Mariusz Jarocki, Artur Niewiadomski, Wojciech Penczek, Agata Półrola, Maciej Szreter

Towards Automatic Composition of Web Services: Abstract Planning Phase

Nr 1017

Warsaw, February 2010
Abstract

The paper proposes a method of converting the problem of automated composition of web services to the problem of building a graph of worlds consisting of formally defined objects. We present basic rules of defining ontologies for services execution environments, in which automatic reasoning about sequences of service calls leading to satisfying a user’s intention is possible. The intention can be specified in a fully declarative language, without knowledge about services. In turn, the services are treated as independent “black boxes”, realizing their activities regardless of the flow in which they attend. The intentions and the descriptions of the services are the only source of knowledge used to generate abstract plans. The above planning process is the first phase of automated composition. The paper presents also a tool implementing our composition algorithm and some experimental results.

Keywords: webservices, automatic composition, abstract planning
Streszczenie

Automatyczna kompozycja usług sieciowych - faza planowania abstrakcyjnego

Praca zawiera propozycję konwersji zagadnienia automatycznej kompozycji usług sieciowych do problemu budowy grafu światów zbudowanych z formalnie zdefiniowanych obiektów. Przedstawiono podstawy budowy ontologii środowiska wykonania usług, w której możliwe jest automatyczne wnioskowanie przebiegu wywołań, prowadzących do realizacji intencji użytkownika. Język wyrażania takiej intencji jest całkowicie deklaratywny i wraz z ontologią, w której usługi pozostają nie związanymi ze sobą “czarnymi skrzynkami”, stanowi jedynie źródło wiedzy dla budowy abstrakcyjnych planów wykonania. Opisany tutaj proces stanowi pierwszą z trzech faz automatycznej kompozycji, będącej tematem badań zespołu. Praca prezentuje również narzędzie implementujące algorytm kompozycji oraz przykład jego zastosowania.

Słowa kluczowe: usługi sieciowe, automatyczna kompozycja, planowanie abstrakcyjne
1 Introduction

In recent years there is a growing interest in automating the composition of web services. This can be explained by moving from experimental to real-world scale of environments, with significantly bigger numbers of services, becoming more and more complex. One of the problems hindering the development of automatic composition approaches is scalability. Since most existing composition methods work with concrete instances of web services, even a simple query requires taking all the instances of all the types of services into account. In our work, strongly influenced by the idea of Entish [1], one of the fundamental ideas behind that project is followed, i.e., we distinguish between the abstract and the concrete phase of the composition. Moreover, our approach uses no other semantic information than that contained in the ontology. Therefore, the services are seen as black boxes, without taking their internal structures into account. This is aimed at making registration of services much easier and feasible from the practical point of view.

1.1 Motivations

There are many Internet services nowadays. Several standards describe how services can be invoked (WSDL [15]), how they exchange information (SOAP [11]), how they synchronise the executions in complex flows (BPEL [14]), and finally how they can be found (UDDI [13]). However, still there is a lack of automatic methods for arranging and executing flows of services, which would use reasoning methods similar to human ones. The main problems are incompatibilities of inputs/outputs of services, and difficulties in comparing their capabilities and qualities. Two services can offer the same functionality, but this fact cannot be detected automatically without unification of their interfaces, done by the service providers.

Our automatic composition system is based on the following concepts:

- Web services provide their functionalities using stubs, generated from WSDL (or other) descriptions for both automatic composition systems and other systems used so far. We do not require a different interface for each of these forms of using services. However, we need a uniform semantic description of service types in order to build adapters (called
proxies) translating specific interfaces of concrete services to the common one. The services handled this way can be used in automatic composition systems. Service registration, aimed to be easy to comprehend by a provider, involves building an appropriate proxy for the service which is being registered. In the future we are going to have the registration process based on interface descriptions both in WSDL and in the languages which contain a semantic information, like OWL-S or Entish. This allows us to adapt a possibly wide class of existing web services.

- A complex client’s goal is expressed in a fully declarative intention language. A flow of services which should be invoked to reach the goal (i.e., a plan) is built automatically, using no other information than these of the client’s query and present in the ontology. The user does not need to know a priori any relations between services. He describes two worlds: the initial and the final one; the goal of the system consists in finding a way of transforming the former into the latter. The user specifies his intention using notions from the ontology, but the relations between these notions and the services are unknown to him.

