Chapter 11

Service Composition

Bastian Blankenburg, Luis Botelho, Fábio Calhau, Alberto Fernández, Matthias Klusch, Sascha Ossowski

11.1 Introduction

One of the striking advantages of Web Service technology is the fairly simple aggregation of complex services out of a library of other composite or atomic services. The same is expected to hold for the domain of Semantic Web Services such as those specified in WSMO or OWL-S. The composition of complex services at design time is a well-understood principle which is nowadays supported by classical workflow and AI planing based composition tools (cf. Chapter 4).

In CASCOM, we developed two composition planners for OWL-S services, OWLS-XPlan and MetaComp, together with an approach to heuristically pre-filtering the set of all available services which are delivered by the SMA (cf. Chapter 10) to significantly reduce the search space for both planners. Accordingly, the CASCOM service composition planning agent, called SCPA, can be configured to use one of the planners and either exploiting the pre-filtering module, or not.

This chapter is structured as follows. We briefly summarize the CASCOM composition planner agent SCPA, followed by the detailed description of the pre-filtering module, and both the OWLS-XPlan and MetaComp planning modules of the SCPA.

11.2 CASCOM Service Composition Agent SCPA

In CASCOM, two different Service Composition Agents (SCPA) have been developed which differ in the planning engine used: one SCPA is based on XPlan [11] while the other relies on SAPA [8]. In any case, the CASCOM SCPA takes a set of OWL-S services, a description of the initial state and the goal state to be achieved as input, and returns a plan that corresponds to a composite service that gets invoked using the FIPA-Request interaction protocol.

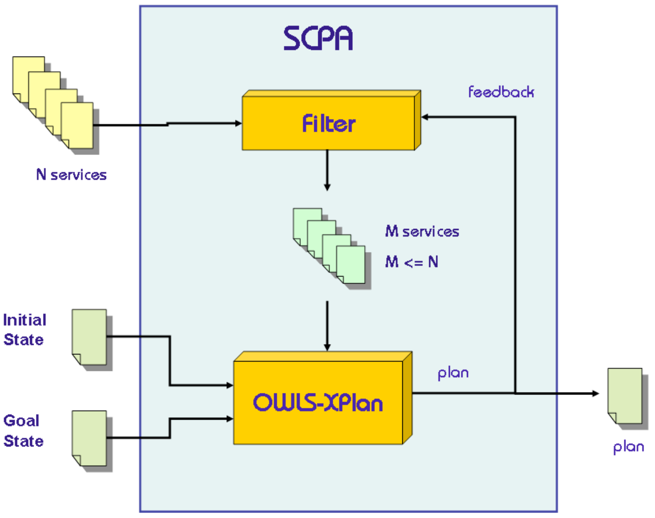


Figure 11.1: OWLS-Xplan Service Composition Agent

The first type of SCPA, called OWLS-XPlan SCPA, relies on the service composition planner OWLS-XPlan (cf. Section 11.4). Figure 11.1 shows its internal architecture, which also contains a pre-filter component which is detailed in Section 11.3. The OWLS-XPlan SCPA may be configured to apply this prefiltering component to the set of available OWL-S services returned by the SDA which, according to the CASCOM architecture (cf. Chapter 7), is in charge of retrieving services from accessible service directories. This reduction is expected to further increase the efficiency of the overall planning process. The final service composition plan is generated by the OWLS-XPlan planner component from the given set of services, the initial and goal state ontologies, and returned for execution and to its internal prefiltering component for experience based learning.

The MetaComp SCPA (cf. Section 11.5) uses the SAPA planner instead of the XPlan planner, and does not use the prefiltering component. In fact, MetaComp asks the SDA itself for a reduced number of services so that fewer service descriptions have to be conveyed between the two agents.

The availability of two different kinds of service composition agents in CASCOM provides potential clients with added flexibility to adapt composition planning to their individual needs. More concrete, the client agent can ask the context acquisition and management system (cf. Chapter 13) for the following context information regarding the two Service Composition Agents:

- Agents availability: if the agent is available (on-line) or not

- Average waiting time per request
- Service waiting list: number of requests waiting for a service
- Average execution time

In addition, the client agent might have built its own model based on past experiences or uses third party services (such as trust and reputation) for making its decision on the selection of the composition planner agent.

11.3 Pre-Filtering for Service Composition

According to the CASCOM Architecture, the SCPA (Service Composition Planning Agent) is in charge of creating a composite service that includes several pre-existing services. In order to be able to generate such a plan that matches the original query, the SCPA needs a set of input services to set out from.

Ideally, the set of services taken into account by the SCPA to create a composite plan should comprise all services registered in the directory. However, this can be impracticable as the number of services increases, as it is expected to occur in the open IP2P environments that CASCOM targets. To overcome that problem, CASCOM suggests to reduce the set of input services that are passed on to the SCPA's composition planning component. For this purpose, filters that sort out those services registered within the directories that are less relevant to the planning process are proposed. This activity is also called *plan based service matching* of a respective service matchmaker that is cooperating with a service composition planner like in CASCOM.

Pre-selecting the set of candidate services for composition planning is not an easy task. Several ad-hoc heuristics can be thought of (e.g. services that share at least one input or output with the query, etc). In this section a more informed method for filtering services that make use of *service class information* is proposed. First, a generic framework for service-class based filtering is described, and then it is instantiated for different filters on the basis of (a) organizational information obtained from the CASCOM role ontology and (b) the service category derived from the directory structure.

11.3.1 Generic Pre-Filtering Framework

At a high level of abstraction, the service composition planning problem can be conceived as follows: let $P = \{p_1, p_2, ..., p_m\}$ be the set of all possible plans (composite services) for a given service request R, and $D = \{s_1, s_2, ..., s_n\}$ the set of input services for the proper service composition planner (i.e. the directory available). The objective of a filter F is to select a given number l of services from D, such that the search space is reduced, but the best plan of P can still be found.

Put in another way: the larger the subset of plans $P' \subseteq P$ that the planner can choose from, the higher the probability that the plan of maximum quality is

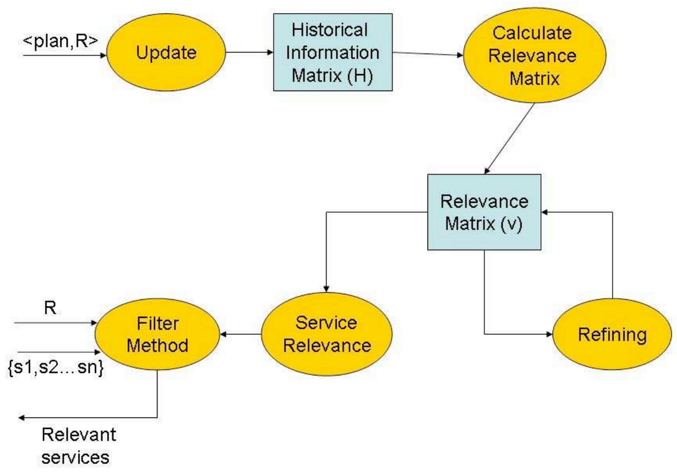


Figure 11.2: Architecture of the filter component

among them. A good heuristic to this respect is based on *plan dimension* and on the *number of occurrences* of services in plans: a service is supposed to be the more important, the bigger the number of plans from P that it is necessary for, and the shorter the plans from P that it is required for. This information can be approximated by storing and processing the plans historically created. So, in principle, matrices might store, for every possible query, the number of plans in which each service appeared, classified by each plan dimension.

However, it soon becomes apparent that the number of services and possible queries is too big to build up all matrices of the above type. Furthermore, the continuous repetition of a very same service request R is rather unlikely. And, even more important, this approach would not be appropriate when a new service request (not planned before) is required (which, in fact, is quite usual). To overcome this drawback, it is assumed the availability of *service class information*, so as to cluster services based on certain properties. If the number of classes is not too big, the aforementioned approach becomes feasible computationally.

Figure 11.2 depicts the structure of the CASCOM approach to service composition filtering. With each outcome of a service composition request, a Historical Information Matrix H is updated. Setting out from this information, a Relevance Matrix v is revised and refined. Based on this matrix, service relevance can be determined in a straightforward manner. For each service composition request, the filtering method is based on this estimated service relevance function.

H^R: Historical information about plans for service class R (Request)				
Dimension	1	2	3	...
# of plans	0	50	70	...
C_1	0	7	24	...
C_2	0	10	55	...
C_3	0	30	21	...
...	...	...	...	...

Table 11.1: Example of class information about historical plans

Computing Pre-filter Information

The *Historical Information Matrix* (Table 11.1) for a service class r compiles relevant characteristics of plans (composite services) that were created in the past in response to requests for services belonging to that class. In particular, for each plan dimension i and service class s it stores the number of plans of length i that made use of services of class r. Historical Information Matrices are updated as newly generated plans come in. If the service request is a logical formulae (given in disjunctive normal form), the contribution of the resulting plan is *distributed* among the affected Historical Information Matrices.

