Semantic Web Service Composition Planning with OWLS-Xplan[*]

Matthias Klusch, Andreas Gerber
German Research Center for Artificial Intelligence,
Stuhlsatzenhausweg 3, 66123 Saarbruecken, Germany,
{klusch, agerber}@dfki.de

Marcus Schmidt
DIaLOGIKa GmbH,
Albertstrasse, 66125 Saarbruecken, Germany
Markus.Schmidt@dialogika.de

Abstract

We present an OWL-S service composition planner, called OWLS-Xplan, that allows for fast and flexible composition of OWL-S services in the semantic Web. OWLS-Xplan converts OWL-S 1.1 services to equivalent problem and domain descriptions that are specified in the planning domain description language PDDL 2.1, and invokes an efficient AI planner Xplan to generate a service composition plan sequence that satisfies a given goal. Xplan extends an action based FastForward-planner with a HTN planning and re-planning component.

Introduction

One of the striking advantages of web service technology is the fairly simple aggregation of complex services out of a library of simpler or even 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 many broadly available tools and other composition planners such as SHOP2.

Hierarchical task network (HTN) planners such as SHOP2 perform well in domains for which complete and detailed knowledge on at least partially hierarchically structured action execution patterns is available, such as, for example, in scenarios of rescue planning. In domains in which this is not the case, i.e., no concrete set of methods and decomposition rules that lead to an executable plan are provided, an HTN planner would not find the solution due to the fixed structure of hierarchical action decompositions stored in its database. That inherently limits the degree of quality of any HTN planner to that of its used methods that are created by human experts.

In contrast, action based planners are able to find a solution based on atomic actions as they are described in the methods, but without using the structure of the latter. Atomic actions can be combined in multiple ways to solve a given planning problem. But how to cope with planning problems that are in part hierarchically structured according to decomposition rules and methods but not solvable exclusively by means of HTN planning?

For this purpose, we developed a hybrid AI planner Xplan (Schmidt 2005) which combines the benefits of both approaches by extending an efficient graph-plan based FastForward-planner with a HTN planning component. To use Xplan for semantic Web-Service composition, XPlan is complemented by a conversion tool that converts OWL-S 1.1 service descriptions to corresponding PDDL 2.1 descriptions that are used by Xplan as input to plan a service composition that satisfies a given goal. In contrast to HTN planners, Xplan always finds a solution if it exists in the action/state space over the space of possible plans, though the problem is NP-complete. Xplan also includes a re-planning component to flexibly react to changes in the world state during the composition planning process. Together the implementations of Xplan and OWLS2PDDL converter make up the semantic Web service composition planner OWLS-Xplan.

The remainder of this paper is structured as follows. Section 2 provides an overview of the OWLS-Xplan system architecture, followed by a brief description of the integrated converter module OWLS2PDDL in section 3. The core of OWLS-Xplan, the hybrid planner Xplan, is presented and compared with SHOP2 in section 4 and 5, respectively. We conclude in section 6.

OWLS-Xplan Overview

OWLS-Xplan consists of several modules for preprocessinf and planning. It takes a set of available OWL-S services, a domain description consisting of relevant OWL ontologies and a planning query as input, and returns a plan sequence of composed services that satisfies the query goal.

For this purpose, OWLS-Xplan first converts the domain ontology and service descriptions in OWL and OWL-S, respectively, to equivalent PDDL 2.1 problem and domain descriptions using its OWLS2PDDL converter. The domain description contains the definition of all types, predicates and actions, whereas the problem description includes all objects, the initial state, and the goal state. Both descriptions are then used by the AI planner Xplan to create a plan (representing one composed web service) in PDDL that solves the given problem in the actual domain and initial state. For reasons of convenience, we developed a XML dialect of PDDL,

[*]This work has been supported by the German Ministry of Education and Research (BMBF 01-IW-D02-SCALLOPS), and the European Commission (FP6 IST-511632-CASCOM).

called PDDXML, that simplifies parsing, reading, and communicating PDDL descriptions using SOAP. Table 1 shows the corresponding notions of both the AI planning and semantic web service domain.

planning domain	semantic web service domain
(atomic) operator	service profile
(atomic) action	atomic web service, atomic process
complex action	service model
method	composed web service, workflow, composite process

Table 1: Correlating notions of the planning and semantic web service domain

An operator of the planning domain corresponds to a service profile in OWL-S: Both operator and profile, in essence, describe a pattern or template of how an action or web service as an instance should look like. A method is a special type of operator, that allows the user to describe workflows or composed web services. The planner may use methods as a hierarchical task network during its planning process.

Converter OWLS2PDDL

The conversion of OWL-S 1.1 service descriptions to PDDXML requires not only the straight forward transcription of types and properties to PDDL predicates but the mapping of services to actions (cf. figure 1). Due to space limitations, we only describe the essential translation process.

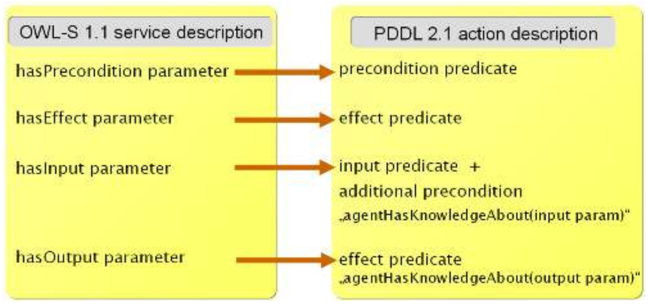
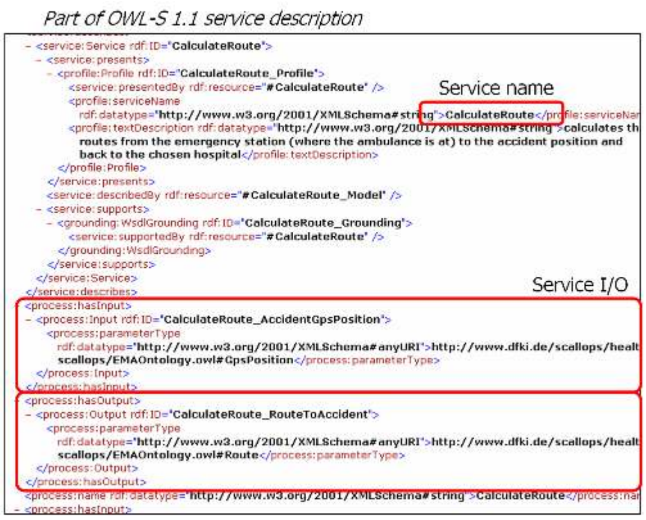