- Applicability of systems for automatic composition can be seen in environments containing many services which offer the same functionalities, but are of different quality or of other features justifying searching for an optimal solution. To make such a choice, the services should be used in an “offer mode” (not involving irreversible changes of their internal states), and in an “execution mode”. This can be reached using the mechanism of proxies.

- Our automating composition consists of three phases. 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 allows to accomplish the goal (expressed in the query). The second phase makes these scenarios “concrete”, which consists in replacing the types of services by their concrete instances. The last phase consists in supervising the execution of the optimal run, with a possibility of correcting it if necessary (i.e., in the case of a service failure). The intermediate results, resulting from the above phases, are to be presented in formats
which enable using them in other tools. The tool presented in the paper can export its results to the BPEL format.

Our further plans of developing the approach involve applying the well-known methods of model checking and theorem proving. We are going to use them mainly (but not only) for building mechanisms of automatic reasoning. To this aim, we change the model of services from functional (meeting the remote procedure call paradigm) to an automata-like, which is more convenient in our research.

The rest of the paper is organised as follows: Sec. 2 presents the related work. In Sec. 3 we sketch the main ideas of our approach. The notions behind them are introduced in Sec. 4 and illustrated by an example in Sec. 5. Sec. 6 shows an algorithm for abstract planning and defines the graph being its result. Sec. 7 presents an implementation of the abstract planner, illustrated by the results for the example of Sec. 5. Sec. 8 contains final remarks and sketches directions of our future work.

2 Related Work

There is a significant number of papers dealing with the topic of web services composition [3, 6, 8, 7, 9, 12]. Some of these works consider static approaches where flows are given as a part of the input, while other deal with dynamically created flows. One of the most active research areas is a group of methods referred to as AI Planning [3]. Several approaches use Planning Domain Definition Language (PDDL [4]). Another group of methods is built around the so-called rule-based planning. Composite services are generated from high-level declarative descriptions. Compositionality rules describe the conditions under which two services are composable. The information obtained is then processed by some designated tools. The composition is described in the Composite Service Specification Language (CSSL). The project SWORD [5] uses an entity-relation formalism to specify web services. The services have pre- and postconditions; a service is represented as a Horn rule denoting that the postconditions are achieved when the preconditions are true. A rule-based expert system generates a plan. Another methodology is the logic-based program synthesis [8]. Definitions of web services and user requirements, specified in
DAML-S [2], are translated to formulas of Linear Logic (a propositional logic; LL in short) so that the descriptions of web services are encoded as LL axioms, while a requirement is specified as a LL sequent to be proven by the axioms. Then, a theorem prover determines if such a proof exists. If so, a process model is extracted from it, and a composite service is described either in DAML-S Service Model or in BPEL4WS. While the approaches described above are automatic, there are also semi-automatic methods assuming human assistance at certain stages [10]. Some approaches are based on specifying a general plan of composition manually; the plan is then refined and updated in an automatic way.

Although most of the approaches presented above are oriented to practice, one can conclude that the presented tools have not been developed to become established frameworks and setting standards. Most of them are not even publicly available. Moreover, it seems that after the significant activity in the beginning of current decade in its second part the development slowed down. Some papers are developed for relatively small and closed domains, and present a preliminary work without delivering a complete solution.

### 3 Main Ideas

Most problems with automatic composition occur when trying to compose services coming from different providers, as there is no unified semantics for functionalities which are offered. This problem can be solved by a unified dictionary of notions/types describing inputs and outputs of services (similarly as in the Entish project [1]). The problem of service composition is then transformed into a problem of searching for compositions of functions able to create from a given set of data (documents in the Entish approach) a set of data (documents), which is requested by the user.

In order to deal with more complicated examples from the real world, one can create a model which allows to describe not only document processing, but also an arbitrary process causing changes in a uniformly described data space. Such a data space will be called a *world*. The worlds can be described by the use of an *ontology*, i.e., a formal representation of knowledge about them. In order to ensure an easy integration with other solutions developed, the ontologies used in our project are defined in the OWL language. The concepts used
to model individuals are a hierarchy of types as well as classes and objects.

