AUTOMATIC GENERATION OF CONCRETE COMPOSITIONS IN ADAPTIVE CONTEXTS

Luca Bevilacqua[2], Angelo Furno[1], Vladimiro Scotto di Carlo[2], Eugenio Zimeo[1]

[1] Department of Engineering, University of Sannio, Benevento, 82100 Italy

[2] Engineering, Napoli, 80142 Italy

ABSTRACT

Service composition is a fundamental facet of Service Oriented Architecture to burst the creation of new services and knowledge throughout the Internet. Automating this aspect has been for many years an interesting research topic for people working in several research areas. In spite of the several scientific results already achieved, generating a concrete and runnable service composition from the semantic descriptions of the domain services and the problem to solve is still an open issue. This paper presents an approach to automatic service composition in the context of autonomic workflows and a related tool developed for an IT industrial context. The tool is able to retrieve service descriptions from a repository, to support the definition of the problem to solve, to generate an abstract plan and to translate it into an executable process language, such as WS-BPEL. This way, the tool covers the planning and re-planning phases of autonomic workflows. The paper compares the approach with other proposals and shows its effectiveness through a case study that exploits automatic service composition to handle an emergency situation caused by a hydrogeological disaster.

KEYWORDS

SOA, Semantic Web services, Automatic Composition, Autonomic Workflow, Business Process Generation.

1. INTRODUCTION

The adoption of Web services and Service Oriented Architectures is promoting a novel approach for developing web applications, since they can be created by composing distributed services hosted by servers in different administration domains. We refer to this new kind of large-scale, distributed application as “multi-organization Web application”.

With the term large-scale, we intend the involvement of a large numbers of services available throughout the Internet. Services can be modified or replaced; they can disappear, and new services with different features may become available.

This class of applications needs a new level of exception handling to address the variability of execution context.

To handle this level of dynamicity, autonomic computing (AC) represents a viable solution. As it allows systems to manage themselves, service compositions can benefit from this approach to properly react to external events in order to change their structure accordingly, reducing human intervention to the minimum.

In this direction, we have defined the concept of autonomic workflow [1][27], a composition of automatic or manual services that is able to proceed towards the goal even if external events significantly change the execution context. To survive the changes, a service composition needs to be modified, taking into account the new environment.

An important role is performed by the *configurator*, a component of an autonomic composition engine that is in charge of implementing self-configuration of service compositions through the knowledge coded at design-time or collected at run-time. The configurator acts on every aspect of a concrete service composition by changing the overall composition graph to make it runnable within the new conditions. To this end, it can: change a link between an activity and a concrete service (re-bind); insert, delete or replace an activity; change the endpoints of a transition; substitute an activity with a sub-process that is able to perform the same actions and to produce the same effects on the external world.

The configurator exploits some internal components to generate or change compositions (see Fig. 1).

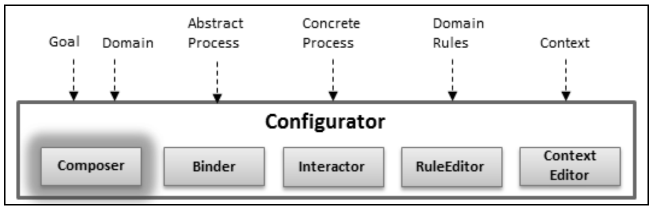


Figure 1. Configurator component of a workflow engine

An important component of the configurator is the *composer*. It can be used either for the initial definition (plan) of a service composition or to re-plan an already defined composition, which could need to be changed completely or in part to react to external events.

The composer exploits planners to transform the descriptions of a goal and a domain in an abstract process. This can be further concretized through the binder, a component in charge of linking an abstract activity with a concrete service. Moreover, domain rules can be exploited to validate automatic compositions generated by the composer.

As Fig. 1 shows, the approach needs several information to support its autonomy during execution. In particular, the Domain refers to a formalized *knowledge* related to the specific application domain where a service composition takes place. It is an ensemble of (1) concepts and relations (ontology) that enrich service descriptions and (2) causal constraints (e.g. pre- and post-conditions) that cause services to be correctly ordered in a service composition. Domain rules, instead, represent higher-level knowledge that is useful to express some constraints that invalidate or validate the generated plans, so reducing the space of admissible solutions. Finally, context rules are used to observe the external world during execution in order to generate new knowledge that potentially can enrich domain ontology and rules.

For illustrating the idea of service composition, we introduce an example scenario related to emergency handling in natural disaster management. The example considered, which will be detailed in Section 4, is about population alert by using multiple communication channels like mobile networks, SMS, MMS, automatic calls, TV, etc.

A possible service composition for alerting people, living in the specific geographic area where the disaster occurred, could consist in the following subprocesses (each constituted by one or more services), to be executed concurrently:

- TV BROADCAST: executing a service to transmit a broadcast alert message on TV channels;
- MOBILE ALERT: executing a service to retrieve the personal details of all the people living in the area; then getting their mobile numbers (MSISDN) from telecom companies, by means of one or more concurrent services, and finally sending SMS (another service);
- LAND LINE ALERT: getting the list of home telephones in the area using a White Pages service; then executing an automatic call center service to call the retrieved phone numbers while concurrently producing a list of citizens without home phone; notifying the local police station with the list of citizens that were not warned (as they do not have home phone or because they did not answered the call) in order to physically alert them at their habitations.

Fig. 2 graphically represents the service composition.

This paper mainly focuses on automatic service composition in the context of autonomic workflow by presenting a tool able to generate concrete and runnable compositions starting from a repository of service descriptions, a domain ontology and constraints. In particular, the domain is expressed as a set of WDSL service descriptions annotated with OWL-S, whereas the target runnable compositions can be generated either in WS-BPEL or in XPDL.

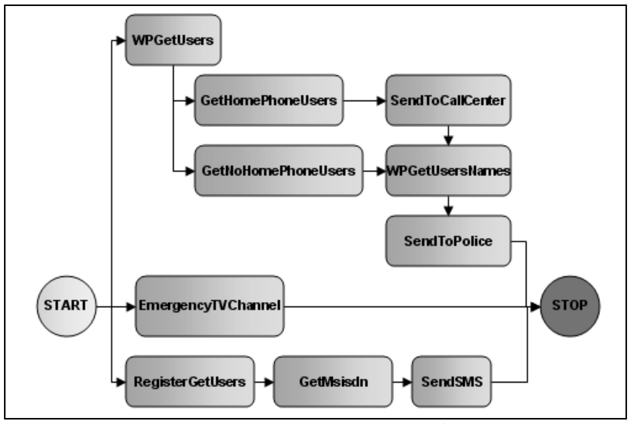


Figure 2. An example service composition for people alert in natural disaster management

The rest of the paper is organized as follows. Section II discusses the related work on tools for automatic service composition. Section III presents the process for automatic generation of service compositions, the proposed tool and its architecture. A detailed description of the WS-BPEL serializer is provided. Section IV analyzes an example of automatic service composition in the context of emergency handling, reporting also a performance analysis for the composition problem considered. Finally, Section V concludes the paper.

2. RELATED WORK

To generate an executable business process description starting from a general, and possibly formal, description of user business requirements and service domain is a complex problem, whose solution needs: (1) a support for formally describing service domain and business problems; (2) an efficient technique for finding a service or combinations of multiple services from the domain, satisfying the specified problem; (3) the automatic generation of a formal business process description, possibly in a standard language (e.g. BPEL, XPDL), from the abstract plan.

In spite of the plethora of efforts devoted to theoretic aspects, up to now only a few proposals have addressed the problem in a comprehensive way and very few tools to generate an executable business process description exist.

In relation to the first problem, the OWL Web Ontology Language [2] allows for representing domain knowledge through a formal and shared XML-based specification of concepts and relations among them. OWL-S [3] supplies Web service providers with a core set of markup language constructs for describing properties and capabilities of their Web services in unambiguous, computer-interpretable form, by referencing concepts and properties from OWL ontologies.

SAWSDL [4] is another W3C recommendation for semantically describing services. It introduces a set of extension attributes to be directly used in WSDL service descriptions to semantically annotate WSDL elements. WSMO-lite [5] is a lightweight set of semantic service descriptions in RDFS for annotating various WSDL elements, using the SAWSDL annotation mechanism.

Regarding the second problem, several approaches, techniques and tools [6-8] have been proposed in literature to efficiently tackle the automation of the composition process. Many of the proposed approaches are based on the use of AI planning techniques, handling the Web service composition problem as a state-space, constraints satisfaction, situation-calculus or other kind of planning problems. Semantics is considered an important support for the automation of the composition process [9].

The Planning Domain Definition Language (PDDL) [10] is considered the *de-facto* standard for classical planning problems input languages. A PDDL planning problem is described in two sections: domain definition and problem specification. The domain describes the possible actions, in terms of inputs, outputs, preconditions and effects, and predicates. The problem essentially describes initial and final states, by specifying the set of predicates assumed to be true in the initial state and the set of predicates to be satisfied in the goal state. Several planners have been developed which use PDDL as input language.

SHOP2 [11], is an HTN (Hierarchical Task Networks) planner, which exploits hierarchical relations among tasks for composing Web services. These relations have to be provided in advance to the planner by designers when describing the planning domain. The planning problem is solved by translating its OWL-S description into a SHOP2 description and by converting the SHOP2 generated plan to an OWL-S runnable process. As pointed in [12], SHOP2 performs well where complete and detailed knowledge on at least partially hierarchically structured action execution patterns is available, but, when no concrete set of methods and decomposition rules are available, an HTN planner is not able to find the solution. This problem inherently limits the planning ability of an HTN planner to the availability of decomposition methods designed by human experts.

In [12,13], Klush et al. proposes the OWLS-Xplan planner, which combines graph-based (by using Graphplan [14]) and HTN planning, using OWL-S descriptions (of both domain and problem) as input, translating them into an XML version of the PDDL language, called PDDXML. The output is a sequence of activities described in PDDXML. The approach combines both the advantages of task decomposition available with HTN planning and the Graphplan capability of always finding a solution, when present. The authors also propose a replanning component, able to update plans during execution.

Another recent composition framework is PORSCE II [15]. Like OWLS-Xplan, the framework input consists of OWL-S service descriptions, which are translated into PDDL. The framework combines a domain-independent planning component (e.g. JPlan, LPG-td) and an ontology concept relevance module for semantic awareness and relaxation during planning. Several plans, with different semantic accuracy levels, can be generated and presented to the user through a graphical component. Moreover, the graphical component can be used to request replanning by selecting a task and asking the system to find an alternative equal or semantically similar service or composition. Subsume relationships among service pre- and post-conditions are considered to find such alternatives.

