Control and Cybernetics

vol. **40** (2011) No. 2

Web Services composition – from ontology to plan by query[*][†]

by

Dariusz Doliwa[1], **Wojciech Horzelski**[1], **Mariusz Jarocki**[1], **Artur Niewiadomski**[2], **Wojciech Penczek**[2,3], **Agata Półrola**[1] **and Maciej Szreter**[3]

[1] Faculty of Mathematics and Computer Science, University of Łódź, Poland

[2] Institute of Computer Science, Siedlce University of Natural Sciences and Humanities, Poland

[3] Institute of Computer Science, Polish Academy of Sciences, Poland

Abstract: The paper proposes a method to cover the world of web services with a uniform semantics, possibly simple but enabling to arrange complex flows of service invocations. The flows are built according to fully declarative user's intentions, specified in a language common for the descriptions of services and for the query. In the approach we model the world of services and of the subjects they operate on using a uniform knowledge database and an object-oriented manner. The current work describes the first phase of the composition: making an abstract plan, i.e., giving an answer how (with what types of services) the required effect can be obtained. The problem of creating a plan is converted to building a specialized graph.

Keywords: automated composition, web services, abstract planning.

1. Introduction

A vision of independently developed parts of software communicating by well-defined network protocols is the conceptual base of the Service Oriented Architecture – SOA (Bell, 2008). From the technical point of view, knowing a standard for communication with a service is sufficient to create a component (*client*) able to use its functionality. The standards mentioned are well founded

[*]Supported by the Project "New information technologies for electronic economy of information society based on SOA paradigm", funded by the Polish Ministry of Science and Higher Education and supported by th European Union in the framework of the European Regional Development Fund program no. POIG.01.03.01-00-008/08.

[†]Submitted: October 2010; Accepted: June 2011.

and widely used. In recent years one can observe growing domination of their certain kind, namely of methods based on XML documents transferred over internet application protocols. With the exception of special applications, standards such as SOAP to encapsulate transferred data, WSDL to define a service interface, and UDDI to maintain indices and classifications of services are widely used in all domains of SOA applications. Additionally, we have languages to describe service flows, such as WS-BPEL.

While technical problems with communication between SOA components are easy to solve, creating methods for automatic aggregation of service functionalities seems to be a greater challenge. There are many tasks which require calling several services — the best examples to be found in e-commerce. Even if we know which types of services need to be called to reach our goal, we often cannot select the services which allow reaching it in an optimal way. One can formulate a fundamental question: what are the minimal conditions under which the services can cooperate with no manual intervention?

A plan how to reach a complex goal by simpler activities may be automatically created by investigating relations between their descriptions. The main problem is a common language to express what the activities do. Even if their functionalities are described using the same language (as WSDL, OWL-S or DAML-S), we do not have any guarantee of a semantic compliance. The common semantics can be made uniform by a centralization of knowledge. Reasoning in the knowledge can build our plan.

The plan which we obtain by the reasoning can be either *abstract* or *concrete* (i.e., *executable* in BPEL terminology). The abstract plan describes how the goal can be reached, without specifying who (i.e., which real web services) should be engaged with the plan. The abstract version of the plan can be built only if services are assigned to a specific type (a *class of services*). The plan can be further transformed to the concrete one by replacing service types by the corresponding service instances.

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 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.

We make endeavours to unify the interface between the "real" and the "virtual" worlds of services. However, reasoning in the "virtual" world can be deterministic and fully automatic. 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: Section 2 presents the related work. The main ideas of our approach and the notions behind them are introduced in Section 4. Section 5 shows an algorithm for abstract planning and defines the graph being its result. Section 6 presents an implementation of the abstract planner, illustrated by some experimental results in Section 7. Section 8 contains final remarks and directions of our future work.

2. Related work

There are many papers dealing with the topic of web services composition (Klusch, Gerber and Schmidt, 2005; Rao, 2004; Rao, Küngas and Matskin, 2004; Rao and Su, 2004; Redavid, Iannone and Payne, 2008; Srivastava and Koehler, 2003). Some of these works consider static approaches, where flows are given as a part of the input, while others deal with dynamically created flows. One of the most active research areas is a group of methods referred to as AI Planning (Klusch, Gerber and Schmidt, 2005). Several approaches use Planning Domain Definition Language (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 SWORD project (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, Küngas and Matskin, 2004). Definitions of web services and user requirements, specified in DAML-S report, 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, Hendler and Parsia, 2003). Some approaches are based on specifying a general plan of composition manually; the plan is then refined and updated in an automatic way.