OWL-S 1.1 service description
PDDL 2.1 action description
hasPrecondition parameter
precondition predicate
hasEffect parameter
effect predicate
hasInput parameter
input predicate +
additional precondition
„agentHasKnowledgeAbout(input param)"
hasOutput parameter
effect predicate
„agentHasKnowledgeAbout(output param)"

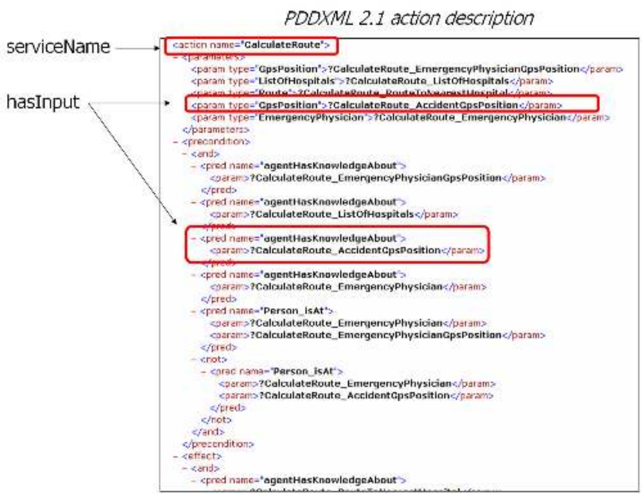
Figure 1: Mapping between OWL-S service and PDDL action description

Any OWL-S service profile input parameter correlates with an equally named one of an PDDL action, and the hasPrecondition service parameter can directly be transformed to the precondition of the action by use of predicates. The same holds for the hasEffect condition parameter. Figure 3 provides an example of such a mapping of an OWL-S 1.1 service that calculates the route from given GPS-position of an accident to the nearest hospital for an emergency physician to the equivalent PDDL 2.1 action description. Either this service already exists, hence its translation is part of the planning *domain* description, or, as a requested service (query) becomes part of the planning *problem* description.



```
- <service:Service rdf:ID="CalculateRoute">
  - <service:presents>
    - <profile:Profile rdf:ID="CalculateRoute_Profile">
        <service:presentedBy rdf:resource="#CalculateRoute" />
        <profile:serviceName
          rdf:datatype="http://www.w3.org/2001/XMLSchema#string">CalculateRoute</profile:serviceNar
        <profile:textDescription rdf:datatype="http://www.w3.org/2001/XMLSchema#string">calculates th
          routes from the emergency station (where the ambulance is at) to the accident position and
          back to the chosen hospital</profile:textDescription>
      </profile:Profile>
    </service:presents>
    <service:describedBy rdf:resource="#CalculateRoute_Model" />
  - <service:supports>
    - <grounding:WsdlGrounding rdf:ID="CalculateRoute_Grounding">
        <service:supportedBy rdf:resource="#CalculateRoute" />
      </grounding:WsdlGrounding>
    </service:supports>
  </service:Service>
</service:describes>
<process:hasInput>
- <process:Input rdf:ID="CalculateRoute_AccidentGpsPosition">
    <process:parameterType
      rdf:datatype="http://www.w3.org/2001/XMLSchema#anyURI">http://www.dfki.de/scallops/healt
      scallops/EMAOntology.owl#GpsPosition</process:parameterType>
  </process:Input>
</process:hasInput>
<process:hasOutput>
- <process:Output rdf:ID="CalculateRoute_RouteToAccident">
    <process:parameterType
      rdf:datatype="http://www.w3.org/2001/XMLSchema#anyURI">http://www.dfki.de/scallops/healt
      scallops/EMAOntology.owl#Route</process:parameterType>
  </process:Output>
</process:hasOutput>
<process:name rdf:datatype="http://www.w3.org/2001/XMLSchema#string">CalculateRoute</process:nar
- <process:hasInput>
```

Figure 2: Part of OWL-S 1.1 service description

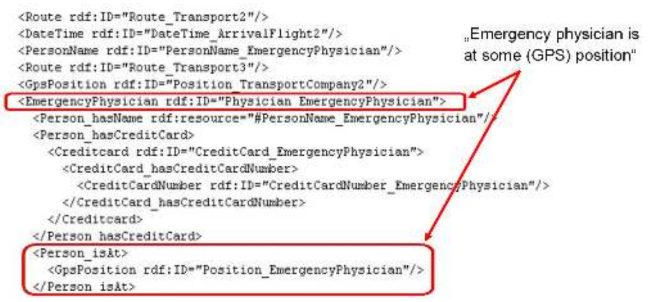


```
<action name="CalculateRoute">
- <parameters>
    <param type="GpsPosition">?CalculateRoute_EmergencyPhysicianGpsPosition</param>
    <param type="ListOfHospitals">?CalculateRoute_ListOfHospitals</param>
    <param type="Route">?CalculateRoute_RouteToNearestHospital</param>
    <param type="GpsPosition">?CalculateRoute_AccidentGpsPosition</param>
    <param type="EmergencyPhysician">?CalculateRoute_EmergencyPhysician</param>
  </parameters>
- <precondition>
  - <and>
    - <pred name="agentHasKnowledgeAbout">
        <param>?CalculateRoute_EmergencyPhysicianGpsPosition</param>
      </pred>
    - <pred name="agentHasKnowledgeAbout">
        <param>?CalculateRoute_ListOfHospitals</param>
      </pred>
    - <pred name="agentHasKnowledgeAbout">
        <param>?CalculateRoute_AccidentGpsPosition</param>
      </pred>
    - <pred name="agentHasKnowledgeAbout">
        <param>?CalculateRoute_EmergencyPhysician</param>
      </pred>
    - <pred name="Person_isAt">
        <param>?CalculateRoute_EmergencyPhysician</param>
        <param>?CalculateRoute_EmergencyPhysicianGpsPosition</param>
      </pred>
    - <not>
      - <pred name="Person_isAt">
          <param>?CalculateRoute_EmergencyPhysician</param>
          <param>?CalculateRoute_AccidentGpsPosition</param>
        </pred>
      </not>
    </and>
  </precondition>
- <effect>
  - <and>
    - <pred name="agentHasKnowledgeAbout">
```

