A Formal Approach to Composing Abstract Scenarios of Web Services*

Mariusz Jarocki¹, Artur Niewiadomski³, Wojciech Penczek²,³, Agata Półrola¹, and Maciej Szreter²

¹ Faculty of Mathematics and Computer Science, University of Lodz, Lodz, Poland
² Institute of Computer Science, Polish Academy of Sciences, Warsaw, Poland
³ Institute of Computer Science, University of Podlasie, Siedlce, Poland

Abstract

An efficient and fully automatic composition of web services is possible when assuming a uniform, consistent semantics for all of them. The paper proposes a method of describing the environment in which the services operate as a set of worlds consisting of formally defined objects. The services are seen as transitions between worlds. Consequently, the problem of automatic composition of services is converted into the problem of searching for the paths in the transition graph which lead from an initial world specified by the user to a world satisfying user’s request, and choosing (possibly) the optimal one. The paper deals with the first stage of the composition, i.e., building an abstract graph representing types of services whose activation can potentially allow to reach the goal. A theoretical description of the approach is supported by some preliminary experimental results.

Keywords: automated composition, web services, abstract planning

1 Introduction

In recent years there has been a growing interest in automating the composition of web services. The number of more and more complex Internet services is still growing nowadays; several standards describe how services can be invoked (WSDL), how they exchange information (SOAP), how they synchronise the executions in complex flows (WS-BPEL), and finally how they can be discovered (UDDI). However, still there is a lack of automatic methods for arranging and executing their flows. One of the problems to deal with is the size of the environment—most existing composition methods work with concrete instances of web services, so even a simple query requires taking all the instances of all the types of services into account. Another problem follows from incompatibilities in inputs/outputs of services, and difficulties in comparing their capabilities and qualities—two services

*Supported by the Project “Nowe technologie informacyjne dla elektronicznej gospodarki społeczeństwa informacyjnego oparte na paradigmacie SOA”, funded by the Polish Ministry of Science and Higher Education and supported by th European Union in the scope of the European Regional Development Fund program no. POIG.01.03.01-00-008/08.
can offer the same functionality, but this fact cannot be detected automatically without unification of their interfaces made by the providers.

Our approach to automatic composition of web services, proposing a solution to the above problems, is based on the following concepts. We introduce a uniform semantic description of service types. In order to adapt a possibly wide class of existing services, specific interfaces of concrete services are to be translated to the common one by adapters (called proxies), built in the process of service registration. The process is to be based on descriptions of interfaces of services, specified both in WSDL and in the languages containing a semantic information, like OWL-S or Entish (Ambroszkiewicz, 2003). The mechanism of proxies is also to allow for using services in an “offer mode” (not resulting in changes in internal states of the services) and in an “execution mode”. Collecting “offers” enables choosing an optimal solution.

The client’s goal is expressed in a fully declarative intention language. The user describes two worlds: the initial and the final one, using the notions coming from an ontology, and not knowing any relations between them or between the services. The task of the composition system consists in finding a way of transforming the initial world into the final one. The composition is three-phase like in the Entish project (Ambroszkiewicz, 2003). In the first phase, called abstract planning or planning in types, we create an abstract plan, which shows sequences of service types whose executions possibly allow to accomplish the goal. The second phase makes these scenarios “concrete”, which means replacing the types of services by their concrete instances. This can also involve choosing a plan which is optimal from the user’s point of view. Finally, the last phase consists in supervising the execution of the optimal run, with a possibility of correcting it in the case of a service failure. The current work deals with the first phase of the composition.

The rest of the paper is organised as follows: Sec. 2 presents the related work. The main ideas of our approach and the notions behind them are introduced in Sec. 3. Sec. 4 shows an algorithm for abstract planning and defines the graph being its result. Sec. 5 presents an implementation of the abstract planner, illustrated by some results. Sec. 6 contains final remarks and sketches directions of our future work.

2 Related Work

There are many papers dealing with the topic of web services composition (Klusch et al., 2005; Rao, 2004; Rao et al., 2004; Rao and Su, 2004; Redavid et al., 2007; Srivastava and Koehler, 2003). Some of these works consider static approaches, where flows are given as a part of the input, while the others deal with dynamically created flows. One of the most active research areas is a group of methods referred to as A1 Planning (Klusch et al., 2005). Several approaches use Planning Domain Definition Language—PDDL (McDermott et al., 1998). Another group of methods is built around the so-called rule-based planning, where composite services are generated from high-level declarative descriptions, and compositionality rules describe the conditions under which two services are composable. The information
obtained is then processed by some designated tools. The project SWORD (Ponnekanti and Fox, 2002) uses an entity-relation formalism to specify web services. The services are specified using pre- and postconditions; a service is represented as a Horn rule denoting that the postcondition is achieved when the preconditions are true. A rule-based expert system generates a plan. Another methodology is the logic-based program synthesis (Rao et al., 2004). Definitions of web services and user requirements, specified in DAML-S, are translated to formulas of Linear Logic (LL): the descriptions of web services are encoded as LL axioms, while a requirement is specified as a sequent to be proven by the axioms. Then, a theorem prover determines if such a proof exists.

While the approaches described above are automatic, there are also semi-automatic methods assuming human assistance at certain stages (Sirin et al., 2003). Some approaches are based on specifying a general plan of composition manually; the plan is then refined and updated in an automatic way.