In our approach, services are understood as transitions between worlds, or functions which transform these worlds. 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 state to a given final state. Descriptions of both the worlds in the specification of a task do not need to be complete.

As in most cases there are many paths which satisfy user’s request, there is a need of a technique which allows to evaluate them in an automatic way in order to choose a most optimal one. The evaluation is based on features of services which take part in modifying worlds.

The ontologies used in the approach collect the knowledge not only about the structure of worlds, but also about the ways the world 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 a declaration what all the services of a given class do - in the project such a pattern of behaviour is referred to as abstract service), and on the level of objects (a concrete service in the project). The description of services involves, besides specifying input and output data types, also declarations of introducing certain changes to a world, i.e., creating, removing and modifying objects.

A user describes its goal in a unified language defined by the ontology. The knowledge contained in the ontology is sufficient to automatic composition of services which can satisfy the user’s request. In Sec. 7 we show that our approach enables to model automated composition based on matching input and output types of services, characteristic for the Entish project [1].

4 Basic Notions

We start with introducing the basic notions used in the paper.

4.1 Classes and Inheritance

In order to model all the elements of our approach OWL ontologies are applied.

**Definition 1 (OWL ontology)** An ontology is a set of definition of classes (ordered in an inheritance hierarchy), their instances and relations between
Definition 2 (Class) A class is a named OWL template, which defines names and types of attributes.

4.1.1 Inheritance.

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. The class \(B\) is called subclass, child class or derived class; the class \(A\) is called superclass, parent class or base class. Contrary to object-oriented programming languages, names of attributes are unique in the whole name space (instead in the class definition only). Attribute encapsulation and overloading is not supported\(^2\).

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 in this way do not correspond to any real-world entity. Abstract classes are used, among others, for defining formal parameters of services, which allows an actual value of such a parameter to be any object instantiating an arbitrary non-abstract subclass of the given class. Moreover, on the top of the inheritance hierarchy there is an abstract base class Thing of no attributes\(^3\).

4.1.2 Using subclasses instead of instances of classes.

If using an object of a certain class is required in the planning process, then an object of an arbitrary subclass of this class can be used instead. This follows from the fact that the set of the attributes of this object is not smaller that the one of the class required.

4.2 Worlds

Assume we have a set of objects \(O\) containing instances of classes defined in an OWL ontology.

---

\(^1\) OWL ontology definition

\(^2\) The last two restrictions follow from assuming OWL as the ontology description language.

\(^3\) The rules of class inheritance, no formal definition of abstract class and the common root of the inheritance tree are also taken from OWL.
Definition 3 (World and objects) The universum is the set of all the objects having 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\(^4\).

- 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 initializing 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,
- **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 \(\text{ObjectName.AttributeName}\).

\(^4\)Such an approach is analogous as in C++, and different than in Java.
Definition 4 (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 $(\text{AttributeName}, \text{AttributeValue})$, where $\text{AttributeName}$ varies over all the attributes of $O$).

- A state of a world is a set of states of all its objects.

The worlds are modified by services. A formal definition of a service will be given after describing the methods of reasoning about worlds.

There are two ways of adding a new object to the world:

1. The object, with some of its attributes set to the values given, is created in the initial world according to the initial clause (see Sec 4.4),

2. Adding the object results from invoking a service (see Sec. 4.3).

An object can be removed from a world only by means of invoking a service.

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):

- $\text{Exists}$ - a function whose first parameter is an object, and which says whether the object exists in the world,

- $\text{isSet}$ - a function whose first parameter is an attribute of an object, and which says whether the attribute is set (has a nonempty value),

- $\text{isConst}$ - 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 $\text{const}$ flag; when called for an object it returns the conjunction of the $\text{const}$ flags of all the attributes of this object.

Some aspects of using the above functions should be explained. $\text{Exists}$ can be used in the effect clause (an element of a query, see Def. 11) to express that an object which appears in the user’s query as an element of the initial world should not be present in the final world. The function $\text{isSet}$ aims at testing initialization of attributes.
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.

4.3 Services

The definition of a service is as follows:

Definition 5 (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),
- validation formulas (the attributes preCondition and postCondition).