The most complete specification of an automatic composition system was described in Ambroszkiewicz (2003). The author proposed a solution based on a multi-phased composition using a uniform semantic description of services. Our approach uses similar methods, adapting them to the existing SOA standards and making them easy to implement without revolutionary changes in the running web service infrastructure.

This article is an extended version of Jarocki et al. (2010a).

3. Our approach

Assume that all the web services in our domain of interest can be strictly classified in a certain hierarchy of types. Each type (a class of services) has a description in a common, unified language. The descriptions use a common base of concepts, consisting of types and their instances (i.e., objects[1]). The description of each type expresses declarations of changes, caused by any service of this type in some world. We define worlds to be sets of objects. Concrete services (i.e., instances of service types) have descriptions as well. They define the service activities in a more precise way.

The main idea of our project is to separate the phases of the planning. The first phase occurs in the space of types, while the second one - in the space of concrete services. The first phase results in an abstract plan, which becomes a concrete plan in the second phase.

The language describing services, both abstract (classes of services) and concrete (service instances), is uniform, fully declarative and possibly simple, in order to make an adaptation process possibly easy for any external service provider.

In Fig. 1 we present the architecture of our composition system currently developed. *Abstract Planner* using the knowledge from OWL ontology and from other knowledge bases (available by specialized interfaces) implements the concepts described in this article. The system is now able to create abstract flows (in abstract BPEL form) starting from a non-imperative user's query (expressed in our language QLa, presented in next sections). *Arranger* (the planner of concrete services) and *Registration Service* will be investigated and implemented in the nearest future.

4. 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

[1]In OWL, which can be used as a skeleton of the descriptions, the objects are called *individuals.*

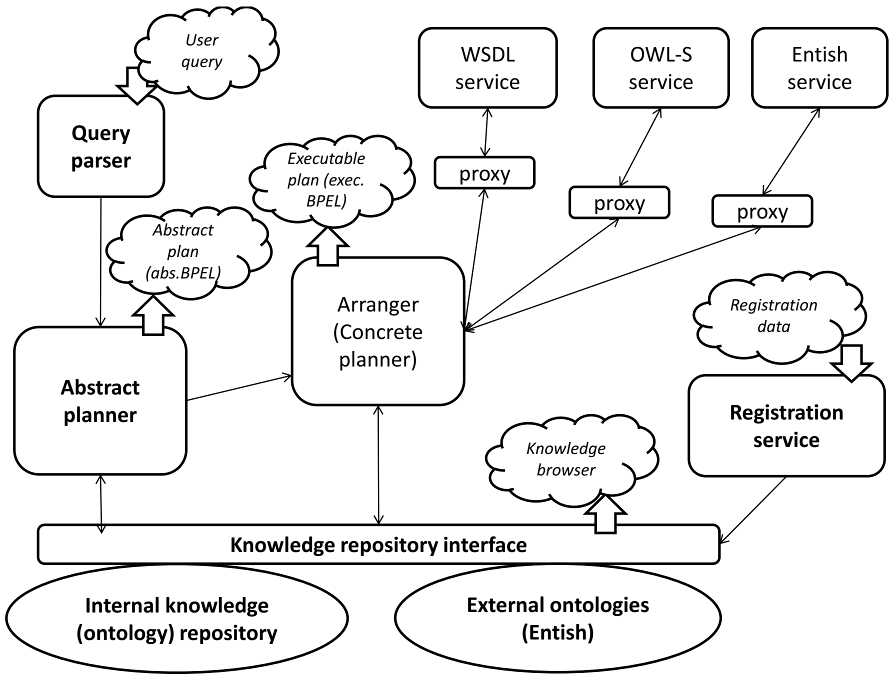


Figure 1. The architecture of the composition system

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 means 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[2]. 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 all the attributes of A, and the attributes specified in its own definition[3]. 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

[2]OWL ontology definition.

[3]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).

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.

A class is called *abstract* if instantiating it (i.e., creating objects of this class definition) is useless in the sense that the objects obtained this way do not correspond to any real-world entity[4]. 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` with no attributes[5].

4.1. Worlds

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

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

- *each object is either* concrete *or* abstract object*,*
- *each object contains named attributes whose values can be either other objects or:*
  - *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`*.*

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

The values of an attribute for an abstract object are interpreted 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`.