Inspired by the Entish project (Ambroszkiewicz, 2003), our approach enables to model automated composition based on matching input and output types of services.

A full version of this paper will be published in the book Ambroszkiewicz et al. (2010, editors).

3 Basic Notions

One of the main elements of our approach consists in introducing a unified semantics for functionalities offered by services, which is done by defining a dictionary of notions/types describing their inputs and outputs. A service is then understood as a function which transforms a set of data into another set of data (or as a transition between them). The sets of data are called worlds. The worlds can be described by the use of an ontology, i.e., a formal representation of knowledge about them. The concepts used to model individuals are a hierarchy of types as well as classes and objects.

In order to ensure an easy integration with other solutions, we use the OWL language for defining ontologies. Formally, an ontology is a set of definitions of classes (ordered in an inheritance hierarchy), their instances and relations between them\(^1\). A class is a named OWL template which defines names and types of attributes. All the classes are ordered in a multiple inheritance hierarchy. The fact that a class \(B\) inherits from a class \(A\) means that \(B\) contains both all the attributes of \(A\), and the attributes specified in its own definition\(^2\). The class \(B\) is called a subclass, a child class or a derived class; the class \(A\) is called a superclass, a parent class or a base class. The inclusion between the sets of attributes of a parent and a derived class implies that if in some context an object of a certain class is required, then an object of an arbitrary subclass of that class can be used instead.

---

\(^1\)OWL ontology definition.

\(^2\)We assume that names of attributes are unique in the whole name space. Moreover, attribute encapsulation and overloading are not supported (which follows from using OWL as the ontology description language).
A class is called *abstract* if instantiating it (i.e., creating objects following the class definition) is useless in the sense that the objects obtained this way do not correspond to any real-world entity\(^3\). Abstract classes can be used, among others, for defining formal parameters of services. Moreover, on the top of the inheritance hierarchy there is an abstract base class *Thing* of no attributes\(^4\).

### 3.1 Worlds

Assume we have a set of objects containing instances of classes defined in an ontology.

**Definition 1 (World and objects)** The *universum* is the set of all the objects. The objects have the following features:

- each object is either a concrete object or an abstract object,
- each object contains named attributes whose values can either be other objects or be:
  - values of simple types (numbers, strings, boolean values; called simple attributes) or NULL (empty value) for concrete objects,
  - values from the set \{NULL, SET, ANY\} for abstract objects.

If an attribute \(A\) of the object \(O\) is an object itself, then \(O\) is extended by all the attributes of \(A\) (of the names obtained by adding \(A\)’s name as a prefix). Moreover, when an object having an object attribute is created, its subobject is created as well, with all the attributes set to NULL.

- each simple attribute has a boolean-valued flag \(\text{const}\).

A world is a set of objects chosen from the universum. Each object in a world is identified by a unique name.

By default each \(\text{const}\) flag is set to false. If the flag of an attribute is true, then performing on the object any operation (service) which sets this attribute (including services initialising it) is not allowed (the value of the attribute is considered to be final). Interpretation of the values of the attributes for abstract objects is as follows: NULL means that no value of the attribute has been set (i.e., the attribute has the empty value), SET means that the attribute has a nonempty value, while ANY means that the state of the attribute cannot be determined (i.e., its value can be either SET or NULL). The attributes are referred to by ObjectName.AttributeName.

**Definition 2 (Object state, world state)** A state of an object \(O\) is a function \(V_o\) assigning values to all the attributes of \(O\) (i.e., is the set of pairs (AttributeName, AttributeValue), where AttributeName ranges over all the attributes of \(O\)). A state of a world is a set of states of all its objects.

In order to reason about worlds and their states we define the following two-argument functions (the second default argument of these functions is the world we are reasoning about):

\(^3\)There is no explicit possibility in OWL to express the fact that a class is not instantiable.

\(^4\)The rules of class inheritance, no formal definition of abstract class and the common root of the inheritance tree are also taken from OWL.
• **Exists** – a function whose first parameter is an object, and which says whether the object exists in the world,

• **isSet** – a function whose first parameter is an attribute of an object, and which says whether the attribute is set (has a nonempty value),

• **isConst** – a function whose first parameter can be either an attribute or an object. When called for an attribute, the function returns the value of its `const` flag; when called for an object it returns the conjunction of the `const` flags of all the attributes of this object.

In the process of abstract planning, i.e., when values of object attributes are not known but we know whether an attribute or an object was modified by the services used before, one can use the above three functions only. Therefore, the abstract planner allows us only to judge whether the object of interest exists, and what the status of its attributes (NULL, SET or ANY) is.

### 3.2 Services

The ontologies collect the knowledge not only about the structure of worlds, but also about the ways they can be transformed, i.e., about services. The services are organised in a hierarchy of classes, and described both on the level of classes (by specifying what all the services of a given class do—such a pattern of behaviour is referred to as an abstract service or a metaservice), and on the level of objects (concrete services). The description of a service includes, besides specifying input and output data types, also declaration of introducing certain changes to a world, i.e., of creating, removing and modifying objects. The definition of a service is as follows:

**Definition 3 (Service)** A service is an object of a non-abstract subclass of the abstract class Service. A service contains (initialised) attributes, inherited from the base class Service. The attributes can be grouped into processing lists (the attributes produces, consumes, requires), modification lists (the attributes mustSet, maySet, mustSetConst, maySetConst), and validation formulas (the attributes preCondition and postCondition). Moreover, a service can contain a set of quality attributes.

