the operator's type specification using context. For example, the signature of the `BOOL` operators `AND`, `OR`, `EQ` and `NOT` can be defined as follows.

```
AND: [BOOL] BOOL -> VOID;
OR:  [BOOL] BOOL -> VOID;
EQ:  [BOOL] BOOL -> VOID;
NOT: [BOOL] VOID -> VOID;
```

The semantics of an operator with a context sort is equivalent to the operator with the context sort added to the lists of sorts of the domain and co-domain. The only difference is that terms formed by an operator with a context sort can be written in the object-oriented style, i.e. in the form of $C.f(x_1, x_2, \cdots, x_n)$, where C is a term of the context sort of operator f. Context sort is only a syntax sugar to improve the readability of specifications. Therefore, in the sequel, we will only discuss operators without context sorts unless explicitly stated.

In general, the syntax in BNF of the signature of an operator is given below.

```
<Type> ::= <Sort Name> [ , <Type> ]
<Domain Type> ::= <Type> | VOID
<Co-domain Type> ::= <Type> | VOID
<Context Sort> ::= <Sort Name>
<Operation> ::=
  <Operator ID>: [ '[' <Context Sort> ']' ]
     <Domain Type> -> <Co-domain Type>;
<Operator ID>::= <Identifier>
```

We classify the operators defined for a sort s into the following kinds. This classification helps the uses of algebraic specifications in automated software testing [17], [18].

Let $\varphi^s : w \to w'$ be an operator in the specification of sort s, where $w = (s_1, s_2, \cdots, s_n)$ and $w' = (s'_1, s'_2, \cdots, s'_k)$.

The operator φ^s is called a *creator* of sort s, if for all $i = 1, \cdots, n$, $s_i \neq s$, and for some $j = 1, \cdots, k$, $s'_j = s$.

The operator φ^s is called a *transformer* of sort s, if there are $i \in \{1, \cdots, n\}$ and $j \in \{1, \cdots, k\}$ such that $s_i = s'_j = s$.

The operator φ^s is called an *observer* of sort s, if for all $j = 1, \cdots, k$, $s_j \neq s$, and for some $i = 1, \cdots, n$, $s'_j = s$.

For example, consider the signatures of the operators of the Boolean algebra. According to the above definition, TRUE and FALSE are creators. AND, OR, NOT and EQ are transformers.

A *signature* part of a specification unit for a sort s, denoted by Σ^s, consists of a finite family of non-empty disjoint sets $\Sigma_{w,w'}$ indexed by (w, w'), where w and w' $\in W_s = \{x \in S | x \prec s \vee x = s\}^*$. Each element φ of set $\Sigma_{w,w'}$ is an *operator symbol* of type $w \to w'$, where w is the *domain type* and w' the *co-domain type* of the operator.

The CASOCC-WS syntax of unit signature in BNF is given below.

```
<Signature>::= [<Import clause>;] <Operations>;
<Operations>::= Operations:[<Creator>;]
  [<Transformers>;][<Observers>]
<Creators>::= Creators:<Operation List>
<Transformers>::= Transformers: <Operation List>
<Observers>::= Observers:<Operation List>
<Operation List>::=<Operation>[;<Operation List>]
```

The signature of a software system is an ordered pair $(\mathbf{S}, \Sigma)$ that consists of $\mathbf{S}$ that is a set S of sorts ordered by the importation relation $\prec$, and a collection Σ of unit signatures Σ^s for sorts $s \in S$.

B. Axioms

Let $(\mathbf{S}, \Sigma)$ be a system signature and $\{V_s | s \in S\}$ be a collection of disjoint sets of variables, where elements of V_s are called variables of sort s. Let $s \in S$ be any given sort. The set of *s-terms*, which are terms that can occur in the specification of sort s, is inductively defined as follows. Let $s_1, s_2, \cdots, s_n, s' \preceq s$, where $s' \preceq s$ means $s' \prec s$ or $s' = s$.