Other notable planning solutions for service composition are based on the Golog language. Golog is a logical programming language and has been extended in [16] to support customized constraints and non-determinism in sequential executions and have been used in order to support service composition, by means of a translation into PDDL. In [17], a process for translating OWL-S descriptions into situation calculus has been proposed, while, in [18], DL reasoning techniques are used together with extended Golog to calculate conditional Web service compositions.

The Haley framework [19, 20] includes a Golog-based planning system for Web service composition. The system uses SAWSDL semantically described services as input, contains a planning Golog-domain generator and the eDT-Golog planner. Differently from the previous described framework, Haley is able to generate a WS-BPEL description of the plan and execute it on a WS-BPEL engine. However, Haley tackles the service composition problem from the perspective of generating complete business processes from user business requirements, by assuming the presence of concrete services with specified QoS parameters. In this sense, scalability is a very important problem and the hierarchical approach, as in SHOP2, is a way to reduce the planning effort, but requires a designer to know how to decompose tasks in subtasks. From our perspective, planning has to be a support especially to the generation of small business sub-processes, which concretize tasks from an already defined main workflow and is guided by the Binder component of the Configurator (Fig. 1). Consequently, the scalability problem is reduced in our perspective. Moreover, our proposed composition tool is also able to work with already composite service. Haley's authors believe classical planning techniques are not well suited to the Web service domain, because of its inner non-determinism. As presented in [1], we argue that non-determinism can be handled through events observation and proper reaction: this way, the autonomic approach can be exploited to fill the gap with the classical planning techniques in Web service composition. Another important difference between Haley and our composition tool is that Haley is not able to generate concurrent sub-processes.

Finally, in relation to the third problem, the generation of formal and standard business process representation of the service compositions, several languages have been proposed. Among these, Business Process Modeling Notation (BPMN) [21], Xml Process Definition Language (XPDL) [22] and Web Service Business Process Execution Language (WS-BPEL) [23] are the most important and widespread ones. In the following, we focus our attention on WS-BPEL (v. 2.0) which can be considered the de-facto standard for business process description languages in the web service domain.

In the following we propose a composition tool based on: (1) the use of OWL and the OWL-S ontology for the semantic descriptions of domain services; (2) a classical planning-based approach for creating service compositions, using the PDDL language for domain and problem specification; (3) autonomic workflows to handle non-determinism and enact proper service re-composition as a reaction mechanism; (4) WS-BPEL as the language for describing the resulting service composition.

3. COMPOSITION TOOL

The proposed tool is intended to support the initial plan of a service composition or to re-plan an already defined one.

3.1 Composition Process

Fig. 3 shows a graphical representation of the notion of Web service composition in workflow design. In the top part of the figure, workflow tasks represent complex activities, which can be implemented as service compositions. Automatic support for generating executable service compositions is the focus of this paper.

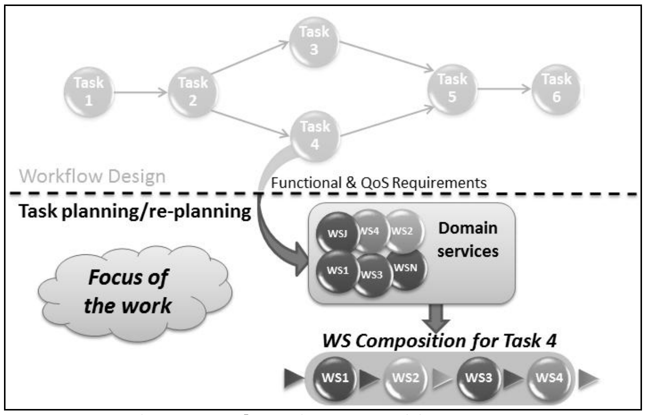


Figure 3. Web service Composition Process

Starting from a task description (the problem, e.g. task 4 in the picture), a set of candidate Web services (the service domain) is inspected in order to find a chain of services (a plan), which is consistent with the task description. Consistency means that, starting from a provided description of the initial state (i.e. the set of predicates which are true before the task beginning), the chain is able to reach the goal state (i.e. the predicates in the goal state have to be true at the end of the service chain).

The initial state includes a description of all the input data available at the beginning of the execution, while the goal state specifies the desired outputs to be obtained by the task execution.

The problem task may correspond to a single task in an already defined business process or to the whole business process. The first case can be related to a re-planning process, where an activity of the process is replaced by a service composition, while the second case relates to the planning process, that is the definition of a new complete business process, satisfying some user's specific business needs.

We assume both the service domain and the problem to solve are described by using the OWL-S ontology, according to the IOPE semantics. The set of semantically described services (the domain) has to be known before starting the composition process and provided as an input to the system together with the semantic description of the problem. The service domain can be retrieved manually or in a (semi-) automatic way, by relying on a service registry and exploiting the semantic description of the goal to reach during a matchmaking process.

Any service operation is associated to an OWL-S file, which includes the definition of an OWL-S atomic process (<*process:AtomicProcess*>), specifying the inputs, the outputs, the preconditions and the effects (IOPE) of the specific operation performed by a service. Inputs and outputs may refer to concepts imported from OWL ontologies or XML-Schema data types. Preconditions and effects are specified within the OWL-S process description in the <*process:hasPrecondition*> and the <*process:hasResult*> sections respectively. Semantic Web Rule Language (SWRL) [24] expressions are used to define these conditions.

The problem is considered as a desired operation and is specified as an OWL-S process with inputs, outputs, preconditions and effects, like a service operation. Inputs and preconditions make up the initial state, which is data known to be available and predicates known to be true when the related task starts, while outputs and effects make up the goal state, that is desired data outcomes and true predicates at the end of task execution.

Fig. 4 describes the main logic flow associated to our composition tool.

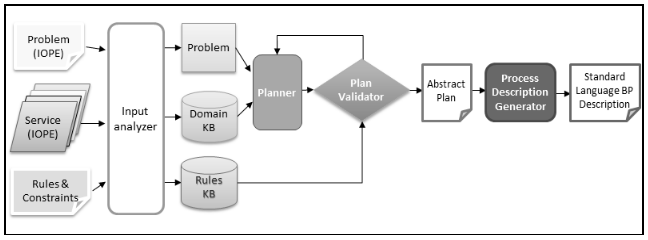


Figure 4. Planner-based composition process

The OWL-S service domain and problem descriptions represent the main inputs for the composer. Also, it is possible to specify simple business rules that have to be validated on the generated plans. Provided inputs are then processed and transformed into proper internal data structures.

The processed information (service domain and rules) is captured into internal incremental knowledge bases, used to quickly solve future requests related to the same domain or business rules. The internal input data are then supplied to the planner component to find some solution plan, or *abstract plan*, (set of activities) from the domain, satisfying the specified problem and business rules (through the plan validator). If the validation fails, a new plan may be found. The generated abstract plan is finally bound to concrete Web services and transformed into a standard business process representation (e.g. BPEL, XPDL), or *concrete plan*. The produced description can be executed on standard execution engines, like RiftSaw [25], Apache ODE [26], SAWE [27].

Since the focus of the paper is not on devising a new planning approach, but on providing the highest automatic support to discovery executable service composition as a result of an adaptation rule, we focused our attention on the well-known Graphplan algorithm to produce the abstract plan. In the following we briefly describe the approach used in the Graphplan planner, which has been adopted in the implementation of the proposed tool, as will be described in Section 3.2.

The Graphplan algorithm [14] was proposed by Blum and Furst in 1997 to efficiently solve planning problems in STRIPS-like domains (made of objects, operators and propositions). It consists of two interleaved phases: a forward phase, where a data structure called "planning-graph" is incrementally extended, and a backward phase where the planning-graph is searched to extract a valid plan. The planning graph can be created in a polynomial time, with respect to the size of the problem domain, while the search phase has an exponential complexity in the worst-case.

The planning graph structure is organized in multiple levels, each containing proposition or action nodes, connected by three kinds of edges: pre-condition, add and delete edges. The 0-level is a propositional list, containing one node for each proposition in the initial state of the problem. Each other level contains both a proposition and an action node list. The 1-level action list will contain all the actions which can be executed given the 0-level propositions, while the 1-level proposition list will include all the effects of the 1-level actions. In general, given a k-level planning graph, the extension of the structure to level k+1 involves introducing all actions (no-operations included), whose preconditions are present in the k-th level proposition list. The k+1 propositional list includes all the propositions added or deleted as effect of the actions belonging to the k+1 level. Since no-operations (a.k.a. persist operations) are included in the k-th action list, all the proposition from level k will also be contained in level k+1 proposition list. The planning graph construction takes into account mutual exclusion constraints among actions and propositions, which are propagated over the levels of the graph.

The search phase on a k-level planning-graph starts by searching, at level k, the propositions corresponding to problem goals. If all the goals are not present, or if they are present but a pair of them are marked mutually exclusive, the search is abandoned right away, and planning-graph is grown another level. For each of the goal propositions, an action from the level k action list, supporting it with its effects, is selected. Mutual constraints are considered in this step, in order not to select actions for supporting two different goals which are mutually exclusive. If they are, backtracking is performed to select new pairs of actions. At this point, the search process is recursively called on the k-1 level planning-graph, with the preconditions of the actions selected at level k as the goals for the k-1 level search. The search succeeds when level 0 is reached and the selected actions for each level represent a partially ordered plan.

3.2 Tool Architecture

The main composition process, described in the previous paragraph, has led to a specific architecture for composition tool that is detailed in Fig. 7.

The most relevant components are:

- The **OWL-S Analyzer**
- The **Planner**
- The **Plan Validator**

- The **Plan Converters** (or serializers)

In order to introduce flexibility in the proposed planning-based approach to composition, we have defined a **meta-model** for describing all concepts, and their relationships, which define our notion of autonomic workflow.

Fig. 5 represents a Domain and a related Problem, while Fig. 6 gives a detailed description of the concept of Plan, composed of a Workflow, which includes simple or complex (composite) Activities and several control flow structures as Parallel and Sequence. The Plan element joins the two models.