Example 1 *A user can define a required world consisting of concrete objects as a set which contains one instance of the class* `Book` *with the attribute* `title` *set to* "Introduction to Service-Oriented Modeling" *and* `owner` *set to* "John Smith" *(i.e., the name of the user). He can also specify that the attribute* `id` *should be set. This models that the user requires from the system to "create" a concrete item (which is ensured by setting* `id`*) - the book titled as above, which belongs to*

[4]There is no explicit possibility in OWL to express the fact that a class is not instantiable.

[5]The rules of class inheritance, no formal definition of abstract class and the common root of the inheritance tree are also taken from OWL.

the person specified (and not to any shop - setting `id` *guarantees that the item was sold).*

During the abstract planning phase, the object is replaced with its abstract incarnation - the attributes `id`, `title` *and* `owner` *have* `SET` *values, while the others are set to* `ANY` *(the user does not specify any requirements about them).*

Definition 2 (Object state and world state) *A* state of an object o *is a function V_o assigning values to all the attributes of* o *(i.e., it is the set of pairs*

$$(\texttt{AttributeName}, \texttt{AttributeValue}),$$

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):

- `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 it is known whether an attribute or an object was modified by the services used before, one can apply 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.

Example 2 *In our running example of book selling, the "intermediate" worlds, leading finally to a world which contains the book, are transformed by services. Each particular service can test a current world state using the above functions.*

4.2. Services

The ontologies collect the knowledge not only about the structure of the worlds, but also about the ways they can be transformed, i.e., about the 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 a 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` *and* `maySet`*), 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 its state. The world modified by a service is called its *pre-world* (*input world*), while the result of the modification 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: by a service which operates on a subset of its elements, or by many services which operate concurrently on disjoint subsets of its elements.

Example 3 *The* `Selling` *service can transform the world by setting the* `id` *attribute of an class instance of the class* `Ware` *(which corresponds to making concrete the item sold, or, in other words, to determining a unique identifier of the item sold) and modifies the previously set attribute* `owner` *(the selling is modelled as changing the owner of an item).*

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

- `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 (the precise grammar specification can be found in Jarocki et al., 2010b, the object example in Jarocki et al., 2010a). The formal parameters from the above lists define an alphabet for the modification lists and the validation formulas.

Example 4 *As already stated, the* `Selling` *service requires an instance of* `Ware`*. The service consumes and produces no other objects. Contrary to it, the* `SelectBook` *service, which models a book selection, can produce an object with* `id` *not set and the attributes* `title` *and* `owner` *set. So, the service answers a question which of the book providers offers the title of interest.*

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.*

The attributes of the objects appearing in the processing lists which do not belong to the list `mustSet` are changed when the service is called.

In the process of abstract planning, each attribute from the list `mustSet` has to be `SET` (the function `isSet` called for this attribute returns the value `true`).

Example 5 *The* `Selling` *service sets the attributes* `id` *and* `owner`*, so they are listed in the* `mustSet` *attribute of 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` *and* `Exists`*.*
- `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` 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. (2010b). Some examples illustrating the above definition are provided in the further part of this paper.

In order to be able to provide some additional information enabling comparison of the quality of services, the 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 Section 4.3). They can be introduced, among others[6], 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.

EXAMPLE 6 *In our running example of book selling, to disable activation of* `Selling` *on a* `Ware` *item already sold we can extend its* `preCondition` *by adding a conjunct specifying that* `id` *must be unset.*

4.2.1. Service types, inheritance, metaservices

Each class of services (service type) has certain features common to all the instances of this type. They are specified by the instance of the class called *metaservice* or *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 from being inconsistent with the definition of the metaservice. Considering the validation formulas, each formula `preCondition` (`postCondition`) of a concrete service is a conjunction of the precondition (postcondition respectively) of the metaservice and the explicitly given `preCondition` (`postCondition` respectively) of the concrete service.

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

[6]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 7.

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. The 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 have mentioned before, the attributes of the objects appearing in the processing lists which do not belong to the union of the lists `mustSet` and `maySet` are not changed when the service is called. This, however, does not apply to the attributes added to the modification lists in the 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[7].