The grammar and usage of these attributes are presented in Def. 6-8.

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 (services which create new objects are not taken into account):

- 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 6 (Processing lists) We have the following processing lists:
• produces - a list of named objects of classes whose instances are created by the service in the world resulting from its execution,

• consumes - a list of names of classes whose objects are taken from the world when a service is invoked, and do not exist in the world resulting from the service execution (the service removes them from the world),

• requires - a list of names of classes whose instances are required to exist in the current world to invoke the service and are still present in the world resulting from its execution.

The elements of each of these lists are of the following form

<formal parameters> ::=  
  <formal parameter>::<parameter class> |  
  <formal parameter>::<parameter class>;  
  <formal parameters>

The structure of the lists is similar to the lists of the formal parameters of the procedures. Names of objects are unique in a description of a class of services / an object of a service class. The formal parameters from these lists define an alphabet for modification lists and validation formulas.

**Definition 7 (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 non-empty 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.
Each of the above four lists is generated by the following grammar:

\[
\begin{align*}
\text{<attribute spec> ::= } & \langle\text{formal parameter} \rangle.\langle\text{attribute name} \rangle \mid \\
& \langle\text{formal parameter} \rangle.\ast \\
\text{<attribute list> ::= } & \langle\text{attribute spec} \rangle \mid \\
& \langle\text{attribute spec} \rangle; \langle\text{attribute list} \rangle
\end{align*}
\]

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

The expression \text{objectName}.* used in the above lists is understood as “all the attributes of the object \text{objectName}”. It is the only way of referring to the attributes which cannot be listed directly in a given moment, due to the fact that the processed object can be of a subclass of the class specified in the list (so have a bigger set of attributes).

The attributes of the objects appearing in processing lists which do not belong to the union of lists \text{mustSet} and \text{maySet}, are not changed when the service is called.

Definition 8 (Validation formulas) The validation formulas are as follows:

- \text{preCondition} is 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 \text{consumes} and \text{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 functions calls \text{isSet}, \text{isConst} and \text{Exists}.

5Using \text{Exists} in \text{preCondition} is redundant w.r.t. using an appropriate object in the list \text{consumes} or \text{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.

14
- **postCondition** is 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, currently (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, can be transformed either trivially, i.e., by replacing arguments of functions by the conjunction of calls of isSet over attributes (and not Exists over objects in future versions), or in a more sophisticated way, which allows to reason about additional consequences in the resulting plan. We are going to deal with the above problems in the future work.

The following grammar builds a proper validation formula for the abstract planner:

```
<objectName> ::= <objectName from consumes> | pre(<objectName from requires>) | post(<objectName from requires>) | <objectName from produces>
<objectAttribute> ::= <objectName>.<attributeName>
<atomicPredicate> ::= Exists(<objectName>) | isSet(<objectAttribute>) | isConst(<objectAttribute>) | not <atomicPredicate>
<conjunction> ::= ...
```
It should be noticed that the grammar allows using post also in pre-
Condition. However, evaluation of such predicates gives always the value false.

4.3.1 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 (i.e., an object of the service which describes the whole class of
services). The (unique) name of a metaservice is derived from the name of the
class of services, i.e., is of the form _class_ServiceClass.

A description of a concrete service should be understood as a list of differ-
ences or as narrowing the template introduced as the metaservice. More pre-
cisely, 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 ap-
propriate list, a formal parameter of the same name and of a restricted domain.
Concidering modification lists, a concrete service can extend the lists of its
metaservice only by declaring that it modifies attributes added by a narrowed
class of parameters. The definition of a concrete service cannot be inconsistent
with the definition of the metaservice.

Comparing with a base class of services, its child class can extend the sets
of objects which are consumed, produced and required, 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. As the processing lists
define the alphabet for the modification lists and the validation formulas, one
should ensure the uniqueness of the formal parameters on all the levels of
inheritance. 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.

Considering validation formulas, each formula preCondition (post-
Condition, respectively) of a concrete service is a conjunction of the precon-
dition (postcondition resp.) of the metaservice and explicitly given precondition (resp. postcondition) of the concrete service. The case of inheritance
is similar - in the derived class, a condition 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 at-
tributes added to modification lists in narrowed types introduced by concrete
services. Potential inconsistencies, resulting from concatenation of process-
ing and modification lists in child classes of services, are treated as ontology
errors.