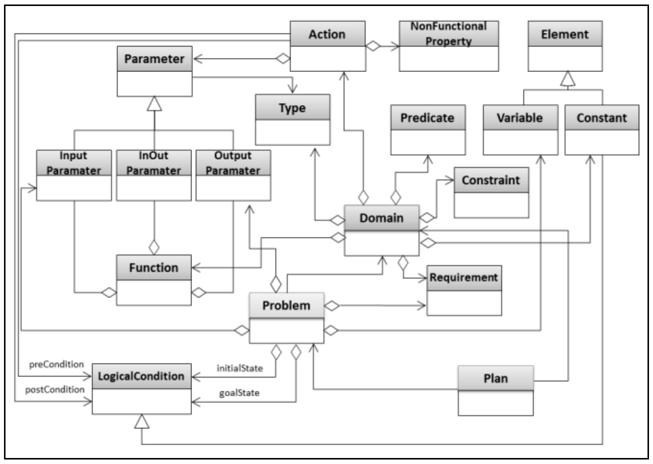


Figure 5. Problem and Domain Model

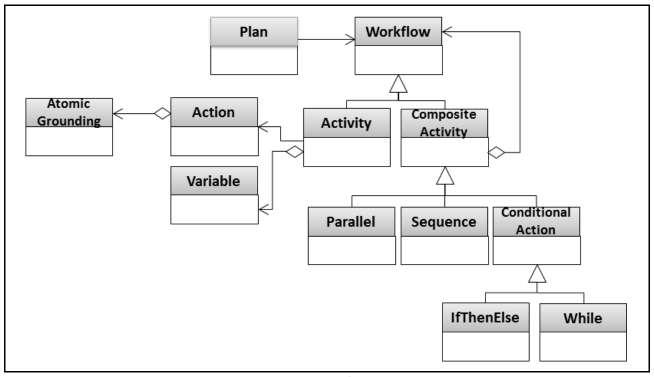


Figure 6. Workflow Model

The **meta-model** is used to define abstract representations of the service domain, the problem under analysis and workflow plans for solving it in the domain. By defining converters (problem and domain serializers) from the meta-model to planning specific representation languages, it is easy to support several planners. This way, the composer is independent from specific planning tools and languages. Since PDDL represents the de-facto standard for classical planning problems and there are several planning systems supporting this language, including PDDL4J, in the current implementation of our tool, we focused our attention on this language and implemented specific converters from the meta-model to the PDDL 3.0 specification (Section 3.3).

The **OWL-S Analyzer** is the component responsible of analyzing both the OWL-S files, which describe the available services in the planning domain, and the OWL-S of the problem to solve. This component parses the provided inputs and converts them to an instance of the meta-model. It also integrates reasoning capabilities about semantic concepts referred in service and problem descriptions.

The main rules used by the analyzer to generate a meta-model instance from OWL-S files are the following:

- The service operation name (<*service:Service*>) defines the name of a new *Action*;
- The name of input and output parameters (<*process:Input*> and <*process:Output*>) of the atomic process describing the service operation defines the name of the action *Parameters*. Associations between the *Action* instance and its input/output *Parameters* are introduced;
- The types of input and output parameters (<*process:parameterType*>), possibly referring ontology concepts, define new *Types* of the domain object and are associated to the corresponding Parameter objects.
- SWRL conditions, defining preconditions or effects of a service operation over its parameters and constants (<*process:hasPrecondition*> and <*process:hasEffect*>), are used to define domain *Predicates*, related to action *Parameters* and associated to action objects as preconditions or effects respectively.
- Any action input parameter, retrieved from an OWL-S process, generates a *hasKnowledge* predicate, added as precondition for the relative action object and having the input parameter associated as a predicate variable.
- Any action output parameter, retrieved from an OWL-S process, generates a *hasKnowledge* predicate, added as effect for the relative action object and having the input parameter associated as a predicate variable. The *hasKnowledge* predicates for the input/output parameters are added in order to consider parameter dependencies among planning actions during the plan generation, when strips-like planners as Graphplan are used.
- Any element in the OWL-S description, different from input or output parameter, or type names, defines a new domain *Constant*.
- References to WSDL information, required in the later phase of *Plan Serialization* for describing an action plan in an executable standard business process representation, are available in the OWL-S grounding section (<*grounding: WsdlAtomicProcessGrounding*>). This section contains information like a WSDL document URI, a *portType*, an *operation*, an *inputMessageMap* and an *outputMessageMap*, which specifies how an OWL-S atomic process maps to a concrete Web service. This information is retrieved by the OWL-S Analyzer and stored in an *AtomicGrounding* object associated to the action related to the OWL-S described service operation.

In a very similar manner, it is possible to build the meta-model *Problem* object, by applying the previous specified parsing rules to the problem OWL-S description. Obviously, no grounding section is supposed to be available for the OWL-S problem description, since the problem is still to be solved with a concrete (combination) of web service(s). For this reason, the last rule does not apply to the case of the OWL-S problem parsing.

The **Planner** is the component deputed to the processing of domain and problem inputs in order to produce a plan of domain actions satisfying the problem. To this end, several solutions are available. We implemented a PDDL problem serializer, which take as input the meta-model representation of the problem and produce a PDDL 3.0 compliant serialization of it. Similarly, we implemented a PDDL domain serializer, which does the same on the domain meta-model object built by the OWL-S Analyzer. By having the PDDL representation of both the domain and the problem, a PDDL planner can be used for generating plans.

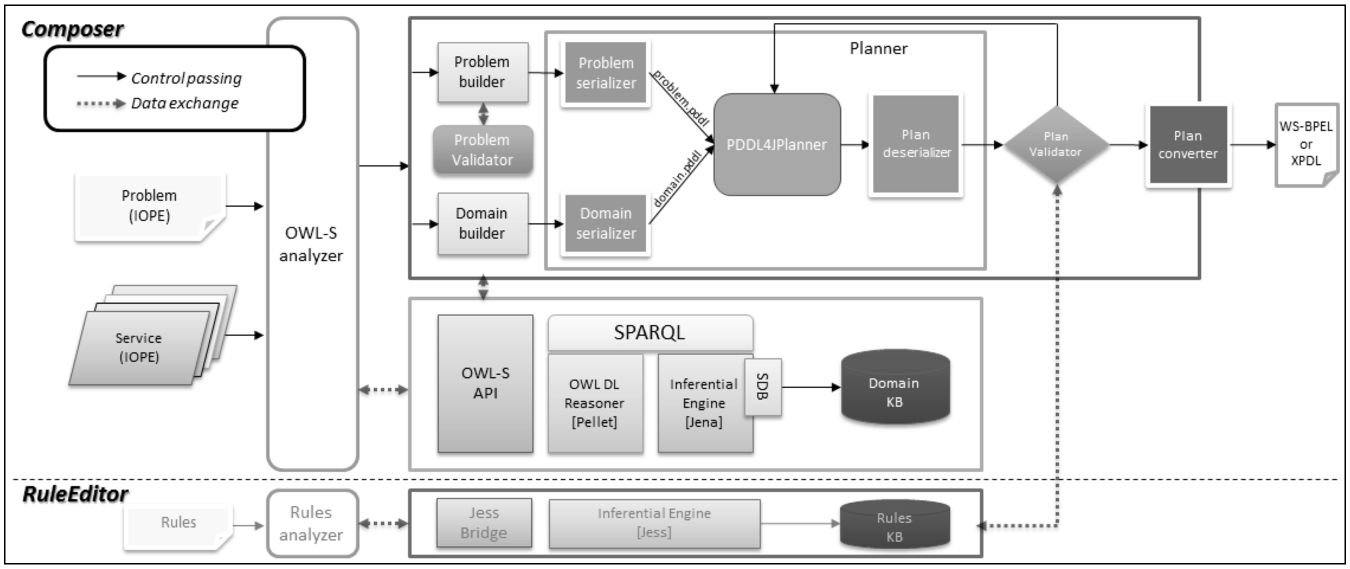


Figure 7. Detailed Composer Architecture

The current implementation of the tool uses PDDL4J [28] for planning, a product released under the GNU General Public License (GPL), which is based on a Java implementation of the Graphplan algorithm, described in Section 3.1. The PDDL4J output plan is represented in a PDDL-like representation, which is converted back to its meta-model representation (the Plan object in Fig. 6), through the *PlanDeserializer* component.

The grammar described in Table 1 defines the language supported by the Composer to express business rules that has to be satisfied by the generated plans. In the current implementation, the rule language only supports the definition of dependency (`<->` in the table) and mutual exclusion constraints (`->!`) between pairs of activity. As an example of these constraints, rule `A <-> B` means that if the plan contains the `A` activity, `B` has to be present too and vice-versa (whereas the order is inferred by IOPE descriptions). Rule `A ->! B` means that `A` and `B` activities have never to be both present in the same plan.

Table 1. An excerpt of the rule language grammar

```
EXP ::= EXP_TYPE ; | ;
EXP_TYPE ::= RULE | CONSTRAINT
CONSTRAINT ::= ACTIVITY <-> ACTIVITY |
               ACTIVITY ->! ACTIVITY
ACTIVITY ::= IDENTIFIER
RULE ::= …
```

The **PlanValidator** component has the role to check if the business rules, specified by the user as input to the composer, are satisfied by a plan produced by the Planner.

If these rules are present and the plan does not satisfy them, a new plan has to be searched by the Planner component until rules are satisfied or there are no other feasible plans.

The behavior of the composer, in relation to the described components, is shown in the UML sequence diagram of Fig. 8.

The architecture described in this section has been implemented using the Java language. The Meta-model in Figures 5 and 6 has been implemented through a set of Java interfaces and classes. The *Domain* and *Problem* classes have been equipped with proper methods for serialization into a PDDL 3.0 language representation (PDDL Problem/Domain serializers in Fig. 4).

A PDDL plan deserializer has been developed in order to analyze the PDDL plan produced by the PDDL4J planner and generate a corresponding instance of the *Plan* class. The associated instance of the *Workflow* class contains the Meta-model representation of the control flow for the plan (see Fig. 6).

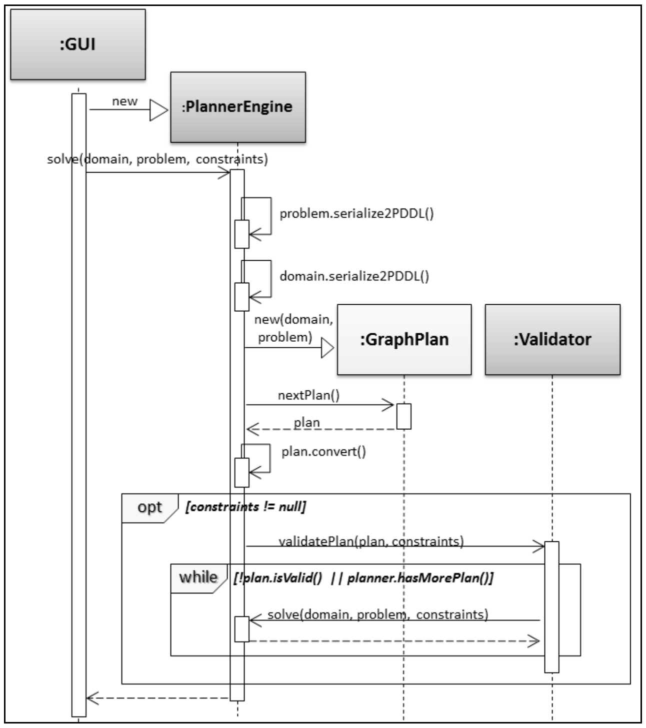


Figure 8. Plan generation and validation

The workflow is generated by analyzing the partially ordered plans (POPs) produced by PDDL4J and, if the plan contains more than one action, by properly constructing a composite activity. This activity can be a sequence or a parallel, containing other parallels or sequences at any level. Since the POPs produced by GraphPlan are simply lists of steps, each containing a list of actions to be performed, any list of actions for a specific step is analyzed in order to identify parallel branches possibly ranging over more than one single step. This condition occurs when, in the steps following the one currently being analyzed, there is no action whose input parameters or pre-conditions depend from the output parameters or post-conditions of the action in the current step. This analysis is done in order to optimize the plans produced by the PDDL4J planner (i.e. maximize the parallelism) in the final control flow of the business process (i.e. the produced *Workflow* instance for the *Plan* object).