Figure 3: Part of action description in PDDXML converted by OWLS2PDDL

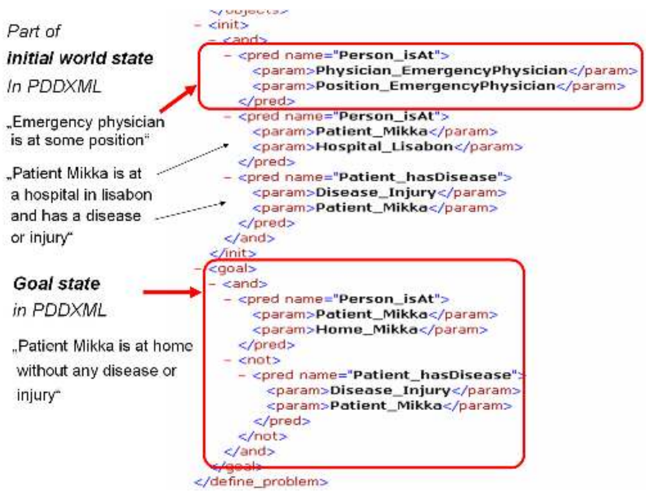
However, the conversion of the output of an individual OWL-S service, that is the information the service offers to the world, to PDDL turns out to be more problematic. The problem is that the service hasEffect condition explicitly describes how the world state will change while this is not necessarily the case for an hasOutput parameter value, though it indeed could implicitly influence the composition planning process. However, PDDL does not allow to describe non-physical knowledge such as train connections produced as an output of a service.

This problem can be solved by mapping the service output parameter X to a special type of the service hasEffect parameter. In particular, every output variable X is described in, and added to the current (physical) planning world state by means of a newly created add-effect predicate in PDDL uniquely named "$agentHasKnowledgeAbout(X)$". Similarly, each input variable Y is mapped to an input parameter Y of an PDDL action complemented by precondition predicate "$agentHasKnowledgeAbout(Y)$". OWLS-Xplan would only use a service description during its planning process, if the additional precondition predicate "$agentHasKnowledgeAbout(Y)$" on available knowledge about service input data is satisfied such that $X = Y$ holds. Otherwise the service execution could fail since checking the service preconditions may reveal that they are not satisfied in the actual world state.



```
<Route rdf:ID="Route_Transport2"/>
<DateTime rdf:ID="DateTime_ArrivalFlight2"/>
<PersonName rdf:ID="PersonName_EmergencyPhysician"/>
<Route rdf:ID="Route_Transport3"/>
<GpsPosition rdf:ID="Position_TransportCompany2"/>
<EmergencyPhysician rdf:ID="Physician_EmergencyPhysician">
  <Person_hasName rdf:resource="#PersonName_EmergencyPhysician"/>
  <Person_hasCreditCard>
    <Creditcard rdf:ID="CreditCard_EmergencyPhysician">
      <CreditCard_hasCreditCardNumber>
        <CreditCardNumber rdf:ID="CreditCardNumber_EmergencyPhysician"/>
      </CreditCard_hasCreditCardNumber>
    </Creditcard>
  </Person_hasCreditCard>
  <Person_isAt>
    <GpsPosition rdf:ID="Position_EmergencyPhysician"/>
  </Person_isAt>
```

Figure 4: Part of initial world state semi-automatically built by OWLS-Xplan editor



```
</objects>
- <init>
  - <and>
    - <pred name="Person_isAt">
        <param>Physician_EmergencyPhysician</param>
        <param>Position_EmergencyPhysician</param>
      </pred>
    - <pred name="Person_isAt">
        <param>Patient_Mikka</param>
        <param>Hospital_Lisabon</param>
      </pred>
    - <pred name="Patient_hasDisease">
        <param>Disease_Injury</param>
        <param>Patient_Mikka</param>
      </pred>
    </and>
  </init>
- <goal>
  - <and>
    - <pred name="Person_isAt">
        <param>Patient_Mikka</param>
        <param>Home_Mikka</param>
      </pred>
    - <not>
      - <pred name="Patient_hasDisease">
          <param>Disease_Injury</param>
          <param>Patient_Mikka</param>
        </pred>
      </not>
    </and>
  </goal>
</define_problem>
```

Part of
initial world state
In PDDXML
„Emergency physician is at some position"
„Patient Mikka is at a hospital in lisabon and has a disease or injury"
Goal state
in PDDXML
„Patient Mikka is at home without any disease or injury"

Figure 5: Part of problem description in PDDXML converted by OWLS2PDDL

Figure 4 shows an example of an inital world state that has been semi-automatically built by the OWLS-Xplan editor. In particular, it currently provides application-oriented templates to the user that allow her to quickly enter, modify, and validate the initial world state and the query, i.e., the goal state, depending on the specific situation and problem at hand. If the user wants to query the agent for a medical transporation service, she only has to fill in a few pre-given templates, thereby setting the values of default parameters of world state and one requested service with related OWL ontologies attached to the template. This initial state and request description is then automatically converted to the corresponding PDDXML problem description by OWLS2PDDL (cf. figure 5). This, in turn, is fed into the planner Xplan to find a solution, i.e. a plan sequence of services or actions on the initial world state that satisfy the given goal.