1) Every s'-term τ of type w is a s-term of type w.
2) For all variables $v \in V_{s'}$, v is a s-term of type s'.
3) For all s-terms $\tau_1, \cdots, \tau_n$ of types $s_1, \cdots, s_n$, respectively, $\langle \tau_1, \cdots, \tau_n \rangle$ is a s-term of type $(s_1, \cdots, s_n)$.
4) For every operator $\varphi^{s'} : w \to w'$ and s-term τ of type w, $\varphi(\tau)$ is a s-term of type w'.
5) For every operator $\varphi^{s'} : [s']w \to w'$ and s-terms τ_C of type s' and τ of type w, $\tau_C.\varphi(\tau)$ is a s-term of type w', and $\tau_C.[\varphi(\tau)]$ is a s-term of type s'. □

In particular, a s-term is called a *ground* s-term, if it contains no variable, i.e. it is formed without using rule 2 in the above definition. For the sake of convenience, we will also write $\varphi^s(\tau_1, \cdots, \tau_n)$ for the s-term $\varphi(\langle \tau_1, \cdots, \tau_n \rangle)$.

For example, assume that p and q are variables of the BOOL sort, the following are BOOL-terms.

```
AND(p,q), OR(FALSE,q), AND(p,OR(q,p)).
```

Let x and y be variables of type NAT. The following are examples of NAT-terms. They are of BOOL type, but they are not BOOL-terms. They can only be used in the axioms of NAT, not in axioms of BOOL.

```
IS_ZERO(x), EQ(S(x),y), AND(IS_ZERO(x),EQ(x,y))
```

The following are BOOL-terms using the signature that contains context.

```
p.AND(q), FALSE.OR(q), p.OR(q).AND(p)
```

The BNF syntax rules for terms are given below.

```
<Term> ::= <Variable> | "<" <Term List> ">"
   | <Operator ID> [ "(" [ <Parameters> ] ")" ]
   | <Term> "." <Term>
<Parameters> ::= <Term List>
<Term List> ::= <Term> [ "," <Term List> ]
```

Let $(\mathbf{S}, \Sigma)$ be a given system signature and Σ^s be the unit signature for sort s. Let τ and τ' be s-terms of type w, $c_1, c_2, \cdots, c_n$ and $d_1, d_2, \cdots, d_n$ be s-terms such that for all $i = 1, 2, \cdots, n$, c_i and d_i are of the same type, a conditional equation of signature Σ^s is

$$\tau = \tau', \ if \ c_1 = d_1, c_2 = d_2, \cdots, c_n = d_n.$$

For example, the following is an equation for NAT.

```
S(x) = S(y), if EQ(x, y)=TRUE.
```

We consider BOOL as predefined unit signature and write $f(x_1, \cdots, x_n)$ to denote the condition $f(x_1, \cdots, x_n) =$ TRUE, if the co-domain type of operator f is BOOL. Thus, the above equation can be rewritten as follows.

```
S(x) = S(y), if EQ(x, y).
```

To further improve the readability of axioms, we also introduce local variable declaration, which defines variables to be used in an equation or a set of equations. The format of local variable declarations is as follows.

$$Let \ \ x_1 = \tau_1, \cdots, x_n = \tau_n \ \ in \ \ Equs \ \ end.$$

For example, the following is an axiom that contains a local variable declaration.

```
Let aID = B.OpenAccount(customer),
    B′  = B.[OpenAccount(customer)]
in  B′.Account(aID).CustomerInfo = customer end;
```

The BNF syntax rules for equations are given below.

```
<Equation>::=
  <Label> : <Term> = <Term> [, if <Conditions>]
  | Let <Var Definition> in <Equations> end
<Conditions> ::= <Condition>[(,|"or")<Conditions>]
<Condition>  ::= <Term> = <Term> | <BOOL Term>
  |<Num Term> <Rel Op> <Num Term> |"~"<Condition>
<Rel Op> ::= "<" | "<=" | ">" | ">=" | "<>"
```

where `<Num term>` are terms of type `NAT`, `INT` or `REAL`, which are predefined sorts. Each equation can also be associated with a unique label.

An axiom consists of a list of variable declarations and a list of equations. A variable declaration declares a list of variables and their types. For example, the following is an example of variable declaration together with an equation that forms an axiom for `NAT`.

```
For all x, y: NAT that S(x) = S(y), if x = y.
```

The BNF syntax rules for axioms are given below.

```
<Var Dec>::= For all <Var-Sort Pairs> that
<Var-Sort Pairs>::=
  <Var IDs>:<Sort Name> [, <Var-Sort Pairs>]
<Var IDs>::= <Var ID> [, <Var IDs>]
<Axiom>::= <Var Dec> <Equations>
<Equations>::= <Equation> [ ; <Equations>]
```

An *algebraic specification* in CASOCC-WS is a triple $(\mathbf{S}, \Sigma, \mathbf{E})$, where $(\mathbf{S}, \Sigma)$ is a system signature, $\mathbf{E} = \{E_s | s \in S\}$ is a collection of equation sets that E_s is a finite set of equations of signature Σ^s.