The current implementation of the tool has been equipped with two main serializers: a WS-BPEL Serializer and an XPDL Serializer (Fig. 9).

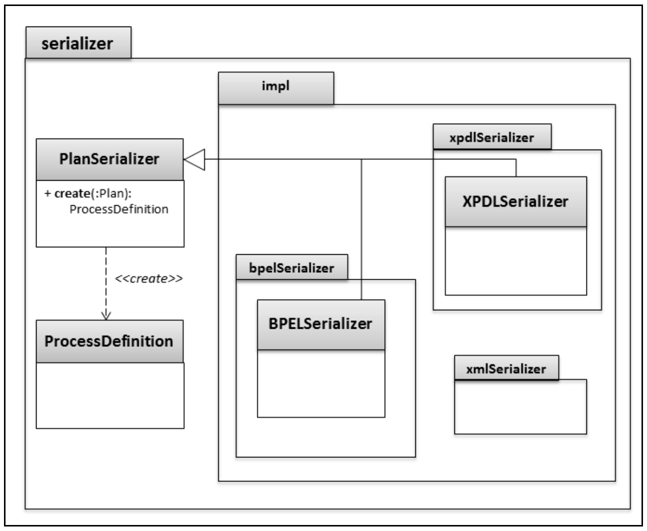


Figure 9. Plan serialization to a BP representation language

In the next section, the WS-BPEL Serializer is presented. The XPDL serializer is quite similar, using the XPDL language for the process representation.

3.3 WS-BPEL serializer

A serializer for a business process representation language is responsible to retrieve all the information required by the corresponding language specification, and possibly demanded by the business process execution engine, to generate a compliant representation of the plan, executable on that engine. For example, the WS-BPEL representation of a process to be executed on the RiftSaw business process engine is composed of a *.bpel* file, a *.wsdl* artifacts file, a deploy *.xml* file and the set of imported *.wsdl* files. The first file (*.bpel*) contains, among other details, the WS-BPEL language description of the business process control flow and data flow. The second one (*.wsdl* artifacts file) contains any WSDL information needed to expose the WS-BPEL process as a Web service (e.g. *portType*, *operation*, *input/output messages*, *XML Schema types*, *partnerLinkTypes*). The third one is a file describing how to deploy the business process on the engine by associating endpoints to any provided or invoked service over a partner link and specifying other details for the deploy process (e.g. if the process is active or not). Finally, the set of the *.wsdl* imported files refers to the Web service descriptions used within the *.bpel* file.

In the following, we describe the process of generating a concrete WS-BPEL process definition (concrete plan), executable on the RiftSaw engine, from a non-empty *Plan* object (abstract plan). With these assumptions, the class diagram in Fig. 10 contains classes for a *BPELFile*, a *BPELDeployFile*, a *BPELArtifactsFile* and a set of imported WSDL descriptions. These classes are the object representation of the files required for building a complete BPEL description. They provide methods for their own construction, whose invocation is coordinated by the *BPELSerializer* and, in turn, by the *BPELProcessDefinition* objects, according to a two-levels builder pattern approach.

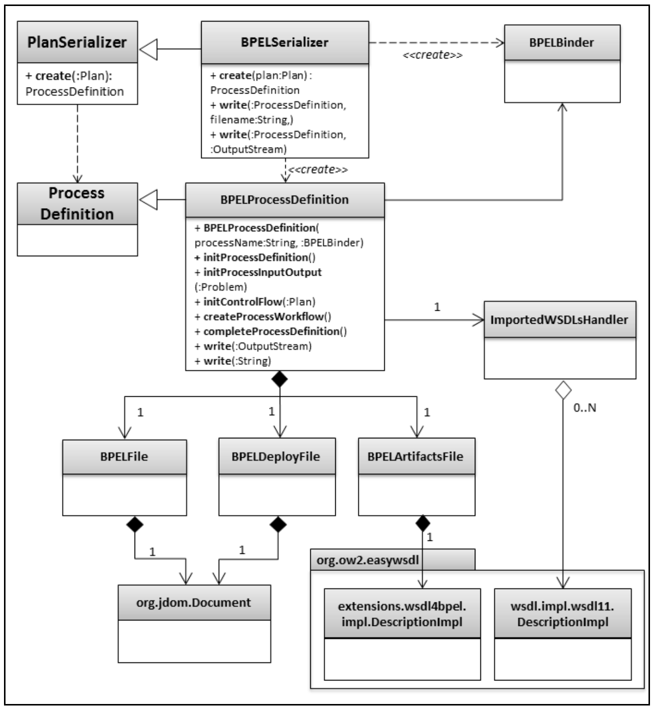

PlanSerializer
+ create(:Plan): ProcessDefinition
BPELSerializer
+ create(plan:Plan) : ProcessDefinition
+ write(:ProcessDefinition, filename:String,)
+ write(:ProcessDefinition, :OutputStream)
<<create>>
BPELBinder
Process Definition
BPELProcessDefinition
+ BPELProcessDefinition(processName:String, :BPELBinder)
+ initProcessDefinition()
+ initProcessInputOutput (:Problem)
+ initControlFlow(:Plan)
+ createProcessWorkflow()
+ completeProcessDefinition()
+ write(:OutputStream)
+ write(:String)
1
ImportedWSDLsHandler
0..N
BPELFile
BPELDeployFile
BPELArtifactsFile
org.ow2.easywsdl
org.jdom.Document
extensions.wsdl4bpel. impl.DescriptionImpl
wsdl.impl.wsdl11. DescriptionImpl

Figure 10. WS-BPEL serializer architecture

In order to execute a plan generated by the planner engine on common business process execution engines, it is necessary to convert its meta-model representation (the *Plan* object) to a standard business process representation language (e.g. WS-BPEL, XPDL, etc.).

To this purpose, according to the architecture depicted in Fig. 7, several **Plan Converters** can be defined, which serialize the meta-model *Plan* object to a specific language representation (Fig. 9). More details about Plan Converters are given in Section 3.3.

In the design of our tool, the *BPELSerializer* class is in charge of creating the *BPELProcessDefinition* object (and the component objects) and invoking its methods in order to create a complete and executable specification of the BPEL process, corresponding to the activity plan found by the planner component. The *create* method triggers the process generation from the specified instance of the *Plan* meta-model class. The *write* method generates files for the process description, by triggering the *write* methods provided by the component objects. The main sources of information available to the *BPELSerializer* are:

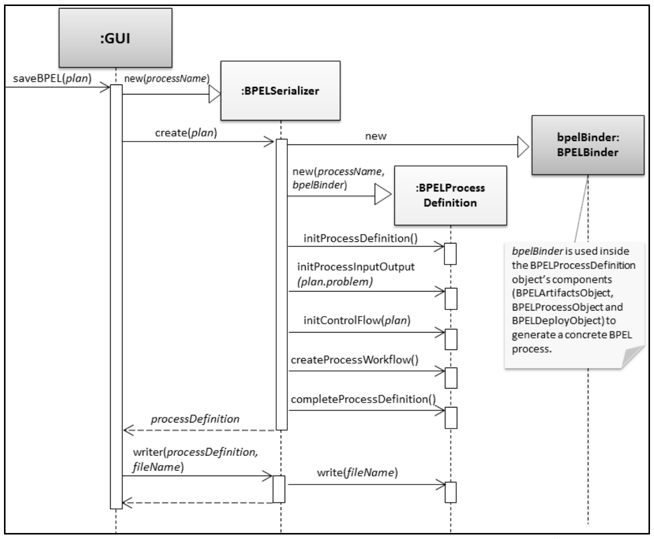
1. the Plan meta-model object and, in particular, the associated Workflow instance;
2. the domain/problem OWL-S files;
3. the WSDL service files, containing the endpoints required to make executable the final business process representation.

The association between the plan actions, the OWL-S and WSDL files lies in the OWL-S grounding section. It is retrieved during the OWL-S parsing and is stored in the Grounding of the Action class in the meta-model (Fig. 7).

WSDL files have been handled (see Fig. 10) by using the

EasyWSDL libraries [29], which offers support for parsing and generating WSDL 1.1 or WSDL 2.0 compliant service descriptions. Also, the *EasyWSDL4BPEL* library enables the handling of partner links BPEL extensions in service description.

The main steps for the generation of the *BPELProcessDefinition* are described in the following (Fig. 11).



bpelBinder is used inside the BPELProcessDefinition object's components (BPELArtifactsObject, BPELProcessObject and BPELDeployObject) to generate a concrete BPEL process.

Figure 11. WS-BPEL process generation, main sequence of actions

1. ***initProcessDefinition()*** initializes the object representing the concrete BPEL process. The generated WS-BPEL process is exposed as a WSDL service, offering one *operation a*nd one *portType* for executing the activity plan. Also the *partnerLink* and the *partnerLinkType* for interacting with the WS-BPEL process are created.
2. ***initProcessInputOutput****(:Problem)* by using the *Problem* Meta-model instance, available input data are used to initialize the process input message, while requested output data are used to initialize the output message of the process. Since the WS-BPEL process is exposed as a web service, input and output messages for the provided operation are associated with simple or complex XML types specified in the WSDL artifacts file. The concrete structure of the types for the input/output messages is created later (see the *buildInvokeActivity(…)* description), when analyzing the concrete services to be invoked by the process using as input (parts of) the process input message and producing as output (parts of) the process output message, respectively;
3. ***initControlFlow****(:Plan)* from the Meta-model *Plan* instance, initializes the basic control flow of the process: an initial receive input message activity and a final reply output message activity are added to the main sequence of activity of the process, which will contain also the whole workflow associated to the plan;
4. ***createProcessWorkflow()*** adds to the control flow skeleton, built by the *initControlFlow()* method, the BPEL activities corresponding to the plan instance. The activities can be sequences and parallels (in any combination) of invocation of web services (BPEL <*invoke*> activities). Also, the *createProcessWorkflow()* method handles the construction of the data flow among web services. By using the binding component (*BPELBinder*), which relies on the grounding information extracted by the OWL-S semantic descriptions of the services to be invoked, partner link and partner link type definition is completed: the former are included in the BPEL file, the latter in the WSDL artifacts file.
5. ***completeProcessDefinition()***, completes the definition of the process by generating the binding information for the invoked and provided partner links of the process, inserting them into the deploy file (bpel-deploy.xml).