As commented above, the aim of ranking services is to try and select a set of services that cover the largest subset of the plan space, as an attempt to maximise the chance of the best plan to be contained in it. Services that formed smaller plans in the past are considered more relevant, since it is easier to cover small plans that large ones, so with less services more plans can be covered.

The Relevance Matrix specifies the relevance of a service class s to be part of a plan (composite service) that matches the query for a certain service class r. The following function is used to aggregate the information about plans contained in the *Historical Information Matrixes* (remember that all this information is about a single request class r):

$$Relevance(C, R) = \frac{\sum\limits_{d=1}^{m} \frac{n_d}{d^c}}{\sum\limits_{d=1}^{m} \frac{N_d}{d^c}} \tag{11.1}$$

where d is the dimension of the plan, m is the dimension of the longest plan stored, n_d is the number of times that C was part of a composite plan of dimension d for the request R, and N_d is the total number of plans of dimension d for that request. Note that each appearance of class C in a plan contributes to the relevance value, and that this contribution is the higher the smaller the plan dimension. c is a constant > 0 that allows adjusting the level of importance of plan dimensions. A relevance value between 0 and 1 is obtained with this calculus for every given service class C with respect to the composition of a service of class R.

The *Relevance Matrix* $v(s, r)$ can be further refined in order to take transitivity into account. Consider the following situation: A plan that achieves C_1 is searched for, and that a potential solution is to compose the services C_2 and C_3 ($C_2 \oplus C_3$ for short). However there is no service provider for C_3, but instead C_3 can be composed as $C_4 \oplus C_5 \oplus C_6$, so the final plan is $C_2 \oplus C_4 \oplus C_5 \oplus C_6$. Unfortunately, the value $v(C_4, C_1)$ is low and the service providing C_4 is discarded and not taken into account in the planning process, so the aforementioned plan cannot be found by the planner. Therefore, the relevance matrix is refined by taking *transitivity* into account, e.g. through the following update: $v(C_4, C_1) = v(C_4, C_3) \cdot v(C_3, C_1)$. The same holds for third-level dependencies (e.g.: $v(C_7, C_1) = v(C_7, C_4) \cdot v(C_4, C_3) \cdot v(C_3, C_1)$). This example motivates the definition of the $v^k(s, r)$ as a k step relevance matrix

$$\begin{aligned} &v^1(s,r) = v(s,r) \\ &v^k(s,r) = Max(v^{k-1}(s,r), v^{k-1}(s,s_1) \cdot v^{k-1}(s_1,r), \\ &\qquad\qquad v^{k-1}(s,s_2) \cdot v^{k-1}(s_2,r), ..., v^{k-1}(s,s_n) \cdot v^{k-1}(s_n,r)) \end{aligned} \tag{11.2}$$

As shown in the equation, the product is used as combination function and the maximum to aggregate the results. Note that the higher the value of k the better the estimation of the relevance of service classes. The refinement of the relevance matrix is repeated until it converges or until a timeout is received. The elevated time complexity of $O(n^3)$ for each refinement step is attenuated by the *anytime properties* of the approximation algorithm. Furthermore, recall that the number of classes n is supposed to be fixed and not overly high. Finally, note that several updates and refinements can be combined into a "batch" to be executed altogether when the system's workload is low.

There are several ways of obtaining the initial relevance matrix. If there are historical records of plans they can be used to calculate the matrix. Also, an a priori distribution can be assigned using expert (heuristic) knowledge. Still, the simplest solution is to let the service composition planning component work without filtering services until the number of plans generated is representative enough to start computing and refining the matrixes.

Service Relevance Calculus

The first step to calculate the *relevance* of a service s for a request r is the mapping of both to *classes of services*. Then, the relevance between the classes is calculated. $v(s, r)$ is used to represent the relevance of *class* s for the *class* r in the request, and $V(S, R)$ as the relevance of service S for the service request R.

Considering that, in general, the service S belongs to several classes (s_1,s_2, ..., s_n), if a request R only includes a class (r) in its description, then

$$V(S,R) = \max(v(s_1,r), v(s_2,r), ..., v(s_n,r))$$

However, if the request specifies a logical expression containing several classes of services ($r_1, r_2, ..., r_m$), logical formulas are evaluated using the *maximum* for disjunctions and the *minimum* for conjunctions; and inside the *maximum* is used to aggregate the service classes specified by the provider. For example, if the request R includes the formula $r_1 \vee (r_2 \wedge r_3)$, and the service S belongs to the classes s_1 and s_2, the calculus is as follows:

$$V(S,R) = \max[\max(v(s_1,r_1), v(s_2,r_1)), \min(\max(v(s_1,r_2), v(s_2,r_2)), \max(v(s_1,r_3), v(s_2,r_3)))]$$

Types of Pre-filter Composition

When a service request is analysed by the pre-filter, the set of services are first ranked by an estimation of the relevance of the service class for that request. Then, only the services belonging to the best ranked classes are passed on to the planner. In order to determine the concrete services that pass the filter three major options are considered:

a) To establish a *threshold* and filter out those services whose classes have a degree of relevance lower than that threshold.

b) To return the estimated *k best* services based on the relevance of their corresponding classes. In this case the number of services that pass the filter is pre-determined.

c) To return a *percentage* of the original set of services (based on the relevance of their corresponding classes). In this case the number of services considered in the planning process depends on the directory size.

When designing the algorithms corresponding to these filters configurations, an additional problem needs to be taken into account. Services with low (or even zero) relevance values would never be considered for planning, so they could never be part of a plan (composite service), remaining with low relevance forever. This is obviously too restrictive, as our relevance values are only estimations based on the information available at some point in time. To overcome this some services are allowed to be fed into the planner even though they are not supposed to be relevant enough according to the filter policy. Those additional services are chosen randomly. This random option is combined with the three aforementioned filter types to allow for an exploration of the service (class) space.

11.3.2 Instantiation of Pre-Filters

In the following we present two different approaches to apply the filtering framework proposed in this section. For each approach the mapping of services to classes is defined. Both methods are based on information available in the OWL-S service descriptions used by CASCOM.

Role-based Pre-filtering

In many service-oriented systems, agents are conceived as mere wrappers for Web Services. However, agents are not only able to execute a service but may also engage in different *types of interaction* related to that service, in the course of which they play several *roles.* For example, in a medical emergency assistance scenario, an agent providing a *second opinion* service should not only be able to provide a diagnostic; it may also be required to explain it, give more details, recommend a treatment, etc. Therefore, a service provider may need to engage in several different interactions, and play a variety of different *roles*, during the provision of a service.

Our role-based filtering method relies on taxonomies of roles and type of interactions (see Figure 10.16 in Chapter 10) to determine service classes. The idea is to relate roles searched in the query to roles played by agents in the composite service, that is, which are the roles typically involved in a plan when a role r is included in the query. For example, it is common that a *medical assistance* service include *travel arrangement*, *arrival notification*, *hospital log-in*, *medical information exchange* and *second opinion* interactions.

Following the CASCOM role-based service description approach (Section 10.7 in Chapter 10), each service provider advertises a set of possible roles from the role ontology that it can play. Similarly, in service requests it is allowed to specify the roles searched from the role ontology as a logical expression in disjunctive normal form. By establishing a mapping from the elements of the role ontology to service classes, the above filtering framework becomes applicable.

In the CASCOM role based modelling approach, the role taxonomy is supposed to be static over significant amounts of time. Still, the ontology *can* be extended to include new roles and types of interaction not considered before. In that case, the relevance matrix is updated with new rows and columns for those new roles. The relevance values for those new roles are unknown initially, but this can be overcome by randomly including some services with low relevance and, in general, by applying the bootstrapping techniques, both described in Section 11.3.1.

Category-Based Pre-Filtering

Another pertinent strategy for service classification is based on the categories (travel, medical) they belong to. Such categories are considered important information in service descriptions (in fact, the OWL-S language includes a specific field for this characteristic). There are several well known category taxonomies (NAICS, UNSPSC,. . .). However, CASCOM does not choose one in particular, keeping it open to the service describer.

In this filtering framework, each category is considered a class of service. Service descriptions include a set of categories. In the case of a service advertisement, this fits exactly our classes approach (set of classes). In the case of service requests, the set of categories specified are interpreted as a logical formula by connecting them with the operator or ($\vee$).

If the number of different classes (categories) is too big, the computational complexity (regarding both space and time) can become rather high. In that case, the granularity of the classes can be decreased by clustering several categories into the same class based on inheritance relations in the taxonomy tree.