C. Semantics of Specifications

We now define the semantics of CASOCC-WS algebraic specifications. It is fairly standard, for example, in the definition of the semantics of first order logic [22].

Given a system signature $(\mathbf{S}, \Sigma)$, a $(\mathbf{S}, \Sigma)$*-algebra* $\mathcal{A}$ is a mathematical structure $(\mathbf{A}, \mathbf{F})$ consists of a collection $\mathbf{A} = \{A_s | s \in S\}$ of sets indexed by S, and a collection $\mathbf{F}$ of functions indexed by the set $\bigcup_{s \in S} \Sigma^s$ such that for each operator $\varphi^s : w \to w'$, the function $f_\varphi \in F$ has domain A_w and co-domain $A_{w'}$, where $w = (s_1, \cdots, s_n)$, $A_w = A_{s_1} \times \cdots \times A_{s_n}$, $w' = (s'_1, \cdots, s'_n)$, and $A_{w'} = A_{s'_1} \times \cdots \times A_{s'_n}$.

Let $\mathcal{A} = (\mathbf{A}, \mathbf{F}_A)$ and $\mathcal{B} = (\mathbf{B}, \mathbf{F}_B)$ be two $(\mathbf{S}, \Sigma)$-algebras. A *homomorphism* β from $\mathcal{A}$ to $\mathcal{B}$ is a mapping β from $\mathbf{A}$ to $\mathbf{B}$ such that for all operators $\varphi : w \to w'$ in the signature and all elements $a_1 \in A_{s_1}, \cdots, a_n \in A_{s_n}$, we have that

$$\beta(f_{A,\varphi}(a_1, \cdots, a_n)) = f_{B,\varphi}(\beta(a_1), \cdots, \beta(a_n)).$$

The evaluation of a term in an algebra depends on the values assigned to the variables that occur in the term. An assignment α of variables V_s, $s \in S$, in an algebra is a function from V_s to A_s. Given an assignment $\alpha : V \to \mathbf{A}$, the evaluation of a term τ, written $[\![\tau]\!]_\alpha$, is defined as follows.

1) $[\![v]\!]_\alpha = \alpha(v)$;
2) $[\![\langle \tau_1, \cdots, \tau_n \rangle]\!]_\alpha = \langle [\![\tau_1]\!]_\alpha, \cdots, [\![\tau_n]\!]_\alpha \rangle$;
3) $[\![\varphi(\tau)]\!]_\alpha = f_{A,\varphi}([\![\tau]\!]_\alpha)$;
4) $[\![\tau.\varphi(\langle \tau_1, \cdots, \tau_n \rangle)]\!]_\alpha = f_{A,\varphi}([\![\langle \tau, \tau_1, \cdots, \tau_n \rangle]\!]_\alpha)$

Let $\mathcal{S} = (\mathbf{S}, \Sigma, \mathbf{E})$ be an algebraic specification and e be an equation $\tau = \tau'$, $if\ c_1 = d_1, \cdots, c_n = d_n$. An $(\mathbf{S}, \Sigma)$-algebra $\mathcal{A} = (\mathbf{A}, \mathbf{F})$ satisfies e, write $\mathcal{A} \models e$, if for all assignments α, we have that $[\![\tau]\!]_\alpha = [\![\tau']\!]_\alpha$ whenever $[\![c_i]\!]_\alpha = [\![d_i]\!]_\alpha$ is true for all $i = 1, \cdots, n$. $\mathcal{A}$ satisfies specification $\mathcal{S}$, written $\mathcal{A} \models \mathcal{S}$, if for all equations e in $\mathbf{E}$, we have that $\mathcal{A} \models e$, and we say that $\mathcal{A}$ is an $\mathcal{S}$-algebra.

The semantics of an algebraic specification is the final algebra $\mathcal{A}$ that satisfies the specification. Formally, an $\mathcal{S}$-algebra is *initial*, if for all $\mathcal{S}$-algebras $\mathcal{B}$, there is a unique homomorphism from $\mathcal{A}$ to $\mathcal{B}$. The $\mathcal{S}$-algebra $\mathcal{A}$ is *final*, if for all $\mathcal{S}$-algebras $\mathcal{B}$, there is a unique homomorphism from $\mathcal{B}$ to $\mathcal{A}$.