4.3.2 Examples.

In order to explain some less intuitive aspects of the definitions, we provide
several examples. As the paper deals with abstract planning only (not concrete
planning), the (concrete) validation formulas in the examples are specified us-
ing a language which is not introduced in the paper, but is similar to the lan-
guage 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
$x>4$).

The first example shows the relation between processing list of a metaser-
service and these of a concrete service.

**Example 1** If the metaservice of a class SelectWare produces an object of
a class Ware, then a concrete service (e.g. electroMartSelect) of the

---

6We 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. 19), and on the third level classes of services with definitions
of their metaservices.
*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.*

\[
_class_{SelectWare}.produces = \{ w:Ware \}
\]

\[
electroMartSelect.produces = \{ \}
\]

\[
electroMartSelect.produces = \{ w:HouseholdWare \}
\]

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

**Example 2** *If the metaservice of the class* SelectWare *sets the attribute* \( x \).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* \( x \).installed introduced *in the class HouseholdWare (and therefore not present in Ware).*

\[
_class_{SelectWare}.produces = \{ x:Ware \}
\]

\[
_class_{SelectWare}.mustSet = \{ x.name \}
\]

\[
electroMartSelect.produces = \{ x:HouseholdWare \}
\]

\[
electroMartSelect.mustSet = \{ x.installed \}
\]

*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).*

Next we show some examples of atomic predicates and validation formulas:

**Example 3**

- **Atomic predicates:**
  \[ x.number>0, \text{ not } isSet(x.id) \]

- **preCondition:**
  \[ x.number>0 \text{ and } isSet(x.id) \]
4 Basic Notions

- **postCondition:**
  
  not x.number = pre(x).number

The next example illustrates relations between validation formulas.

**Example 4** Let **postCondition** of the metaservice SellingService be of the form \( x.\text{id}>0 \), and **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 **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.

Finally, we show an example of a validation formula and its transformation by the abstract planner:

**Example 5** The formula **preCondition**

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

will be transformed in the abstract planner into

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

4.3.3 Additional attributes of a service.

Service classe can contain additional *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. 4.4). Such attributes can be introduced, among others\(^7\), 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 process.

\(^7\)The decision whether the additional quality attributes are introduced by separate abstract “second-level” classes, or are defined on the level of non-abstract classes of services (with concrete representatives) is left to an ontology designer. The suggestions of the authors are presented in Footnote 6.
4.3.4 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 it to the current form. The result of executing a service is a (possibly new) world with a new state.

Definition 9 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 pre-Condition 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.

Definition 10 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 consumes,

- there is a one-to-one mapping between all the other objects in \( S' \) and the objects in the list produces of \( U \), such that each object \( O \) from the list 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 postCondition holds,

- in the objects which are actual values of the appropriate parameters the flags const of the attributes listed in mustSetConst of \( U \) are set, and the attributes listed in 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.

4.4 Queries

The task of the composition system is to provide sequences of service calls (runs) which can transform a world specified by a user into a desired world. Services which can be used for building the above sequences are listed in the repository of classes and objects. From the obtained sequences the system should choose those satisfying execution conditions specified by user, and then select those optimal from the user’s point of view (i.e., considering the criteria he gave).

User defines a query by describing an initial state of the world as well as a final state required.

Definition 11 (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 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 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 the abstract planning phase as well as in the first phase of concrete planning, which adjusts types and analyses pre- and postconditions of concrete services. Due to the fact that in this work we deal with abstract planning only, the language of quantification and aggregation used in execution conditions is in an early stage of development. Its complete formalization will be a subject of our future work.

On the abstract level, the initial clause and the effect clause are specified as DNF formulas over the predicates $\text{isSet}$ and $\text{Exists}$. 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.

Examples of queries are presented below.

