University of Zurich
Department of Informatics

Dynamic and Distributed
Information Systems

Semi-Automated Semantic Web Service Composition Planner Supporting Alternative Plans Synthesis and Imprecise Planning

Diploma Thesis January 6, 2007

Peter Birchmeier
of Zürich ZH, Switzerland

Student-ID: 00-702-738
pbirchmeier@gmx.ch

Advisor: **Michael Dänzer**

Prof. Abraham Bernstein, PhD
Department of Informatics
University of Zurich
http://www.ifi.unizh.ch/ddis

Abstract

NExT (Next Experiment Toolbox) is a system that provides guidance to the user during the whole lifecycle of experiments in the life science domain. These are typically composed out of many atomic complex and potentially long-running tasks, which are grounded by a heterogeneous tool set.

This thesis extends NExT with a planning component for semi-automated composition of OWL-S services while coping with non-determinism and incomplete knowledge. The solution offers a set of non-standard planning features, namely synthesizing plan alternatives, imprecise planning in cases of no or too few solutions, and planning with complex goals and QoS optimization criteria. The concept decouples planning tasks from their problem solving. It proposes a novel approach to encode planning problems and user requested cooperative planning features as PDDL 3.0 planning problems, and to allocate them to appropriate distributed planners. Aside the deployment of state-of-the-art planners, planner concepts for synthesizing plan alternatives and for dealing with the no-solution case are shown.

Zusammenfassung

NExT (Next Experiment Toolbox) ist eine Applikation, welche den Benutzer während des gesamten Ablaufs von Experimenten in angewandten Wissenschaften unterstützt. Experimente bestehen aus vielen komplexen atomaren und teilweise zeitaufwendigen Aufgaben, welche mittels heterogenen Tools bewerkstelligt werden.

Diese Diplomarbeit erweitert NExT mit einer Planungskomponente, welche halb-automatisiert OWL-S Services zusammenstellt und gleichzeitig dem Nicht-Determinismus und unvollständigem Wissen Rechnung trägt. Die Lösung bietet eine Reihe von aussergewöhnlichen Planungsfähigkeiten, nämlich Synthese von alternativen Plänen, ungenaues Planen in Fällen von keiner oder zu wenigen Lösungen, und Planen mit komplexen Planungszielen und Qualitäts-Optimierungskriterien. Das Konzept entkoppelt Planungsaufgaben von ihren Problemlösungen. Ein neuer Ansatz wird vorgeschlagen, der Planungsprobleme und vom Benutzer gewünschte kooperative Planungsfähigkeiten als PDDL 3.0 Planungsprobleme kodiert, und der diese Probleme geeigneten verteilten Planern zuweist. Neben dem Einsatz von existierenden Planern werden Konzepte zu zwei weiteren Planern vorgestellt: der eine generiert Planalternativen, der andere widmet sich dem Problem, dass keine Lösung existiert.

Contents

1

Introduction

Imagine one has a laboratory environment where scores of tasks, which are sometimes very simple and similar and sometimes complex, have to be composed in a sequence in order to carry out recurrent experiments. Moreover, tasks would have to be fulfilled at different locations. One would wish that those experiment processes could be supported or automated by computer help, i.e. processes could be modeled by a computer application, executed in a process execution environment and reused once created, or the application helps to automatically compose the needed tasks for a given experiment.

Michael Dänzer has followed this approach in his master's thesis *NExT - The NMR Experiment Toolbox* [Dänzer, 2005] by analyzing the research environment of NMR experiments at the Institute of Physical Chemistry from the Swiss Federal Institute of Technology (ETH) in Zurich. He mapped needed processes for those experiments and domain specific properties to a process ontology in OWL-S, a Semantic Web service description language capable of defining processes in a machine readable manner. Using the application NExT[1], processes can be composed and then be executed in a process execution environment. Furthermore, the user can monitor and influence the execution.

This thesis extends the project NExT and addresses the issue of how processes can be semi-automatically composed. The idea is to relieve the user from composing the necessary process steps by hand, and to let the user profit from the proposed process plans which are generated either from atomic processes or from existing and proven composite processes. The proposed solution focuses on the integration of the user into the planning process and thus enables planning with cooperative methods, i.e. it applies a set of non-standard planning features such as the synthesis of plan alternatives and imprecise planning. Equipped with these features, it ensures a valuable support for the user.

[1] *NExT* is both the name of the master's thesis and the name of the software project created by M. Dänzer. Henceforth, the term NExT will be used to refer to the software application.

1.1 Goal of the Thesis

The planner builds on top of the goals of NExT which in turn evaluate from the NMR experiment domain. The development of the planner component targets the improvement of experiment workflows and has to be aligned to their requirements.

Since the planner has to deal with pre-defined Semantic Web services, it should be capable of dealing with the typical wattles in the domain of Semantic Web services like potential non-deterministic behavior of the services and the uncertainty about knowledge completeness at the time of planning. While the automated composition of Web services is an emerging topic of high interest, the scope of the planner presented goes beyond that of pure Web Service composition solutions as they normally exist. The planner namely focuses on some specific additional sub-goals in order to seemly extend the NExT project:

- **User Interaction**
  A main goal of the NExT Project is to guide the user at his work, for example during plan composition. Since planning is a complex task and sometimes the planner depends on the support of the user, for example in the scenario that no plan can be found on a first attempt , the planner should interact with the user to successfully optimize a planning problem and the resulting problem solution.

- **Plan Alternatives**
  It is never guaranteed that a calculated plan, even if it is the optimal one for a given problem, fits the requirements and wishes of the user – it lies in the nature of a human being to have an affinity toward making decisions upon proposed alternatives. Therefore the planner should provide the user with some alternative plan propositions. If possible, the plan propositions should be annotated with some sort of quality information.

- **Imprecise Planning**
  Aside consistent plan solutions, the planner should calculate plans that are not fully perfect and in cooperation with the user. This is necessary to not withhold the user with a plan path direction only because some knowledge misses, or more probably because the planner could not find a plan.

The concept presented in this thesis not only contributes to the project NExT in an effective way, but represents a practical and innovative solution for the field of composing Semantic Web services with the help of planning techniques.

1.2 Structure

Please find next short descriptions of all chapters to get a rough overview about the thesis structure.

Section 2 gives insight into the motivation reasons of this thesis, namely NExT embedded in the environment of NMR experiments, and the research topic of automated Web service composition. Some background aspects are discussed as well to introduce the reader into the problematics of the topics.

Section 3 lightens some fundamentals about problems arising in the domain of AI planning and explains selected planning techniques such as algorithms that are relevant for the thesis.

Section 4 provides a collection of all accumulated assumptions, requirements and conditions. They were obtained under different aspects, for example from NExT, deduced from the thesis goals or from their analysis, and partially from design decision as well.

Section 5 is all about deciding which design concepts could best fulfill the requirements demanded for the development of the planning component. It starts with an extensive analysis of the thesis goals, evaluates several plausible design alternatives afterwards and concludes with decisions in favor of a main concept.

Section 6 presents the proposed concept about the planning component. It explains fundamental principles influencing all other concepts, the various elements of the architecture and its functions. It moves forward to describe mappings from OWL-S to PDDL. One Section only attends to all details about AI planners, including their organization and some concrete solutions. Sub-concepts are addressed at the end.

Section 7 gives an overview about the software parts that were implemented according to the main concept in order to confirm assumptions and decision made with the concept. A discussion about the results and other aspects is attached.

Finally, Section 8 concludes with a summary about the concepts in respect of the thesis goals and some propositions for future work.

2
Motivation

The major motivation comes from the underlying project NExT which targets the simplification and automation of the working conditions in the NMR experiments domain. Some important work has already been done and some is open. An integration of a planner component into NExT would greatly contribute to the project and help in achieving the defined goals. (see Chapter 2.1)

Furthermore, the consolidation of Semantic Web services and planning techniques is currently a striking topic of research. Several solutions have been presented that try to merge the benefits from both domains. An analysis should therefore address the question how far the approaches go, and it might be interesting to develop some new ideas. (see Chapter 2.2)

2.1 Project NExT

[Dänzer, 2005] addresses the problems existing in the domain of NMR experiments and presents an overall solution to them. The problems are mainly the lack of explicit domain knowledge and the insufficient tool support. The latter can be described as following: For most experiment tasks there exist tools, but most of them were developed for single aspects only and thus have to be used in rather monolithic-like fashion. All of them have their own specific interface and usage method. Furthermore, in some circumstances tools for certain tasks do not even exist. Also, a corporate tool that could integrate these tools at least to a certain degree for making process plans does not currently exist. In this heterogeneous environment, the user is obliged to manually execute the various tools, to adapt his work style to the peculiarities of the tools and to have the data of the experiments spread across different locations.