4.2.2. Modifying a world

Separating calls of single services is one of the key concepts in our approach. A service transforms a world, irrespectively of how many services have been activated on this world so far. 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 in the following definitions:

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 the objects which are actual values of the parameters the flag* `const` *is set.*

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* `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*;*

[7]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*, and on the third level classes of services with definitions of their metaservices.

- *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* `mustSet` *nor in* `maySet` *have the same values as in the world* S*. Moreover, all the attributes listed in* `mustSet` *or* `maySet` *which are of nonempty values in* S*, in* S' *are of nonempty values as well.*

Fig. 2 presents an example of modifying a world by a service. The service requires one object of a particular type (A) with one attribute (x) set to a nonempty value. The service changes another attribute (y) of the object and additionally produces a new object of a different class (B) with a partially initialized state.

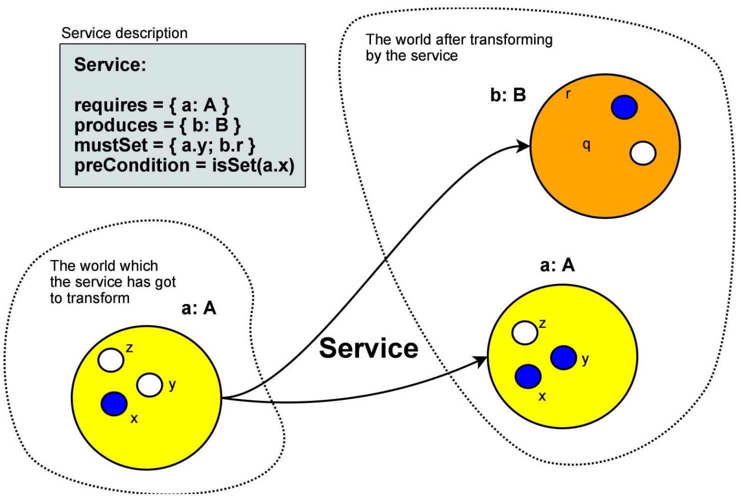

Service description
Service:
requires = { a: A }
produces = { b: B }
mustSet = { a.y; b.r }
preCondition = isSet(a.x)
The world after transforming by the service
b: B
r
q
a: A
z
y
x
The world which the service has got to transform
a: A
z
y
x
Service

Figure 2. An example of modifying a world by a service

4.3. 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, by means of a query. 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* `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 the final world (i.e., a subset the final world must contain);*
- *an* effect clause *specifying a condition which has to be satisfied by the final world. The clause is a propositional formula over names of objects and their attributes from both domains defined above; references to the initial state of an object, if ambiguous, are specified using the notations* `pre(objectName)` *and* `post(objectName)`*, analogously as in the language used in the formulas* `postCondition` *of services. The grammar of the effect clause is analogous to the grammar of the* `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 constructing 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 the 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 information as to 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.

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

Example 7 *A user can formulate a query by the following description: the initial domain is empty ("I have nothing"), the effect domain contains one object of the* `Book` *class ("I need a book"), satisfying a condition of setting of* `id` *and equipped with certain* `title` *and* `owner` *attributes ("I want to be an owner of the book titled as follows").*

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 the 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 objects are the same).*

4.4. Summarizing the concepts by examples

To make the above definitions more comprehensible, we describe below these concepts in a very short and intuitive form, followed by some simple examples.

The objects model things of the real world, aggregated by the concept of classes. Each object is equipped with a set of attributes, defined in the class the object belongs to. On the abstract level we do not consider exact values of the attributes, but only the fact that they are set, not set, or their value is meaningless or not defined. For example, the class `Ware` models the subset "sellable items" of all the things. It equippes its objects with the attributes `name`, `owner`, `location`, and the attribute `id` which enables to distinguish between an abstract and a particular instance of the class (a "concrete" ware has a concrete identifier, e.g. a serial number).

We use the inheritance mechanism in a classical manner. A derived class carries new attributes (comparing with its base class). For example, the class `Book` extends the class `Ware` by the attributes which describe its author and title. Objects of a derived class can be used in any context in which objects of its base class are expected to occur.

A world is a set of objects. A subset of a world (including the empty subset) can be transformed by a service. Invoking a service results in enriching the world by new objects (according to the contents of the list `produces` of the service), deleting the objects from the subset transformed (according to the contents of the list `consumes`) or changing their attributes (which follows from the contents of the list `requires`, `mustSet` and `maySet` of the service). The description of the service restricts its invocation ability to worlds whose objects satisfy conditions on attributes (specified by the `preCondition`). It also defines conditions which must be satisfied after invoking the service (`postCondition`). For example, any service of the `SelectWare` class produces a new object of the class `Ware` (in particular, a `Book`) and does not require any objects in the subset transformed. In contrast, any service of the class `Selling` works on a subset containing one object of the class `Ware`, which is transformed by setting the previously unset attribute `id` and resetting the attribute `owner`.