The two types of classification of services presented in this section can be combined as follows:

$$Relevance(S,R) = \alpha \cdot Rel^{RB}(S,R) + (1-\alpha) \cdot Rel^{CB}(S,R) \text{with } \alpha \in [0..1] \quad (11.3)$$

where $Rel^{RB}(S,R)$ and $Rel^{CB}(S,R)$ are the relevance values obtained by the role and category-based filtering approaches, respectively.

11.4 Service Composition With OWLS-XPlan

Though the composition of complex Web Services attracted much interest in different fields related to service oriented computing, there are only a few implemented composition planning tools publicly available for the semantic Web such as the HTN composition planner SHOP2 for OWL-S services [15]. One problem with HTN planners is that they require task specific decomposition rules and methods developed at design time, hence are not guaranteed to solve arbitrary planning problems. That, in particular, motivated the development of our hybrid composition planner OWLS-XPlan for OWL-S 1.1 services which always finds a solution if it exists, though the corresponding planning problem remains to be NP-complete. Like SHOP2, OWLS-XPlan does perform closed world planning prepared through its integrated converter OWLS2PDDL (cf. Section 11.4.2).

While its original version enables static composition of OWL-S services in static domains, an upgraded version OWLS-XPlan 2.0 (cf. Section 11.4.4) also allows to compose OWL-S services in dynamic and stochastic environments in which changes of the world state can non-deterministically (stochastic) occur during (dynamic) planning. Such changes concern the availability of services; changes of predicates, facts, objects of the plan base. In such environments XPlan 2.0 offers an event based dynamic sequential planning of composite services. It listens for events of state changes during its planning process with heuristic partial re-planning of a new minimal and valid composition plan. This is in contrast to non-classic reactive planning with interleaved service execution, and non-classic off-line planning such as conformant, conditional, or contingency planning.

11.4.1 Architecture

The Semantic Web Service composition planner OWLS-XPlan consists of several modules for pre-processing and planning (cf. Figure 11.3). It takes a set of available

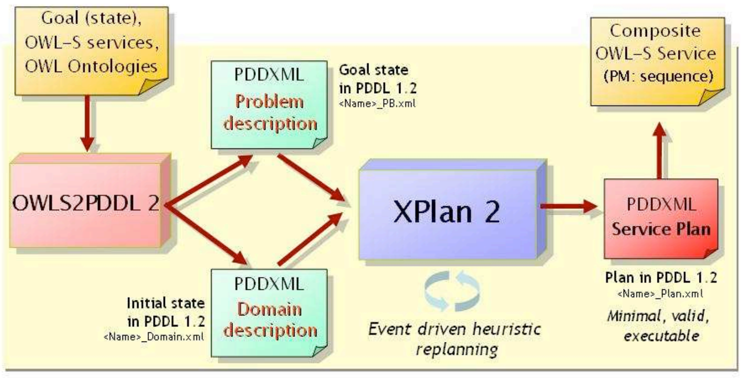

Goal (state), OWL-S services, OWL Ontologies
PDDXML Problem description
Goal state in PDDL 1.2 <Name>_PB.xml
Composite OWL-S Service (PM: sequence)
OWLS2PDDL 2
XPlan 2
PDDXML Service Plan
Initial state in PDDL 1.2 <Name>_Domain.xml
PDDXML Domain description
Event driven heuristic replanning
Plan in PDDL 1.2 <Name>_Plan.xml
Minimal, valid, executable

Figure 11.3: Architecture of OWLS-XPlan

OWL-S 1.1 services, related OWL ontologies, and a planning request (goal) as input, and returns a planning sequence of relevant services that satisfies the goal.

For this purpose, it first converts the domain ontology and service descriptions in OWL and OWL-S, respectively, to equivalent planning problem and domain descriptions in PDDL 2 ("Planning Domain Definition Language") using the integrated OWLS2PDDL converter (cf. Section 11.4.2). For reasons of convenience, we developed a XML dialect of PDDL, called PDDXML, that simplifies parsing, reading, and communicating PDDL descriptions using SOAP.

The planning domain description contains the definition of all types, predicates and actions, whereas the problem description includes all objects (grounded predicates, constants), the initial state, and the goal state. An operator of the planning domain corresponds to a service profile in OWL-S since both operator and profile describe patterns of how an action or service as an instance should look like. A method is a special type of operator for fixed complex services that OWLS-XPlan may use during its planning process. Both descriptions are then used by the state based action planner XPlan to create a plan in PDDL that solves the given problem in the actual domain.

Key to the translation from OWL-S to PDDXML is that any service in OWL-S corresponds to an equally named action with the same set of input parameters, logical preconditions, and effects. However, for classical (STRIPS like) action planning in AI, PDDL does not allow to describe concrete input or output values of operators such as information on specific train connections returned by a service. So we have to add special precondition and effect predicates to tell the planner that it does in general know about the output values as an effect of executing the respective action on the current world state, or the values of typed input variables allowing to match value based restrictions in preconditions of possible successor actions.

Its core AI planning module called XPlan is a heuristic hybrid FF planner based on the FF planner developed by Hoffmann and Nebel [9] (cf. Section 11.4.3). It combines guided local search with relaxed graph planning, and a simple form of hierarchical task networks (HTN) to produce a plan sequence of actions that solves a given problem. If equipped with methods, XPlan uses only those parts of methods for decomposition that are required to reach the goal state with a sequence of composed services. For stochastic domains in which the world state is changing during planning, we developed an event driven heuristic planning module XPlan 2.0 for dynamic composition of services (cf. Section 11.4.4).

11.4.2 Converter OWLS2PDDL

The purpose of the OWLS2PDDL converter is to tranlate a given OWL-DL expression in OWL-S 1.1 service descriptions and a given service composition problem into an equivalent PDDL planning problem which can be understood by AI planners such as XPlan. More concrete, the structured functional service composition problem (SWS, I, G) consists of two (user-provided) OWL-DL ontologies that represent an intial (I) and a goal (G) world state, respectively, and a set SWS of OWL-S services. In the following, we assume familiarity with OWL-DL and OWL-S. The initial state I consists of the domain knowledge base KB_0 (available services, imported OWL ontologies T with asserted instances), and the goal state G represented by a goal service functionality S_G (IOPE = input, output, precondition, effect). The problem is to find a composition sequence $P = S_1 \circ ... \circ S_n$, $S_i \in SWS$ that satisfies G (P reaches G from I).

Overview

The converter OWLS2PDDL is mapping this service composition problem to a classical action based AI planning problem in PDDL. PDDL is a modular language that allows to control its expressiveness by specifying certain *requirements*. The converter itself uses a XML dialect of PDDL 1.2, called PDDXML, with the *ADL* and *open-world* requirements for both PDDL 1.2 [2], and the Action Description Language (ADL) [13] and additional syntax for predicate cardinality restrictions [1]. An action planning problem is defined as a triple $(Init, Goal, Ops)$ consisting of

1. an initial state $Init$,
2. a goal state $Goal$,
3. and a set of *Operators Ops*, where each operator describes a possible action in the considered domain. An operator is characterised by its parametrised precondition and effect, such that

[1]The PDDXML grammar in compact RelaxNG (see [5]) can be found at *http://www.dfki.de/˜blankenb/owls2pddl/PDDXMLDomain.rnc* and *http://www.dfki.de/˜blankenb/owls2pddl/PDDXMLProblem.rnc* for domain and problem instance definitions, respectively.

(a) an action is applicable in a given world state if and only if its precondition is fulfilled in that state.

(b) the effect describes how a state s is transformed to its successor state if the action is applied to s.

PDDL aggregates *Init* and *Goal* states in a *PDDL problem* definition. Operators are contained in a PDDL *domain definition. Init* is a conjunction of predicates, whereas *Goal* is a function-free first-order logical sentence. An action precondition is, like the goal state, a function-free first-order logical sentence, whereas an effect can only be a conjunction of predicates or negated predicates, a universal quantified effect, or a conditional effect; non-deterministic disjunctions are not allowed in effect constraints (in contrast to ADL).

In summary, the OWLS2PDDL converter implements a function

$$(O \times O \times 2^S) \mapsto (D \times P)$$

where O is the set of all OWL DL ontologies, S is the set of all OWL-S 1.1 services, D is the set of all PDDL domains and P is the set of all PDDL problems. The main idea of the conversion is to map OWL-S services to planning domain operators, and to produce the PDDL problem from given OWL "Initial State" and "Goal State" ontologies. In particular, expressions in PDDL are then interpreted with standard FOL semantics corresponding to those of the decidable FOL subset OWL-DL. The interpretation of PDDL operators corresponds to that of respective ADL (Action Description Language [13]) operators which can be reduced to STRIPS operators (see [7]), which in turn are interpreted using Lifschitz' semantics (see [12]). In contrast to STRIPS, both PDDL and ADL assume the open world, but only ADL allows both disjunctions and negated literals in effect constraints (PDDL disallows disjunctive effects). The additional cardinality restrictions are interpreted under the standard description logic (DL) semantics for non-qualifying number restrictions (DL part "N").