**Example 6** 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**:
  \[ \text{sum(s.price, s:ChargeableService)} \leq 1000, \]

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

*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 7** Assume a user’s query is:

- **Initial domain**: f:Fruits

- **Initial clause**:
  \[ f.\text{capacity}=100 \text{ or } \]
  \[ \text{not isSet(f.capacity) and f.number=100 and f.weight=1} \]

- **Effect domain**: j:Juice
• **Effect clause:** \( j.\text{capacity} > 0 \)

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

• **Initial domain:** \( f: \text{Fruits} \)

• **Initial clause:**
  
  \[
  \text{isSet}(f.\text{capacity}) \text{ or } \begin{cases} 
  \text{not isSet}(f.\text{capacity}) \text{ and isSet}(f.\text{number}) \\
  \text{and isSet}(f.\text{weight})
  \end{cases}
  \]

• **Effect domain:** \( j: \text{Juice} \)

• **Effect clause:** \( \text{isSet}(j.\text{capacity}) \)

*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.*

**Definition 12 (Highlighted objects)** An object is called highlighted w.r.t. a user’s query if its name occurs both in the initial domain and in the effect domain of this query.

**Definition 13 (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 \( o_1 \) from one set there is exactly one corresponding object \( o_2 \) from the second set, such that \( o_1 \) and \( o_2 \) are objects of the same class, and the values of all the attributes in both the objects are the same).

## 5 A Complete Example

In order to illustrate our approach we first 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.
5.1 Model for Objects of Services

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

<table>
<thead>
<tr>
<th>Class</th>
<th>Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ware</td>
<td>id: integer</td>
</tr>
<tr>
<td></td>
<td>name: string</td>
</tr>
<tr>
<td></td>
<td>owner: string</td>
</tr>
<tr>
<td>Measurable</td>
<td>capacity: float</td>
</tr>
<tr>
<td>Juice</td>
<td>extends Ware, Measurable</td>
</tr>
<tr>
<td>Fruits</td>
<td>extends Ware, Measurable</td>
</tr>
</tbody>
</table>

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.

5.2 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, initializing 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 processed to an instance of the class Fruits. Comparing with
Selling, it declares additionally 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.

5.3 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 will be shown in Sec. 7.1.

6 Abstract Planning

The result of the abstract planning phase is an abstract graph. 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
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;

end if

if $s$ satisfies $\varphi_E$ then

mark $s$ as final, mark $s$ as processed

continue;

end if

if depth of $s$ greater than $k$ then

mark $s$ as processed

continue;

end if

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 $S$'s $\text{consumes, requires}$ then

continue;

end if

if $s$ does not satisfy $S$'s $\text{preCondition}$ then

continue;

end if

generate all the subsets of the objects in $s$ satisfying $S$'s $\text{preCondition}$ and such that the attributes listed in $S$'s $\text{mustSet}$ have the flag $\text{const}$ 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
nodes of the graph - these which have no input edges represent alternative ini-
tial worlds, while these with no output edges are alternative final worlds.

A formal definition of the abstract graph is as follows:

**Definition 14** 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.

### 6.1 A Composition Algorithm

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,
- users query specifying an initial world and a final world,
- 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 \((\varphi_E)\).
7 Implementation

The abstract planner is a tool which performs the first phase of web services composition, i.e. with abstract planning (planning in types). Our tool can be accessed via a graphical user interface, or from the level of Java via API.

Figure 1: A structure of the application

As it has been described before, a state of a world is given by a set of objects of classes defined in an ontology, of attributes set in an appropriate way. A composition of services consists in searching for sequences of services which can transform a world which (together with its state) is specified as an initial one, into a world satisfying conditions requested. In our implementation:

- ontologies are modelled using the OWL language and the Protege environment;

- OWL ensures a convenient way to 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 (modelling items to be processed by services);

- conditions on the input and on the output world of each service are specified in the ATLA\textsuperscript{8} 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{9} language.

The composition is a two-stage process. The first stage builds an abstract graph which describes possible solutions using types of services (and not their concrete instances). We distinguish two worlds (together with their states): a start world (at which the procedure starts) and an end world (a termination condition which means that the user’s request is satisfied). Although the algorithms operate in BFS mode, they 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 $W_B$).

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,

- Jena library (accessing OWL files generated by the Protege tool).