The inheritance among services enables to create some more precise classes than `SelectWare` mentioned above, for example a class `SelectBook`. The class differs from the previous one in such a way that we can be sure that it produces an object of the class `Book` (and not any other `Ware` item which is not a `Book`) with the attribute `title` set.

A user can express his goal by defining the initial world he "has" and the world which he requires, using the same mechanism which is utilized in the services' descriptions. For example, he can specify that he wants to transform the empty world into another, containing one object of the class `Book` which has the attributes `id`, `title` and `owner` set (more precisely: `owner` should be set to "him", and `title` - to a given title of his interest). This can model buying in the real world. We can see that the goal can be attained by the sequence of invokings: `SelectBook` and then `Selling`.

5. 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*.

The abstract graph is a directed multigraph. Its nodes 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:

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.consumes, S.requires then
        continue;
      end if
      if $s$ does not satisfy S.preCondition then
        continue;
      end if
      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
    end for
    mark $s$ as processed
  end while
```

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*
- *$L : E \longrightarrow U$ is a function labelling the edges with services.*

Algorithm 1 presents 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 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 one 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 (φ_S) and a part referring to a final world (φ_E).

6. Implementation

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 parser ANTLR of the AtLa and QLa languages (see below),
- Jena library (accessing OWL files generated by the Protege tool).

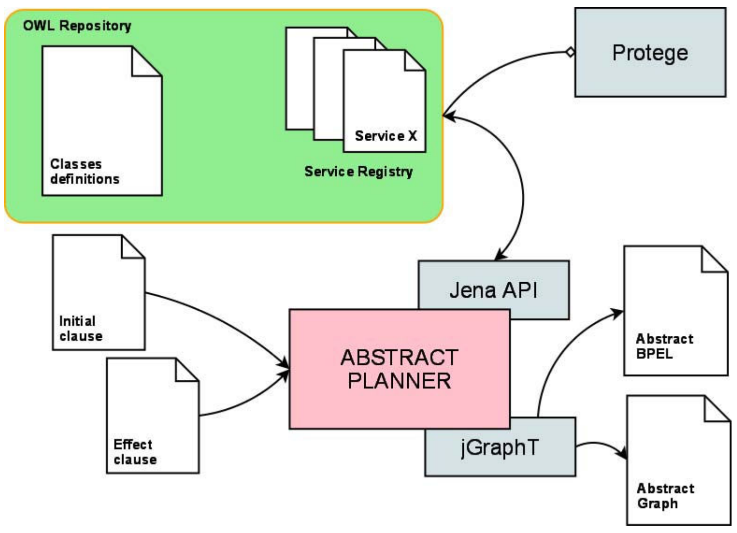


Figure 3. An architecture of the application

The architecture of the tool is shown in Fig. 3. In the implementation, the ontologies 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 (modelling items to be processed by services)). The conditions on the input and on the output world of each service are specified in the AtLa (ATtribute LAnguage) 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 Wb and a final world Wf, using the QLa (Query LAnguage) 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 Wb, and the termination condition is given by Wf) or as a backward search (the start state is then given by Wf, and the termination condition - by Wb).