PDDL has close to SOIN expressivity with only subsumption, equivalence and transivity of roles from OWL-DLs SHOIN expressivity missing. But these missing features can be represented by fully expanding any role specification to include also any parent and transitively holding roles. This is explained in Subsection 11.4.2. Thus, PDDL's expressiveness is sufficient to equivalently represent an OWL-S service composition problem.

The converter OWLS2PDDL generates the planning problem in PDDL for the planner XPlan, and comes in two versions: OWLS2PDDL 1.0 (2.0) of OWLS-XPlan (OWLS-XPlan 2.0) converts expressions of EXPTIME description logic SI(D) (NEXPTIME description logic SHOIN(D), corresponding to OWL-DL) to PDDL 1.2. However, there are a few obstacles to be discussed in the following together with a simple example of a conversion.

Operators and Service Outputs

While an OWL-S service has inputs, outputs, preconditions and effects, a planning domain operator only has the latter two. The OWL-S specification states that as opposed to preconditions and effects, which refer to the world *state*, inputs and outputs represent *information* that is made available for or *produced by the service*. For an PDDL object, however, its existence cannot be bound to certain states (even with the open-world requirement).

As a solution of this problem, we model the possible creation of information by services with the help of a special `agentHasKnowledgeAbout` predicate. This predicate is set

- in the PDDL initial state: for each object representing an individual of the OWL initial state ontology;
- in the PDDL goal state: for each object representing an individual of the OWL goal state ontology;
- in the effect definition of a PDDL operator: for each PDDL operator parameter representing an output parameter of the respective OWL-S service.

We require this predicate to hold for all PDDL operator parameters representing input parameters of OWL-S services. This ensures that an action is rendered unapplicable in a state if the corresponding service's required input information is not available in that state.

Service Preconditions and Effects

OWL-S does not prescribe a specific language for defining preconditions and effects of services. Instead, one can specify the language with the OWL-S 1.1 `expressionLanguage` property. We extended the to OWL-S 1.1 `Expression` and `Condition` classes to allow for PDDXML preconditions and effects. The definition of these extended classes, `PDDXML-Expression` and `PDDXML-Condition`[2]. Currently, the converter converts only such PDDXML expressions and conditions.

Restrictions of Initial State and Action Effect

In OWL-DL, the class description of an OWL individual in any given ontology can be arbitrarily complex in the scope of OWL-DL. Since there is no notion of "initial ontologies" in OWL, restrictions resembling those of PDDL can hardly be imposed. Thus, expressions in the given OWL initial ontology which violate these restrictions are ignored by the converter when generating the initial state and action effects. We assume that individuals which are stated to be in a given OWL class do indeed fulfill all necessary restrictions of that class.

[2]see http://www2.dfki.de/~babla/owls2pddl/pddxml.owl

Open vs. Closed World

Both OWL and PDDL make the open world assumption (OWA) calling for monotonic reasoning. However, XPlan does perform closed world reasoning like many action planners and service composition planners like SHOP2. Thus, the initial state is implicitly "closed" when feeding the generated PDDL planning problem description into XPlan[3]. It is not possible to include the latter (disjunctive) expression in a PDDL initial state or action effect, since disjunctions are not allowed there. Thus, when interpreting the problem as being closed-world, the conversion might not be complete.

Other Issues of Conversion

Other issues of converting OWL-DL to PDDL are as follows.

- **The PDDL type system is not general enough** to reflect the possibly complex relationships of OWL classes. Thus, the explicitly specified classes of an OWL individual are represented for the corresponding PDDXML object by unary predicates of the classes' names (this includes all superclasses).
- **No language construct for enumerations**: OWL classes which are defined via *oneOf* are converted using a disjunction of special *identity* predicates. These predicates are defined for every object in the initial state and for output parameters.
- **No domain axioms**. Our XML dialect of PDDL does not support domain axioms (which XPlan also does not support). Thus, the conversion of an OWL class definition has to be inserted at every place in the PDDXML where an object or parameter which corresponds to an OWL individual or OWL-S service parameter of that class occurs.

Conversion Rules

Figures 11.4 and 11.5 illustrate how OWL-S service descriptions, intial and goal state ontologies are translated to PDDXML. Table 11.2 summarizes how each SHOIN-expressivity OWL DL construct is converted to an equivalent PDDXML condition. In this table, the leftmost column "DL Ex." denotes the expressiveness class, Δ denotes the domain, and X^I denotes the interpretation of X. Table 11.3 summarises the conversion of transitive properties and subsumption of properties. Follwing these rules, an equivalent PDDL planning problem representation of the service composition problem can be obtained, albeit with factorial runtime and space requirements in the worst case. Please note that in the current version of the converter implementation (OWLS2PDDL 2.0), the conversion of the description of equivalent classes in the goal state, transitive properties and subsumption of properties are not implemented yet; this is ongoing work.

[3]That is, anything which cannot be deduced (e.g. property predicate p) in the initial state is assumed to be false (i.e. $\neg p$), as opposed to being unknown (i.e. $p \vee \neg p$).

DL			OWL	PDDXML
Ex.	Syntax	Semantics		
$\mathcal{S}$	A	$A^I \subseteq \Delta^I$	Class	Unary Predicate
	$\top$	$\top^I = \Delta^I$	Thing	PDDL type “object”
	R	$R^I \subseteq \Delta^I \times \Delta^I$	Property	Binary Predicate
	$R \in R_+$	$R^I = (R^I)^+$	Trans. Property	Multiple Predicates, effect
	$C \sqcap D$	$C^I \cap D^I$	conjunctionOf	`<and/>`
	$C \sqcup D$	$C^I \cup D^I$	disjunctionOf	`<or/>`
	$\neg C$	$\Delta^I \backslash C^I$	complementOf	`<not/>`
	$\exists R.C$	$\{x \mid \exists y.(x,y) \in R^I$ and $y \in C^I\}$	someValuesFrom	`<exists/>`
	$\forall R.C$	$\{x \mid \forall y.(x,y) \in R^I$ implies $y \in C^I\}$	allValuesFrom	`<forall/>`
$\mathcal{H}$	$R \subseteq S$	$R^I \subseteq S^I$	subPropertyOf	Multiple Predicates, effect
$\mathcal{I}$	R^-	$\{(x,y) \mid (y,x) \in R^I\}$	inverseOf	Predicate
$\mathcal{N}$	$\geq nR.C$	$\{x \mid \#\{y.(x,y) \in R^I$ and $y \in C^I\} \geq n\}$	minCardinality	`<cardinality min=''..."/>`
	$\leq nR.C$	$\{x \mid \#\{y.(x,y) \in R^I$ and $y \in C^I\} \leq n\}$	maxCardinality	`<cardinality max="..."/>`
$\mathcal{O}$	$\{o\}$	$\{o\}^I = \{o^I\}$	XML Type + RDF-value	Object
	$\exists T.\{o\}$	$\exists C : \{o\}^I \in T$	hasValue	`<exists/>` + special `identity` predicate

Table 11.2: Conversion and semantics of OWL DL class descriptions to PDDXML conditions

Context	OWL	PDDXML
Initial state	Transitive Property p	For all predicates $p(i,k), p(k,m)$, $i \neq k \neq m$: predicate $p(i,m)$
	p subPropertyOf p'	For all predicates $p(i,k)$: predicate $p'(i,k)$
Actions’ effect	Transitive Property p	`<forall>` i,k,m `<if>` `<and>` $(i \neq k \neq m)$ $p(i,k)$ $p(k,m)$ `</and>` $p(i,m)$ `</if>` `</forall>`
	p subPropertyOf p'	`<forall>` i,k `<if>` $p(i,k)$ $p'(i,k)$ `</if>` `</forall>`

Table 11.3: Expansion of transitive and subsumed properties

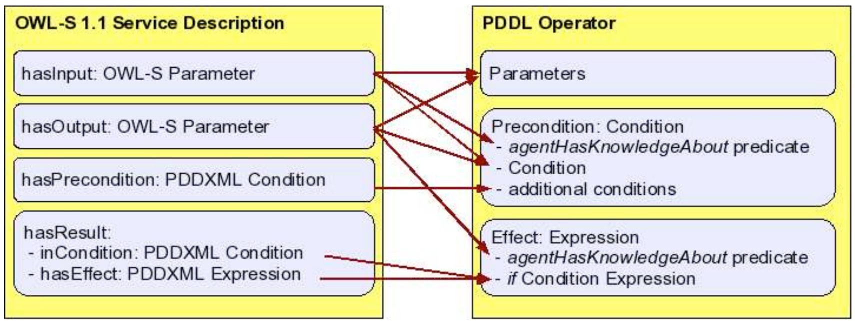

OWL-S 1.1 Service Description
hasInput: OWL-S Parameter
hasOutput: OWL-S Parameter
hasPrecondition: PDDXML Condition
hasResult:
- inCondition: PDDXML Condition
- hasEffect: PDDXML Expression
PDDL Operator
Parameters
Precondition: Condition
- *agentHasKnowledgeAbout* predicate
- Condition
- additional conditions
Effect: Expression
- *agentHasKnowledgeAbout* predicate
- *if* Condition Expression