The existence of final $\mathcal{S}$-algebra is omitted for the sake of space.

D. Observability

Informally, a software entity is observable if we can compare the equality of two values (or states) x and y of the entity by invocation of a binary predicate $EQ(x, y)$, i.e. operator with `BOOL` as the codomain, provided by the entity. In that case, we say that the software entity is observable by the predicate. For example, the `BOOL` and `NAT` data types are observable. However, many complex data types and software entities are not observable. For example, the equality of two streams of natural numbers cannot be determined in such a way. Thus, `STREAM` is not observable.

The observability by an operator EQ means that whenever $EQ(\tau, \tau')$ returns `TRUE`, the values of terms τ and τ' must be the same. Therefore, observability imposes an addition requirement for an algebra to satisfy the specification. This requirement is formally defined as follows.

Let $\mathcal{S} = (\mathbf{S}, \Sigma, \mathbf{E})$ be a given specification in CASOCC-WS and $\mathcal{A} = (\mathbf{A}, \mathbf{F})$ be a $\mathcal{S}$-algebra. Assume that sort $s \in S$ is specified as observable by operator EQ. We say that algebra $\mathcal{A}$ satisfies the condition of "*observable by EQ*", if for all ground s-terms τ and τ' of type s, we have that

$$\mathcal{A} \models (\tau = \tau') \Leftrightarrow \mathcal{A} \models (EQ(\tau, \tau') = TRUE).$$

For example, assume that `NAT` is observable by `EQ`. Then, the following two equations are equivalent.

```
S(x) = S(y), if EQ(x, y).
S(x) = S(y), if x = y.
```

The specification of observability in CASOCC-WS is in the format defined by the following BNF syntax rule.

```
<Observability> ::=
  observable by <Operation Id> | unobservable
```

III. Specifying Web Services in CASOCC-WS

In this section, we discuss how to specify web services in CASOCC-WS. We will first present a set of rules to automatically derive algebraic signatures from the descriptions of web services in WSDL. We will then give a set of heuristic rules for writing algebraic axioms in order to define the semantics of the services.

A. *Web Service Description Language WSDL*

WSDL stands for Web Service Description Language. It is an XML language for describing the programmatic interfaces to web services. Its current version, WSDL 2.0, is recommended by W3C, but its tool supports are still underdevelopment. In contrast, there are good tool supports to its previous version WSDL 1.1, which is still widely used although not endorsed by the W3C. Therefore, in this paper, we will use WSDL 1.1. The principle can be easily adapted to WSDL 2.0.

In WSDL, a web service description has the following structure.

```
<definitions>
  <types> <!--types definitions--> </types>
  <message> <!--message definition--> </message>
  <portType definition>
     <operation>
       <!--operation definition--></operation>
     <input> <!--input definition--> </input>
     <output> <!--output definition> </output>
  </portType>
  <binding> <!--binding definition--> </binding>
  <service> <!--service definition-->
    <port> <!--port definition> </port>
  </service>
</definition>
```

In a WSDL description of a web service, the type definitions define the data types and their representations in XML. They are often in the form of XML schema definition. The message definitions define the typed data input to or output from an operation of the service. There are three kinds of messages: IN, OUT and INOUT. The portType definition defines a collection of operations provided by the service together their input and output. Each operation is an atom of the functionality of the service. The binding definition associates a port type to a protocol and a data format. An example is binding to SOAP and further identifying its style to be RPC, encoding to be literal and transport to be HTTP. The port definition gives a network address and binding where operations reside. The algebraic structure of a web service can be derived from type definition, message definition and portType definition.

B. *Rules for Transforming WSDL to CASOCC-WS Signature*

The basic idea of the algebraic approach to the formal specification of web services is to regard a service as an algebraic structure with the operations provided by a service as the operators. However, the parameters of the operators cannot simply be string of characters. Instead, we also regard each data type and message type as an algebraic structure. The elements and attributes of an XML document are accessed through a set of setters and getters. The constraints on the instance documents are specified by the axioms through these setters and getters. Therefore, we have the following rules that transform WSDL descriptions into CASOCC-WS specifications.