The solution includes among others

1. a process model definition in OWL-S to formally describe processes in the NMR domain,

2. a software solution (NExT) which allows (a) the definition of process plans for NMR projects, (b) the execution of these in a (semi-) automated mode while the user is guided through all stages, and

3. the knowledge exchange within the NMR community.

One fundamental approach to solve the problem was to analyze the processes of the NMR experiments regarding the specificity frontier [Bernstein, 2000]. The specificity frontier corresponds to the specificity spectrum of processes, ranging from highly specified (fixed work processes) to highly unspecified (ad-hoc processes like e-mail communication). The processes in the NMR experiments domain were identified either to be fully specifiable or in case of changeable composite processes to have a specificity lying somewhere in between the two extremes. [Bernstein, 2000] presents ways to cope with processes of different specificities: (1.) dividing the specificity frontier into sub-spectra and (2.) providing runtime transfer-mappings between the sub-spectra. These key ideas were incorporated into NExT by designing the process models according to them and by leaving the altitude of either composing process plans by hand or requesting the software to propose automatically derived process plan alternatives to the user. The procedure is known as the *Mixed-Initiative* feedback loop [Veloso, Mulvehill, & Cox, 1997].

On one hand, parts of the software NExT have already been implemented, for example, it is possible to create new experiment cases, to compose process plans and to execute and monitor them. On the other hand, the implementation of a key aspect is open, namely the software's ability to provide the user with plan alternatives. Regarding the specificity frontier, this would address the software support for under-specified process plans. There are various approaches for a realization[1]:

- Synthesis of new process plans from scratch.
- Case-based reuse and adaption of existing process plans (case-based planning).
- Plan-merging: The problem is decomposed into sub-problems and solved separately (for example by several distributed agents), whereon the resulting sub-plans are merged.
- Plan-rewriting: A single plan solution is repeatedly modified to get better solutions.
- Any combination of the above.

This thesis attends to the synthesis of new process plans by elaborating a planner based solution.

[1] The classification is done from the point of view of the synthesis form, though case-based planning in the form of pure reuse of existing plans [Spalzzi, 2001] might not be regarded as a synthesis. Mentioned are approaches relevant for NExT, but there are others, for example situated planning which adapts planning according to the need for reactivity by choosing planning layers with increasing reactivity and decreasing planning depth.

To get an impression of the planner's functions, let us consider the following sample scenario: A user has created a new experiment case and is currently composing a process plan in the composition editor. He first defines the tasks which have to be done in order to achieve the experiment goals. He then starts with attaching some atomic process steps. He decides to use the planner which could be due to various reasons:

- The user does not know how to continue the process step composition. So, the planner can help him find some solutions, if there exist some at all.
- The user has an idea of a process plan, but wants to check whether he overlooked a solution that could be better than the one in his mind. The planner will find some alternative solutions.
- The user wants to save time, regardless of the complexity of the experiment tasks. The resulting plan is then used as template.

After the execution of the planner, it might be possible that no plan could be found or that the found plan alternatives are not satisfying. In such a case, the user can relax some constraints, for example mark a sub-goal that not necessarily has to be attained, or add some new constraints, for example set the duration of process steps, in order to optimize the retained plans. Subsequently, the planner is started again. The use case in Figure 2.1 abstracts all possible scenarios.

Recovering the issue about the specificities of processes, we remark that all just outlined sub-scenarios dealt with processes that can be placed on the specificity frontier somewhere between enough specified to be usable for planning and highly specified.

Use Case UC1: Plan Generation Process

Primary Actor: User
Stakeholders and Interests:
- User: Wants to find a plan.
- Research Group:
Preconditions:
Success Guarantee (Postconditions): A satisfying plan was found.

Main Success Scenario (or Basic Flow):
1. *User* decides to start the planner component.
2. *User* configures the planner.
3. *System* calculates 1-n plan solutions.

User repeats steps 2-3 until plans are enough satisfying.

4. *User* selects among the plan alternatives the most appropriate one.

Extensions (or Alternative Flows):
2a. *User* relaxes some domain, initial state or goal state constraints.
2b. *User* gives some additional information.
3a. *System* can't find any plan solution.
    1. *System* indicates the relaxation potential.
        1a. *User* is interested in partial plans.
            1. *System* provides the User with 1-n partial plans.
        1b. *User* quits the plan generation process.
3b. *System* consumes too much calculation time;
    eventually some plans are already calculated.
    1a. *System* stops the calculation, if a timeout has been defined.
    1b. *User* stops the planning process.

Use case format is according to [Cockburn, 2000].

Figure 2.1: Use Case: Plan Generation Process

2.2 Semantic Web Services and AI Planning Techniques

The increasing need for application integration inside companies and more and more between companies entails the request for a corresponding technical infrastructure. Web services have jumped into the gap and could become popular thanks to their versatility and platform independence. The business world has built on various W3C passed XML-standards to describe Web services (WSDL). Unfortunately, service interfaces are of syntactical nature, similar to those of remote procedure calls. This turned out to be a disadvantage for several reasons: (1.) Web services have to be annotated with additional information about their functionality and (2.) flows of Web services have to be manually defined in a service flow specification language like PBEL4WS or WSCL. In other words, the definition of Web services is very complex.

The Semantic Web community has developed Web service frameworks that build on top of Semantic Web ontologies which bring semantical enrichment to Web services. For example, by means of preconditions and effects world altering facts can be described. The logical descriptions in the underlying ontologies and the possibility to reason about their statements suggest the appliance of AI to automatically compose Web services. The combination of both techniques holds high potential and has stimulated the industry's interests.

At the time of writing, Semantic Web services combined with the power of AI have rarely been used in practice because Semantic Web services still leave some important questions, such as security, unanswered. Furthermore, the combination is hardly trivial, due to the fact of their diametrically opposed origins. While research on AI planning techniques has concentrated during the last three decades on how world knowledge like objects and constraints can be reduced to minimal, deterministic and closed-world assumed representations in order to move the problem's complexity to a machine computable dimension. Semantic Web services followed the spirit of the World Wide Web and enables information exchange and reasoning about resources in an open world. The semantic web is much closer to the reality.

Bringing together these techniques is a big challenge which in turn yields some crucial problems:

- **Non-Determinism**
  The domain of Semantic Web services is highly non-deterministic due to several reasons: Web services can fail or return different types of results on which the subsequent process execution might depend, or some relevant world facts can change during process execution. These fully or partly unpredictable behaviors are the contrary of what an offline planner normally wants to have. A classical planner prefers fully predictable world changes.

- **Incomplete Knowledge**
  Planning techniques assume to have complete information about the initial world state (closed world assumption). In reality much information is not given at planning time but available at a later time. To this belongs the fact, "that in contrast to classical planning, where all objects are available in the initial state and the actions change the state of objects, web services create new objects at runtime" [Srivastava & Koehler, 2003].

- **Complex Process Plans**
  Planning techniques normally generate plans with sequential order of the plan steps (partial order planning is an exception). But in order to cope with the non-deterministic behavior of Web services, a planner should be able to generate plans that include complex control structures such as loops, choices, or even parallel execution [Giunchiglia & Traverso, 1999]. The handling of exceptions and conditions could thus be compiled into plans.

- **Complex Goals**
  Planning techniques require explicit goal specifications. "Unfortunately, [..] explicit goals are usually not available from an industrial perspective." [Srivastava & Koehler, 2003] Goals often include *inexplicit goals* which get satisfied during the process plan instead at the end of it. "For example, the explicit goal of a travel reservation process is to perfectly organize the travel, while its (more or less) implicit goal is the creation of the required travel documents in some data base." [Srivastava & Koehler, 2003]

Several solutions have been proposed to the issue, most of them do a full automated planning stage, i.e. with a given domain and problem specification they try to get out the best possible plan. Since this approach corresponds to the hardest possible way, those solutions normally succeed to solve one problem aspect very well, whereas other aspects are rather suboptimally solved. There are mainly two critiques: (1.) It is questionable whether it is necessary or even possible to find a perfectly suitable plan taking into account all conditions and problems which could arise in future executions of Web services. (2.) Furthermore, not all goals might be known at the time of planning but evolve during execution of the plan. In this context, [Srivastava & Koehler, 2004] argues that "Web services composition can not be seen as a one-shot plan synthesis problem defined with explicit goals but rather as a continual process of manipulating complex workflows, which requires to solve synthesis, execution, optimization, and maintenance problems as goals get incrementally refined".