In the following subsections, the two most relevant phases of the business process generation, i.e. control flow generation and data flow generation, are described in details.

3.3.1 Control Flow generation

The skeleton of the workflow structure is created by the *initControlFlow()* method and consists in a sequence of two activities: a receive input message activity followed by a reply output message activity (by assuming the process will interact with a requestor by at least requiring an input message and returning an output).

This structure is refined in the *createProcessWorkflow()* step, by introducing the activities of the generated plan in between the receive/reply ones. To this end, the *createPlanActivities*, provided by the *BPELFile* object, is invoked. The method relies on the use of the *browseActivity(…)* method which is applied to the *Workflow* instance of the *Plan* object, as described in Fig. 12.

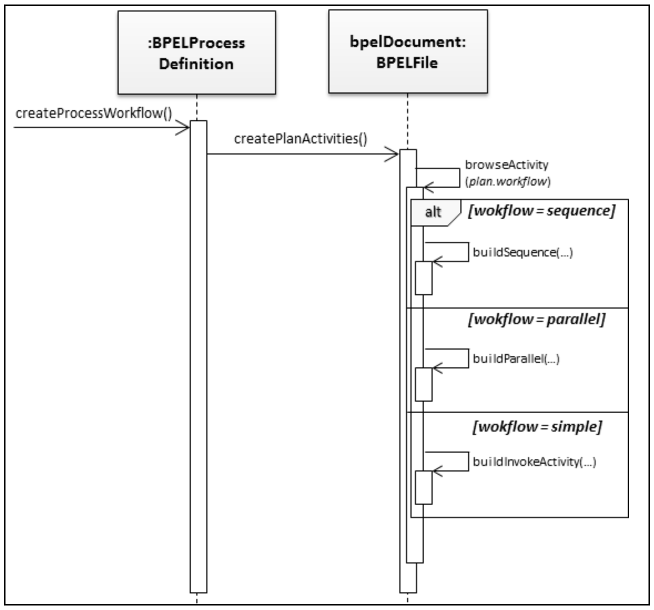


Figure 12. WS-BPEL process generation, main sequence of actions

The method analyzes the type of the Meta-model *Workflow* instance (*Sequence*, *Parallel*, *Activity*), containing the control flow definition of the business process, and builds a new sequence (<*bpel:sequence*>), parallel (<*bpel:flow*>) or an external Web service invocation (<*bpel:invoke*>) activity to the BPEL process accordingly. To this purpose, the proper methods from the *BPELProcess* class have to be invoked, depending on the type of the current *Workflow* instance. Any time a sequence or a parallel is added to the BPEL process (i.e.: when the workflow parameter is a *CompositeActivity*), a recursive invocation of the *browseActivity* method has to be performed over any component *Workflow* instance, in order to add to the control flow the activities inside the current parallel or sequence (which can be both *CompositeActivity* or *Activity*). This way, proper nestings of <*bpel:sequence*>s and <*bpel:flow*>s of basic Web service invocations are produced in the final BPEL process, according to the control flow structure in the *Workflow* object of the *Plan*.

The *buildInvokeActivity(…)* (Fig. 13) is the method used for adding an external web service invocation to the BPEL process. This method is invoked any time an atomic *Activity* is found in the Workflow. In the current implementation of the tool, only synchronous invocations of Web services are addressed. In the BPEL process, this corresponds to introduce an <*invoke*> activity with both the input and the output variables specified, if present.

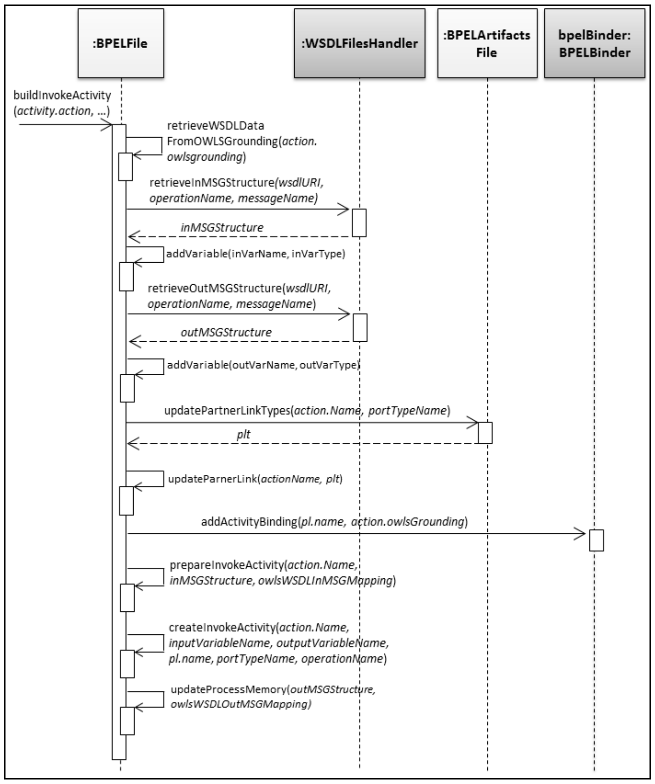

:BPELFile
:WSDLFilesHandler
:BPELArtifacts File
bpelBinder: BPELBinder
buildInvokeActivity (activity.action, ...)
retrieveWSDLData FromOWLSGrounding(action. owlsgrounding)
retrieveInMSGStructure(wsdlURI, operationName, messageName)
inMSGStructure
addVariable(inVarName, inVarType)
retrieveOutMSGStructure(wsdlURI, operationName, messageName)
outMSGStructure
addVariable(outVarName, outVarType)
updatePartnerLinkTypes(action.Name, portTypeName)
plt
updateParnerLink(actionName, plt)
addActivityBinding(pl.name, action.owlsGrounding)
prepareInvokeActivity(action.Name, inMSGStructure, owlsWSDLInMSGMapping)
createInvokeActivity(action.Name, inputVariableName, outputVariableName, pl.name, portTypeName, operationName)
updateProcessMemory(outMSGStructure, owlsWSDLOutMSGMapping)

Figure 13. Generation of an <*invoke*> activity, main sequence of actions

The <*invoke*> activity specification has to include any detail for the correct invocation of the Web service (*binding information*) like: *partnerLink*, *portType*, *input and output variables* for data exchanges between the process and the WS.

As shown in Fig. 13 (*retrieveWSDLDataFromOWLS-Grounding(…)*), the binding information can be retrieved from the WSDL-grounding information of the OWL-S description of the concrete action performing the activity. In the Meta-model, each atomic activity is linked to an action (a virtualization of the service to be invoked) which contains, among other information, the mapping between its own OWL-S semantic description and the WSDL description. From the OWL-S WSDL grounding, it is possible to retrieve the service WSDL description URI, the *PortType*, the specific operation performing the OWL-S atomic process. Also, the mappings between the semantic description of the OWL-S process input/output parameters and the WSDL operation input/output messages are retrieved and used for data flow generation (see Section 3.2.2).

Also, by using proper naming conventions, information about *partnerLinks* and *partnerLinkTypes* is retrieved and included in the <*bpel:invoke*> specification or, if not already available, automatically generated and added to the business process artifacts file (*updatePartnerLinkTypes(…)*) and to the bpel file (*updatePartnerLinks()*). The binder will use instead the WSDL URI and the *portType* retrieved from the WSDL-grounding to get service endpoints, storing them, in association with the *partnerLinks*, in order to use them later during the deploy file generation.

3.3.2 Data Flow generation

To generate an executable WS-BPEL process it is fundamental that a proper data flow among the services to invoke is built. This can be performed by taking into account the semantic data dependencies among the web service activities, which has driven the plan generation. It is worth to note that the plan, both in the PDDL and in the Meta-model representations, does not contain an explicit description of the data flow among the composing activities, while a WS-BPEL process requires its formal definition in terms of variable declarations, initializations, value transfers and assignments (<*bpel:assign*>).

In the current implementation of the tool, we assume that each Web Service operation has, at most, one input message and one output message and that the type of the exchanged messages can only be a simple or complex XML Schema [30] data type. In the latter case, we only consider sequences of simple XML Schema types (string, integer, float, etc.). In the following discussion, we will refer to the most general case of service operations with both input and output messages specified and typed with a complex XML Schema. Also, in the grounding section of the OWL-S descriptions, parameters of the atomic process are considered to be explicitly mapped to simple typed parts of the message of the web service corresponding operation. This process should naturally take place during the semantic description of the web service in its OWL-S file and can rely on the use of the <*grounding:WsdlInput(Output)MessageMap*>, <*grounding: owlsParameter*> and <*grounding:wsdlMessage-Part*> elements provided by the OWL-S grounding ontology.

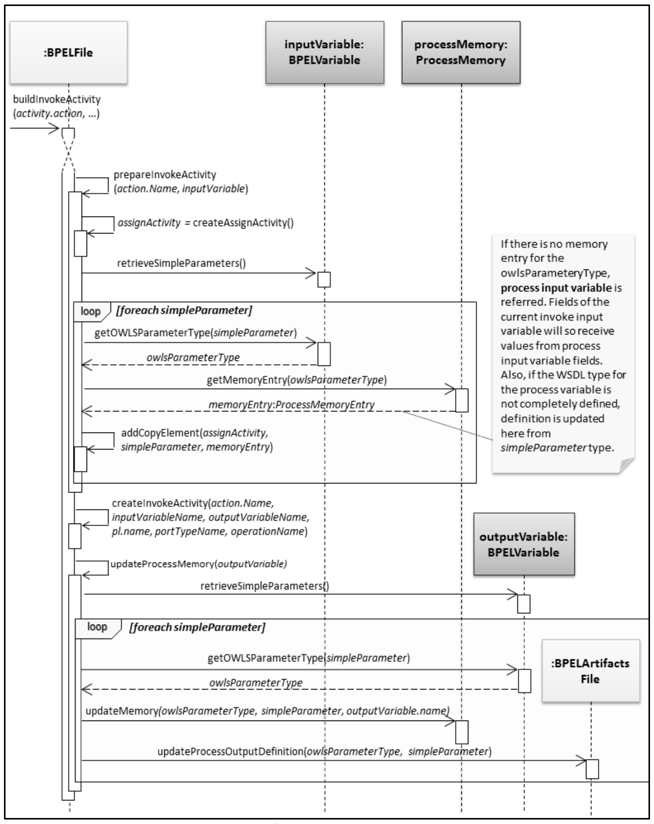
When a new <*bpel:invoke*> is processed (*browseActivity(…)* in Fig. 13), the *retrieveInMSGStructure(…)* and *retrieveOut-MSGStructure(…)* methods are used to retrieve the XML Schema structure of the input and the output variables respectively, from the WSDL description of the service operation to invoke. Then, a new couple of variables with the retrieved type structure are added to the WS-BPEL process as global variables (*addVariable(…)* method) and referred inside the *inputVariable* and *outputVariable* fields of the <*bpel:invoke*> specification.