A service modifies (transforms) a world, as well as the world’s state. The world to be transformed by a service is called its pre-world (input world), while the result of the execution is called a post-world (output world). Modifying a world consists in modifying a subset of its objects. The objects being transformed by one service cannot be modified by another one at the same time (i.e., transforming objects is an atomic activity). A world consisting of a number of objects can be transformed into a new state in two ways\(^5\): by a service which operates on a subset of its elements, or by many services which operate concurrently on disjoint subsets of its elements.

**Definition 4 (Processing lists)** The processing lists are as follows:

\(^5\)Services which create new objects are not taken into account.
• **produces**—a list of named objects of classes whose instances are created by the service in the post-world,

• **consumes**—a list of named objects of classes whose objects are taken from the input world, and do not exist in the world resulting from the service execution (the service removes them from the world),

• **requires**—a list of named objects of classes whose instances are required to exist in the current world to invoke the service and are still present in the output world.

The structure of the lists is similar to the lists of the formal parameters of the procedures (see Jarocki et al. 2010 for the grammar). An example is presented in the appendix (see Example 1). The formal parameters from the above lists define an alphabet for modification lists and validation formulas.

**Definition 5 (Modification lists)** The modification lists are as follows:

• **mustSet**—a list of attributes of objects occurring in the lists **produces** and **requires** of a service, which are obligatorily set (assigned a nonempty value) by this service,

• **maySet**—a list of attributes of objects occurring in the lists **produces** and **requires** of a service, which may (but not must) be set by this service,

• **mustSetConst**—a list of attributes of the objects which occur in the lists **produces** and **requires** of a service, which are obligatorily set as being constant in the worlds after executing this service,

• **maySetConst**—a list as above, but of the attributes which may be set as constant.

A grammar for the above lists can be found in Jarocki et al. 2010. An example is provided in the appendix (see Example 2). The attributes of the objects appearing in processing lists which do not belong to the union of lists **mustSet** and **maySet** are not changed when the service is called.

In the process of abstract planning, each attribute from the list **mustSet** is considered to have the value **SET** (the function **isSet** called for this attribute will return the value **true**), whereas each attribute from the list **maySet** is seen as having either the value **SET** or the value it had before executing the service.

**Definition 6 (Validation formulas)** The validation formulas are as follows:

• **preCondition**—a propositional formula which describes the condition under which the service can be invoked. It consists of atomic predicates over the names of objects from the lists **consumes** and **requires** of the service and over their attributes, and is written in the language of the propositional calculus (atomic predicates with conjunction, disjunction and negation connectives). The language of atomic predicates contains comparisons of expressions over attributes with constants, and functions calls with object names and attributes as arguments. In particular, it contains calls of the functions **isSet**, **isConst** and **Exists**

---

Using **Exists** in **preCondition** is redundant w.r.t. using an appropriate object in the list **consumes** or **requires**. However, the future directions of developing the service description language mentioned in the final part of the paper, include moving modification lists to validation formulas.
• **postCondition**—a propositional formula which specifies conditions satisfied by the world resulting from invoking the service. The formula consists of atomic predicates over the names of objects from the lists `consumes`, `produces` and `requires` of the service and over their attributes. To the objects and attributes one can apply pseudofunctions `pre` and `post` which refer to the state of an object or an attribute in the input and the output world of this service, respectively. By default, the attributes of objects listed in `consumes` refer to the state of the pre-world, whereas these in `produces` and `requires`—to the state of the post-world.

The validation formulas are built in a way similar to the expressions in the high-level programming languages. However, for abstract planning we use their reduced forms which are DNF formulas (i.e., formulas in a disjunctive normal form), with atomic predicates being (possibly negated) calls of the functions `isSet`, `isConst` or `Exists`. We assume that an arbitrary predicate, being a function over the set of objects and their attributes, is transformed by replacing arguments of functions by the conjunction of calls of `isSet` over attributes. Again, a complete grammar of validation formulas for the abstract planner can be found in Jarocki *et al.* 2010. Some examples illustrating the above definition are provided in the appendix.

In order to be able to provide some additional information enabling comparison of the quality of services, service classes can contain **quality attributes** which are set while a service is executed. These attributes can be used in user queries (in an execution condition and in a quality function, see Sec. 3.3). They can be introduced, among others, by base abstract services which collect certain common features of services, e.g. “chargeable services” (assigned with the attribute of price) or “time-taking services” (assigned with timing interval). The above attributes are not used in the abstract planning.

### 3.2.1 Service Types, Inheritance, Metaservices

Each class of services (service type) has certain features common for all the instances of this type. They are specified by the instance of the class called a **metaservice** or an **abstract service** (i.e., an object of the service which describes the whole class of services).

A description of a **concrete service** should be understood as a list of differences or as narrowing the template introduced as the metaservice. More precisely, a concrete service can overload the processing lists of its metaservice by narrowing the class of objects it works on. This is done by using, in an appropriate list, a formal parameter of the same name and of a restricted domain. Moreover, a concrete service can extend the modification lists of its metaservice only by declaring that it modifies attributes added by a narrowed class of parameters. This prevents the definition of a concrete service to be inconsistent with the definition of the metaservice. Considering validation formulas, each formula `preCondition`
(postCondition, respectively) of a concrete service is a conjunction of the precondition (postcondition resp.) of the metaservice and explicitly given precondition (resp. postcondition) of the concrete service. Some examples illustrating the above dependencies can be found in the appendix.