I argue too, for both aspects, that a Web service composition problem is more complex and depends on additional support from the user and on results of the continuous execution of the plan path as it has been calculated so far. The sub-goals of this thesis, namely (a) user interaction, (b) plan alternatives and (c) compromised planning, follow that idea and give a good starting position to see the Web service composition problem from a different and promising angle.

2.3 Related Work

At the time of writing, there is no project that could completely fulfill the goals of the planning component to be developed. Instead of that, several solutions have been presented to the automated Web service composition problem, each of them with a slightly different focus. Next, the most relevant works are outlined, giving an idea of the different approaches. The various planning techniques that could be used for the planning component are not covered here since their research scope has a more general nature and they would require a large effort to be usable in the project as-is.

2.3.1 SHOP2

SHOP2 [Sirin, Parsia, Wu, Hendler, & Nau, 2004] is a well-known representative of HTN (Hierarchical Task Network) based planners for composing Semantic Web services. SHOP, the predecessor of SHOP2, was one of the first out-of-the-box planning solutions for the automated Web service composition.

SHOP2 extracts decomposition methods from composite processes of OWL-S domain ontologies and uses them to decompose a given planning problem to atomic process steps. This process can be viewed as the specialization of pre-written composite processes which results in a sequential process plan. In contrast to other HTN planners, SHOP2 plans for tasks in the same order that they will later be executed. Hence, current state of the world is known at each step in the planning process which enables the SHOP2s precondition-evaluation mechanism to incorporate significant inferencing and reasoning power.

SHOP2 offers additional capabilities to optimize for certain variables such as costs due to the fact that the planner considers the entire execution path. Since the decomposition of composed processes is a straight forward approach for automated web service composition, SHOP2 has good performance attributes and scales well to large numbers of methods and operators.

WSC-SHOP2 [Kuter, Sirin, Nau, Parsia, & Hendler, 2006] extends SHOP2 to support information gathering during planning. It does so by querying the truth values of certain atoms when there is not enough information in the knowledge base to determine their values. The planning process does not need to wait for the responses of the services and can continue planning while the service is still executing.

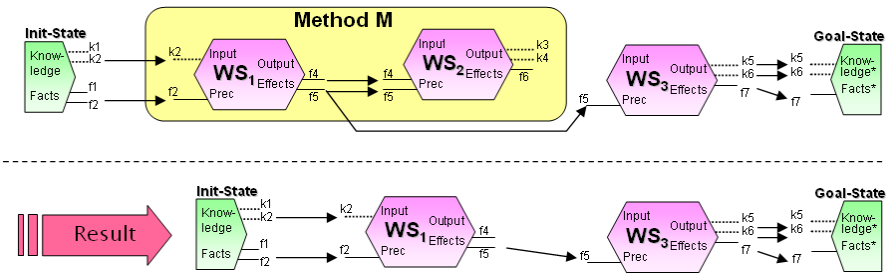
Software Used

SHOP2 was originally written in Lisp and was then also ported to Java (JSHOP2). The sources are available under Open Source license.

2.3.2 OWLS-Xplan

OWLS-Xplan [Klusch, Gerber, & Schmidt, 2005] allows for automatic composition of Semantic Web services defined in OWL-S. The software component was developed in the context of the SCALLOPS project[2].

OWLS-Xplan is a hybrid planner based on state-space forward search and an additional HTN like web service decomposition module. In order to achieve high performance, the Fast-Forward planning system [Hoffmann & Nebel, 2001] was used and extended. The HTN module helps to decompose composite Web services as they exist in domains with partly or fully domain specific and detailed knowledge, and utilizes the necessary atomic parts only (see Figure 2.2). This enables OWLS-Xplan to cope with domain-unspecific as well as domain-specific planning problems, i.e. it is able to find solutions in either case. The planner returns sequential plans and allows after the plan generation for exchanging equivalent web services while optimizing plans by means of QoS metrics.



Source: [Klusch et al., 2005]

Figure 2.2: OWLS-Xplan: Use of decomposed processes

At the time of writing, OWLS-Xplan was not fully implemented yet. A complete version should be available in February 2007.

Software Used

OWLS-Xplan's data collection and optimization module is written in Java, whereas the planner itself is a windows executable whose source files are not provided. A proprietary language PDDXML according to PDDL is used to retain the extracted data from OWL-S web services and domain knowledge. Since PDDXML has XML syntax, it "simplifies parsing, reading, and communicating PDDL descriptions using SOAP" [Klusch et al., 2005].

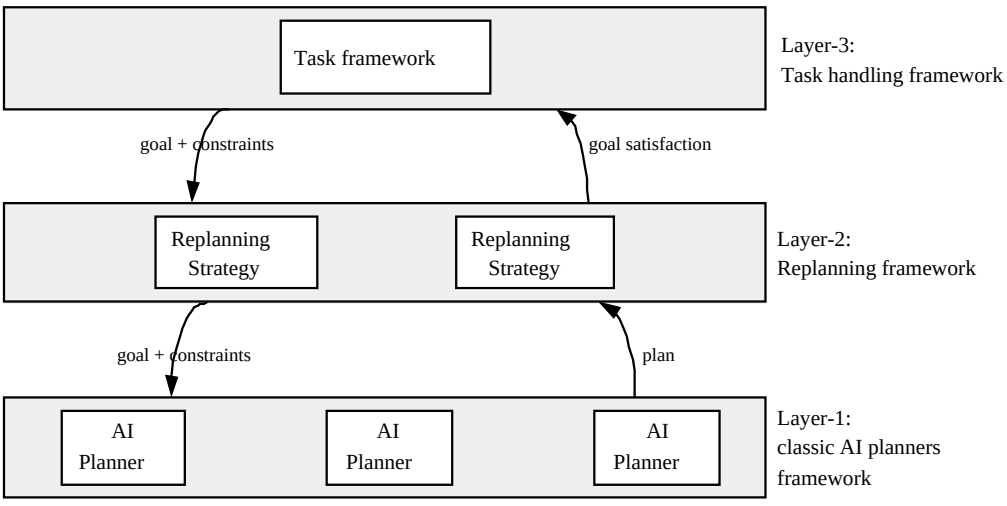
[2]Secure Agent-Based Pervasive Computing, see http://www-ags.dfki.uni-sb.de/~klusch/scallops/

2.3.3 WSPlan

WSPlan, the PDDL based tool [Peer, 2004], has been elaborated in the dissertation of Peer. It treats the Web service composition problem as a whole and tries to take all aspects into consideration. It argues that the diversity of the Web service domains is best addressed by a flexible combination of complementary reasoning techniques and planning systems.

The presented tool decouples the different concerns like goal description, planning and plan execution from particular planning technologies and implementations: A Web service composition problem is first converted to an equivalent AI planning problem. Then the most suitable planner is chosen for the particular planning task. The decoupling has been further expanded in [Peer, 2005] to the *three layer concept* (Figure 2.3 illustrates the collaboration of the three layers):

- Layer 3: Task Handling
  Finds an executable path for a defined task.
- Layer 2: Replanning/Monitoring
  Solves a Web service composition problem considering the typical issues like incomplete knowledge, non-deterministic behavior of operations and the construction and the destruction of objects.
- Layer 1: Classical AI Planner
  Constructs a sequential plan with complete knowledge and fully deterministic operations.

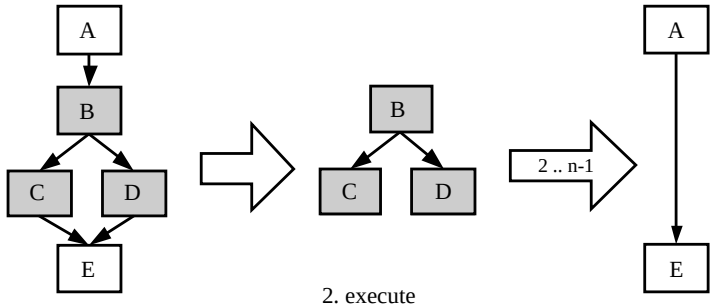


Source: [Peer, 2006]

Figure 2.3: WSPlan: Three Layer Concept

More specifically, layer two takes as input Web services defined in WSDL and marked up with semantics on services and operations, and converts the given problem to PDDL. At the same time, a table of causal links is created with whom the world state after a plan step execution can be verified. The generated plan is then step wise executed and if the causal links indicate an unexpected state, a replanning process is started.