The AI planner Xplan

Xplan is a heuristic hybrid search planner based on the FF-planner developed by Hoffmann and Nebel (Hoffmann & Nebel 2001). It combines guided local search with graph planning, and a simple form of hierarchical task networks to produce a plan sequence of actions that solves a given problem. This yields a higher degree of flexibility compared to pure HTN planners like SHOP2 (Sirin *et al.* 2004) whereas the use of predefined workflows or methods improves the efficiency of the FF-planner. In contrast to the general HTN planning approach, a graph-plan based planner is guaranteed to always find a solution independent from whether the given set of decomposition rules for HTN planning would allow to build a plan that contains only atomic actions. In fact, any graph-plan based planner would test every combination of actions in the search space to satisfy the goal which, of course, can quickly become prohibitively expensive.

Xplan combines the strengths of both approaches. It is a graph-plan based planner with additional functionality to perform decomposition like a HTN planner. Figure 6 shows an example of how Xplan of OWLS-Xplan uses only those parts of a given method for decomposition that are required to reach the goal state with a sequence of composed services WS_1 and WS_3. In contrast, HTN planning would completely decompose M into WS_1 followed by WS_2, hence output also WS_2 which is of no use for reaching the goal.

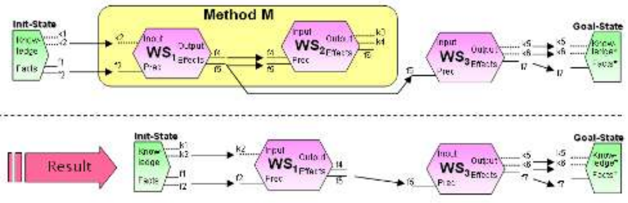


Figure 6: Using parts of methods to reach a goal state in OWLS-Xplan

Overview

The Xplan system consists of one XML parsing module, and following preprocessing modules. First, required data structures for planning are created and filled, followed by the generation of the initial connectivity graph and goal agenda. The latter two actions are interleaved with replanning. The core (re-)planning modules concern the heuritically relaxed

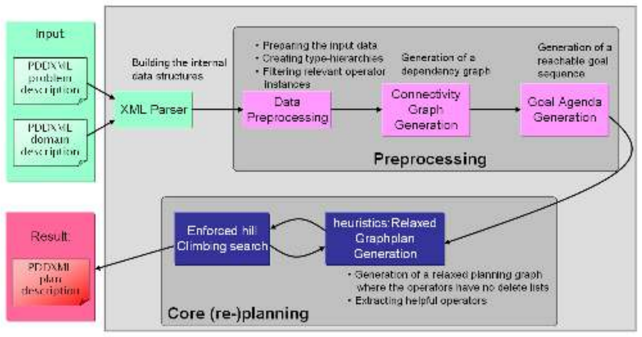

Input
PDDXML problem description
PDDXML domain description
Building the internal data structures
XML Parser
• Preparing the input data
• Creating type-hierarchies
• Filtering relevant operator instances
Data Preprocessing
Generation of a dependency graph
Connectivity Graph Generation
Generation of a reachable goal sequence
Goal Agenda Generation
Preprocessing
Result:
PDDXML plan description
Enforced hill Climbing search
heuristics: Relaxed Graphplan Generation
• Generation of a relaxed planning graph where the operators have no delete lists
• Extracting helpful operators
Core (re-)planning

Figure 7: Architecture of Xplan

graph-plan generation and enforced hill-climbing search (cf. figure 7).

After the domain and problem definitions have been parsed, Xplan compiles the information into memory efficient data structures. A connectivity graph is then generated and efficiently realized by means of a look up table, which contains information about connections between facts and instantiated operators, as well as information about numerical expressions which can be connected to facts. This connectivity graph is maintained during the whole planning process and used for the actual search. In case of a replanning situation, the connectivity graph is adjusted according to the changed new world state.

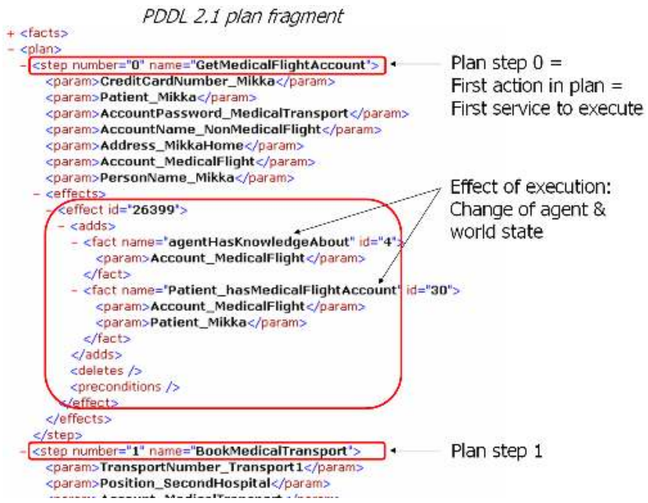
Xplan uses an enforced hill-climbing search method to prune the search space during planning, and a modified version of relaxed graph-planning that allows to use (decomposition) information from hierarchical task networks during the efficient creation of the relaxed planning graph, if required, such as in partially hierarchical domains. Information on the quality of an action (service) are utilized by the local search to decide upon two or more steps that are equally weighted by the used heuristic.