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 BPEL.

7. Example

In Doliwa et al. (2010), where implementation of ideas from Penczek, Półrola and Zbrzezny (2010) is described, the authors present some experimental results, based on a test ontology created using the presented formalism. We provide one of these examples.

We model an environment of services related to publishing, including several aspects of preparing publications: in our ontology there are services providing contents, photos, typesetting, publishing in the Internet, printing etc., as well as objects like books, servers and web sites. Figs. 4 and 5 display the ontology used in the example in a graphical form. Because of the complexity we do not mention here a complete formal description of the dependencies between the classes, presenting only some selected relations between the objects and the services. All these relations are based on the same idea: some objects can be **produced** under the condition that we have provided the objects which are **required** by the producing service to its activation. The required objects must have specific features, which are limited to **set/unset** conditions on the abstract level of planning. For example, if we require a web site with photos, an object of the `Photo` class must occur at the path of modified worlds; it can be created by another service (e.g. `PhotoService`) or provided by the user (in the definition of the initial world). The `ipAddress` of the `Host` object can be modified only by some service of the class `Hosting`, so if the requirement on the address appears in the query, then the plan will contain an invocation of that service.

A simple user's query could be stated informally as *"I want to obtain a website on a particular topic, of a specified functionality and with a certain graphical layout, under a given address; I have a fixed budget to spent, and everything should be done until a certain date"*, or *"I need a book on some topic, printed in n copies, of a high quality, for a price not bigger than k, at the date dd–mm–yyyy"*. The queries are converted to enumerations of objects in the

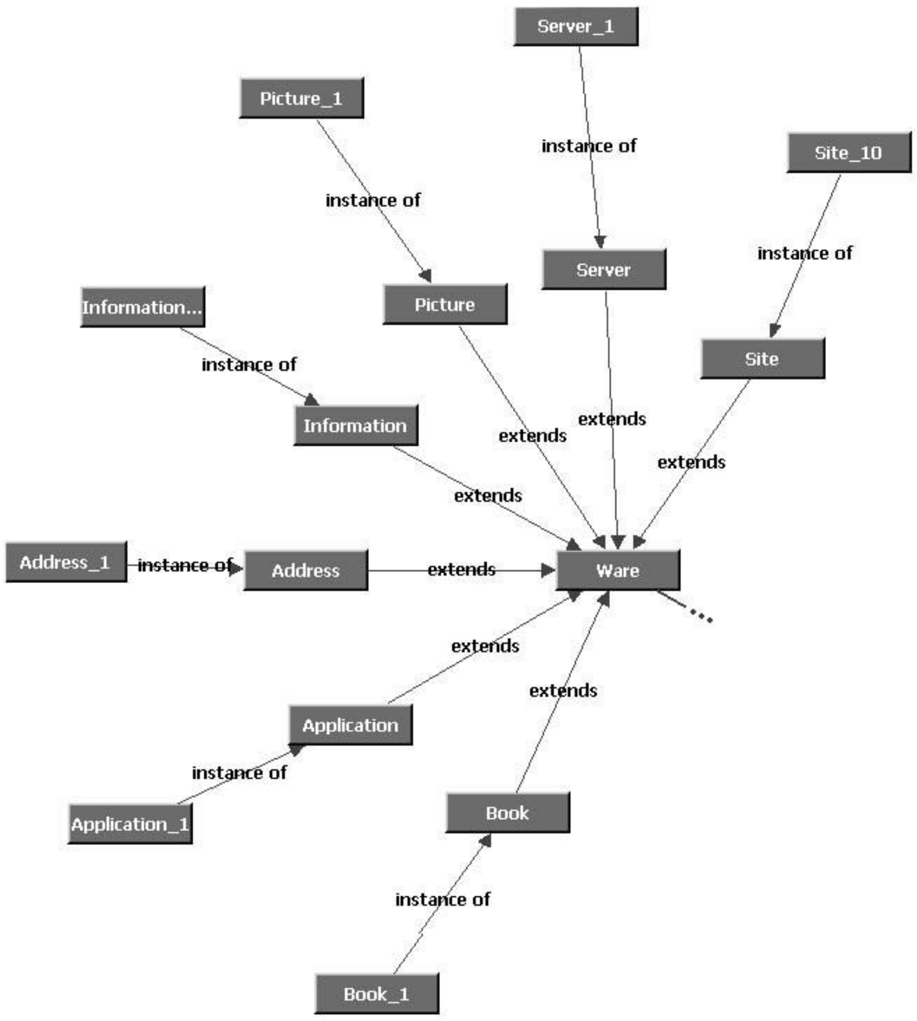


Figure 4. The ontology for the example (objects part)

initial world and the effect world. In the abstract planning phase the conditions referring to values of attributes are converted to requirements of having the attributes set. Depending on the query, the planner can produce various plans without any imperative assumptions not included in the ontology.