Figure 11.4: Conversion of OWL-S services to PDDXML actions

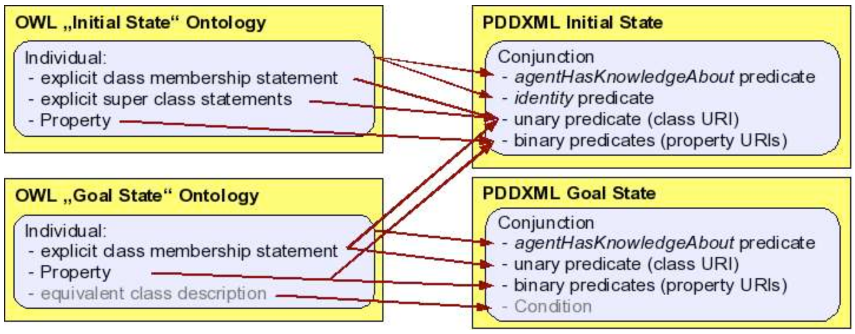
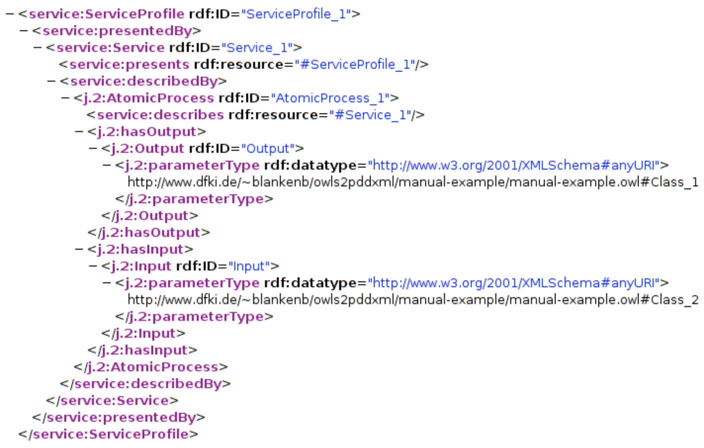

OWL „Initial State" Ontology
Individual:
- explicit class membership statement
- explicit super class statements
- Property
OWL „Goal State" Ontology
Individual:
- explicit class membership statement
- Property
- equivalent class description
PDDXML Initial State
Conjunction
- *agentHasKnowledgeAbout* predicate
- *identity* predicate
- unary predicate (class URI)
- binary predicates (property URIs)
PDDXML Goal State
Conjunction
- *agentHasKnowledgeAbout* predicate
- unary predicate (class URI)
- binary predicates (property URIs)
- Condition

Figure 11.5: Conversion of OWL DL state ontologies to PDDXML states

Example of OWLS2PDDL Conversion

In the following, we provide a brief and simple example of conversion by OWLS2PDDL. Suppose that the given service, initial and goal ontologies all import a common ontology which includes some class definitions, and that there is just one service to convert as shown in Figure 11.6. The converter generates the PDDXML action shown in see Figure 11.7 for this service. The service has one input of type `Class_2`, whose definition is shown in Figure 11.8.

First, the `agentHasKnowledgeAbout` predicate is required on the input. Second, it must be ensured that only objects of the required type can be instantiated with the action. The service input class is a defined class, i.e. any individual which has a minimum cardinality of 2 on property `objectProperty_1` is a member of this class. Thus, the generated PDDXML condition contains a disjunction that states that either the parameter must explicitly be stated to be of type `Class_2`, or the cardinality restirction on the property must hold. Similarly, the conversion of the output type is also included in the action's precondition. The `agentHasKnowledgeAbout`, on the other hand, is only set in the effect, reflecting



```
– <service:ServiceProfile rdf:ID="ServiceProfile_1">
  – <service:presentedBy>
    – <service:Service rdf:ID="Service_1">
        <service:presents rdf:resource="#ServiceProfile_1"/>
      – <service:describedBy>
        – <j.2:AtomicProcess rdf:ID="AtomicProcess_1">
            <service:describes rdf:resource="#Service_1"/>
          – <j.2:hasOutput>
            – <j.2:Output rdf:ID="Output">
              – <j.2:parameterType rdf:datatype="http://www.w3.org/2001/XMLSchema#anyURI">
                  http://www.dfki.de/~blankenb/owls2pddxml/manual-example/manual-example.owl#Class_1
                </j.2:parameterType>
              </j.2:Output>
            </j.2:hasOutput>
          – <j.2:hasInput>
            – <j.2:Input rdf:ID="Input">
              – <j.2:parameterType rdf:datatype="http://www.w3.org/2001/XMLSchema#anyURI">
                  http://www.dfki.de/~blankenb/owls2pddxml/manual-example/manual-example.owl#Class_2
                </j.2:parameterType>
              </j.2:Input>
            </j.2:hasInput>
          </j.2:AtomicProcess>
        </service:describedBy>
      </service:Service>
    </service:presentedBy>
  </service:ServiceProfile>
```

Figure 11.6: OWL-S Example Service

the information gain that is achieved by the service execution. The resulting initial and goal states are rather simple. The initial state consists of the following definition of `Individual_1`:

```
<Class_3 rdf:ID="Individual_1">
    <j.0:objectProperty_1>
        <Class_4 rdf:ID="Individual_2"/>
    </j.0:objectProperty_1>
</Class_3>
```

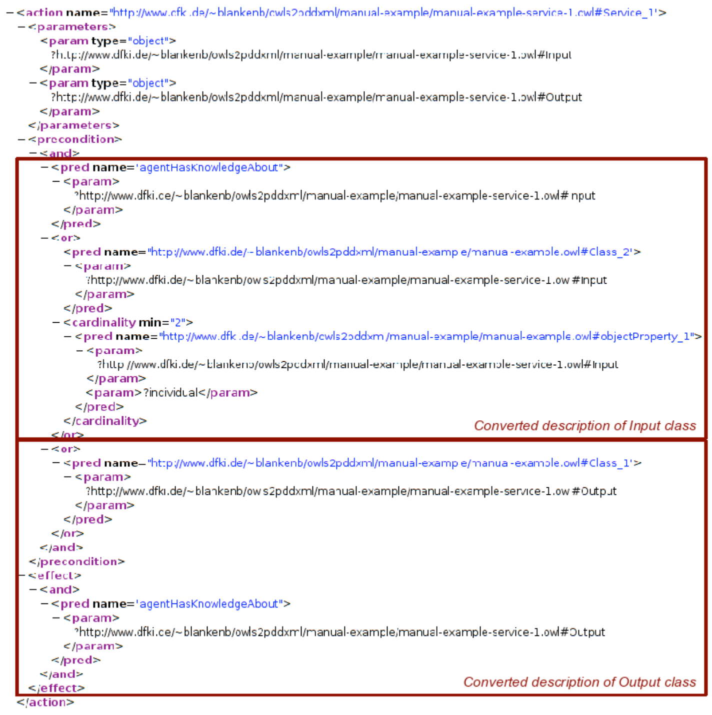
The resulting PDDXML expression which is written into the PDDXML problem definition is shown in Figure 11.9. It includes the `agentHasKnowledgeAbout` predicate, the explicit class membership statement, the `identity` predicate and the binary predicate that represents OWL property `objectProperty_1`

11.4.3 Static Composition

As mentioned above, XPlan performs a static composition under closed world assumption. In fact, the solution of XPlan to the problem of structured functional service composition at hand corresponds to finding a sequence of services that globally plug-in matches with the given goal service functionality (cf. Figure 11.10).