In addition, Xplan includes a replanning component which is able to re-adjust outdated plans during execution time. Therefore, the execution engine informs the planning module about changed world states, and the Xplan replanning component decides whether the remaining plan fragment to execute is still valid or whether a re-planning has to be initiated. Figure 9 shows a fragment of the plan description produced by Xplan, i.e., a sequence of actions, that is the composed sequence of corresponding OWL-S services, that can be executed by the agent.

We implemented Xplan modularly in C++, using the Microsoft MSXML Parser for reading PDDXML definitions and generating plans in XML format. Alternatively, Xplan also provides an interface for direct interchange of planning data without having to use PDDXML as interchange format. Each component of Xplan will be described in more detail in subsequent sections.

Data preprocessing component

Solving a planning process can be viewed as a search problem in the space of all possible combinations of action sequences. Xplan starts off with preprocessing the input data assigning initial values to each predicate of the given (problem and domain) state description in PDDL.


PDDL 2.1 plan fragment
Plan step 0 = First action in plan = First service to execute
Effect of execution: Change of agent & world state
Plan step 1

```
+ <facts>
- <plan>
  - <step number="0" name="GetMedicalFlightAccount">
      <param>CreditCardNumber_Mikka</param>
      <param>Patient_Mikka</param>
      <param>AccountPassword_MedicalTransport</param>
      <param>AccountName_NonMedicalFlight</param>
      <param>Address_MikkaHome</param>
      <param>Account_MedicalFlight</param>
      <param>PersonName_Mikka</param>
    - <effects>
      - <effect id="26399">
        - <adds>
          - <fact name="agentHasKnowledgeAbout" id="4">
              <param>Account_MedicalFlight</param>
            </fact>
          - <fact name="Patient_hasMedicalFlightAccount" id="30">
              <param>Account_MedicalFlight</param>
              <param>Patient_Mikka</param>
            </fact>
          </adds>
          <deletes />
          <preconditions />
        </effect>
      </effects>
    </step>
  - <step number="1" name="BookMedicalTransport">
      <param>TransportNumber_Transport1</param>
      <param>Position_SecondHospital</param>
```

Figure 8: Part of plan description in PDDXML.

Type relation, conversion and simplification of formulas. In the second step, Xplan creates a matrix, that describes all type relations and type inclusions. Predicates which are neither negative nor positive in the effect list of an operator are considered static for the complete planning process, hence are removed from all preconditions and effect lists. Then, the preconditions and effects are converted to disjunctive normal form.

Operator-templates, instantiation and reduction of search space. Xplan creates templates from these simplified operators which are instantiated by all possible combinations of input data based on object instances as described in the PDDL problem description. The set of instantiated operators is then reduced by means of fixed point computation leading to useable and relevant operators. This is achieved by iteratively starting with applying all operators to the initial state. Facts that are added to the state by operators will be stored in a *potential positive facts* list. The respective operators are marked as relevant. This process is repeated until either no new facts nor operators are added to the lists. Operators and facts that are neither reachable nor able to be fulfilled, are removed from the basis set of instantiated operators. Relations between instantiated methods, complex actions and atomic actions are built, to speed up the search and decomposition later on. Furthermore, to guarantee completeness while searching, all negative facts that have a corresponding fact in the potential positive facts list are also stored in the list of relevant facts. Both relevant facts and operators are used to build the connectivity graph.

Generation of the connectivity graph and goal agenda

The connectivity graph is built upon the list of relevant facts, and relevant operators in an iterative process that detects the dependencies between the precondition, add- and delete lists of operators and facts. Once created, the connectivity graph remains static during the search and planning process. In contrast to traditional plan graph algorithms, Xplan does not consider the complete set of goals as a whole but computes an ordered list of goals, the so called goal agenda. The corresponding goal graph is generated based upon this agenda and the FALSE-sets of each goal. Finally, the transitive hull over the goal graph is being computed which is then used to classify goals into goal sets.

Let $(\mathcal{O}, \mathcal{I}, \mathcal{G})$ be a planning problem for which a goal agenda with n goal-sets $Gs_0, \ldots, Gs_n$ exists. The search algorithm starts with the initial state $I_0 = \mathcal{I}$ and the first goal-set Gs_0 as the planning goal G. If a solution P_1 is found which leads from I_0 to Gs_0, then the plan is used on P_1 and I_0. The resulting state $I_1 = Result(I_0, P_0)$ is then used as the starting point for the search using I_1 as initial state and planning goal $G = Gs_0 \cup Gs_1$. Thus all reached goals Gs_0 to Gs_{k-1} remain valid while searching for a solution for Gs_k. For the current planning goal G_k in iteration k it holds that

$$G_k = \bigcup_{i=0}^{k} Gs_i$$

The Xplan search algorithm uses a *no-ops first*-strategy, i.e., goals achieved in previous iterations are marked and only temporally deleted if they will be generated again later on. This guarantees that the planner generates no sub-optimal plans with loops.

The Relaxed Graphplan heuristic

After the goal-agenda has been generated, the search process starts. The search consists of two interleaved processes. The Relaxed Graphplan heuristic (Hoffmann 2000) approximates the distance between the initial state $\mathcal{I}$ to all reachable states S. These distance values are then used to guide the forward directed search. After each successful step the distance values are updated again using the heuristic.

Definition 0.1 *A state $S = (F_S, h_S, N_S)$ is defined as*

- *F_S is a set of all facts which are true in state S.*
- *h_S is distance to the goal given by a heuristic.*
- *N_S is a set of helpful action which can be used in state S.*

Complex actions and hierarchical task networks within relaxed Graphplan We have expanded the Relaxed Graphplan heuristic based algorithm by adding an HTN planner component, and utilization of numerical and boolean facts that can be updated online during the planning phase by external function calls. As a consequence, not only atomic operators but also complex actions and methods are allowed during planning. If a complex action is used while generating a plan graph of which preconditions on some graph layer E_i are satisfied, the HTN component then tries to decompose the complex action using a method-structure element or complex task. A relevant method is searched for by looking up the connectivity graph. Since more than one method could be relevant for decomposition, a heuristic h^d_{htn} is used to determine the most useful one. The selected partial task network itself may contain complex actions that have to be recursively decomposed. Through selection of useable operators O_i of plan graph layer E_i the algorithm first tries to select complex actions. If a solution cannot be found by decomposition, Xplan tries to find a solution without using the HTN component.