WSPlan uses *sensing sub-plans* to get around the incomplete knowledge problem at planning time. A plan is calculated and if parts of it consist of services that only gather information but do not alter the world state, these sub-plans are executed before all others (see Figure 2.4). The resulted knowledge acquisition simplifies the further (re-)planning and plan execution process.



Source: [Peer, 2005]

Figure 2.4: WSPlan: Sensing Sub-Plans

Software Used

The application was developed in Java and makes use of planners compiled for Linux and sometimes for Windows too. The code is available under an Open Source license.
WSDL is used as the Web service description language and PDDL as the planner data exchange language.

2.3.4 Web Service Composition Using ConGolog

Golog (alGOl in LOGic) [Levesque, Reiter, Lesperance, Lin, & Scherl, 1997] is a high level logic programming language built on top of the Situation Calculus. Golog is a mixture of imperative and declarative programming. It sequentially executes do-instructions (actions). The configuration of these instructions is done by logical inferring. Therefore,

a Golog agent consists of a program of do-instructions and a mechanism logical inferring. ConGolog is an extension to Golog to support concurrent execution.

In [McIlraith & Son, 2002] ConGolog is used for the composition of Semantic Web services. Instead of planning, the agent's task is to execute a high-level program corresponding to a plan. An interpreter is used to search for a way to execute the program. Since logical reasoning relies very much on sufficient information and Semantic web is sensitive in this respect, [Phan & Hattori, 2006] has extended the work, more precisely the interpreter to support information gathering with search in an open world initial database.

Planning with ConGolog has an outstanding advantage: It manages the non-deterministic behavior of Web services so to say by nature. The interpreter always tries to find the best path independent of the current situation, i.e. it does not make any difference whether a control construct like If-Then-Else has to be executed next, or the execution of a Web service has failed, or simply a choice between different execution paths has to be made.

Software Used

Golog[3] should be run on top of a Prolog interpreter or compiler. Different interpreters exist in regard of Web service composition, two were mentioned above. In addition, a Golog compatible Prolog software is needed, like ECLIPSE Prolog[4] which is freely available under Open Source license (Cisco-style Mozilla Public Licence).

2.3.5 A *M*odel *B*ased *P*lanner for Automated Composition of Semantic Web Services

Planning as model checking was shown to be suited for domains of non-deterministic actions, partial observability, and complex goals [Giunchiglia & Traverso, 1999]. In [Traverso & Pistore, 2004] a Web service composition solution is presented that uses planning as model checking. It basically creates a new Web service that incorporates the behavior of all participating domain Web services and separates out the relevant parts for achieving the goals.

The solution translates OWL-S Web services ($W_1..W_n$) into non-deterministic and partially observable state-transition systems ($E_{W_1}..E_{W_n}$) (see Section 3.1.1 for an explanation of state-transition systems). Each service is thereby encoded to a separate state-transition system having its state-based dynamic system that can evolve, i.e. change state, and that can be partially controlled and observed by external agents. The actions created for the state-transition system correspond to invocations and responses of atomic processes that

[3] Any further information and software can be found on http://www.cs.toronto.edu/cogrobo/main/systems/index.html.

[4] see http://eclipse.crosscoreop.com/

```
<process: CompositePr ocess rdf:ID="Shipper">
 <process: Sequence>
  <process:AtomicProcess rdf:about="# DoShippingRequest" />
  <process:CompositeProcess>
   <process: IfThenElse>
    <process: ifCondition rdf:resource="# ShippingPossible" />
    <process: then>
     <process:CompositeProcess>
      <process: Choice>
       <process:AtomicProcess rdf:about="# AcceptShippingOffer" />
       <process:AtomicProcess rdf:about="# RefuseShippingOffer" />
      </process:Choice>
     </process:CompositeProcess>
    </process:then>
   </process:IfThenElse>
  </process:CompositeProcess>
 </process:Sequence>
</process:CompositeProcess>

<process: AtomicPr ocess rdf:ID="DoShippingRequest">
 <process: Input rdf:ID="size">
  <process:parameterT ype rdf:resource="# Size" />
 </process:Input>
 <process: Input rdf:ID="destination">
  <process:parameterT ype rdf:resource="# Location" />
 </process:Input>
 <process: ConditionalOutput rdf:ID="price">
  <process: coCondition rdf:resource="# ShippingPossible" />
  <process:parameterType rdf:resource="# Cost" />
 </process:ConditionalOutput>
 <process: ConditionalOutput rdf:ID="duration">
  <process: coCondition rdf:resource="# ShippingPossible" />
  <process:parameterT ype rdf:resource="# Delay">
 </process:ConditionalOutput>
 <process: ConditionalOutput rdf:ID="na">
  <process: coCondition rdf:resource="# NoShippingPossible" />
  <process:parameterT ype rdf:resource="# NotAvailable" />
 </process:ConditionalOutput>
</process:AtomicProcess>

<process: AtomicPr ocess rdf:ID="AcceptShippingOffer" ... >

<process: AtomicPr ocess rdf:ID="RefuseShippingOffer" ... >
```

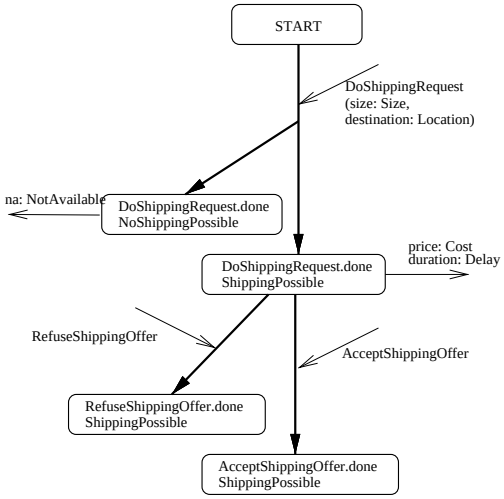
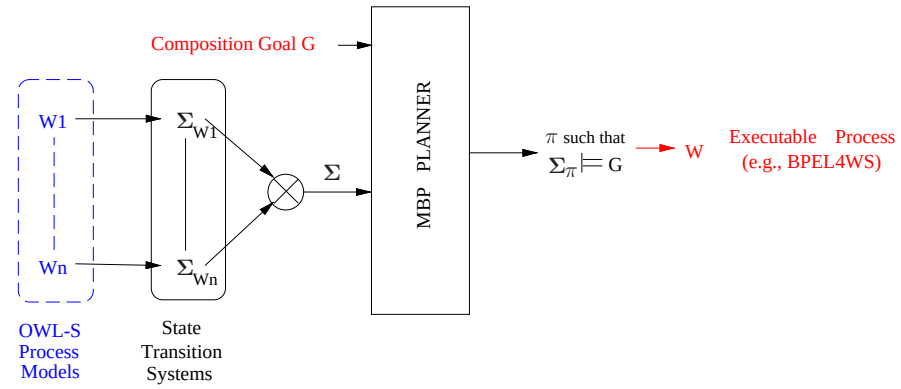


Figure 2.5: MBP: Sample translation from an OWL-S Web service to a state-transition system, illustrated by a graph

were collected from atomic or decomposed composite processes of the specific Web service and from any dependent processes which are involved in the interaction with that Web service (see Figure 2.5 for an example). The translation is similar to [Narayanan & McIlraith, 2002], whereas the states of state-transition systems can be seen as the markings in the Petri nets.

The new Web service W to be created has to operate in the environment of the combined state-transition system Σ constituted by the state-transition systems $\Sigma_{W_1}..\Sigma_{W_n}$. The state-transition systems can independently evolve over time and the planner's task is, given a composition goal G (defined in EAGLE which allows expressing goals as conditions and sub-goals with preferences on the whole behaviour of a service), to generate a plan π that controls the planning domain, i.e. interacts with the external services in a specific way such that the evolutions satisfy the goal G (see Figure 2.6). The plan π contains imperative logic in order to cope with the behavior of the environment of the new Web service W, and can thus be translated to any language capable of expressing flows of executable processes like BPEL4WS.

MBP (Model Based Planner) [Bertoli, Cimatti, Pistore, Roveri, & Traverso, 2001] is used to generate plans based on the extracted state-transition systems and the complex goals.



Source: [Traverso & Pistore, 2004]

Figure 2.6: MBP: OWL-S based automated composition

MBP has proved to scale very well for large domains.

Software Used

The approach depicted uses MBP[5]which is available in binary form, currently for Linux PCs, and for evaluation purposes only. The code of the translation itself is not available.

[5]MBP is availably on http://sra.itc.it/tools/mbp/, and NuSMV, on which MBP is based, on http://nusmv.itc.it/.