XPlan Solution of the Service Composition Problem

As mentioned in previous section, the functional service composition planning problem can be mapped to a classical action planning problem by (a) identifying the given services with actions and (b) describing the domain together with the requested service in an initial, respectively, goal state ontology in the standard



```
– <action name="http://www.dfki.de/~blankenb/owls2pddxml/manual-example/manual-example-service-1.owl#Service_1">
  – <parameters>
      <param type="object">
        ?http://www.dfki.de/~blankenb/owls2pddxml/manual-example/manual-example-service-1.owl#Input
      </param>
    – <param type="object">
        ?http://www.dfki.de/~blankenb/owls2pddxml/manual-example/manual-example-service-1.owl#Output
      </param>
    </parameters>
  – <precondition>
    – <and>
      – <pred name="agentHasKnowledgeAbout">
        – <param>
            ?http://www.dfki.de/~blankenb/owls2pddxml/manual-example/manual-example-service-1.owl#Input
          </param>
        </pred>
      – <or>
          <pred name="http://www.dfki.de/~blankenb/owls2pddxml/manual-example/manual-example.owl#Class_2">
          – <param>
              ?http://www.dfki.de/~blankenb/owls2pddxml/manual-example/manual-example-service-1.owl#Input
            </param>
          </pred>
        – <cardinality min="2">
          – <pred name="http://www.dfki.de/~blankenb/owls2pddxml/manual-example/manual-example.owl#objectProperty_1">
            – <param>
                ?http://www.dfki.de/~blankenb/owls2pddxml/manual-example/manual-example-service-1.owl#Input
              </param>
              <param>?individual</param>
            </pred>
          </cardinality>                                          Converted description of Input class
        </or>
      – <or>
        – <pred name="http://www.dfki.de/~blankenb/owls2pddxml/manual-example/manual-example.owl#Class_1">
          – <param>
              ?http://www.dfki.de/~blankenb/owls2pddxml/manual-example/manual-example-service-1.owl#Output
            </param>
          </pred>
        </or>
      </and>
    </precondition>
  – <effect>
    – <and>
      – <pred name="agentHasKnowledgeAbout">
        – <param>
            ?http://www.dfki.de/~blankenb/owls2pddxml/manual-example/manual-example-service-1.owl#Output
          </param>
        </pred>
      </and>
    </effect>                                                     Converted description of Output class
</action>
```

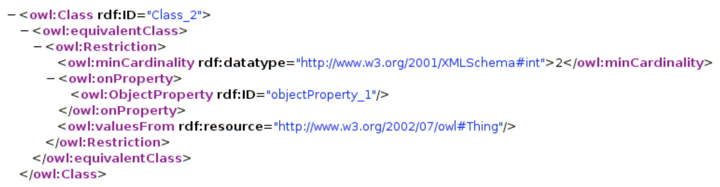
Figure 11.7: The generated PDDXML action

planning language PDDL. The solution of XPlan for this problem corresponds to a plug-in match of the plan P considered as one composed service to the goal service S_G with goal ontology G together with an IOPE (input, output, precondition, effect) chaining of the sequecence of services within P (cf. Figure 11.10).

In addition, the executability of P on the grounding level of the reconverted actions to OWL-S services can be guaranteed by the interleaved checking of whether the I/O parameter data types in XMLS of subsequent services of P grounded in WSDL are compatible with each other which ensure the data flow within the sequence of services to be executed after planning. The reconversion is done by OWLS2PDDL 2.0 and the compatibility check is performed by the planner by means of its integrated OWLS-MXP component (partially reused from the OWLS-MX matchmaker, cf. Chapter 10).

Graphbased FF Planning with XPlan

For each sub-goal g of the determined goal agenda, at each planning step i, XPlan quickly builds a relaxed planning graph $RPG(i)$ in a fast goal reachability test heuristically ignoring negative effects of actions A, and the corresponding relaxed



```
- <owl:Class rdf:ID="Class_2">
  - <owl:equivalentClass>
    - <owl:Restriction>
        <owl:minCardinality rdf:datatype="http://www.w3.org/2001/XMLSchema#int">2</owl:minCardinality>
      - <owl:onProperty>
          <owl:ObjectProperty rdf:ID="objectProperty_1"/>
        </owl:onProperty>
        <owl:valuesFrom rdf:resource="http://www.w3.org/2002/07/owl#Thing"/>
      </owl:Restriction>
    </owl:equivalentClass>
  </owl:Class>
```

Figure 11.8: Common class definitions

plan $RP(i)$ in a backward pass from g to S_i. The relaxed plan contains all paths of applicable actions that lead from g to S_i, of which only those in its first action-layer 0 are called helpful.

In the following, XPlan focuses on the helpful actions of $RP(i)$ only, hence reduces the search space. Please note that the relaxed plan is not necessarily correct due to ignorance of the Del-lists, i.e., negative effects of actions. In order to decide which helpful action to select as the next action in a valid plan sequence, XPlan applies each of them to S_i and adds the previously ignored Del-list facts yielding the complete state S_{ij}, where $j \in \{1,..,l\}$, denotes the j-th helpful action applied to state S_i.

For each of these states the relaxed plan $RPG(i,j)$ is then built to heuristically search for the relaxed plan $RP(i,j)$ with heuristically minimal length $h(RP(i,j))$. In this context, the "plan length" $h(RP(i,j))$ just denotes the sum of all actions in all action-layers of the RP.

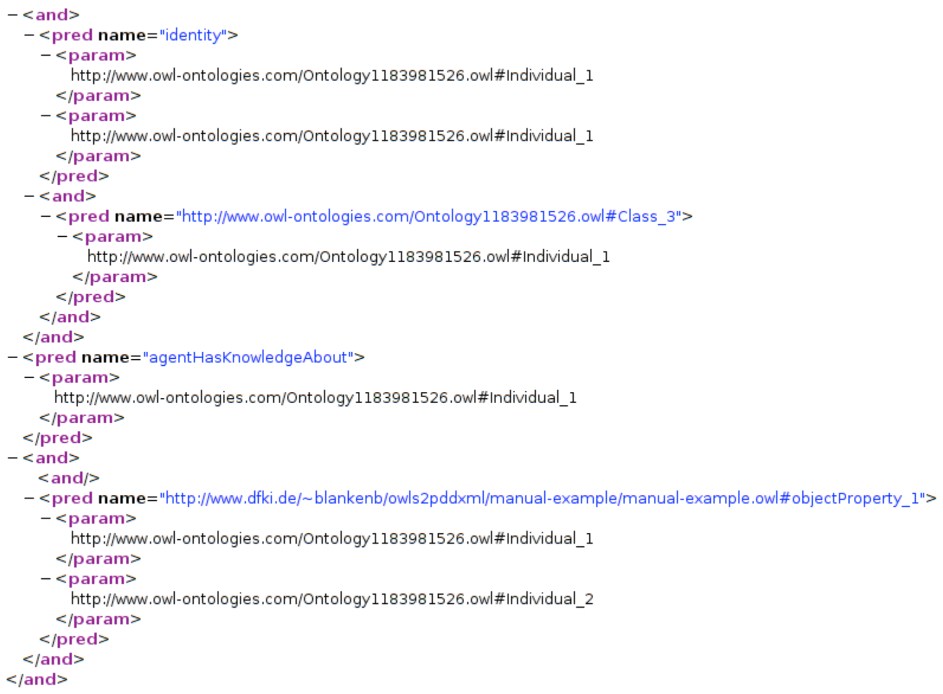
Finally, XPlan retains the action A_{ij} with heuristically minimal goal distance, and starts the next planning step $i+1$ with S_{ij}. If there are multiple RPs of equal length, it repeats the same decision process starting at state S_{i1} (like a breadth first search restricted on helpful actions), and then $S_{i2}, ..., S_{il}$ until a minimum is found.

Eventually, all created plans for sub-goals g of the goal agenda are respectively concatenated which yields the final plan sequence P. The plan then gets executed, and if it fails, XPlan allows re-planning from the most recent valid state produced by action execution, to avoid a total re-planning, if possible.

As mentioned above, XPlan also checks at each planning step whether the selected pairs of services to be composed are data type compatible to ensure the executability of the generated plan. For this purpose, it utilizes respective information it got from the service matchmaker (OWLS-MXP) about the available services prior to the planning process. For more details on OWLS-XPlan in general, and XPlan in particular, together with examples of service translation from OWL-S to PDDXML we refer the reader to [10, 14].