As far as inheritance is concerned, a child class can extend the sets of objects which are consumed, produced and required by its base class, using the appropriate processing lists to this aim (so, it is allowed to extend the subset of a world influenced by a service). It can also narrow the types of parameters in the processing lists, which is done by using the same names of formal parameters as in the lists of the base class. By default (when its lists are empty) a child class inherits the specification of the parent class. The modification lists can be extended in a similar way (i.e., by extending the sets of attributes which are modified). The declarations of setting attributes are not restricted only to the attributes added by the child class in the lists produces and requires—it is also allowed to modify attributes not changed by the metaservice of the parent class. Validation formulas are handled in a way similar to the case of concrete services—an explicitly specified condition of a derived class is conjuncted with the appropriate condition from the parent class.

As we mentioned before, the attributes of objects appearing in processing lists which do not belong to the sum of lists mustSet and maySet are not changed when the service is called. This, however, does not apply to the attributes added to modification lists in narrowed types introduced by concrete services. Potential inconsistencies, resulting from concatenation of processing and modification lists in child classes of services, are treated as ontology errors\(^8\).

### 3.2.2 Modifying a World

Separating calls of single services is one of key concepts in our approach. A service processes a world, irrespectively of how many services brought this world to the current form. The result of executing a service is a (possibly new) world with a new state. However, the pre- and post-world of a service satisfy certain conditions described below:

**Definition 7** A service \( u \) is enabled (executable) in the current state of a world \( s \) if:

- each object \( o \) from the lists consumes and requires of \( u \) can be mapped onto an object in \( s \), of the class of \( o \) or of its subclass; the mapping is such that each object in \( s \) corresponds to at most one object from the above lists;
- for the objects in \( s \) which, according to the above mapping, are actual values of the parameters in consumes and requires the formula preCondition of \( u \) holds,
- the list mustSet of \( u \) does not contain attributes for which in objects which are actual values of the parameters the flag const is set.

\(^8\)We do not assume an “expanded” inheritance hierarchy of services, contrary to hierarchy of types of their “objects”. A suggested model of service inheritance is three-level: on the first level the class Service as a “carrier” of basic attributes, on the second level classes carrying additional quality attributes (see p. 9), and on the third level classes of services with definitions of their metaservices.
Definition 8  A service $u$ executable at the current world $s$ produces a new world $s'$ in which:

- there are all the objects from $s$, besides these which in the mapping done for executing $u$ were actual values for the parameters in $\text{consumes}$,
- there is a one-to-one mapping between all the other objects in $s'$ and the objects in the list $\text{produces}$ of $u$, such that each object $o$ from the list $\text{produces}$ corresponds to an object in $s'$ which is of a (sub)class of $o$;
- for the objects which, according to the above mappings, are actual values of the parameters in the processing lists the formula $\text{postCondition}$ holds,
- in the objects which are actual values of the appropriate parameters the flags $\text{const}$ of the attributes listed in $\text{mustSetConst}$ of $u$ are set, and the attributes listed in $\text{mustSet}$ of $u$ have nonempty values,
- assuming the actual values of the parameters as above, all the attributes of all the objects existing both in $s$ and in $s'$ which do not occur neither in $\text{mustSet}$ nor in $\text{maySet}$ have the same values as in the world $s$; the same holds for the flags $\text{const}$ of the attributes which do not occur neither in $\text{mustsetConst}$ nor in $\text{maySetConst}$. Moreover, all the attributes listed in $\text{mustSet}$ or $\text{maySet}$ which are of nonempty values in $s$, in $s'$ are of nonempty values as well.

3.3 Queries

A user describes its goal in a declarative language defined by the ontology. He specifies (possibly partially) an initial and a final (desired) world, possibly giving also some evaluation criteria. The query is defined in the following way:

Definition 9 (Query) A query consists of the following elements:

- an initial domain—a list of named objects which are elements of the initial world. The form of the list is analogous to the form of the list $\text{produces}$ in the description of a service;
- an initial clause specifying a condition which is to be satisfied by the initial world. The clause is a propositional formula over the names of objects and their attributes, taken from the initial domain. The grammar of the clause is analogous to the grammar of the $\text{preCondition}$;
- an effect domain—a list of named objects which have to be present in a final world (i.e., a subset the final world must contain);
- an effect clause specifying a condition which is to be satisfied by the final world. The clause is a propositional formula over the names of objects and their attributes from both the domains defined above; references to the initial state of an object, if ambiguous, are specified using the notations $\text{pre}(\text{objectName})$ and $\text{post}(\text{objectName})$, analogously as in the language used in the formulas $\text{postCondition}$ of services. The grammar of the effect clause is analogous to the grammar of the $\text{postCondition}$;
- an execution condition—a formula built over services (unknown to the user when specifying the query) from a potential run performing the required transformation of the initial world into a target world. While construction of this formula, simple methods of quantification and aggregation are used;
• a quality function—a real-valued function over the initial world, the final world and services in a run, which specifies a user’s criterion of valuating the quality of runs. The run of the smallest value of this function is considered to be the best one.

The last two parts of a query are used after finishing both the abstract planning phase and the first part of concrete planning, which adjusts types and analyses pre- and postconditions of concrete services.