Fig. 6 presents a part of the plan generated for the user's query displayed in the left panel of the screenshot (requiring a website of a certain IP address, and a certain layout and code). The full version of the example with the queries, generated plans and a performance analysis can be found in Doliwa et al. (2010).

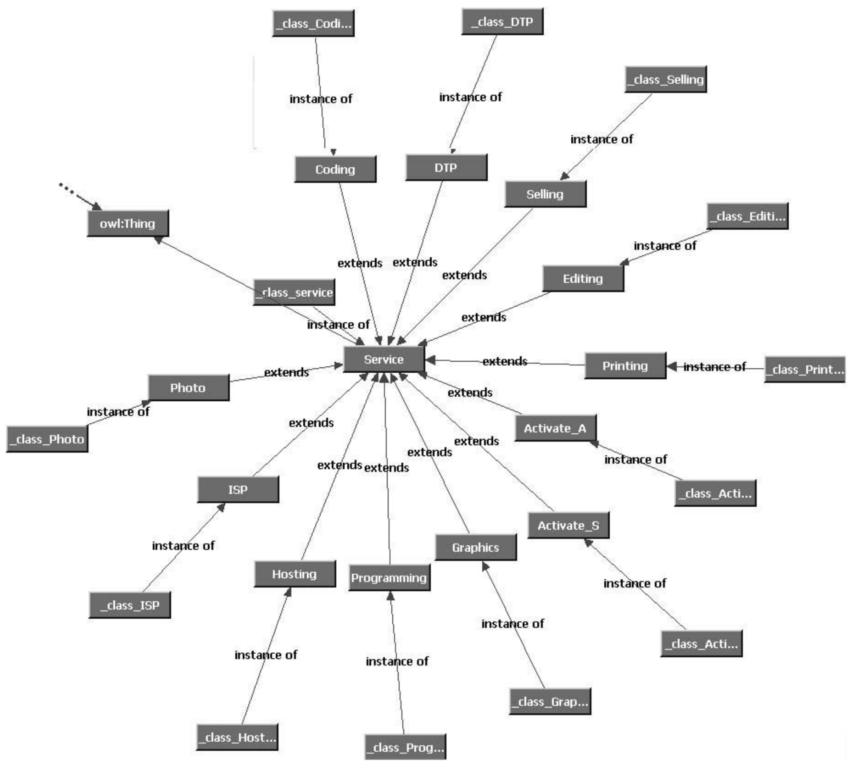
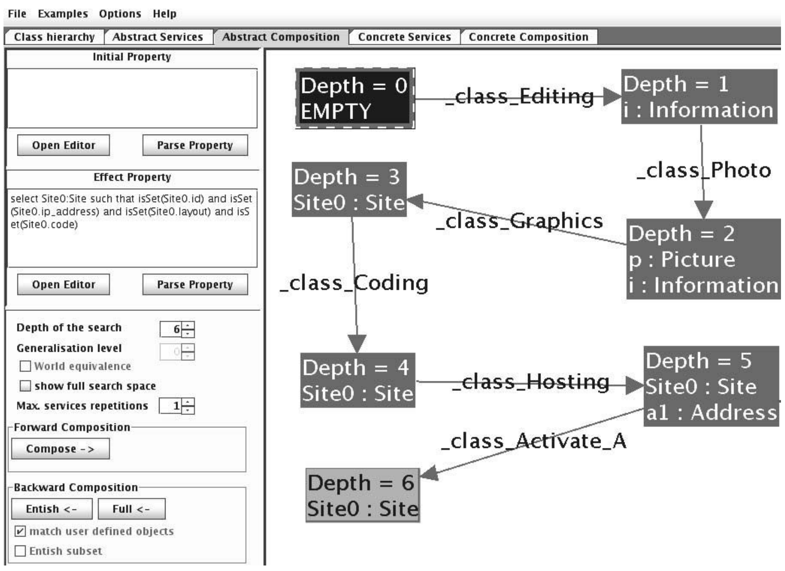


Figure 5. The ontology for the example (services part)

8. Final remarks

The system presented in this article is on its early stage of development. Our aim is to prepare an easy mechanism of an automatic composition, which can be applied to various domains. The areas of our particular interest are e-commerce and web support for medical services. The approach presented seems to be applicable. The first stage of concretising an abstract plan, based on translation to automata and using a SAT-solver to searching for an appropriate scenario, was presented in Penczek, Półrola and Zbrzezny (2010).

Besides a further development of concretising methods, the directions of our future research involve 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.



select Site0:Site such that isSet(Site0.id) and isSet(Site0.ip_address) and isSet(Site0.layout) and isSet(Site0.code)

Depth of the search: 6
Generalisation level: 0
☐ World equivalence
☐ show full search space
Max. services repetitions: 1
Forward Composition
Backward Composition
☑ match user defined objects
☐ Entish subset

Figure 6. An example plan

References

Ambroszkiewicz, S. (2003) *EnTish: An Approach to Service Description and Composition.* ICS PAS, Warsaw.

Bell, M. (2008) *Introduction to Service-Oriented Modeling.* John Wiley & Sons.

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

Doliwa, D., Horzelski, W., Jarocki, M., Niewiadomski, A., Penczek, W., Półrola, A., Szreter, M., and Zbrzezny, A. (2010) Web Service Composition Toolset. In: *Proc. of the Int. Workshop on Concurrency, Specification and Programming (CS&P'10). Informatik-Berichte*, **237** (1), Humboldt University, 131–141.

Jarocki, M., Niewiadomski, A., Penczek, W., Półrola, A. and Szreter, M. (2010a) A Formal Approach to Composing Abstract Scenarios of Web Services. In: *Proc. of the 18th Int. Conf. Intelligent Information Systems (IIS 2010).* Wydawnictwo Akademii Podlaskiej, Siedlce, 3–22.

Jarocki, M., Niewiadomski, A., Penczek, W., Półrola, A. and Szreter, M. (2010b) Towards Automatic Composition of Web Services: Abstract Planning Phase. Technical Report 1017, ICS PAS, Warsaw.