11.4.4 Dynamic Composition

For OWLS-XPlan 2.0, which has been eventually used in CASCOM, we modified the original XPlan module of OWLS-XPlan to allow for event driven heuristic re-



```
- <and>
  - <pred name="identity">
    - <param>
        http://www.owl-ontologies.com/Ontology1183981526.owl#Individual_1
      </param>
    - <param>
        http://www.owl-ontologies.com/Ontology1183981526.owl#Individual_1
      </param>
    </pred>
  - <and>
    - <pred name="http://www.owl-ontologies.com/Ontology1183981526.owl#Class_3">
      - <param>
          http://www.owl-ontologies.com/Ontology1183981526.owl#Individual_1
        </param>
      </pred>
    </and>
  </and>
- <pred name="agentHasKnowledgeAbout">
  - <param>
      http://www.owl-ontologies.com/Ontology1183981526.owl#Individual_1
    </param>
  </pred>
- <and>
    <and/>
  - <pred name="http://www.dfki.de/~blankenb/owls2pddxml/manual-example/manual-example.owl#objectProperty_1">
    - <param>
        http://www.owl-ontologies.com/Ontology1183981526.owl#Individual_1
      </param>
    - <param>
        http://www.owl-ontologies.com/Ontology1183981526.owl#Individual_2
      </param>
    </pred>
  </and>
</and>
```

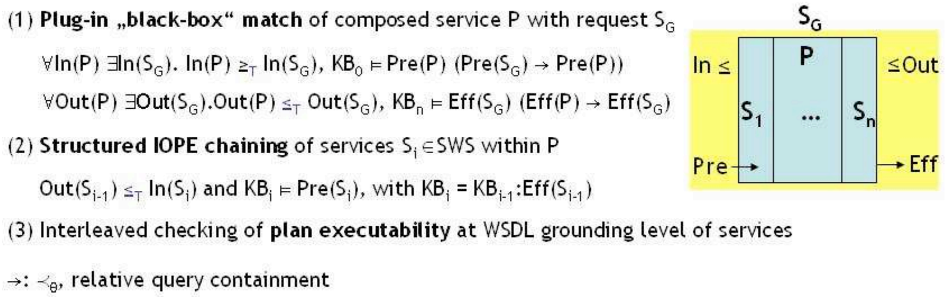
Figure 11.9: OWL initial state ontology

planning of composite services during the actual planning process. The modified planner XPlan 2.0 does perform, in essence, highly frequent event driven off-line re-planning under closed world asumption with heuristic computation of best re-entry points for re-planning at the end of each planning step if the currently produced plan, or plan fragment is affected by the observed change.

External changes of the world state concern converted OWL ontologies, individuals and the set of available services during the internal planning process each of which potentially affecting the respective operators, actions, predicates, facts and objects in the PDDXML problem and domain descriptions as well as already generated partial plans. For event monitoring, we equipped XPlan 2.0 with an event listener for distinguished classes of events.

In particular, in each plan step i, before applying selected helpful action A to the state S_i, however, XPlan 2.0 listens for events of state changes. If no events are in its event queue, it applies A to S_i and proceeds with plan step i+1. The plan fragment from initial state S_0 to S_i is correct and, due to the selection of helpful actions in the minimal relaxed plan, approximatively optimal.

XPlan 2.0 triggers re-planning in the following cases of observed events of world state changes: (1) An operator (service) instantiation (action) becomes avail-



(1) **Plug-in „black-box“ match** of composed service P with request S_G

$\forall In(P)\ \exists In(S_G).\ In(P) \geq_T In(S_G),\ KB_0 \models Pre(P)\ (Pre(S_G) \rightarrow Pre(P))$

$\forall Out(P)\ \exists Out(S_G).Out(P) \leq_T Out(S_G),\ KB_n \models Eff(S_G)\ (Eff(P) \rightarrow Eff(S_G))$

(2) **Structured IOPE chaining** of services $S_i \in SWS$ within P

$Out(S_{i-1}) \leq_T In(S_i)$ and $KB_i \models Pre(S_i)$, with $KB_i = KB_{i-1}:Eff(S_{i-1})$

(3) Interleaved checking of **plan executability** at WSDL grounding level of services

$\rightarrow$: $\prec_\theta$, relative query containment

Figure 11.10: Structured functional service composition

able. This is the case if (a) a new operator has been introduced, or (b) the world state (set of facts) changed such that an operator whose instantiation was impossible before can be instantiated now, or (c) new predicates which are part of the preconditions or effects of an operator are introduced, making it possible to instantiate this operator; (2) An operator (service) of the plan is not possible anymore, if any of the opposites of cases 1.a – 1.c holds; (3) The goal state changed due to a change of the original planning request.

Each of these cases is handled separately as described in subsequent sections. If facts or objects change, the planner searches for the first operator which precondition is satisfied by the new fact, and starts re-planning from there, while the helpful actions get instantiated with the new fact(s). The case in which a predicate p changes can be reduced (a) to the latter case of changed facts, if new facts are added; (b) to the case of change of operator o (action A), if preconditions or effects of o include p; or (c) to the case of fact changes, if the deletion of p implies the deletion of all instances of p. It is assumed that the planning state consistency is checked by means of an appropriate module as intergal part of both XPlan and XPlan 2.0.

Both versions of OWLS-XPlan have been implemented in Java and are available at the semantic Web community portal semwebcentral.org.

11.5 Service Composition With MetaComp

MetaComp is one of the service composition agents developed in the CASCOM project. Although MetaComp has been designed and implemented following basically the same approach as the OWLS-XPlan module described in the previous section, we emphasize two main differences.

First, MetaComp service discovery approach is different from that used in the filtering component of OWLS-XPlan. Second, MetaComp uses the SAPA planner [8] instead of the XPlan planner. Whereas the filtering process of OWLS-XPlan is applied to the services returned by the service discovery agent (SDA), the service discovery strategy designed for MetaComp asks the SDA for a reduced number of services so that fewer service descriptions have to be conveyed between the two agents.

Besides, MetaComp service discovery strategy is simpler than the one used in the filtering component of OWLS-XPlan. It is based on service categories (which have to be provided by the agent client), on service inputs, outputs, preconditions and effects, and it uses context information. Although simpler, we feel this strategy might yield reasonable results. However, for the purpose of the CASCOM selected problems, any of the planners (SAPA or XPlan), means either MetaComp or OWLS-XPlan would have been a good choice. The remaining of this section provides some details regarding MetaComp development and results.

11.5.1 Architecture

MetaComp receives service composition requests from its clients. Service composition requests include a partial OWL-S description specifying the service to be composed, that is, the initial state and the composition goal. The service specification (initial state and composition goal) and the descriptions of the services available to be integrated in the final compound service are sent to the planning algorithm for it to generate the compound service.

However, since both the desired service specification and the descriptions of the available services are represented in OWL-S whereas the used planning algorithm accepts only PDDL, these OWL-S descriptions, as in OWLS-XPlan, are first translated to PDDL. The planning algorithm output is merely the sequence of the elemental services that actually make up the compound service. This has to be reconverted to OWL-S so that it can be sent to the agent's client.

This conversion involves two steps. First, it is necessary to generate the compound service (global) inputs, outputs, preconditions and effects from the (local) inputs, outputs, preconditions and effects of the elemental services that make up the compound service. Second, the sequence of elemental services comprising the compound service and its global inputs, outputs, preconditions and effects (generated in the first step) are converted to OWL-S. Figure 11.11 shows MetaComp component based architecture and the interactions between its components.

The agentified service composition module MetaComp consists of the following five key components:

1. the MetaComp agent interaction component AIC;
2. the converter OWLS2PDDL as described in previous section;
3. the planning component SAPA;
4. the IOPE generation component;
5. the OWL-S description generation component.

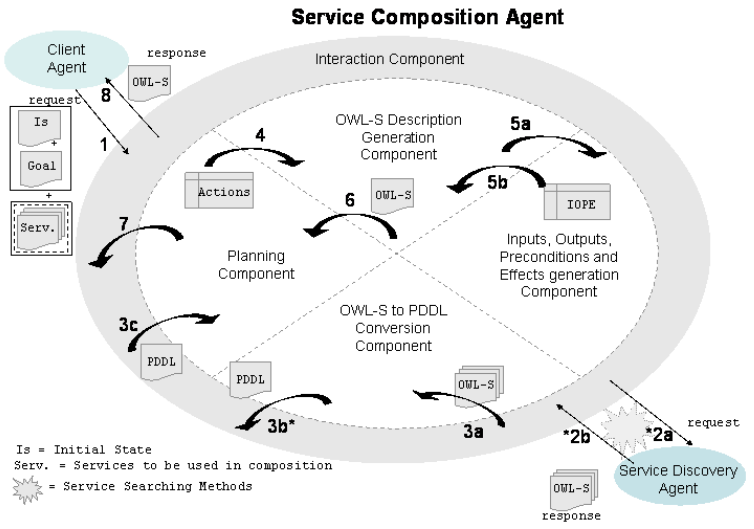

Service Composition Agent
Client Agent
response
OWL-S
request
Is
+
Goal
+
Serv.
8
1
Interaction Component
OWL-S Description Generation Component
4
5a
5b
Actions
6
OWL-S
IOPE
7
Planning Component
Inputs, Outputs, Preconditions and Effects generation Component
3c
OWL-S to PDDL Conversion Component
PDDL
PDDL
OWL-S
3b*
3a
*2b
*2a
request
Service Discovery Agent
OWL-S
response
Is = Initial State
Serv. = Services to be used in composition
= Service Searching Methods

Figure 11.11: MetaComp Architecture

The AIC of the MetaComp agent was developed as an extension of the JADE platform. Its main purpose is to provide an interaction framework to FIPA-compliant agents. It uses the FIPA-Request interaction protocol [6] when interacting with its clients and when interacting with the service discovery agent SDA requesting the services to be used during composition. AIC is responsible for receiving/sending messages and parsing them into a suitable format for the interaction with other components and with agents.