External procedure calls Many planners offer the possibility to use numerical values with the standard operators $+, -, *, \div, \ldots$ during the planning process. In most cases these functions are not only bounded in their number but rather hard-coded in the planner such as in the Metric-FF planner (Hoffmann 2003). This is a drawback because the system cannot be expanded without having to change the code. In contrast, Xplan offers the use of so-called external call-back functions. A call-back function is linked to a predicate by means of a fluent variable which contains its return value. This way, Xplan is able to obtain actual information on the value of predicates during the planning proces by calling the linked call-back function. The function returns a boolean value which indicates whether the linked predicate is set or removed from the world state in the next layer of the plan graph. New call-back functions can be added without changing the code of the planner itself.

The fluents are utilized by the planner on both the global fluent-layer of the plan graph, that represents the current state in the computed plan, and the local fluent-layer storing the changed new states of the fluents for their use in the next planning steps. An update of the global fluent-layer is performed each time the fast-forward search finds a better state with respect to a given utility function. The values of the local fluent-layer are used for calculating those facts that are satisfied by executable actions of some layer of the plan graph. The pseudo-code of the algorithm 1 for generating the relaxed plan graph is provided in the annex of this paper.

Extraction of a relaxed plan from the planning graph Let $(\mathcal{O}', \mathcal{I}_x, \mathcal{G})$ the planning problem to solve and PG the relaxed plan graph with k layers. Figure 9 shows a simple example of problem and domain description together with initial part of a corresponding plan graph.

The search for a relaxed plan starts at the top most layer E_{k-1}. For every goal of the current layer E_i, $i < k$, an action of E_{i-1} is selected that satisfies one or multiple goals. Let F_i be the set of facts of layer E_i, then the selection of goals is done by use of a heuristic h_d that measures the barrier for executing an action.

$$h_d(o) := \sum_{p \in pre(o)} min\{i \mid p \in F_i\} + (1 - QoS(o))$$

with $QoS(o) \in]0, 1]$ quality of service of action o. The intersection of the set of selected actions' preconditions of E_{i-1} and the facts of E_{i-1} makes up the goal set G_{i-1} of layer E_{i-1}. This goal set then has to be fulfilled by use of actions of the subsequent layer E_{i-2}. The recursion is going on until the lowest layer of the planning graph with initial facts is reached. This is then the relaxed plan, that consists of an action sequence $A_0 \ldots A_{k-1}$. $k - 1$ is the index of the first layer which contains all goals of the original problem.

PDDL problem and domain description

```
Init (Have(X))
Goal(Have(X) ∧ Used(X))
Action(Use1(X)
        Precondition: Have(X)
        Effect: ¬ Have(X) ∧ Used(X))
Action(Use2(X)
        Precondition: ¬ Have(X)
        Effect: Have(X))
```

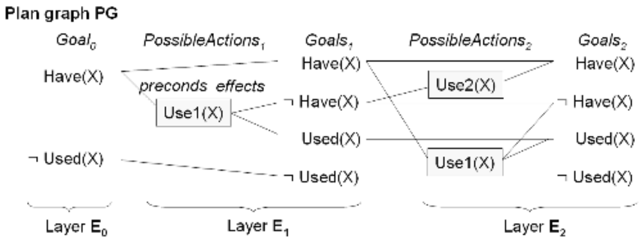


Figure 9: Part of a plan graph with $k = 3$ layers

To get an approximation how far away it is from state $\mathcal{I}_x$ to goal $\mathcal{G}$, a heuristic $h(\mathcal{I}_x)$ (described in (Hoffmann 2000)):

$$h(\mathcal{I}_x) := \sum_{i=0}^{k-1} |A_i|$$

The value $h(\mathcal{I}_x)$ indicates the length of the relaxed, sequential solution of $\mathcal{O}$ starting with $\mathcal{I}_x$. This value in combination with the state information $\mathcal{I}_x$ constitutes the new search state S which is included into the search space of the fast forward search.

Detecting helpful actions

In addition to the heuristic, Xplan computes the set H_S of helpful executable actions for every search state S such that the goal $\mathcal{G}$ eventually can be reached.

Definition 0.2 *A* helpful action *of a search state S is an action, that satisfies at least one proposition of the goal set G_1 of the first layer in the plan graph. The set of helpful actions is described as follows:*

$$H_S(S) := \{o \mid (pre(o) \subseteq S) \cap (add(o) \cap G_1(S) \neq \emptyset)\}$$

If there are many helpful actions, then actions of an HTN decomposition are preferred. The reason is, that such actions are more highly probable to be succeeded by an useful action in the task network as part of the relaxed plan.

Local search with enforced hill-climbing

Xplan uses an enforced hill-climbing search algorithm to search for best reachable states during generation of the global plan to satisfy a given goal. It combines the standard search strategy with a breadth search for a better state than the given one not only in its direct neighborhood but within the set of successor states of S that are reachable by applying a helpful action of N_S. This search strategy performs as follows.

- Compute the distance between the starting state $\mathcal{I}$ and the goal state G by use of Relaxed Graphplan, and the set of helpful actions I.
- Initialize the enforced hill-climbing with $\mathcal{I} = (F_{\mathcal{I}}, h_{\mathcal{I}}, N_{\mathcal{I}})$ as input.
- Enforced Hill-Climbing analyzes all reachable states that have been computed. It assigns each state with its approximative distance to the goal by use of Relaxed Graphplan.
- If a better state is found, then include this state into the current plan, and use it as a basis for further search. Update all fluents on the current layer by invoking the respective call-back functions.
- Terminate if a state $S' = \mathcal{G}$ is reached in which all given goals are satisfied. Otherwise, if not at least one goal has been achieved, the search failed. In this case, a new complete breadth search is instantiated on $\mathcal{I}$ to find a solution, if it exists.