- The service name corresponds to the main sort name for the web service.
- For each type definition, an algebraic specification of the type is generated with the type name as the sort name. And, this sort name is then added into the import list of the algebraic specification of the web service.
- For each message definition, an algebraic specification of the message type is generated with the message type name as the sort name. This sort name is also added to the import list of the specification of the web service.
- For each operation defined in portType part, an operation signature is generated and added into the transformer part. Its domain and co-domain are the input and output message type names, respectively. The web service name is the context sort.

The rules for generating algebraic specifications for a type U are as follows. Note that, for each simple type of XML schema definition language, we provide a predefined algebraic specification in the CASOCC-WS language. Therefore, we do not need to generate the specification for simple types, but just to use the corresponding sort name. If the type U is not a simple type, the following rules are applied.

- Generate U as the sort name of the algebraic specification of the type.
- For each attribute A in the type definition of U, where E is in the following form.
  `<xsd:attribute name="X" type="V"/>`
  - An algebraic specification of the type V is generated by applying this same set of rules with a sort name V.
  - The sort name V is added to the import list of the algebraic specification of U.
  - An operator $setX : U, V \to U$ is added into the transformer part of the specification of U.
  - An operator $getX : U \to V$ is added into the observer part of the specification of U.
  - The following equations are added to the axiom part of the specification of U:
    ```
    Forall v : V, u : U that
      u.setX(v).getX = v;
      u.setX(v).getY = u.getY
    ```
    where Y is a name of an element or attribute in the definition of type U and $Y \neq X$.
- For each element E is the type definition of U, the transformation rule is similar to that for attribute;
- For each constraint, an axiom is generated according to the meaning of the constraint. For example, for the constraint that the length of string is fixed to be 8. The following axiom will be generated.
  ```
  Forall v : V that v.length=8
  ```

For example, consider the following type definition.

```
<xsd:complexType name="PurchaseOrder">
  <xsd:attribute name="shipTo" type="USAddress"/>
```

```
  <xsd:attribute name="items"  type="Items"/>
  <xsd:attribute name="orderDate"
                 type="xsd:date"/>
</xsd:complexType>
```

We have the following algebraic specification.

```
Spec PurchaseOrder Unobservable
  Import USAddress, Items, date;
  Operators
    Transformer:
      set_shipTo: [PurchaseOrder] USAddress->VOID,
      set_items: [PurchaseOrder] Items-> VOID
      set_orderDate: [PurchaseOrder] date -> VOID;
    Observer:
      get_ShipTo: [PurchaseOrder] VOID->USAddress;
      get_items: [PurchaseOrder] VOID ->Items;
      get_orderDate: [PurchaseOrder] VOID -> date;
  Axioms:
    Forall ord: PurchaseOrder, addr: USAddress,
           itm: Items, dt: date that
      ord.set_shipTo(addr).get_shipTo = addr;
      ord.set_shipTo(addr).get_items
        = Ord.get_items;
      ord.set_shipTo(addr).get_orderDate
        = Ord.get_orderDate;
      ord.set_items(itm).get_Items = itm;
      ord.set_items(itm).get_shipTo
        = ord.get_shipTo
      ord.set_items(itm).get_orderDate
        = ord.get_orderDate
      ord.set_orederDate(dt).get_orderDate = dt;
      ord.set_orederDate(dt).get_shipTo
        = ord.get_shipTo;
      ord.set_orederDate(dt).get_Items
        = ord.get_Items.
End
```

The rules for generating algebraic specifications from message type definitions are similar. The details are omitted.

C. Guidelines for Writing Axioms

Now, we address the question how to write axioms. Because axioms heavily depend on the semantics of the web services, they cannot be derived from the descriptions in WSDL, which contains very limited information about semantics. The following gives a few heuristic rules for the manual writing of axioms.
(*R1-Setter*): For each setter $setX(v)$ that sets the value of attribute X to v, write two axioms as follows:

$$\forall s, v \cdot (s.setX(v).getX = v), if\ PreCond_{setX(v)}$$

$$\forall s, v. \cdot (s.setX(v).getY = s.getY),$$

where $X \neq Y$.
(*R2-Getter*): For each getter $getX$ that gets the value of attribute X, write the following axiom:

$$\forall s, v \cdot (s.[getX] = s).$$

(*R3-Creator*): For each creator $C(x_1, \cdots, x_n)$, write the following axioms for $i = 1, \cdots, n$.