The purpose of the OWL-S to PDDL conversion component, the OWLS2PDDL converter taken from OWLS-XPlan, is to generate the PDDL descriptions from the received initial state description, composition goal specification, and the OWL-S descriptions of the services available for composition. For more information about the OWLS2PDDL converter we refer to Section 11.4.2.

The planning component SAPA [8] of MetaComp is responsible for generating a sequence of component services (i.e., actions) that satisfies the client request (i.e., planning goal) from the specified initial state. MetaComp uses SAPA, a domain-independent heuristic forward planner that can handle durative actions, metric resource constraints, and deadline goals. SAPA is designed to be capable of handling the multi-objective nature of metric temporal planning.

Though SAPA accepts PDDL level three (version 2.1) descriptions, following a CASCOM project decision, we have used only SAPA PDDL level one capabilities. The processing in SAPA, since it receives the two PDDL sections until it produces the plan, is made up of three steps:

1. reading and parsing the PDDL descriptions, which in case they are grammatically correct, should be transformed into a data structure to be processed by SAPA;
2. instantiation of the parameters of the available actions with object instances represented in the planning problem;
3. searching for a planning solution.

In the instantiation step, all static conditions presented in the initial state are evaluated. Static conditions are those whose truth value do not change as a result of some service execution. If a certain static condition is true in the initial state, it continues to be true in all subsequent states that result of service execution. Since their truth value never changes, once SAPA checks that they are true in the initial state, static conditions are removed from the PPDL description. This increases the planner performance.

The IOPE generation component receives the generated action sequence and the (local) parameters, preconditions and effects of each of the actions of the new composite service and generates the (global) inputs, outputs, preconditions and effects of the compound service.

The purpose of the OWL-S description generation Component is to generate the OWL-S description of the generated compound service from the sequence of component services generated by the planning component (SAPA) and the service inputs, outputs, preconditions and effects produced by the Inputs, Outputs, Preconditions and Effects generation Component.

11.5.2 Service Selection Methods

The service composition process requires a set of existing services that may be chained to form the compound service. The first step of service composition is to request the descriptions of those services to the service discovery agent. If this is not a carefully crafted process, it may result either in a huge, computationally intractable collection of services, most of which may turn out to be useless for the composition problem at hand, or in a small set of services which may be insufficient to create the desired compound service.

In this respect, we assume context information of great importance since it allows reducing the set of services requested to the service discovery agent (SDA) to only those matching the current context. This will improve efficiency in two ways. First, the SDA will only return fewer but relevant services. Second, service composition with fewer services is more efficient. Besides efficiency, context compliant services will hopefully be more adequate to the current state of affairs. Service availability and cost, and user profile are the context information considered in the service selection process.

Two service selection methods have been designed: a service category based method, where services are selected according to their category; and a method in which services are selected if at least one of their inputs, outputs, preconditions or effects matches the composition problem. However, currently, none of the designed methods has been integrated in the implemented MetaComp; this is future work.

Search Based on Service Categories

The method is focused on the service categories specified in the service composition request. In this approach, service categories are organized in a hierarchic taxonomy. After receiving the composition request, the service composition agent MetaComp asks the SDA for all available services that match the specified categories and the current context information (service availability and cost, and user profile).

The returned services (after transformation to PDDL) are used by the planning component SAPA to create the new compound service. In case the service composition is successful, the new compound service is sent to the client agent. If the services of the specified categories are not enough to perform the composition, the solution is to look for services of the category immediately above the specified category, in the given hierarchy. This process will repeat itself until the composition is successful, the maximum composition time specified by the client is reached, or no more upper levels can be found in the categories hierarchy.

Search Based on Problem Characteristics

In this method, the service composition agent MetaComp asks the SDA for all available services that match the context information and have at least one input or one precondition, one output, or one effect of the desired service. MetaComp uses the returned services in order to try to create the desired compound service. If the composition is successful, the compound service is sent to the client agent. If the service composition fails, MetaComp will ask the SDA for more services. This time, MetaComp will ask for all available services that match the context information and at least one precondition, input, output or effect that matches with the previously provided services.

MetaComp uses the newly received services plus the previously received ones and starts a new composition. The process continues until a compound service is created, the maximum composition time specified by the user is reached, or the maximum number of considered services, as specified by the client, is reached.

11.5.3 Implementation

MetaComp (with the exception of the service selection methods) was implemented in the Java programming language. Several Java based tools were used in its development: JADE (Java Agent DEvelopment Framework) [3], SAX (Simple API for XML) [4], ALL (Abstract Logic Language) [1], OWLS2PDDL (cf. Section 11.4.2) and OWL-S API [16].

JADE was used as the agent platform and for the development of MetaComp interaction component. SAX was used for reading PDDXML preconditions and effects. ALL provides support for the internal representation of PDDXML preconditions and effects. The OWLS2PDDL of OWLS-XPlan was used for the conversion of OWL-S service descriptions, and OWL descriptions of the initial state of the world and goal, to PDDL. The OWL-S API was used for writing the OWL-S descriptions of the compound services generated by MetaComp agent.

11.6 Summary

In this chapter, we presented that the CASCOM composition planner agent SCPA, a detailed description of the prefiltering module, and both the OWLS-XPlan and MetaComp planning modules of the SCPA. For static SWS composition planning, the SCPA can use MetaComp while OWLS-Xplan2 allows for advanced dynamic service composition (cf. Chapter 4). In any case, the search space can be tuned by prefiltering of relevant services according to the non-functional role-based matchmaker described in the previous chapter. The SCPA has been fully implemented in Java and successfully demonstrated in the CASCOM e-health application scenario.

References

[1] Adetti: Abstract Logic Language; ALL Specification. Available online at http://clts.we-b-mind.org/files/all.doc. 2002.

[2] The AIPS-98 Planning Competition Committee: PDDL the planning domain definition language. Technical Report CVC TR-98-003/DCS TR-1165, Yale Center for Computational Vision and Control, October 1998. Available at: ftp://ftp.cs.yale.edu/pub/mcdermott/software/pddl.tar.gz.

[3] F. Bellifemine, G. Caire, A. Poggi and G. Rimassa: JADE - A White Paper. EXP Magazine, In search of innovation, 3(3). 2003. Available on-line at http://exp.telecomitalialab.com

[4] D. Brownell: SAX2. O'Reilly; ISBN: 0596002378. 2002.

[5] J. Clark (Ed.): Relax NG Compact Syntax, November 2001. http://relaxng.org/compact-20021121.html.

[6] FIPA Commitee Members: Foundation for Intelligent Physical Agents: Interaction Protocol Specifications. 2002. Available on-line at http://www.fipa.org/repository/ips.php3

[7] B.C. Gazen and C.A. Knoblock: Combining the expressivity of ucpop with the efficiency of graphplan. Proceedings of the 4th European Conference ECP on Planning, London, UK, Springer-Verlag, 1997.

[8] M. B. Do and S. Kambhampati: Sapa: A Scalable Multi-objective Heuristic Metric Temporal Planner. Journal of AI Research, 20:155–194, 2003.

[9] J. Hoffmann and B. Nebel: The FF Planning System: Fast Plan Generation Through Heuristic Search. Journal of Artificial Intelligence Research (JAIR), (14):253302, 2001.

[10] M. Klusch, A. Gerber and M. Schmidt: Semantic Web Service Composition Planning with OWLS-XPlan. Proceedings of the AAAI Fall Symposium on Semantic Web and Agents, Arlington VA, USA, AAAI Press, 2005.

[11] M. Klusch and K-U. Renner: Fast Dynamic Re-Planning of Composite OWL-S Services. Proceedings of 2nd IEEE Intl Workshop on Service Composition (SerComp), IEEE CS Press, Hongkong, China, 2006.

[12] V. Lifschitz: On the semantics of STRIPS. MP. Georgeff, Amy L. Lansky (eds): Proceedings of the Intl Workshop on Reasoning about Actions and Plans, Timberline, Oregon, Morgan Kaufmann, 1986

[13] E.P. Pednault: ADL: Exploring the middle ground between STRIPS and the situation calculus. Proceedings of the Conference on Knowledge Representation and Reasoning KRR, San Francisco, CA, USA, Morgan Kaufmann, 1998.

[14] K-U. Renner, B. Blankenburg, P. Kapahnke and M. Klusch: OWLS-XPlan 2.0 - Dynamic Composition Planning of OWL-S Services (Reference Manual). SCALLOPS Project Report, 2007. Available at www.dfki.de/ klusch/owlsx-plan2.pdf

[15] E. Sirin, B. Parsia, D. Wu, J. Hendler and D. Nau: HTN planning for Web Service composition using SHOP2. Journal of Web Semantics, 1(4), 2004.

[16] E. Sirin and B. Parsia: The OWL-S Java API. Proceedings of the Third International Semantic Web Conference. 2004.