3

Background

In this chapter, I revisit some fundamentals that are relevant to this thesis. Whereas a knowledge about Web services and Semantic Web services is assumed to be known for the understanding of the thesis, planning techniques deserve to be lighted here a bit more in details[1].

A variety of features is asked from AI planning techniques, but due to the complexity of planning there is not an all-in-one technique coping with all aspects. Rather several different techniques exist for a certain problem. Following, some basic ideas of planning are shortly covered. A selection of proved planning algorithms are appended. An introduction into formalizing planning problems is given in Appendix A and a detailed description of the Planning Domain Definition Language (PDDL) can be found in Appendix B.

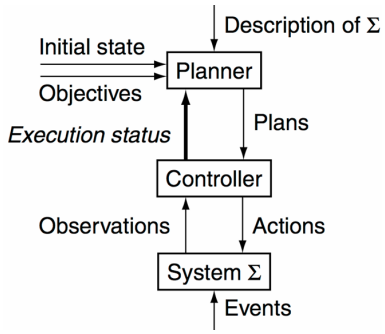
3.1 Complexity of Planning

Planning is simply spoken the task to decide what actions should be used to achieve some objectives in a particular environment. Planning is known to be a hard problem, due to good reasons: Looking at the planning problem in the real world, we have uncertainty about the effects of actions, i.e. we might know the normal effect, but we cannot predict whether that case occurs. We do not have complete information about the environment, for example some important information can miss at the beginning, and how can we say for sure that the observed domain state is actually correct? External events or other agents might alter the environment and thus affect the goal achievement. Another point is that we might have time, resource or preference constraints or want to optimize for certain aspects.

Planning seems to be a challenge and it is really. Two approaches making planning easier are discussed in this section. But before let us introduce a conceptual model of the planning task for a common understanding.

[1] The theory about planning techniques depicted in this chapter emanate to a major amount from [Ghallab, Nau, & Traverso, 2004] and [Russell & Norvig, 1995] which both give a wide-spread introduction. [Peer, 2005] [Rao & Su, 2004] give a good overview about planning techniques usable for the automated composition of (Semantic) Web services.

3.1.1 Conceptual Model



System Σ corresponds the a state-transition system. Source: [Ghallab et al., 2004]

Figure 3.1: Conceptual model for planning

A conceptual model helps to understand the various concerns of a problem. The *state-transition system*[2] is commonly used as a conceptual model for the planning problem, illustrated in Figure 3.1.

The model distinguishes

- the state-transition system Σ that represents the world of the planning problem and evolves according to the actions and events it receives,
- a controller which, given a description of the system Σ, provides an action a according to some plan, and
- a planner which, given a description of the system Σ, an initial state and some objectives, provides a plan.

Formally, the state-transition system is a 4-tuple $\Sigma = (S, A, E, \gamma)$, where:

- $S = \{s_1, s_2, ...\}$ is a finite set of states;
- $A = \{a_1, a_2, ...\}$ is a finite set of actions;
- $E = \{e_1, e_2, ...\}$ is a finite set of exogenous events;
- $\gamma : S \times (A \cup E) \rightarrow 2^S$ is a state-transition function.

The system, more precisely the system's state, evolves over time when actions or events happen. The difference between actions and events lies in whether the controller has any control over the execution. Whereas it can explicitly execute actions, events are based upon

[2]State-transition systems are actually theoretical models used in the automata theory and describe the possible states of state based systems and the possible transitions between those states.

on the internal dynamics of the system. If u is either an action a or an event e, and $\gamma(s, u)$ is not empty, one state s' of a possible set of states will result.

The objectives can be described in different ways: either

- by a single goal state s_g or a set of goal states S_g,
- by some conditions, that the controller should avoid or pass some states, or should stay at a certain state,
- by a utility function to be optimized, or
- by tasks that the system should perform.

Finally, the controller might not be able to fully observe the state of the system, so an observation function $\eta : S \rightarrow O$ can model the set $O = \{o_1, o_2, ...\}$ of possible observations.

3.1.2 Restriction of the Assumptions Made in the Conceptual Model

A planning problem taking into account all aspects of the state-transition system can be much worse than NP-complete. In order to make planning easier, a first approach, which is characteristic to classical planning, is to restrict the assumptions made in the conceptual model.

Assumption		Description
A0	Finite Σ	The System Σ has a finite set of states.
A1	Fully Observable Σ	One has complete knowledge about the state of Σ (also about the initial state).
A2	Deterministic Σ	For every state s and for every event or action u, $\mid \eta(s, u) \mid \leq 1$, i.e. there is at most one resulting state.
A3	Static Σ	The set E of events is empty, thus only controlled transitions take place.
A4	Restricted Goals	Only objectives specified as an explicit goal state s_g or set of goal states S_g are handled.
A5	Sequential Plans	A solution plan is a linearly ordered finite sequence of actions.
A6	Implicit Time	Actions and events have no duration; they have instantaneous state transitions.
A7	Offline Planning	The planner plans for the given initial and goal states, regardless of possible changes in Σ.

Table 3.2: Classical planning: restrictive assumptions

Classical planning problems normally combine all eight restrictive assumptions. But nevertheless, a planning problem is still very hard, in best cases NP-complete.

3.1.3 Modern Approach

Classical planning has mainly concentrated on domain independent algorithms, and has normally taken the restrictive assumptions. The focus on domain independence was not bad, since many effective algorithms have emerged and since domain dependent algorithms, as contrary, can be applied to only a specific type of problems. Nevertheless, recovering the need for making planning easier, another approach has become more important with the ongoing International Planning Competitions (IPC) all two years: The idea is to take advantage of the characteristics of a given problem domain, i.e. to provide the planner with additional domain-specific knowledge and to extend a domain independent planning engine with algorithm artifacts that can deal with the additional information in order to support the planner engine. These extra information can be about process compositions (harnessed for example by the hierarchical task network (HTN) planning), about non-deterministic outcomes of actions (can be used for example with the planning by model checking planning paradigm) or about time and resource constraints (which can be managed by means of a scheduler integration).

Aside, some important progress could be achieved in the domain of classical planning techniques, also thanks to the IPCs. These techniques are known under the term *neoclassical* techniques.

3.2 Planning Algorithms

3.2.1 State Space Planning

State space planning is a classical planning approach. The solution space is represented as a connection graph consisting of states (nodes) and actions (connections) according to the state-transition system. It mainly performs a search, either forwards beginning at the initial state (*progressive search*) or backwards from the goal state (*regressive search*). The search procedure, which is a typical element for the classical and neoclassical planning algorithms, can usually be abstracted to a recursive procedure having following steps (u is a node):

- a **refinement step**: modifies the collection of actions and/or constraints associated with u
- a **branching step**: generates child nodes for u
- a **pruning step**: removes unpromising child nodes
- selection of a child node v
- calling the search procedure with v

The procedure is executed until the goal state is reached, or backtracked if a dead path has been navigated. The challenge of a search algorithm is to choose the right child nodes and to recognize wrong or cycling paths as early as possible. Several search algorithms exist (for example depth-first search as illustrated in Figure 3.2).

```
Depth-first-search(u)
    if Terminal(u) then return(u)
    u ← Refine(u)
    B ← Branch(u)
    C ← Prune(B)
    while C ≠ Ø do
        v ← Select(C)
        C ← C - {v}
        π ← Depth-first-search(v)
        if π ≠ failure then return(π)
    return failure
end
```

Figure 3.2: Basic depth-first search algorithm for state space planning

Properties

State space planning algorithms generate sequential plans. Thanks to their rather simple base structure, they have been reused many times and extended to become fast and solid planning representatives nowadays (e.g. SGPlan 5 [Hsu, Wah, Huang, & Chen, n.d.]).

Heuristics

The thought behind heuristics is that search should be guided to first explore the most promising solutions. A heuristic function $h(n)$ expresses the relative desirability of the candidate node n. A sample $h(n)$ estimates the minimal cost from node n to a final node.

Considering the abstract search procedure, the child node having the minimum value for $h(n)$ is selected first. To extend a typical depth-first search algorithm as shown in Figure 3.2 for heuristics, *Select(n)* would have to be exchanged by *argmin*$\{h(n) \mid n \in C\}$.