In order to provide nice presentation of the results, both abstract and concrete graphs (i.e., graphs with concrete instances of services, not described in this work) are optionally represented in BPEL. This enables using visualisation and processing tools designed for graphs described in that language. As the abstract graph models directly the elements represented in BPEL (service

\textsuperscript{8}ATtribute LAnguage

\textsuperscript{9}Query LAnguage
calls, concurrent executions of services), transforming it to the BPEL format involves only simple translations of its structures into XML descriptions of BPEL constructions.

The approach presented enables also an integration with the Entish project [1]. In the project mentioned, the composition of services is based on matching their input and output data types. A service is understood as a function which creates a set of objects (documents) of precisely specified types, starting from another set of documents. A service can be executed if it provided with objects of the appropriate types. The above objects can be obtained in the following ways:

- an object of the appropriate type “is in the user’s interface” (and was either placed there by executions of other services, or introduced there by hand),

- such an object was created by a service executed just before the current one.

If some objects required to execute a service are missing, then a user has to create them “by hand”.

The above composition model can be transformed to the one proposed in the paper in an easy way. The mechanisms of hierarchy of types, validation formulas and modification lists are not used, as there are no such mechanisms in Entish types space. The services read from Entish repository are added to our ontology, augmented with “our” processing lists (the input types of a service are used to build the list requires, the output types - the list produces, the list consumes is not used). The system of types of objects not being services is simplified (Entish inheritance is not taken into account). The problem of missing documents which are to be created by an interaction with the user can be solved by using services which produce these documents (of empty lists consumes and requires).

The transformation described above is implemented in the current version of our tool.
7 Implementation

Figure 2: The main screen of the tool. A diagram on the right shows the ontology, i.e., hierarchy of classes read from an OWL file for the example in Sec. 5.

7.1 Using the Abstract Planner

The current version of the application requires installing Java Runtime Environment (in a version at least 1.6). After running the planner, one should load a file containing an ontology, or to import an Entish repository (menu File). One can also use a built-in generator of examples (menu Examples). The ontology loaded is shown in both a graphical and text form in the bookmark ClassHierarchy (see Fig. 2). The bookmark Services enables to see the details of the descriptions and the internal representations of services (Fig. 3).

The main part of the tool is the bookmark Composition presented in Fig. 4. The panel in the left-hand side enables specifying the parts of a query which describe an initial and a final world, setting options required, and then running the composition process. The result of composition (a graph) is displayed in the window on the right. The to BPEL button enables exporting the graph to
the BPEL format. The tool generates one or more BPEL files describing the results, depending on the ontology loaded, on the query and on the options chosen.

Fig. 5 shows a part of the graphical interface of the tool with the results for the example of Sec. 5. The left-hand side of the screen contains a query describing the initial world (empty) and the final world. The right part presents a visualisation of the abstract graph generated by the tool using a backward-search algorithm. The nodes marked in blue correspond to the initial domain (these are empty worlds). The green node represents the final world, satisfying user’s request. Each path from an initial world to the final world represents a sequence of services whose execution results in an effect which satisfies the user’s request. The sequences obtained are:

- **SelectWare, then FruitSelling and then MakingJuice**,  
- **SelectWare and then JuiceSelling**,  
- **SelectWare, then Selling and then MakingJuice**,
Final Remarks

Figure 4: Specifying a query for the example in Sec. 5.

- 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.

8 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.

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 lan-
guage (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


Pracę zgłosił Józef Winkowski.

Adresy autorów: Wojciech Penczek, Maciej Szreter
Instytut Podstaw Informatyki PAN
ul. Ordona 21
01-237 Warszawa, Polska

Mariusz Jarocki, Agata Półroła
Wydział Matematyki i Informatyki UŁ
ul. Banacha 22
90-238 Łódź, Polska

Artur Niewiadomski
Instytut Informatyki, Akademia Podlaska
ul. Sienkiewicza 51
08-110 Siedlce, Polska

E-mail: penczek,mszreter@ipipan.waw.pl
jarocki,polrola@wmi.uni.lodz.pl
artur@iis.ap.siedlce.pl

Symbol klasyfikacji rzeczowej: II.C.2.4

Na prawach rękopisu
Printed as manuscript

Nakład 100 egzemplarzy. Papier kserograficzny klasy III. Oddano do druku w lutym 2010 r. Wydawnictwo IPI PAN. ISSN 0138-0648.