In order to properly build the data flow, the input variable of the invoke activity has to be initialized by using a <*bpel:assign*> activity, placed just before the <*bpel:invoke*>. We call this step *prepareInvokeActivity(…)* (Fig. 14).

The input variable may in general be complex and the values of the composing elements can come from both the process input variable or the output of any previously executed activity.

By knowing the structure of the input variable (*inMsgStructure* in Fig. 13), it is possible to retrieve, for each simple type parameter of the service input message, its semantic description (the *process:parameterType*, i.e. an ontological concept) by using the message map in the grounding section of the OWL-S service description. The *BPELVariable* class is an abstraction of the concept of BPEL variable and contains both the structure of the complex type variable (e.g. *inMsgStructure*) and the mapping between each simple field of the complex variable and the corresponding ontological concept (e.g. *owlsWSDLInMSGMap*).

A process memory data structure (*ProcessMemory* in Fig. 13, hold by the *BPELFile* object) has been used to progressively store, for each different ontological concept discovered during the generation of the <*bpel:invoke*> activity, the references to the BPEL global variable and the specific simple type field from a complex WSDL message containing the most updated value for that parameter type. This information can be used by the *addCopyElement(…)* method as the source value for a <*bpel:copy*> inside the <*bpel:assign*> prepare activity for the corresponding input variable field of the <*bpel:invoke*> activity, used as the <*bpel:copy*> destination.


:BPELFile
inputVariable: BPELVariable
processMemory: ProcessMemory
buildInvokeActivity (activity.action, ...)
prepareInvokeActivity (action.Name, inputVariable)
assignActivity = createAssignActivity()
retrieveSimpleParameters()
loop [foreach simpleParameter]
getOWLSParameterType(simpleParameter)
owlsParameterType
getMemoryEntry(owlsParameterType)
memoryEntry:ProcessMemoryEntry
addCopyElement(assignActivity, simpleParameter, memoryEntry)
createInvokeActivity(action.Name, inputVariableName, outputVariableName, pl.name, portTypeName, operationName)
outputVariable: BPELVariable
updateProcessMemory(outputVariable)
retrieveSimpleParameters()
loop [foreach simpleParameter]
getOWLSParameterType(simpleParameter)
owlsParameterType
:BPELArtifacts File
updateMemory(owlsParameterType, simpleParameter, outputVariable.name)
updateProcessOutputDefinition(owlsParameterType, simpleParameter)

If there is no memory entry for the owlsParameteryType, **process input variable** is referred. Fields of the current invoke input variable will so receive values from process input variable fields. Also, if the WSDL type for the process variable is not completely defined, definition is updated here from *simpleParameter* type.

Figure 14. Generation of an <*invoke*> activity, details on the *prepareInvokeActivity(…)* operation

It is important to consider that, when an ontological concept is encountered for the first time (no entry is defined in the *ProcessMemory* for a specific input field ontological description), the reference is assumed to be retrieved from the process input variable (the variable specified in the first <*receive*> activity of the process). Also, since the concrete structure of the business process input variable could still be not completely defined (in the bpel artifacts), the knowledge of the destination variable type (i.e. the input variable of the service to be invoked) can be used to complete this definition, updating the artifacts file.

After this last step, the specification of the <*bpel:invoke*> is complete and the *ProcessMemory* can be updated with the structural information coming from the output variable used for the external Web service invocation. This way, the updated memory entry can be used for any next <*bpel:invoke*> generation working on the same ontological data (*process:parameterType*).

It is worth to note that the proposed approach requires a unique semantic characterization of each message field used by the services: if a service message contains multiple fields referring to the same conceptual data, the semantic description should distinguish the two fields anyway (e.g. referring to the ordering of the fields to distinguish them as related to two different concepts). This is required both to have a correct plan generation with a STRIPS-like planner (like Graphplan) and to correctly reconstruct the data flow during the WS-BPEL process generation.

The *updateProcessOutputDefinition(…)* is required to complete the structure definition of the process output variable, used inside the last <*bpel:reply*> process activity.

4. AN APPLICATION EXAMPLE

To evaluate the potential of the tool, we tested automatic service composition on the specific application scenario (*population alert*) introduced in Section 1.

4.1 Domain description and problem solution

The proposed scenario is about the handling of a hydrogeological disaster (e.g. extreme rain, flooding, inundations, etc.) that strikes a human community (city, town, rural village, etc…). Our tool can be used to plan emergency management flows of actions to be executed by a standard workflow engine.

Disaster response management is a particularly meaningful test bed for the tool since action flows must be timely planned and executed. Such actions may be concerned with the use of specific resources (such as telecommunication facilities) and/or the coordination of static and mobile resources (volunteers, policemen, ambulances).

Planning has to take in account resources capabilities, availability and readiness. To this end, the proposed tool is able to generate, automatically, quickly and almost effortlessly, all the planning processes needed. E.g. variations in services availability in the domain, changes in the state of resources, changes in the overall goal, can be immediately considered to get updated plans.

The specific example is about population alert by using multiple communication channels (mobile networks, SMS, MMS, automatic calls, TV, etc.). To design a realistic scenario, we have derived it from the analysis of real disaster management plans defined by the organization and emergency procedures of the Italian *Protezione Civile*, the national body in charge of prevention and management of disaster events. The *Protezione Civile* adopts a specific model (“metodo Augustus”); such model emphasizes operational flexibility by using to the largest possible extent resources located close to the emergency, and involving all organizations (institutional or not) that can be useful in the specific situation.

The assumption is that such cooperating organizations would have made some or all of their disaster management resources or capabilities available as web services. Such web services could be used to access resources, to get info, to alert volunteers. Each service would have a WSDL and an OWL-S semantic description of its behavior. OWL-S description refers to a general domain ontology that describes the emergency context.

We considered as cooperating organizations local police, fire brigade, telecom companies, white pages, register offices, community volunteering etc. For each considered organization, we defined a specific OWL domain ontology to classify the concepts involved and their inner relationships. The whole set of domain services refer to several dozens of entities. Some concepts included in the ontology are: *citizen*, *address*, *personal data*, *message*, *mobile or fixed-line telephone number*, *deliver status of a message*, etc. The semantic OWL-S descriptions of each web service refer to concepts described in the defined domain ontology. Some examples of the domain services are:

- *register office service*, to get personal data of the people that must be warned;
- *white pages service*, to get personal data of people that could

be involved in the alert process;

- *SMS/MMS broadcast* (telecom companies) to alert people in a specific neighborhood;
- *SMS/MMS send* to specific people directory;
- *send phone calls*, using prerecorded messages or via human call center (telecom companies);
- *alert local police or fire brigade*,

In Fig. 15, we report a representation of OWL-S ontology for the "Send SMS" service, which is the archetype of a service made available by a mobile telecom company to send SMS to a list of users.

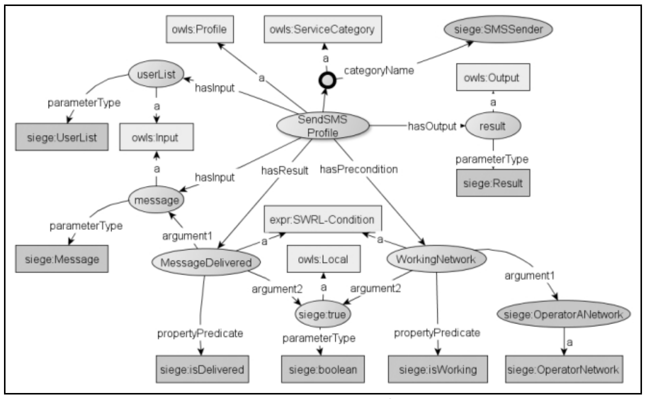


Figure 15. Ontological description of the service SendSMS

OWL-S descriptions are converted into PDDL, the internal language used by the PDDL4J planner. In Table 2, we report a sample of PDDL translation, related to the *Send SMS* service. By using the backward strategy of the Graphplan algorithm, if the problem goal is equal to (or contains in conjunction with other predicates) the effects of the SendSMS service, the PDDL4J planner may include the above PDDL action in the plan and try to reach the specified precondition through a single service, or a service chain, beginning with the specified initial state.

As the aim of the whole scenario is about alerting the population living in a specific area, the goal includes the following post-conditions: to get home phone numbers of all people living in the area and to alert them by phone calls, by SMS, using TV channel, etc.

Table 2. PDDL code generated by the PDDL Domain serializer

```
(:action SendSMS
    :parameters (?userList ?message ?result)
    :precondition (and (hasKnowledge ?userList)
                       (hasKnowledge ?message)
                       (isWorking OperatorANetwork))
    :effect       (and (hasKnowledge ?result)
                  (isDelivered ?message))
)
```

The resulting plan process includes several services and three different parallel branches. The three branches contain the following actions:

1. Alert using land line phones:
   - get the list of home telephones in the area using a White Pages service;
   - give such phone numbers to a call center for automatic call procedure;
   - get a list of citizens without home phone;
   - send the list of citizens that were not warned (as they do not have home phone or because they did not answered the call) to local police.
2. Broadcast an alert message using a specific service made available by TV corporations.
3. Alert using SMS channel:
   - use a service made available by the birth register to get the names of all people living in the area;
   - get mobile numbers (MSISDN) from telecom companies;
   - send SMS to everyone.

Fig. 2, which has been already discussed in Section 1, is the graphical representation of the solution (abstract) plan produced by the Graphplan component, which includes ten services (*WPGetUsers*, *GetHomePhoneUsers*, *GetNoHomePhoneUsers*, *SendToCallCenter*, *WPGetUsersNames*, *SendToPolice*, *EmergencyTVChannel*, *RegisterGetUsers*, *GetMsisdn*, *SendSMS*) to perform the activity described above.