Heuristics have first been used for state space planning [McDermott, 1996] only, and have then been applied to other techniques such as partial order planning, too. There are several approaches for the heuristic evaluation (e.g. A* as shown in Figure 3.3), a favored one is the usage of a relaxed planning graph.

```
Q ← initial node
C ← empty
repeat
    if Q is empty, return failure
    n ← first element of Q, Q ← rest(Q)
    if n is a final node, return n
    if n ∉ C, or has lower cost that its copy in C then
        add n to C
        S ← succ(n)
        S ← sort(S,f)
        Q ← merge(Q,S,f)
            (Q is ordered in increasing order of f(n) = g(n) + h(n))
    endif
endrepeat
```

Figure 3.3: A* algorithm is a state space search algorithm with a heuristic function $f(n)$ that sums the costs of the path from the initial node to the candidate node ($g(n)$) and the estimated costs of the path from the candidate node to the goal node ($h(n)$). It maintains a list Q with all passed nodes in increasing order of $f(n)$, and branches with the first node of Q. It guarantees to find the optimal solution, but consumes much space for the maintenance of the list Q.

3.2.2 Partial Order Planning

Partial order planning (POP) can be seen as searching in the space of partial plans. The partial plans are thereby the nodes of the connectivity graph. A partial plan can be defined as a five-tuple: $\mathcal{P} = (\mathcal{A}, \mathcal{O}, \mathcal{L}, \mathcal{OC}, \mathcal{UL})$, where[3]:

- $\mathcal{A}$ is a set of actions,
- $\mathcal{O}$ is a set of ordering constraints over $\mathcal{A}$, for example $a_i \prec a_j$,
- $\mathcal{L}$ is a set of causal links over $\mathcal{A}$
  (a **causal link** is of the form $a_i \xrightarrow{p} a_j$ and denotes a commitment that the precondition p of action a_j will be supported by an effect of action a_i),

[3] A complete description of the POP algorithm can be found in [Weld, 1994].

- $\mathcal{OC}$ is a set of open conditions
  (an **open condition** is a precondition of an action in the partial plan which has not yet been achieved), and
- $\mathcal{UL}$ is a set of unsafe links
  (a causal link $a_i \xrightarrow{p} a_j$ is **unsafe** if an action a_k exists such that a) $\neg p \in \mathit{Eff}(a_k)$ and b) $\mathcal{O} \cup \{a_i \succ a_j \succ a_k\}$ is consistent).

The algorithm starts with a partial plan built by means of the domain actions, the initial state and the goal state. The partial plan and all resulting partial plans are refined by repeatedly resolving the open conditions and unsafe links (flaws), until all actions are partially ordered in $\mathcal{O}$.

Properties

POP is based on the least commitment principle. The plans returned correspond to the partial orderings of $\mathcal{O}$.

3.2.3 Planning Graph Based Planning

Planning graph based planning is a mixture between state space and partial order planning. It uses a special connection graph called *Planning Graph*: "A Planning Graph encodes the planning problem in such a way that many useful constraints inherent in the problem become explicitly available to reduce the amount of search needed." [Blum & Furst, 1995]. It simplifies the planning problem by ignoring some constraints, for example it ignores the deletion of facts as actually caused from action effects.

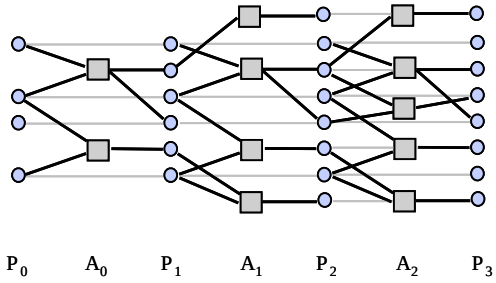
The planning graph is a directed layered graph, whereas a proposition layer and an action layer alternate (see Figure 3.4). Starting with a proposition layer $P_{i=0}$ consisting of the initial state propositions, the succeeding action layer A_i contains all actions applicable to P_i. Layer P_{i+1} takes over all propositions from layer P_i (virtually by no-op actions) and all effects from layer A_i. The planning graph levels off, i.e. after a layer P_k (fix point) all layers are identical. A valid plan may exist as soon as a proposition layer contains all goal propositions and when they are pair-wise non-mutex. If this is not the case when reaching the fix point, no plan exists. Inconsistent relations that occur because of the constraint relaxation are marked as mutual exclusions (see Figure 3.6): Two actions are mutex if (a) an effect of one negates an effect of the other (inconsistent effects), one deletes a precondition of the other (interference), or if (b) they have mutex preconditions (competing needs). Two preconditions are mutex if one is the negation of the other or all ways of achieving them are mutex (inconsistent support).

The planner Graphplan presented in [Blum & Furst, 1995] uses then a recursive search strategy to extract a plan from the graph. The plans generated have a sequential total order but may have several actions per step, so that those could be executed in parallel (a sample plan extraction from a planning graph can be seen in see Figure 3.5).

Properties

A planning graph can be constructed in polynomial complexity (time and space), and is thus often used for heuristics (called a relaxed planning graph). Graphplan finds the shortest-parallel plan, is sound and complete, and will terminate with failure if there is no plan.

Graphplan has a rather specific algorithm, and while the planning graph technique has proved to excel in STRIPS domains, it results in a trade-off concerning extendibility: It is possible to extend Graphplan by mechanisms supporting new features (ADL [Koehler, Nebel, Hoffmann, & Dimopoulos, 1997], sensing actions [Weld, Anderson, & Smith, 1998], temporal planning [Long & Fox, 2003], and others), but at the risk of deteriorating the Graphplan behaviour due to later implications with extensions [Long & Fox, 2003].



$P_0..P_n$ are proposition layers, $A_1..A_n$ action layers.

Figure 3.4: Planning Graph: Basic Structure

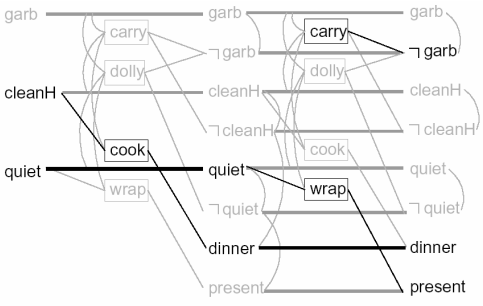


Figure 3.5: Planning Graph: Plan Extraction

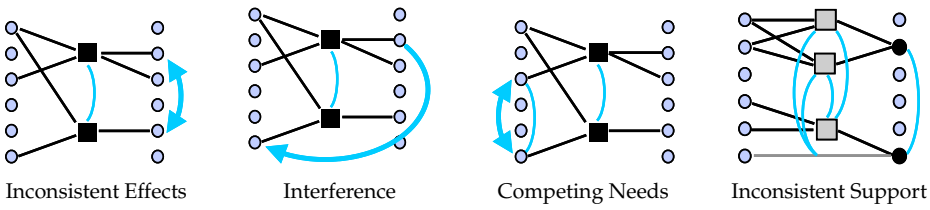


Figure 3.6: Planning Graph: Mutual Exclusions

4

Requirements

This chapter lists the detailed requirements for the development of the planner component prototype. The requirements are available in a categorized form. Each requirement is weighted with priority label expressing its importance in an ordinal scale from low, medium to high. The requirements are separated in a traditional way [Glinz, 2005] into functional (Section 4.2) and non-functional requirements (Section 4.3).

It should be mentioned that requirements might have an inexplicit definition origin, i.e. they were not listed during the first requirements specification process, but are either derived from other requirements, or have arisen with the analysis of the major goals. In the latter case, the deduced requirements were marked as such in Section 5.1.

The first section starts by revisiting the environment of NExT in order gain transparency regarding the requirements specification.

4.1 Assumptions

Having outlined the motivation and objectives of the NExT project, I have not mentioned yet any of its technical details. These technical details do not necessarily have to be seen from an implementation perspective. In fact, they have a not negligible influence on the requirements engineering for the planning component. On one side, they define base requirements for an integration of the planning component into NExT, and on the other side, they may show a possible discrepancy between the concepts of NExT and their implementation we should care about.

- **Composite Processes:**
  Considering the NExT's process model, it envisages a distinction between atomic and unchangeable composite processes for the ground level of the process hierarchy. These basic operations correspond to experimental steps and are described in OWL-S again either as atomic or composite processes. At the time of writing, all experimental

steps are realized as atomic processes, and a complete composition structure does not exist. So a planner implementation cannot rely only on composite processes.

- **Numeric Fluents:**
  In a planning model, numeric fluents represent function symbols that can take an infinite set of values. They are referenced either in preconditions or effects of actions. Regarding an OWL-S process model, they are modeled as property values of their class instances, and get related to by preconditions and effects as well. The question is not whether an implementation is possible in OWL-S, because it is in different ways, but whether an implementation has been designed in NExT. Currently, numeric fluents are not directly supported by NExT, but are very likely to be implemented next, namely by means of the annotation language constructs of OWL. Therefore support for numeric fluents should be required.