On the abstract level, the initial clause and the effect clause are specified as DNF formulas over the predicates isSet and Exists. This means that (not taking into account the variants of worlds following from the disjunctive form) the query can be reduced to enumerating:

• objects in the initial world,
• objects in the final world, carrying an information which of them were present in the initial world (which is done by using the same names of formal parameters),
• objects that are to be removed from the initial world,
• attributes which in the final world must have nonempty values,
• attributes which in the final world must have empty values (must be null).

In other words, a list of objects in the initial domain and the DNF form of the initial clause generates a number of alternative initial worlds whose states (values of attributes) are set according to (possibly negated) predicates occurring in the initial clause. Some examples of queries can be found in the appendix.

For a better efficiency of the composition we introducte equivalence of worlds w.r.t. the query considered.

**Definition 10 (Highlighted objects)** An object is called highlighted w.r.t. a user’s query if its name occurs in both the initial and the effect domain of this query.

**Definition 11 (Equivalent worlds)** Two worlds s and s’ are called equivalent if the sets of their highlighted objects are equal, and their complements are equal when names of objects are left aside (i.e., for each object o1 from one set there is exactly one corresponding object o2 from the second set, such that o1 and o2 are objects of the same class, and the values of all the attributes in both the objects are the same).

4 Abstract Planning

The aim of a composition process is to find a path in the graph of all the possible transitions between worlds which leads from a given initial world to a given final world, specified (possibly partially) in a user’s query, using no other knowledge than that contained in the ontology. The composition is three-phase; the first phase we are dealing with in this work consists in finding all the sequences of service types (abstract services) which can potentially lead to satisfying the user’s goal. The result of the abstract planning phase is an abstract graph.
Algorithm 1 Computing an abstract graph of services

Require: a query $\varphi = (\varphi_S, \varphi_E)$, a maximal search depth $k$.
Ensure: a graph $G$

a queue $Q$
put the world described by $\varphi_S$ into $Q$, assign them the depth 0, mark them as initial
while $Q$ is nonempty do
get $s$ from $Q$
if $s$ processed then
continue;
endif
if $s$ satisfies $\varphi_E$ then
mark $s$ as final, mark $s$ as processed
continue;
endif
if depth of $s$ greater than $k$ then
mark $s$ as processed
continue;
endif
for each service $S$ defined in the system do
// check whether the world $s$ can be processed by $S$:
if $s$ does not contain certain objects from $S$.consumes, $S$.requires then
continue;
endif
if $s$ does not satisfy $S$.preCondition then
continue;
endif
generate all the subsets of the objects in $s$ satisfying $S$.preCondition and such that the attributes listed in $S$.mustSet have the flag const not set
for all element $s_p$ do
create the world $s'_p$ resulting from transforming $s_p$ by $S$
if exists $v \in G$ such that $v \equiv s'_p$ then
add the edge $s_p \xrightarrow{S} v$ to $G$
add $v$ to $Q$
else
add to $G$ the node $s'_p$ and the edge $s_p \xrightarrow{S} s'_p$
add $s'_p$ to $Q$
end if
end for
mark $s$ as processed
end for
end while

The abstract graph is a directed multigraph. The nodes of the graph are worlds in certain states, while its edges are labelled by services. Notice that such a labelling carries an information what part of a input world (node) is transformed by a given service (which is specified by actual values of the parameters in consumes and requires of the service), and what part of the output world (node) it affects (the lists produces and requires of this service). We distinguish some nodes of the graph—these which have no input edges represent alternative initial worlds, while these with no output edges are alternative final worlds. A formal definition of the abstract graph is as follows:

Definition 12 An abstract graph is a tuple $GA = (V, V_p, V_k, E, L)$, where

- $V$ is a subset of the set $S$ of all the worlds,
- $V_p \subseteq V$ is a set of initial nodes,
- $V_k \subseteq V$ is a set of final nodes,
- $E \subseteq V \times V$ is a transition relation s.t. $e = (v, v') \in E$ iff $L(e)$ transforms the world $v$ into $v'$, where
• \( L : E \rightarrow U \) is a function labelling the edges with services.

Below, we show a forward-search-mode algorithm for automatic composition of abstract services. The input for the algorithm consists of an (OWL) ontology containing services and objects they operate on, a users query specifying an initial world and a final world, and a natural number \( k \) which limits the depth of the search. The value \( k \) bounds the maximal number of services which can be composed to transform the initial world into the final world. The algorithm builds an abstract graph which shows how the worlds are transformed by executing services: its nodes correspond to the worlds, while the edges—to the services executed. The graph built by the algorithms presents all the possible scenarios of transforming the initial world into the final world using at most \( k \) services.

For a better readability we present a basic version of the algorithm, without optimisations (see Algorithm 1). For simplicity of description, we distinguish in the query a part referring to an initial world (\( \varphi_S \)) and a part referring to a final world (\( \varphi_E \)).

5 Implementation and Experiments

The composition method presented above has been implemented. Our abstract planner can be accessed via a graphical user interface, or from the level of Java via API. The program is implemented in Java, using the following components:

- the graph library jgraphT (representation on graphs and operations on them),
- the module jgraph (graphs visualisation),
- the parser ANTLR of the AtLa and QLA languages (see below),
- Jena library (accessing OWL files generated by the Protege tool).