$$\forall x_1, \cdots, x_n \cdot C(x_1, \cdots, x_n).getX_i = x_i,\ if\ PreCond_{C(x_1, \cdots, x_n)}$$

(*R4-Transformer 1*): For each transformer $F(x)$ and observer $getX$, write an axiom in the following form.

$$\forall s, x \cdot (s.F(x).getX) = \varphi(x, s.getX),\ if\ PreCond_{F(x)}$$

where φ is the function of the service $F(x)$.
(*R5-Transformer 2*): For each operation $P(x)$ that applies to component A that generates an invocation of operation $Q(exp_x)$ on component B of the system, write an axiom in the following form.

$$\forall s, x, y \cdot (s.[A.P(x)].B = s.B.Q(exp_x)),\ if\ PreCond_{A.P(x)}$$

where $PreCond_{f(x)}$ is the pre-condition to apply $f(x)$.

IV. An Example

In this section, we demonstrate the algebraic specification of a web service in CASOCC-WS by an example.

The service is to provide advises on the amount of personal tax when a user inputs his incomes and the types of incomes according to the Chinese personal income tax rules, which classifies personal incomes into three types: salary, business revenue, and services revenue. Here, for the sake of space, we focus on salary incomes. The tax rates for salary incomes are given in Table I, where the taxable income is calculated according to the following formula.

$$TaxableIncome = Income - Pension - Allowance.$$

The total amount of tax to pay is calculated in two different but equivalent ways as follows.

$$\begin{aligned} Tax(X) &= \sum_{i=1}^{K-1} ((U_i - L_i) \times R_i) + (X - L_K) \times R_K \\ &= X \times R_K - D_K \end{aligned} \quad (1)$$

where K is the tax level, U_i and L_i are the upper and lower boundaries of tax level i, respectively, R_i and D_i are the rate and adjustment for level i, respectively, and X is the taxable income. The correctness of formula (1) depends on the values set to D_i, which should be calculated as follows.

$$\begin{aligned} D_1 &= 0; \\ D_k &= U_{k-1} * (R_k - R_{k-1}) + D_{k-1}, k > 1. \end{aligned}$$

Table I
Tax Rates for salary income

Level	Taxable Income range	Rate (%)	Adjustment
1	[0, 500)	5	0
2	[500, 2000]	10	25
3	(2000, 5000]	15	125
4	(5000, 20000]	20	375
5	(20000, 40000]	25	1,375
6	(40000, 60000]	30	3,375
7	(60000, 80000]	35	6,375
8	(80000, 100000]	40	10,375
9	> 100000	45	15,375

The description of the web services in WSDL 1.1 is given below, where, for the sake of space, we have removed the details of the message types and the binding part.

```
<?xml version="1.0" encoding="UTF-8" ?>
 <wsdl:definitions ... >
 <wsdl:message name="getTaxRateRequest">
    ...  </wsdl:message>
 <wsdl:message name="getTaxLevelRequest">
    ...  </wsdl:message>
 <wsdl:message name="getTaxLevelResponse">
    ...  </wsdl:message>
 <wsdl:message name="getReducerRequest">
    ...  </wsdl:message>
 <wsdl:message name="getBaseResponse">
    ...  </wsdl:message>
 <wsdl:message name="getBaseRequest" />
 <wsdl:message name="getReducerResponse">
    ...  </wsdl:message>
 <wsdl:message name="getTaxResponse">
    ...  </wsdl:message>
 <wsdl:message name="getTaxRequest">
    ...  </wsdl:message>
 <wsdl:message name="getTaxRateResponse">
    ...  </wsdl:message>
 <wsdl:portType name="PersonalTax">
 <wsdl:operation name="getBase">
  <wsdl:input message="impl:getBaseRequest"
    name="getBaseRequest" />
  <wsdl:output message="impl:getBaseResponse"
    name="getBaseResponse" />
  </wsdl:operation>
 <wsdl:operation name="getTaxLevel"
   parameterOrder="kind revenue">
  <wsdl:input message="impl:getTaxLevelRequest"
    name="getTaxLevelRequest" />
  <wsdl:output message="impl:getTaxLevelResponse"
    name="getTaxLevelResponse" />
  </wsdl:operation>
 <wsdl:operation name="getTaxRate"
    parameterOrder="kind revenue">
  <wsdl:input message="impl:getTaxRateRequest"
    name="getTaxRateRequest" />
  <wsdl:output message="impl:getTaxRateResponse"
    name="getTaxRateResponse" />
  </wsdl:operation>
 <wsdl:operation name="getReducer"
   parameterOrder="kind revenue">
  <wsdl:input message="impl:getReducerRequest"
    name="getReducerRequest" />
  <wsdl:output message="impl:getReducerResponse"
    name="getReducerResponse" />
  </wsdl:operation>
 <wsdl:operation name="getTax"
   parameterOrder="kind salary pension">
  <wsdl:input message="impl:getTaxRequest"
    name="getTaxRequest" />
  <wsdl:output message="impl:getTaxResponse"
    name="getTaxResponse" />
  </wsdl:portType>
 <wsdl:binding name="PersonalTaxSoapBinding"
   type="impl:PersonalTax">   ...
  </wsdl:binding>
 <wsdl:service name="PersonalTaxService">
  <wsdl:port binding="impl:PersonalTaxSoapBinding"
   name="PersonalTax">
   <wsdlsoap:address
     location="http:.../PersonalTax.jws" />
  </wsdl:port>
 </wsdl:service>
</wsdl:definitions>
```