- **Simple Plan Structurings:**
  When the user decides to use the planning component for retrieving some alternative plans, NExT expects primarily plans with simple structurings such as a sequence.

4.2 Functional Requirements

4.2.1 Thesis Goals

The purpose of this section is to describe the sub-goals and their direct induced requirements in terms of the development of the software.

ID	Requirement	Priority
R_{A1}	**Interactive plan finding** The planner invites the user to refine the planning problem in several cycle runs until the resulting plans correspond to his best wishes. For that purpose, on one side it receives information from the user by letting him configure the planning problem, on the other side it gives back a qualitative feedback on planning runs. An infrastructure should be provided which supports this session like interaction.	high

R_{A2}	**Planning for alternative plans** Several plan alternatives are calculated, and information about each alternative is provided concerning: - plan length - the quality of the plan - other meta information (e.g. some numeric values for optimisation purposes)	high
R_{A3}	**Imprecise planning** – Imprecise planning involves the relaxation of domain and problem constraints (conditions, actions, types matching) to increase the solution space.	high
	- In case that no plan can be found, the planner should be able to find some useful suggestions for a possible plan.	high
R_{A4}	**Ease of integration into a process execution environment** The planner provides an interface that can be accessed by the process execution environment. It can greatly account for a good integration by: – allowing asynchronous communication which decouples the components, especially with regard to concurrency, – maintaining a communication session.	high
R_{A5}	**Offline planning.** The user should be provided with plan alternatives before any Web services have been executed. This means that the planner should not depend on results of a Web service execution, but should be able to do planning offline. If necessary, Web services that only output state-less information or whose execution does not have any influence on the active experiment environment can certainly be executed.	high
R_{A6}	**Replanning.** While executing a process plan which was calculated from the planner, it could be necessary to do a replanning step. A replanning capability would enable to continue the used planning problem, i.e. the instantiated planning problem, and to partially update the planning domain or problem, respectively.	medium

4.2.2 Description Language for Semantic Web Services

A description language for Semantic Web services should fulfill some requirements in order to be usable in conjunction with planning techniques and in order to cope with the sub-goals given in this thesis.

	Requirement	Priority
R_{B1}	Formal definition of preconditions and effects. The planner component relies on the formal definition to be able to convert them to appropriate logic clauses.	high
R_{B2}	Extension ability to annotate language entities like services or types. Extensibility of language entities or the possibility to annotate them would help to add optimisation or quality of service related information on resource consumption, duration, performance and any other quality aspect.	medium
R_{B3}	Existing API to read and write service descriptions. This would highly facilitate the process of retrieving the necessary information from service descriptions.	medium
R_{B4}	Process composition capability. The planner needs compositional control constructs in order to describe process plans. At least a sequential control construct should be supported.	high
R_{B5}	Process composition capability for complex plans. This emphasizes the above defined requirement with the support for more complex control constructs like parallel, if-then-else, repeat-until execution. They enable for example the description of parallel plans.	low

4.2.3 Planner Features

The next requirements define the demands made to the planning engine after having decided the main concept (Section 5.2).

	Requirement	Priority
R_{C1}	Support for basic planning features, and artifacts of domain and planning problem description languages:	
	- STRIPS-like descriptions	high
	- disjunctive preconditions	high
	- conditional effects	high
	- time constraints†	low
	- resource constraints	low
R_{C2}	Numeric fluents that would enable for example resource constraints†	medium
R_{C3}	Generation of plans that contain control constructs (split/join, if/then/else, repeat/until, ..)	low
R_{C4}	Global constraints:	
	- soft goals: preferences which don't have to be satisfied in either case.	high
	- optimization critera like QoS information (for example on a numeric base)	medium
R_{C5}	Managing non-determinism	medium
R_{C6}	Managing partial observability: The planner has incomplete knowledge and has ways to cope with that problem, for example by means of information gathering.	low
R_{C7}	Probabilistic actions (actions that have a probability distribution for their possible resulting states)	very low

† In case these features are not implemented, the proposed architecture should allow for these features with minimal hassle.

4.3 Non-Functional Requirements

4.3.1 Performance Requirements

CPU Time

Performance is not a killer criterion, but an important one. The benefit of a planner built upon the idea of user interactivity exists only, when a user actually does not have to wait too long for the plans to be generated.

Planning corresponds to one of the hardest computational problems existing (see Section 3.1), usually only solvable in exponential computation steps. Following the nature of planning algorithms is analyzed to gain transparency about which parameters actually affect the computational complexity.

The variables having the most impact on the computational complexity are (a) the complexity of the domain and problem (i.e. constraints) which are more or less given (depending on whether the planner plans for a imprecise solution or not), (b) the dimension of the domain and problem (for example the number of actions) which is also given and can hardly be minimized, and (c) the target quality of a plan in the sense whether a plan is a time-first solution or an optimal solution for the problem. Optimal plans are harder to find and thus consume more calculation time.

In order to minimize the time consumption, it might be useful to investigate in a solution which is capable of distributing the computational load across processors and/or machines.

Memory

Non-transient memory such as hard disk space can unlimitedly be used. For transient memory, one should keep in mind that several planning processes could be executed in parallel and thus consume large amounts of memory.

4.3.2 Quality Requirements

Quality of Architecture

The architecture of the planner component should be designed with state of the art design methods such as decoupling of high cohesion parts for modularity and separating interfaces from implementations. Reusability and extensibility are factors that should also be considered.

4.3.3 Constraints

Concurrency

The planner should be designed to be executable in parallel from the same planning requester, i.e. from the same NExT instance.

Programming Language

The API of the planning component should at least be available in Java, but the more is developed in Java the better, since NExT is developed too in Java and the overall maintenance of the code would thus be easier.

Platform

Linux is mandatory since NExT will run with high probability on research servers which normally use Linux. The support for the Windows platform is very useful for testing and performance measuring purposes.

5

Design Evaluation

The goal of this chapter is the evaluation of different approaches that come into question as problem solutions. It is a trade-off of their benefits and their disadvantageous consequences. Section 5.1 analyzes the main goals as well as the sub-goals in details to crystallize the relevant evaluation aspects, and pre-evaluates partial approaches by giving recommendations. Section 5.2 evaluates several design alternatives and puts the decision for a main concept in a nutshell. Section 5.3 addresses some special aspects and evaluates corresponding sub-concepts.

5.1 Goals Analysis

Let us revisit the major goals: Aside the main goal of the conception of a planning component based on pre-defined Semantic Web services, there are the sub-goals:

- focus on the *user interaction*,
- planning for several *plan alternatives*, and
- *imprecise planning* in case no plan can be found or the user wants other solutions.

Examining these goals, there is to the best of my knowledge no existing solution that could deal with all of them or is generic enough to be configured until it suits. Therefore, all aspects are analyzed following with respect to the evaluation in Section 5.2.

5.1.1 User Interaction

A central idea of user interactivity (requirement R_{A1}) is the overall control lying in the user's hands instead in those of NExT or any of its sub-layered components, i.e. the user triggers a process execution and can interrupt the execution at any time. In addition, he

can tell NExT to run only one experimental process step, like in debug mode. When he invokes the planning feature, he wants to obtain a set of propositions about plan alternatives, and there should be no need for the planner to have to execute any processes in order to retrieve the plans and more generally to alter anything in the world of an experiment case. The depicted scenario looks like an obvious case for offline planning, where the planner is detached from the dynamics of the world and works offline. In contrast to it, online planning would mean that the planner evolves its plan by executing the next plan steps and by adapting the current plan to an eventually unexpected world state. Let us discuss the two alternatives:

- **Quasi-online planning by means of a runtime integration**

  There are several interesting planners in the field of automated Web service composition that make use of online planning (for example a ConColog based planner like [Phan & Hattori, 2006]). Online planning brings with it a smart solution for the problem of non-determinism of services and for the lack of domain knowledge, since it aligns its execution path according to the current world state and can thus react on world changes or new constraints. Because the process execution control should not lie in the hands of the planner component as explained above, the only way to go would be an integration of the planner's process execution and monitoring mechanism with the process execution logic of NExT. So, granted that the user has utilized the planner and has chosen one of the proposed plan alternatives, an execution of a plan step would come along with an assimilation to the planner's process execution control. Apart from the fact that such a tightly coupled integration would increase the complexity of an architectural design and require some changes to NExT itself, an integration might be questionable if the underlying planning engine had a fix control logic or was based on a completely different language such as ConGolog.