The structure of the tool is shown in Fig. 1. In the implementation, ontologies

![Figure 1: A structure of the application](image-url)

are modelled using the OWL language and the Protege environment. We define a hierarchy of types which are either objects of classes derived from the class `Service` (representing services), or objects which do not inherit from the above class (modeling items to be processed by services)). Conditions on the input
and on the output world of each service are specified in the AtLa\textsuperscript{9} language using attributes of services. The ontology contains both types of services and concrete instances of services. An input to the composition system is a query, in which the user specifies an initial world $W_b$ and a final world $W_f$, using the QLa\textsuperscript{10} language. Given a query, the abstract planner builds an abstract graph which describes possible solutions using types of services. Although the algorithm operates in BFS mode, it can also work as a forward search (in such a case the start state is given by $W_b$, and the termination condition is given by $W_f$) or as a backward search (the start state is then given by $W_f$, and the termination condition - by $W_b$).

In order to provide a nice representation of the results, the graph produced is optionally represented in BPEL. This enables using visualisation and processing tools designed for graphs described in that language.

We have tested our tool on the following example: a user requests having some amount of juice. The ontology contains both services which sell juice, and services which can produce juice from fruits, which in turn requires buying fruits before. A complete specification of the example can be found in the appendix. The ontology for the example (a diagram of hierarchy of classes read from an OWL file by our tool) is presented in the Fig. 2.

![Figure 2: The ontology for the Getting Juice Example](image)

The result of the composition is displayed in Fig. 3. The left-hand side of the screen contains a description of the query. The abstract graph is shown on the right. The dark nodes correspond to the initial domain (these are empty worlds). The node in the middle represents the final world. Each path from an initial world to the final world represents a sequence of services whose execution can satisfy the user's request. The sequences are: (1): SelectWare, then FruitSelling and then MakingJuice, (2): SelectWare and then JuiceSelling, (3): SelectWare, then Selling and then MakingJuice, and (4): SelectWare and then Selling. It should be noticed that the planner generates all the possible paths, not only these involving the services specified in a "most detailed" way.

\textsuperscript{9}ATtribute LAnguage

\textsuperscript{10}Query LAnguage
6 Final Remarks

The system presented in this paper is on an early stage of development. Our aim is to prepare an easy mechanism of automatic composition, which can be applied to various domains. The areas of our particular interest are e-commerce and web support for medical services. Moreover, directions of our future research involve a complete specification of the grammars for the validation formulas of concrete services, and the problem of building proxies connecting the composition system with real-world web services, together with mechanisms enabling service registration. It seems also necessary to extend the languages of formulas to a complete first-order language (with quantification). In particular, the modification lists should become elements of validation formulas, which would enable specifying optional modifications of a world. Moreover, solving further, bigger examples modelling real-world business processes will allow to evaluate a practical applicability of the approach.

References

S. AMBROSKUWIECZ (2003), enTish: An Approach to Service Description and Composition, ICS PAS, Ordana 21, 01-237 Warsaw.


DAML-S (2003), DAML-S (and OWL-S) 0.9 Draft Release, http://www.daml.org/services/daml-s/0.9/.


M. KLUSCH, A. GEBER, and M. SCHMIDT (2005), Semantic Web Service Composition Planning with OWLS-XPlan, in Proc. of the 1st Int. AAAI Fall Symposium on Agents and the Semantic Web, AAAI Press.
A Formal Approach to Composing Abstract Scenarios


J. Rao, P. Küngas, and M. Matskin (2004), Logic-based Web Services Composition: From Service Description to Process Model, in Proc. of the IEEE Int. Conf. on Web Services (ICWS'04), IEEE Computer Society.


A Examples Illustrating the Definitions

In order to illustrate the definitions and to explain some their less intuitive aspects, we provide several examples. As the paper deals with abstract planning only (not concrete planning), the (concrete) validation formulas in the examples are specified using a language which is not introduced in the paper, but is similar to the language of the expressions in popular programming languages. The only parts of these formulas seen by the abstract planner are the variables used (e.g., the fact that an expression contains a variable \(x\), but not the fact that the expression is \(x>4\)).

We start with two examples presenting processing lists (Def. 4) and modification lists (Def. 5) of services.

**Example 1** A juice making service \(JMS\), producing juice from fruits and sugar, can have the processing lists of the form:

\[
\begin{align*}
JMS.\text{consumes} & = \{ f: \text{Fruits}; s: \text{Sugar} \} \\
JMS.\text{produces} & = \{ j: \text{Juice} \}
\end{align*}
\]
The processing lists of a transport service $TS$ can be as follows:

\[
\begin{align*}
TS.\text{consumes} & = \{ \} \\
TS.\text{produces} & = \{ \} \\
TS.\text{requires} & = \{ w:\text{Ware} \}
\end{align*}
\]

**Example 2** The juice making service $JMS$ considered in Example 1 can have the modification lists of the form

\[
\begin{align*}
JMS.\text{mustSet} & = \{ j.\text{capacity}, j.\text{name}, j.\text{owner}, j.\text{location} \} \\
JMS.\text{maySet} & = \{ \} \\
JMS.\text{mustSetConst} & = \{ j.\text{name} \} \\
JMS.\text{maySetConst} & = \{ \}
\end{align*}
\]

which means that the juice produced has certain name and capacity, belongs to an owner and is in certain location, and its name cannot be changed.