Klusch, M., Gerber, A. and Schmidt, M. (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, 55–62.

McDermott, D., Ghallab, M., Howe, A., Knoblock, C., Ram, A., Veloso, M., Weld, D. and Wilkins, D. (1998) PDDL - The Planning Domain Definition Language - Version 1.2. Technical Report TR-98-003, Yale Center for Computational Vision and Control.

Penczek, W., Półrola, A. and Zbrzezny, A. (2010) Towards Automatic Composition of Web Services: A SAT-Based Phase. In: *Proc. of the 2nd Int. Workshop on Abstractions for Petri Nets and Other Models of Concurrency and of the Int. Workshop on Scalable and Usable Model Checking (APNOC'10 + SUMO'10).* University of Minho, Braga, 76–96.

Ponnekanti, S.R. and Fox, A. (2002) SWORD: A Developer Toolkit for Web Service Composition. In: *Proc. of The Eleventh World Wide Web Conference (Web Engineering Track)*, Honolulu, Hawaii, USA, 83-107. http://radlab.cs.berkeley.edu/people/fox/static/pubs/pdf/c09.pdf.

Rao, J. (2004) *Semantic Web Service Composition via Logic-Based Program Synthesis.* Ph.D. thesis, Dept. of Comp. and Inf. Sci., Norwegian University of Science and Technology.

Rao, J., Küngas, P. and Matskin, M. (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, 446–453.

Rao, J. and Su, X. (2004) A Survey of Automated Web Service Composition Methods. In: *Proc. of the 1st Int. Workshop on Semantic Web Services and Web Process Composition (SWSWPC'04).* **LNCS 3387**, Springer-Verlag, 43–54.

Redavid, D., Iannone, L. and Payne, T. (2008) OWL-S Atomic Services Composition with SWRL Rules. In: *Proc. of the 4th Italian Semantic Web Workshop: Semantic Web Applications and Perspectives (SWAP'07).* CEUR Workshop Proceedings **314**, CEUR-WS.org, 51–60.

Sirin, E., Hendler, J. and Parsia, B. (2003) Semi-automatic Compositions of Web Services Using Semantic Description. In: *Proc. of the Int. Workshop 'Web Services: Modeling, Architecture and Infrastructure' (WSMAI'03).* ICEIS Press, 17–24.

SOAP (2007) SOAP Version 1.2. http://www.w3.org/TR/soap.

Srivastava, B. and Koehler, J. (2003) Web Service Composition - Current Solutions and Open Problems. In: *Proc. of the Int. ICAPS 2003 Workshop on Planning for Web Services.* AAAI, 28–35.

UDDI (2005) Universal Description, Discovery and Integration v3.0.2 (UDDI). http://www.oasis-open.org/committees/uddi-spec /doc/spec/v3/ uddi-v3.0.2-20041019.htm.

WS-BPEL (2007) Web Services Business Process Execution Language v2.0. http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html.

WSDL (2001) Web Services Description Language (WSDL) 1.1. http://www.w3.org/TR/2001/NOTE-wsdl-20010315.