In Fig. 16, an excerpt of the corresponding WS-BPEL process (concrete plan) automatically generated by the WS-BPEL serializer is shown.

```
<bpel:process ... name="alertingPopulationPlan" targetNamespace=
      "http://ing.unisannio.dslab.it/bpel/alertingPopulationPlan">
  ...
  <bpel:partnerLinks>...</bpel:partnerLinks>
  <bpel:variables>...</bpel:variables>
  <bpel:sequence name="sequence">
    <bpel:receive name="receiveInput" partnerLink="client"
      portType="tns:alertingPopulationPlanPortType"
      operation="alertingPopulationPlanOperation" variable="input" createInstance="yes" />
    <bpel:flow>
      <bpel:sequence>
        <bpel:assign name="WPGetUsersServicePrepareActivity" validate="no">...</bpel:assign>
        <bpel:invoke name="WPGetUsersService" partnerLink="WPGetUsersServicePL"
             portType="ns4:WPGetUsersService"
             operation="WPGetUsers" inputVariable="WPGetUsersRequestVariable"
             outputVariable="WPGetUsersResponseVariable" />
        <bpel:flow>
          <bpel:sequence>
            <bpel:assign name="GetHomePhoneUsersServicePrepareActivity" validate="no">...
             </bpel:assign>
            <bpel:invoke name="GetHomePhoneUsersService" partnerLink=
                   "GetHomePhoneUsersServicePL" portType="ns5:GetHomePhoneUsersService"
                   operation="GetHomePhoneUsers"
                   inputVariable="GetHomePhoneUsersRequestVariable"
                   outputVariable="GetHomePhoneUsersResponseVariable" />
            <bpel:assign name="SendToCallCenterServicePrepareActivity" validate="no">...
            </bpel:assign>
            <bpel:invoke name="SendToCallCenterService" partnerLink=
                    "SendToCallCenterServicePL"
             portType="ns6:SendToCallCenterService" operation="SendToCallCenter"
                   inputVariable="SendToCallCenterRequestVariable"
                   outputVariable="SendToCallCenterResponseVariable" />
          </bpel:sequence>
          <bpel:sequence>
            ...
            <bpel:invoke name="GetNoHomePhoneUsersService" ... />
          </bpel:sequence>
        </bpel:flow>
        ...
        <bpel:invoke name="WPGetUsersNamesService" .../>
        ...
        <bpel:invoke name="SendToPoliceService" ..."/>
      </bpel:sequence>
      <bpel:sequence>
        ...
        <bpel:invoke name="RegisterGetUsersService" ... />
        ...
        <bpel:invoke name="GetMsisdnService" ... />
        ...
        <bpel:invoke name="SendSMSService" ... />
      </bpel:sequence>
      <bpel:sequence>
        <bpel:assign name="EmergencyTVChannelServicePrepareActivity" validate="no">...
        </bpel:assign>
        <bpel:invoke name="EmergencyTVChannelService" .../>
      </bpel:sequence>
    </bpel:flow>
    <bpel:assign name="replyOutputPrepareActivity" validate="no">...
    </bpel:assign>
    <bpel:reply name="replyOutput" partnerLink="client"
      portType="tns:alertingPopulationPlanPortType"
     operation="alertingPopulationPlanOperation" variable="output" />
  </bpel:sequence>
</bpel:process>
```

Figure 16. An (excerpt of) automatically generated WS-BPEL process for the alert workflow

The auto-generated code in Fig. 16 contains a main sequence composed of a receive activity, a flow, a variable assignment and a reply activity, coherently with the considerations about control and data flows generation reported in Section 3.3. The first receive activity (*receiveInput*) creates a new instance of the business process and stores the request message coming from the invoker in the *input* variable, declared in the <*variables*> section. The type of the variable is specified in the *.wsdl* artifacts file associated to the bpel file, not reported due to space limitations.

The variable is composed of the destination area and the message text, among other fields.

The final reply activity (*replyOutput*) returns a response message to the invoker, whose fields (in this case just the outcome of the alert) are properly prepared by the assign activity preceding it (*replyOutputPrepareActivity*). The middle flow contains the <*invoke*> activities required to interact with external services, implementing the abstract tasks in Fig. 2. Coherently with the three-branches structure of the plan, the BPEL flow is composed of three sequences. One of them simply invokes a service for alerting people by a TV message (*EmergencyTVChannelService*). Another sequence contains the chain of service invocations for getting information about people living in the area (*RegisterGetUsersService*), retrieving their mobile numbers (*GetMsisdnService*) and sending SMS to them (*SendSMSService*). The last sequence of the flow is more complex, made up of an invocation for the *WPGetUsersService* followed by a flow (containing the sequence of *GetHomePhoneUsersService* and *SendToCallCenterService* in parallel with *GetNoHomePhone-UsersService*) and a final invocation for the *SendToPoliceService*. It is worth to note that each <*invoke*> activity is preceded by an <*assign*> activity, aimed at preparing the input message of the service to invoke.

Some limitations in the auto-generated BPEL code of Fig. 16 derive from the assumptions described in Section 3.3. A synchronous request-response message exchange pattern is used for invoking external services from the BPEL process (each <*invoke*> has both the input and the output variables specified). Also, the same synchronous model applies to the interactions between the BPEL business process and its invokers (the process starts with a receive and terminates with a reply activity). Correlation sets are not addressed in the current implementation of the tool and the generated flow is related to a normal execution flow. Finally, absence of structural mismatches among services with the same semantic input/output characterization is assumed, when generating the code by exploiting the binding information in the OWL-S groundings. These limitations are intended to be overcome as future work, for example by means of mediation services (for structural mismatches) and callback mechanisms (for also supporting asynchronous service interactions).

4.2 Performance analysis

Execution times for solving the previously described "population alert" composition problem are reported in Fig. 17. It is worth to note that performance is not a critical aspect in this paper, since our focus is mainly on the problems related to automating the generation of an executable service composition from a set of semantic service descriptions. Architectural issues and flexibility of the proposed solution have been considered as the main drivers for tool implementation instead of performance or other non-functional criteria. Also, the tool described in this paper is still a prototypal implementation, which has been useful to demonstrate feasibility and utility of the proposed approach. Nevertheless, the performance measures acquired in our analysis have confirmed the potential of the tool, which can be effectively and efficiently used to support designers in service composition and to implement adaptive replanning, especially when the service domain is not particularly large.

The "population alert" problem has been solved in ten test cases, each characterized by a different size of the service domain, i.e. a different number of OWL-S semantic descriptions available for solving the problem (from 10 to 100 descriptions, by tens). For each of the test cases considered, the ten services for composing a possible solution to the problem (see Section 4.1 and Fig. 2) were included in the domain and our tool was always able to find the correct composite solution. Also, the BPEL generator was able to retrieve a complete WS-BPEL process (.bpel file together with the other files required to deploy it), which was correctly executed on the RiftSaw workflow engine.

Total execution time is the sum of four main contributions: (1) *domain conversion time*, (2) *problem conversion time*, (3) *planning time* and (4) *WS-BPEL generation time*. *Domain conversion time* represents the time needed to access, parse and convert to PDDL each semantic service description, referred by a URI from the set of the OWL-S files constituting the planning domain; *problem conversion time* is the time required to access, parse and convert to PDDL the OWL-S description of the problem to solve; *planning time* is the time required for Graphplan to find a solution to the PDDL problem in the PDDL domain; *WS-BPEL time* is the time required to produce the set of files making up a complete and executable BPEL process from the abstract plan and the WSDL files referred in the OWL-S grounding section, by using the BPEL serializer.

The four time contributions have been measured by using system time, with nanosecond resolution. The machine used for executing the test cases was an Intel Core 2 Duo CPU, with 3 GB RAM, running a Linux Debian distribution. Semantic descriptions and WSDL files were deployed locally to the machine on an Apache server.

The total execution time and the four contributions that compose it are reported, on a log scale, in Fig. 17(a). The points in the figure are the median values obtained after 100 iterations for each test case. Also, 10 initial iterations per test were executed, whose times have not been accounted in order to evaluate tool performance during a steady-phase.

This way, the influence over measured times of dynamic class loading, Java *Just-In-Time* (re-)compilations and other cold-start overheads from the Java Virtual Machine has been reduced. For each point in Fig. 17, observed minimum and maximum values are depicted as error bars around the median value.

As expected, the predominant contribution to the total execution time is *domain conversion time*, since it includes the access to a set of semantic descriptions and ontologies via an HTTP server (local to the testing machine) and their parsing by means of the OWL-S API (v. 2.0).

In relation to *problem conversion time* and *BPEL generation time*, we have observed an approximately constant time, since, in the case of OWL-S problem conversion, one description has to be converted into PDDL, while, in the BPEL case, the found solution plan to convert to BPEL is the same for all the test cases.

Finally, as concerning planning time, Graphplan takes an approximately exponential trend with respect to the number of domain services, as shown in Fig. 17(b). However, on small/medium-size domains, Graphplan exhibits very low time overhead to compute solutions, even lower than 100 ms, due to its efficiency and the fact it works on memory structures only.

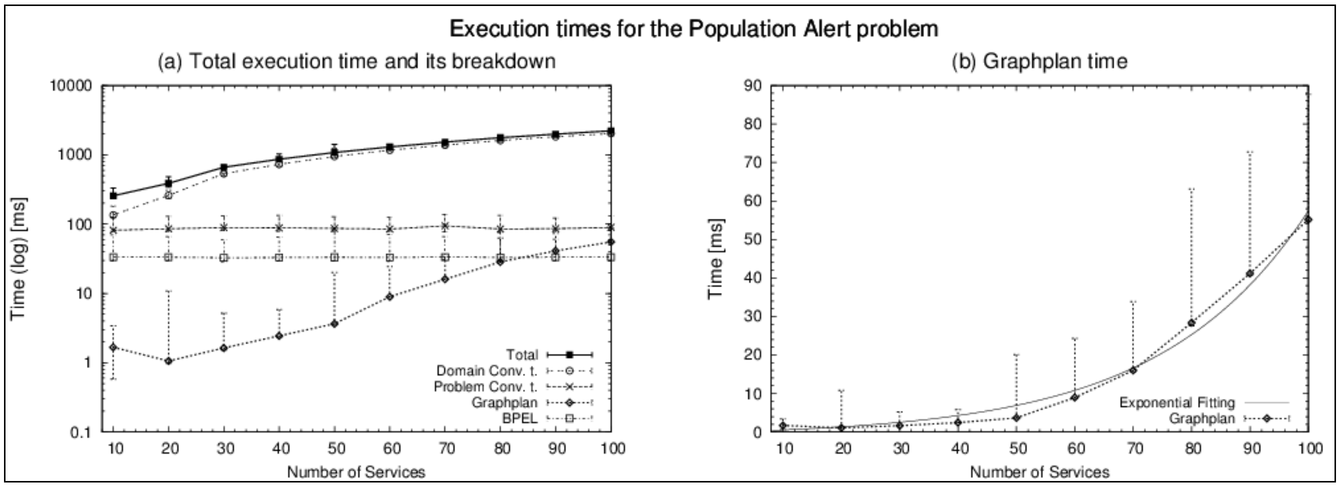


Figure 17. Time analysis for the population alert problem

5. CONCLUSION AND FUTURE WORK

This paper proposes a tool for automatic composition of OWL-S semantically annotated web services. Executable compositions are automatically generated by the tool in either WS-BPEL or XPDL languages, referencing concrete WSDL services retrieved from the OWL-S groundings.