As shown in the WSDL description of the web service, the service consists of five operations. Their functions are explained in Table II.

Table II
OPERATIONS OF THE TAX ADVISE WEB SERVICES

Name	Function
getBase	It returns the amount of taxable income when given the net income and pension.
getTaxLevel	It returns the tax level when given the amount of taxable income.
getTaxRate	It returns the tax rate when given the amount of taxable income.
getReducer	It returns the amount of adjustment when given the amount of taxable income.
getTax	It returns the total amount of tax to be paid when given the amount of taxable income.

From the WSDL description of the web services, we can generate the following algebraic specification, where additional observer operations on the internal state of the web service and axioms were added manually.

```
Spec PersonalTaxService Is unobservable;
  Import
    getTaxRateRequest,   getTaxRateRespons;
    getTaxLevelRequest,  getTaxLevelResponse,
    getBaseRequest,      getBaseResponse,
    getReducerRequest,   getReducerResponse,
    getTaxRequest,       getTaxResponse
  Operations:
   Transformer:
     getBase: [PersonalTaxService]
       getBaseRequest -> getBaseResponse;
     getTaxLevel: [PersonalTaxService]
       getTaxLevelRequest -> getTaxLevelResponse;
     getTaxRate: [PersonalTaxService]
       getTaxRateRequest -> getTaxRateResponse;
     getReducer: [PersonalTaxService]
       getReducerRequest -> getReducerResponse;
     getTax: [PersonalTaxService]
       getTaxRequest -> getTaxResponse;
   Observers:
     TaxRate: [PersonalTaxService] Int -> Real;
     Allowance: [PersonalTaxService] VOID -> Real;
     TaxLevel: [PersonalTaxService] Real -> Int;
     Adjustment: [PersonalTaxService] Int -> Real;
     Lower: [PersonalTaxService] int -> Real;
     Upper: [PersonalTaxService] Int -> Real;
  Axioms:
   For all x, k: Int;
    PTS:PersonalTaxService, gBR: getBaseRequest,
    gTLR:getTaxLevelRequest, gTax:getTaxRequest,
    gRR:getReducerRequest,gTRR:getTaxRateRequest,
   that
     PTS.getBase(gBR).amount
      = gBR.salary - gBR.pension -PTS.Allowance;
     PTS.getTaxLevel(gTLR).Level
      = PTS.TaxLevel(gTLR.amount);
     PTS.getTaxRate(gTRR).Rate
      = PTS.TaxRate(gTRR.Level);
     PTS.getReducer(gRR).amount
      = PTS.Adjustment(gRR.Level);
     Let k=TaxLevel(gTax.amount) in
       PTS.getTax(gTax).amount
      = (gTax.amount * PTS.TaxRate(k))
         -PTS.Adjustment(k)
     end
    Let k = PTS.TaxLevel(x) in
      PTS.Lower(k) > x;
      PTS.Upper(k) <= x;
    end;
    PTS.Lower(k) < PTS.Upper(k), if k >0 & k<=9;
    PTS.Lower(k+1) = PTS.Upper(k), if k>0 & k<9;
    PTS.Adjustment(1)=0;
    PTS.Adjustment(k+1)
     =(PTS.TaxRate(k+1)-PTS.TaxRate(k))
      * PTS.Upper(k) + PTS.Adjustment(k),
        if k>0 & k<9
End
```

Note that, the axioms about the amount of tax to be paid is according to the formula (1).