The pseudo-code of the local search by enforced hill-climbing algorithm is shown in algorithm 2.

Re-planning component

During plan execution, the agent has to check for each action of the plan whether its preconditions hold, or not. If at least one precondition is not satisfied, Xplan gets informed about which facts are invalid, at which position in the plan this problem occurs, and then checks whether the original plan still can be executed. Otherwise, it tries to fix the problem by searching for an alternative path in the connectivity graph from the actual position in the plan to the goal state. In addition, it may temporally block unnecessary actions to reduce the search space, thereby avoiding a complete preprocessing phase.

Plan patterns

Xplan builds ordered service composition plan sequences only whereas OWL-S allows for more complex plan structures such as unordered, choice, if-then-else, iterations, repeat-until loops, repeat-while loops, split, and split+join. However, complex plan structures can be formed out of the produced plan sequence based on its appropriate analysis and interpretation in posterior. For example, figure 10 shows how a plan sequence looks like, that can be transformed into a split+join structure. In this case, the input of WS_2 does not depend on the output of WS_1, hence both services may be executed concurrently. Since both are required to achieve the given goal, their results have to be joined after execution.

Figure 11 shows an example of how to realize a split structure. Like in previous example, both services neither influence each other, nor share a common goal to reach, thus can be executed in parallel.

However, the plan structures "choice" and "unordered sequence" are not realizable by proper interpretation of plan sequences created by Xplan. Though, the latter problem is a hard problem for any AI planner in general, including, for example, Shop2 (Sirin *et al.* 2004).

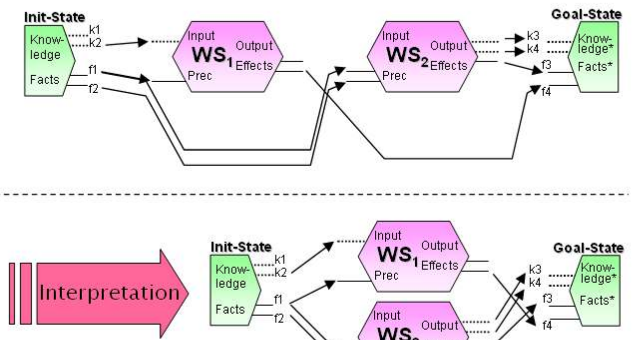


Figure 10: Split + Join interpretation

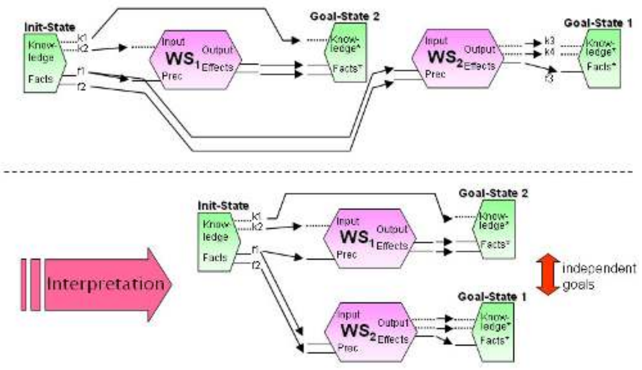


Figure 11: Split structure of an OWLS-Xplan plan

Related work

A logic-based DAML-S composition planner has been developed at the UMBC, USA (Sheshagiri, desJardins, & Finin 2003). This planner uses STRIPS-style services to compose a plan, given the goal and set of basic services. It is implemented with JESS (Java Expert System Shell), and uses a set of JESS rules to translate DAML-S descriptions of atomic services into planning operations.

One of the currently most prominent service composition planners is Shop2 (Simple Hierarchical Ordered Planner 2) developed at the University of Maryland, USA (Wu *et al.* 2003). It is a hierarchical task network (HTN) planner well-suited for working with the hierarchically structured OWL-S process model. The authors proved the correspondence between the semantics of Shop2 and situation calculus semantics of the OWL-S process model. The implemented Shop2 soundly and completely plans over sets of OWL-S descriptions, and treats the output of a web service as effects that either change the planning agent's knowledge, or the world state. Shop2, like HTN planner in general, replaces those elements of the provided methods (workflows) by special methods or atomic actions until the composition plan contains only atomic actions that correspond to available web services. During planning, web services are not executed, hence do not affect the world state.

Both Xplan and Shop2 base on the closed world assumption, use PDDL for problem description, allow external (call-back) functions to be bounded to variables and executed during planning, and generate total ordered, instantiated plan sequences for a given initial state, goal and planning domain. Among others, the main difference between Shop2 and Xplan is inherent to the individual planning processes. In essence, Shop2 plans are generated by use of given decomposition rules (methods), hence a solution to the planning problem is not always guaranteed to be found (Lotem, Nau, & Hendler 1999). In contrast, hybrid Xplan as part of OWLS-Xplan tries to plan by means of (a) method decomposition using only relevant parts of it, discarding useless actions, thereby reducing the plan size, and (b) if this is not successful, uses its relaxed graph plan algorithm to find a solution, if it exists.

Conclusion

We presented an OWL-S service composition planner, called OWLS-Xplan, that allows for fast and flexible off-line composition of OWL-S services by use of an OWLS2PDDL converter, and a hybrid AI planner that combines relaxed Graphplan FF-planner with local search and HTN based planning, and a re-planning component. OWLS-Xplan has been implemented in C++ and Java, and is currently in use in a prototyped medical health information service system. It is intended to make the OWLS-Xplan code package available to the community at www.semwebcentral.org.