The modification lists of the transport service $TS$ considered in Example 1 can be as follows:

\[
\begin{align*}
TS.\text{mustSet} & = \{ w:\text{location} \} \\
TS.\text{maySet} & = \{ \} \\
TS.\text{mustSetConst} & = \{ \} \\
TS.\text{maySetConst} & = \{ w:\text{location} \}
\end{align*}
\]

which means that the services changes the location of the ware, and possibly fixes it (i.e., can transport the ware to a final destination).

The next two examples show atomic predicates and validation formulas (Def. 6):

**Example 3**

- Atomic predicates: \( x.\text{number}>0 \), \( \text{not isSet}(x.\text{id}) \)
- preCondition: \( x.\text{number}>0 \) and \( \text{isSet}(x.\text{id}) \)
- postCondition: \( \text{not } x.\text{number} = \text{pre}(x).\text{number} \)

**Example 4** The juice making service $JMS$ considered in Example 1 can have the validation formulas of the form

\[
\begin{align*}
JMS.\text{preCondition} & = \{ \text{isSet}(f.\text{capacity}) \) and \text{isSet}(s.\text{capacity}) \) and \) s.\text{capacity}>0 \) and \) f.\text{capacity}>100 \}
\end{align*}
\]

which means that the service processes known and nonnegative amounts of sugar and fruits, deals with the portions of fruits exceeding 100 units, and declares to produce an nonnegative amount of juice.

The validation formulas of the transport service $TS$ can be as follows:

\[
\begin{align*}
TS.\text{preCondition} & = \{ \text{isSet}(w.\text{location}) \) and \text{isSet}(w.\text{weight}) \) and \) w.\text{weight}>0 \) and \) w.\text{weight} \leq 10000 \}
\end{align*}
\]

which means that $TS$ can transport wares of a known weight not greater 10000 units, and transport them from a known location to another (different) location.

Finally, we show an example of a validation formula and its transformation by the abstract planner:
Example 5  The formula \(\text{preCondition}\)

\[(\text{isSet}(x.\text{id}) \text{ and } x.\text{number}>0) \text{ or } (\text{isSet}(x.\text{id}) \text{ and } x.\text{capacity}>0)\]

will be transformed in the abstract planner into

\[(\text{isSet}(x.\text{id}) \text{ and } \text{isSet}(x.\text{number})) \text{ or } (\text{isSet}(x.\text{id}) \text{ and } \text{isSet}(x.\text{capacity}))\].

The next group of examples presents the relations between the attributes of concrete services and the attributes of their metaservice, as well as the aspects of inheritance. The first of these examples shows the relation between processing list of a metaservice and these of a concrete service.

Example 6  If the metaservice of a class SelectWare produces an object of a class Ware, then a concrete service (e.g. electroMartSelect) of the class SelectWare can be described as producing an object of the class Ware (which is the default) or as producing an object of its subclass (which requires an appropriate contents of produces). Both versions, together with metaservice specifications, are presented below.

\[
\begin{align*}
_\text{class.SelectWare}.\text{produces} & = \{ w:\text{Ware} \} \\
\text{electroMartSelect}.\text{produces} & = \{ \} \\
\text{electroMartSelect}.\text{produces} & = \{ w:\text{HouseholdWare} \}
\end{align*}
\]

The next example illustrates the relation between the modification lists of a concrete service and its metaservice.

Example 7  If the metaservice of the class SelectWare sets the attribute \(x.\text{name}\), where \(x\) is an object of the class Ware produced by the metaservice, then the service electroMartSelect of this class can narrow the type of the objects produced to a subclass of Ware called HouseholdWare, and can additionally set the attribute \(x.\text{installed}\) introduced in the class HouseholdWare (and therefore not present in Ware).

\[
\begin{align*}
_\text{class.SelectWare}.\text{produces} & = \{ x:\text{Ware} \} \\
_\text{class.SelectWare}.\text{mustSet} & = \{ x.\text{name} \} \\
\text{electroMartSelect}.\text{produces} & = \{ x:\text{HouseholdWare} \} \\
\text{electroMartSelect}.\text{mustSet} & = \{ x.\text{installed} \}
\end{align*}
\]

In the above, the service electroMartSelect declares explicitly that it sets the attribute installed, and implicitly that it sets the attribute name (this follows from the definition of the metaservice).

The next example illustrates relations between validation formulas.

Example 8  Let \(\text{postCondition}\) of the metaservice SellingService be of the form \(x.\text{id}>0\), and \(\text{postCondition}\) of the metaservice of the derived class WashingMachineSelling be of the form \(\text{isSet}(x.\text{capacity})\). An instance RacoonShop of WashingMachineSelling (a concrete service) declares explicitly in its \(\text{postCondition}\) that \(x.\text{brand} = \text{"Racoon"}\). The above means that in the world resulting from invoking this service the conjunction of these two formulas, i.e. \(\text{isSet}(x.\text{capacity})\) and \(x.\text{id}>0\) and \(x.\text{brand} = \text{"Racoon"}\), will be satisfied.
Next, we provide some examples of queries:

**Example 9 User’s query:**

- **Initial domain:** empty set
  
  **Initial clause:** true,

- **Effect domain:** w:WashingMachine,
  
  **Effect clause:** w.id>0 and w.owner="Me" and (w.name="Raccoon 1" or w.capacity>5),

- **Execution condition:** sum(s.price, s:ChargeableService) <= 1000,

- **Quality function:**
  
  \[
  w.\text{capacity} \times 10 - \frac{w.\text{sum}(s.\text{price}, s:\text{ChargeableService})}{100}.
  \]

On the abstract level the above query means that we request creating in the final world an object of the class **WashingMachine**, of nonempty values of attributes: either **id**, **owner** and **name**, or **id**, **owner** and **capacity**.