V. Conclusion

In this paper, we proposed an algebraic specification language CASOCC-WS for web services. A set of rules to transform WSDL descriptions of web services into algebraic structures is also proposed and illustrated by an example.

There are algebraic specification languages, such as CoCasl [20], that allow both algebraic and co-algebraic specifications. However, for each operator, either the domain or the co-domain must be a singleton. Moreover, CoCasl requires that a specification unit is either *algebraic* (i.e. all operators have singleton domain) or *co-algebraic* (i.e. all operators have singleton co-domain), but not a mixture. In [21], a unit of specification can be in the so-called (Ω, Ξ)-structure. That is, a specification unit can contain a set of algebraic operators plus a set of co-algebraic operators, but for each operator one of its domain and co-domain must be singleton.

Although (Ω, Ξ)-structure is much more flexible, its restrictions are still too stringent for software components and services. For example, suppose that an online ticket booking service `BOOKING` provides a service `BookTicket`, which takes two parameters `DATE` and number of seats, and returns a message to inform whether the number of seats is available. Its signature has non-singleton domain and co-domain because the internal state of booking is changed. The CASOCC-WS language relaxes the restrictions on the domains and co-domains of operators so that each service can be naturally represented as one operator. Thus, the ticket booking service can be written in CASOCC-WS as follows, but not in any existing algebraic specification languages.

```
BookTicket:[ BOOKING] DATE,NAT -> MESSAGE.
```

Because each service can be naturally represented as an operator and the sorts can map into types of software entities with one-one correspondence, CASOCC-WS supports modular development of formal specifications of services. This enables us to derive the algebraic structures of service automatically as shown in this paper. A prototype system that implements the rules that transform WSDL descriptions into algebraic structures have been implemented. The example given in this paper was generated by the tool. We are further developing the tool toward automated testing of web services based on CASOCC-WS specifications.

References

[1] A. van Lamsweerde, "Formal specification: a roadmap," in *Proc. of ICSE'00 - Future of SE Track*, 2000, pp. 147–159.

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

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

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

[5] C. Cîrstea, "Coalgebra semantics for hidden algebra: Parameterised objects an inheritance," in *Prof. of WADT'97*, 1997, pp. 174–189.

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

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

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

[9] D. Sannella and A. Tarlecki, "Algebraic methods for specification and formal development of programs," *ACM Comput. Surv.*, vol. 31, no. 3es, p. 10, 1999.

[10] J. D. Gannon, P. R. McMullin, and R. G. Hamlet, "Data-abstraction implementation, specification, and testing," *ACM Trans. Prog. Lang. Syst.*, vol. 3, no. 3, pp. 211–223, 1981.

[11] P. Dauchy, M.-C. Gaudel, and B. Marre, "Using algebraic specifications in software testing: A case study on the software of an automatic subway," *Journal of Systems and Software*, vol. 21, no. 3, pp. 229–244, 1993.

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

[13] R.-K. Doong and P. G. Frankl, "The astoot approach to testing object-oriented programs," *ACM Trans. Softw. Eng. Methodol.*, vol. 3, no. 2, pp. 101–130, 1994.

[14] M. Hughes and D. Stotts, "Daistish: systematic algebraic testing for oo programs in the presence of side-effects," in *Proc. of ISSTA '96*, 1996, pp. 53–61.

[15] 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.

[16] 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.

[17] L. Kong, H. Zhu, and B. Zhou, "Automated testing ejb components based on algebraic specifications," in *Proc. of COMPSAC'07 (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] H.Zhu and B.Yu, "An Experiment with Algebraic Specifications of Software Components," in *Proc. of QSIC'10*, 2010.

[20] T. Mossakowski, L. Schröder, M. Roggenbach, and H. Reichel, "Algebraic-coalgebraic specification in CoCasl," *J. Log. Algebr. Program.*, vol. 67, no. 1-2, pp. 146–197, 2006.

[21] A. Kurz, "Logics for coalgebras and applications to computer science," Ph.D. dissertation, Ludwig-Maximilians Universitat Munchen, July 2000.

[22] I. Chiswell and W. Hodges, *Mathematical Logic*, Oxford University Press, 2007.