References

Hoffmann, J., and Nebel, B. 2001. The FF Planning System: Fast Plan Generation Through Heuristic Search. *Journal of Artificial Intelligence Research (JAIR)* (14):253–302.

Hoffmann, J. 2000. A heuristic for domain indepndent planning and its use in an enforced hill-climbing algorithm. *Proceedings of 12th Intl Symposium on Methodologies for Intelligent Systems, Springer Verlag*.

Hoffmann, J. 2003. The Metric-FF planning system: Translating Ignoring Delete Lists to Numeric State Variables. *Artificial Intelligence Research (JAIR), vol 20*.

Lotem, A.; Nau, D.; and Hendler, J. 1999. Using planning graphs for solving HTN problems. *Proceedings of AAAI/IAAI conference, USA*.

Schmidt, M. 2005. Ein effizientes Planungsmodul fuer die lokale Planungsebene eines InteRRaP Agenten. Master's thesis, Universitaet des Saarlandes.

Sheshagiri, M.; desJardins, M.; and Finin, T. 2003. A planner for composing services described in DAML-S. *Proceedings of AAMAS 2003 Workshop on Web Services and Agent-Based Engineering*.

Sirin, E.; Parsia, B.; Wu, D.; Hendler, J.; and Nau, D. 2004. HTN planning for web service composition using SHOP2. *Journal of Web Semantics, 1(4)* 377–396.

Wu, D.; Parsia, B.; Sirin, E.; Hendler, J.; and Nau, D. 2003. Automating DAML-S web services composition using SHOP2. *Proceedings of the 2nd International Semantic Web Conference (ISWC2003), pages 20-23, Sanibel Island, Florida, USA*.

```
function BuildRelaxedPlanningGraph() computes
relaxedPlanningGraph or fails
input: InitialFacts[] : List of Facts
input: GoalFacts[] : List of Facts
local: CurrentLayerFacts[], NextLayerFacts[] : List of
Facts
local: CurrActivActions[] : List of Actions
local: CurrentLayer : int
begin
    CurrentLayerFacts = IntitialFacts;CurrentLayer = 0;
    while ! AllGoalsActive(GoalFacts) do
        foreach Fact $f$ in CurrentLayerFacts do
            Increment precondition counter of actions
            which $f$ is a precondition of;
        end
        foreach Fact $f$ in CurrentLayerFacts do
            /* First collect all action,
               that are a result of a
               method decomposition and
               compute the layer, when it
               is earliest executed.    */
            CurrActivHTNActions +=
            GetActiveHTNActions(CurrentLayer,$f$);
            /* Select all remaining
               executable actions, that
               are part of the
               current-layer            */
            CurrActivActions +=
            GetActivePrimitiveActions(CurrentLayer,$f$);
        end
        foreach Action $a$ in CurrActivHTNActions do
            if all preconditions of $a$ are satisfied AND
            Layer of $a$ == CurrentLayer then
                /* $a$ is executable, all
                   preconditions are
                   fulfilled and is
                   executable in the layer
                   */
                NextLayerFacts +=
                GetAddedFactsFromAction($a$);
                RemoveFromList($a$,CurrActivHTNActions);
            end
        end
        foreach Action $a$ in CurrActivActions do
            if all preconditions of $a$ are satisfied AND
            Layer of $a$ == CurrentLayer then
                NextLayerFacts +=
                GetAddedFactsFromAction($a$);
                RemoveFromList($a$,CurrentActivateActions);
            end
        end
        CurrentLayerFacts = NextLayerFacts;
        NextLayerFacts = <>;
        /* Increasing layer counter and
           continue with next layer.   */
        $CurrentLayer = CurrentLayer + 1$;
        if CurrentLayerFacts == <> then
        /* If a fix point is reached
           regarding facts and actions
           and the goal isn't fulfilled,
           the problem isn't solvable.
           */
        if ! AllGoalsActive(GoalFacts) then
            return FAILURE;
        end
    end
    return CurrentLayer;
end
```

Algorithm 1: Generating a relaxed plan graph

```
function DoEnforcedHillClimbing() computes
validPlan: Plan or fails
input: InitialState : State input: GoalState : State
    local: S : State /* the current computed
    state                                  */
local: S' : State /* possible successor of S
    */
local: currPlan : Plan /* current plan     */
local: $h_S$ : int /* the distance of S to a
    goal computed by use of Relaxed
    Graphplan                              */
local: $h_{S'}$ : int local: $N_S$[] : List of Actions /* List
    of helpful action based on state S
    */
local: $N_{S'}$[] : List of Actions
begin
    /* The initial plan is empty      */
    currPlan = <>;
    S = InitialState; /* Compute the distance
       from starting state to goal    */
    $h_S$ = BuildRelaxedPlangraph(S,GoalState);
    /* Compute helpful actions for S
       */
    $N_S$ = GetHelpfulActions(S);
    while $h_S \neq 0$ do
        /* Searching with breadth search
           for a state S' with $H_{S'} < H_S$
           within $N_S$ and their
           successors.  BFS_Expand
           computes for every relevant
           state the distance between
           goal and helpful actions.
           This is done by
           BuildRelaxedPlangraph and
           GetHelpfulActions            */
        S' = BFS_Expand(S,$N_S$);
        if S' == NULL then
            return FAILURE;
        else
            /* If a state S' is found,
               the action sequence is
               attached to the end of the
               current plan, that enables
               to get from S to S'.     */
            $currPlan =$
            $currPlan + ActionsPath(S, S')$;
            /* Update fuents of the
               global fluent-layer      */
            UpdateGlobalFluents(S,S');
            /* The search goes on
               beginning with S'. $N_{S'}$ is
               computed before by
               BFS_Expand and can still be
               use.                     */
            S = S';
            $N_S = N_{S'}$;
        end
    end
    return currPlan;
end
```

Algorithm 2: Local Search by enforced hill-climbing