The next example shows an initial world generated from a query, and an “obligatory subset” of a target world.

**Example 10 Assume a user’s query is:**

- **Initial domain:** f:Fruits
  
- **Initial clause:** f.capacity=100 or (not isSet(f.capacity) and f.number=100 and f.weight=1)

- **Effect domain:** j:Juice

- **Effect clause:** j.capacity>0

From the above query we create the following “working query” for the abstract planner:

- **Initial domain:** f:Fruits

- **Initial clause:** isSet(f.capacity) or (not isSet(f.capacity) and isSet(f.number) and isSet(f.weight))

- **Effect domain:** j:Juice

- **Effect clause:** isSet(j.capacity)

which generates two alternative initial worlds—both consisting of a single object of the class **Fruit**, but of different states: the first one with a nonempty value of the attribute **capacity**, the second one with **capacity** set to null, but of nonempty values of **number** and **weight**. Moreover, the query generates a single “obligatory subset” of the final world, consisting of an object of the class **Juice** with nonempty value of **capacity**.

**B The Example of Getting Juice**

In order to illustrate our approach we present a simple ontology, in which abstract planning for our example can be done. For compactness of the description, the notation of triples (similarly as in the language RDF) is used.
B.1 Model for Objects of Services

The classes whose objects are processed by services are defined as follows:

- **Ware**:
  - `id` integer
  - `name` string
  - `owner` string
  - `Measurable` capacity float

- **Juice** extends Ware, Measurable
- **Fruits** extends Ware, Measurable

The class **Ware** models a thing which can be sold. The class contains the attributes:
- `id` carrying the information of “being concrete”; setting it to a nonempty value means that a concrete instance or “portion” of a given thing has been selected for further processing,
- `name` which describes what we are working on (but without referring to any concrete instance/portion in the real world),
- `owner` which specifies an owner of the thing,
- `Measurable` which, using Java terminology, is an interface extending the properties of a class by possibility of being measured (expressed for simplicity by one attribute capacity).

**Juice** and **Fruits** are child classes of both the classes mentioned above.

B.2 Services

A description of services is:

- **SelectWare** produces `w:Ware`
- **SelectWare** consumes null
- **SelectWare** requires null
- **SelectWare** mustSet `w.name`; `w.owner`

- **Selling** produces null
- **Selling** consumes null
- **Selling** requires `w:Ware`
- **Selling** mustSet `w.id`; `w.owner`
- **Selling** preCondition `not isSet(w.id) and isSet(w.name)`
- **Selling** postCondition `w.owner!=pre(w).owner`

- **FruitSelling** extends Selling
  - **FruitSelling** requires `w:Fruits`
  - **FruitSelling** mustSet `w.capacity`
  - **FruitSelling** postCondition `w.capacity>0`

- **JuiceSelling** extends Selling
  - **JuiceSelling** requires `w:Juice`
  - **JuiceSelling** mustSet `w.capacity`
  - **JuiceSelling** postCondition `w.capacity>0`

- **MakingJuice** produces `j:Juice`
- **MakingJuice** consumes `f:Fruits`
MakingJuice  mustSet  j.id; j.name; j.capacity
MakingJuice  preCondition  isSet(f.id) and isSet(f.name) and
                    f.capacity>0
MakingJuice  postCondition  isSet(j.id) and isSet(j.name) and
                    j.capacity>0

Below, the word “service” is used as denoting a class of services or a metaservice.

- **SelectWare** is a “searching service”, typical for e-market, which, given a query, tries
to select from the offers of services (not only selling ones) these which correspond to
the thing(s) requested. The service creates one or many objects, but does not set
their ids, leaving this to services which precise the portion or item. It sets, however,
the attribute owner, initialising it with the place a ware of interest was found (usually
a shop).

- **Selling** is a selling service, operating on an existing object of the class Ware, which
is not precisely pointed to in the seller’s offer (nonempty value of owner, but null id)
but is of a precisely specified kind (nonempty value of name). The service declares
setting the attribute id and changing the owner. The new owner is derived from the
query.

- **FruitSelling** is a child service of Selling. It narrows the type of the objects pro-
cessed to an instance of the class Fruits. Comparing with Selling, it declares addi-
tionally setting the attribute capacity and a positive amount (capacity) of the ware
sold. The value is derived from the query. JuiceSelling is similar to FruitSelling,
besides the fact that the type of the objects processed is narrowed to an instance of
the class Juice.

- **MakingJuice** is a service which makes juice from fruits: it removes an object of the
class Fruits from the world, placing there an object of the class Juice instead. The
portion of fruits must be “concrete” (of a nonempty value of id, we have to know
also what kind and how much fruits we have, so a nonempty value of name and a
positive value of capacity are required). The juice produced is also “concrete”, i.e.,
of the list of nonempty attributes analogous to that for the input.

**B.3 Query**

A query is specified as follows:

InitWorld   null
InitClause  true
EffectWorld  j:Juice
EffectClause  j.id>0 and j.capacity=10 and j.owner="Me"

which should be understood as “create a world which contains a concrete (id>0) instance
of juice of a given volume whose owner is the requester”. The results generated by our
abstract planner for the above set of data are presented in Sec. 5.