OWL-S descriptions of both services and problem are analyzed, converted into PDDL and fed as input to the Graphplan planner PDDL4J. Solution plans are translated into WS-BPEL or XPDL and can be executed by any common business process execution engine (like RiftSaw). An application scenario about the partial handling of hydrogeological disasters has been defined and used for testing the potential of the presented tool. Performance analysis has confirmed the efficiency of the tool, showing execution times in the order of few seconds, in the case of small/medium-size domains.

Instead of optimizing the tool for planning complete workflows, we have chosen to take advantage of it for re-planning (parts of) already defined (autonomic) workflows, when some activities of the original plan are not available and equivalent sub-processes are required to replace them.

Despite of its robustness and usefulness, the tool is still prototypal and further improvements are possible. Among these, current domain services are provided as input by means of a set of known OWL-S descriptions, instead we have planned to introduce more flexibility by integrating the tool with a registry. The registry should be able to automatically retrieve a set of candidate domain services matching the ontology concepts referred within the OWL-S problem specification. The registry can also be used for retrieving groundings of semantic services to concrete services to be referenced in XPDL or WS-BPEL descriptions.

We also aim at improving the data and control flow generation, by introducing support for finer-grained synchronization mechanisms, like WS-BPEL *<link>*, *<source>* and *<target>* in *<flow>* activities. However, new semantic constructs to semantically specify such synchronization requirements in the OWL-S descriptions need to be investigated.

Moreover, during XPDL or WS-BPEL generation, concrete services may ground same ontological concepts to different concrete data types, generating data mismatches when they are included within the same solution plan and present data dependencies over the differently grounded concepts. A possible solution consists of providing a set of Web service data adapters to the composer, to perform data conversions between different representations: they can be automatically selected and interposed between the mismatching services. Finally, when no semantically exact solution is available in the service domain, semantically relaxed plans for partial goal satisfaction could be proposed and ranked.

ACKNOWLEDGEMENTS

The work described in this paper has been partly developed during SIEGE [31], a research project co-funded by the Italian Ministry of Research, which adopted the proposed tool to build specific workflows of services in an e-Government context. We thank Roberto Pratola and Ivano De Furio for their contributions in implementing and validating the tool.

REFERENCES

[1] G. Tretola, E. Zimeo, "Autonomic Internet-scale Workflows", in Proceedings of the 3rd International Workshop on Monitoring Adaptation and Beyond, ACM New York, USA, 2010,

[2] D. L. McGuinness, F. van Harmelen "OWL: Ontology Web Language," W3C Recommendation, [Online] http://www.w3.org/TR/owl-features/, 2004.

[3] D. Martin, M. Burstein, and J. Hobbs et al, "OWL-S: Semantic Markup for Web Services," W3C Member Submission, [Online] http://www.w3.org/Submission/OWL-S/, 2004.

[4] J. Farrell, H. Lausen, "Semantic Annotations for WSDL and XML Schema," W3C Recommendation, [Online] http://www.w3.org/TR/sawsdl/, 2007.

[5] D. Fensel, F. Fischer, and J. Kopecký et al "WSMO-Lite: Lightweight Semantic Descriptions for Services on the Web," W3C Member Submission, [Online] http://www.w3.org/Submission/2010/SUBM-WSMO-Lite-20100823/, 2010.

[6] J. Rao and X. Su, "A Survey of Automated Web Service Composition Methods," in Proceedings of the First International Workshop on Semantic Web Services and Web

Process Composition, SWSWPC 2004, San Diego, California, USA, July 6th, 2004.
[7] S. Dutsdar and W. Schreiner, "A Survey on Web Service Composition," in International Journal of Web and Grid Services, vol. 1, pp. 1-30, August 2005.
[8] Z. Li, L. O'Brien, J. Keung and Xi Xu, "Effort-Oriented Classification Matrix of Web Service Composition," in Proceedings of the 2010 Fifth International Conference on Internet and Web Applications and Services (ICIW), pp. 357 - 362 , May 2010.
[9] B. Medjahed, A. Bouguettaya, A. K. Elmagarmid, "Composing Web Services on the Semantic Web," in the VLDB Journal vol. 12 no. 4, November 2003.
[10] M. Ghallab, A. Howe, C. Knoblock, and D. McDermott, et al, "PDDL: The Planning Domain Definition Language," in AIPS98 planning committee (1998), Volume: 78, Issue: 4, Publisher: Citeseer, Pages: 1-27 DOI: 10.2307/3729517.
[11] D. S. Nau, T. C. Au, O. Ilghami, U. Kuter, J. W. Murdock, D. Wu and F. Yaman "SHOP2: An HTN Planning System," in Journal of Artificial Intelligence Research, Vol. 20, pp.379-404, 2003.
[12] M. Klusch, A. Gerber "Semantic Web Service Composition Planning with OWLS-XPlan," in Proceedings of the 1st Int. AAAI Fall Symposium on Agents and the Semantic Web, 2005.
[13] M. Klusch, K. U. Renner, "Fast Dynamic Re-Planning of Composite OWL-S Services," in Proceedings of 2nd IEEE Intl Workshop on Service Composition (SerComp), IEEE CS Press, Hongkong, China, 2006.
[14] A. Blum, M. Furst, "Fast Planning Through Planning Graph Analysis," Journal of Artificial Intelligence, vol. 90, pp. 281-300, 1997.
[15] O. Hatzi, D. Vrakas, N. Bassiliades, D. Anagnostopoulos, I. Vlahavas, "Semantic Awareness in Automated web service Composition through Planning," in 6th Hellenic Conference on Artificial Intelligence (SETN 2010), Springer, LNCS Vol. 6040, Athens, Greece, May 2010.
[16] S. A. McIlraith, T. C. Son, "Adapting Golog for Composition of Semantic Web Services," D. Fensel, F. Giunchiglia, D. L. McGuinness, and M.-A. Williams, editors, KR, pp. 482-496, 2002.
[17] M. Phan, F. Hattori, "Automatic Web Service Composition Using ConGolog," in ICDCS Workshops, p. 17, 2006.
[18] F. Lecue, A. Leger, A. Delteil, "DL Reasoning and AI Planning for Web Service Composition," in Web Intelligence IEEE 2008, pp. 445-453, 2008.
[19] H. Zhao, P. Doshi, "Haley: An End-to-End, Scalable Web Service Composition Tool: A Hierarchical Framework for Logical Composition of Web Services," in Proceedings of IEEE International Conference on Web Services, ICWS07, Salt Lake City, Utah, 2007.
[20] H. Zhao and P. Doshi, "Haley, A Hierarchical Framework for Logical Composition of Web Services," in Service Oriented Computing and Applications, pp. 285-306, 2009.
[21] OMG, "Business Process Modeling Notation (BPMN) Specification v. 2.0,", [online] http://www.omg.org/spec/BPMN], 2011.
[22] WfMC, "XML Process Definition Language (XPDL) v. 2.1", [online] http://www.wfmc.org/xpdl.html, 2008.
[23] OASIS Standard, "Web Service Business Process Execution Language (WS-BPEL) Specification 2.0," [online] http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html], 2007.
[24] I. Horrocks, P. F. Patel-Schneider, and H. Boley et al, "SWRL: A Semantic Web Rule Language Combining OWL and RuleML," W3C Member Submission, [online] http://www.w3.org/Submission/SWRL/, 2004.
[25] Jboss Community, http://www.jboss.org/riftsaw, v. 2.3.0, 2011.
[26] The Apache Software Foundation, http://ode.apache.org/, v. 1.3.5, 2011.
[27] M. Polese, G. Tretola, E. Zimeo, "Self-Adaptive Management of Web Processes," in Proceedings of the IEEE International Conference on Web Systems Evolution (WSE), 2010.
[28] D. Pellier, "PDDL4J", [Online] http://sourceforge.net/projects/pdd4j/, 2011.
[29] OW2 Consortium, "EasyWSDL toolbox" [Online] http://easywsdl.ow2.org/
[30] W3C, "XML Schema" [Online] http://www.w3.org/XML/Schema
[31] S. Pierno, L. Romano, L. Capuano, M. Magaldi, L. Bevilacqua, "Software Innovation for E-Government Expansion," in Move to Meaningful Internet Systems: OTM 2008, pp. 822-832, 2008.

BIOGRAPHIES

Luca Bevilacqua, got his degree in Electronic Engineering in 1989, and his Master in Business Administration Degree in 1992. He has been working in IT leading companies (Olivetti, Sema, Atos Origin, Engineering Ingegneria Informatica). Since the year 2000 he has been responsible for several R&D projects dealing with topics such as web services and their orchestration, advanced user interfaces (multimedia, perceptual, affective). Today he works in Engineering group, as R&D Director, for all R&D projects co-funded by Italian public funds/bodies.

Angelo Furno is a Ph. D. Student in the Department of Engineering at the University of Sannio (Italy). His research interests are in the fields of software engineering and distributed computing systems, with special emphasis on SOA-based systems. Automatic service composition, self-adaptive workflow systems, semantic Web Services, Context-aware, P2P, and cloud computing are the main specific areas of his current research. He received his master degree at University of Sannio in 2010 and was a research assistant at the same university before joining the Ph.D. program.

Vladimiro Scotto di Carlo had is degree in Physics in 1998. Since 2005 he has been working as researcher in IT companies (Atos Origin, Engineering Ingegneria Informatica). His main interests are in the area of advanced user interfaces. In recent years he has had the role of technical coordinator in R&D projects about natural language interpretation, multimodal interface in mobile devices and innovation for e-Government.

Eugenio Zimeo received the PhD degree in Computer Science from the University of Naples in 1999. Currently he is an assistant professor at the University of Sannio in Benevento (Italy). His primary research interests are in the areas of software architectures, frameworks and middleware for distributed systems, service oriented, grid, cloud and P2P computing, autonomic computing, wireless sensor networks and mobile computing. He has published about 80 scientific papers in journals and conferences of the field and led many large research projects.