- **Offline planning combined with consistency checking and replanning:**

  Offline planning is – simply spoken – nearer to the mind of user interactivity: the planner does its job when asked and returns the control (and hopefully some plans) to NExT, i.e. to the user. There are two issues arising with the nature of offline planning:

  - **Consistency checking**
    Imagine the user has asked the planner for several plan propositions and is now continuously executing one chosen plan. The plan which he uses now was created offline some time steps before, and is based upon the world state at that past time point. In the meantime, the world state might have altered in a different way as the planner has planned. Depending on whether the plan is a simple linearly ordered set of process steps or has built-in control constructs to deal with some possible different flows, the plan might become inconsistent. NExT is therefore most probably obliged to do a consistency check and possibly to append a replanning step. So, offline planning is associated with the characteristic

of iteratively consistency checking and replanning after a process execution. (→ requirement R_{A6})

– **Information gathering**
Web services can be separated into pure information gathering services[1] and services that may alter the world state. The execution of services of the former type is a good way to complement the initial knowledge with the additionally queried information ([Kuter et al., 2006] and [Peer, 2004] both take advantage of this approach). Unfortunately, offline planning restricts the abilities of a planner, and the planner might not have the permission to execute any processes anymore, even if they are of the information gathering type. To overcome that issue, either NExT should gather the information from relevant Web services on its own initiative and incorporate it into the domain knowledge – the difficulty here lies in how to decide which services should be queried and for what actually –, or the planner component provides NExT with accurate instructions on which services should be queried and for what. The latter approach makes more sense since the planner component does a wide spread analysis of the domain and its services in terms of planning anyway.

The offline planning variant is clearly favored over online planning since it fits better for the principle of user interactivity inherited from NExT. For both issues arising with offline planning, there are decent solutions. (→ requirement R_{A5})

Offline planning combined with continuous consistency checking and replanning is a good way to deal with the non-deterministic behavior of Web services, since the system may immediately react on new circumstances such as altered world facts, new information or changed goals.

5.1.2 Plan Alternatives

If the user invokes the planner component, he should be provided with some alternative plans (requirement R_{A2}). Before analyzing possible realizations, some aspects are discussed in more detail while making some assumptions.

- **Plan representation**

  A plan is not equal to a plan, because they can have different representations, even if they describe the same plan solution. First of all, they can be described in different languages, such as PDDL or OWL-S. Secondly, they may have different action structurings. Considering the resulting plans of state-of-the-art planners, we may have plans consisting of plan steps that are either totally ordered – linearly (e.g. state space planning) or complex by means of control constructs (e.g. planning by model checking) – or partially ordered (e.g. partial order planning). The format does not play a

[1] In the AI planning domain, such services are known as sensing actions.

role because any format can be consistently transformed to NExT's process definition language OWL-S as long as the language the plan is described in is not more expressive than OWL-S. The structurings of plan steps however have a wide span ranging from a sequence to complex nested structures. Theoretically, all existing linearly ordered plan solutions could be represented as one single complex structured plan. But the latter version is not exactly what a user intends to receive when he asks for alternatives. Therefore, I assume that the user should be provided with – if possible – at least a handful plans which have a simple structuring of plan steps such as a sequence. I argue that the fast understanding of the proposed simple plans helps the user to get an optimal overall idea of the possible execution paths (→ requirement R_{B4}). If the user is interested in receiving a plan which involves a complete structure of the solution space, the planner might provide with him an additional complex plan for this case. The generation of a complex plan would be an optional feature (→ requirement R_{B5}).

- **Conditional versus non-conditional process outputs**

  One point should be mentioned in addition. If we assume to have only processes with non-conditional outputs (including effects) and for simplicity, processes to be atomic, we will exactly have two execution outcomes of services: (1) a Web service has successfully returned the outputs, or (2) the service or any other involved party has failed. If we further assume the successful execution to be the normal scenario, we have no problem to search for different simple, mostly sequentially ordered plans, provided that such solutions exist, and select among these the few best. Let us now go ahead to the case when we have processes with several conditional outputs and effects, and we can not predict which outcome will emerge, neither now nor at the time of their executions. Since we have to take into account all possible outcomes, many considerations of process inter-dependencies would be necessary to meet the goal constraints in either case and we would automatically come up with one or two complex plans. And it turns out that a division of a complex plan is not trivial, i.e. every division might danger the plan's guarantee to reach the goal state. A possible way out could be the usage of imprecise planning.

- **Quality information of a plan**

  To facilitate the user's choice of a plan, the plans should be assigned with additional information like the quality of the plan or how the plan could be retrieved.
  (→ requirement R_{A9})

- **Optimization**

  Optimization is a useful technique to find the optimal solution according to some optimization criteria, but is contrary to the generation of alternative plans. Planning with optimization normally finds only the single optimal solution automatically.

Approaches

Now, let us investigate how the requirement of plan alternatives can be fulfilled. Normally, planners plan for a single plan. Optimal planners (for example planners based on the planning as satisfiability technique) search indeed explicitly for the best single solution. Having said that, it does not necessarily mean that the underlying algorithms would not be able to generate several plans within the same planning run. Apart from optimal planners which plan for only one solution as a matter of principle, several planning algorithms suggest possibilities. There are three distinct approaches coming into my mind to produce multiple plan alternatives:

- **Extension of AI planning algorithms**

  This approach addresses the roots of the problem and comes along with the idea of using the hidden abilities of a planner algorithm to produce several plans at the same time. As explained above, we analyze only sub-optimal planners that generate primarily sequential plans. A realization should guarantee to produce several plan alternatives, provided they exist. Algorithms coming into question are state space planning with a rather breadth-first like search, Graphplan, partial order planning and planning as model checking.

  Taking an existing algorithm may have the consequence that planning features like sub-goal specifications would have to be implemented in addition. Therefore it might be useful to choose an algorithm of which either some extensions already exist or which can be extended without any hassles.

- **Evaluation of complex plans**

  In the second approach, planners first generate plans upon which several plan alternatives can be extracted afterward. The original plans should thus have either a simple structuring with additional information enabling plan variations or a complex structuring, so that an extraction for alternatives is possible. But let us assume we have an original plan which is large, i.e. consists of many actions, and which is weakly constrained, i.e. actions may be arranged in many different ways, the succeeding extraction could result in a huge pile of plan alternatives. This would help the user in no way, and a randomly selection of a limited amount of plans is just as useless. The only way to go is to combine the plan extraction with a re-analysis of the planning problem in terms of plan quality evaluation.

- **Using several planners in parallel**

  The last approach simply intends the usage of several planners in parallel. They are engaged with the same planning problem to concurrently produce some plan alternatives. This approach is the simplest among the presented, but has a drawback: How can be guaranteed that the different planners actually do not return exactly the

same plan alternative? To cope with that issue, the planning problem is either slightly differently configured for each planner (for example one planner might be configured to take some additional constraints into account), or the set of used planners should be selected as a mixture of diverse planning algorithms. A good differentiation for the latter would be the usage of a few sup-optimal planners combined with one optimal planner.
Another issue is performance. Running several planners in parallel might consume more CPU time than an extended single planner. Having said that, this issue has to be seen proportional, i.e. running three native compiled planners may still be faster than a complex Java bytecode planner.

All three approaches are usable, however, some have explicit advantages. The approach of using several planners in parallel comes of the best, because it suggests a simple but effective architecture and enables the integration of other approaches. For example, one could think of developing an additional AI planner that fits in the multiple-planner architecture.

Diminution of the Solution Space

Retrieving a sizable amount of plans is a nice feature. Loosing track of too many plans is less desirable. In such a case, it is of course possible to select only the best plans according the their quality measures. This might however have the effect that some imperfect plans which could actually have good user ratings are phased out. That is why the user is given a possibility to add constraints in order to be able to produce exactly those plans that correspond to his preferences. These constraints may be state trajectory constraints, constraints with numeric fluents or any other imaginable global constraints. ($\rightarrow$ requirement R_{B5})

Let us recover the complex goal requirement for automated Web service description as described in Section 2.2. The main idea behind it is that additional constraints have to be taken into account which are not expressible by goal states, for example the condition to pass a specific state. The setup of additional constraints, especially state trajectory constraints, can be used to define those complex goals of Web service planning problems.

5.1.3 Imprecise Planning

Imprecise planning (requirement R_{A3}) generally spoken helps to enlarge the solution space and is used for certain cases. These cases are (see Figure 5.1):

- **Case 1: No plan could be found**

  The planner component was not able to find any plan solution in a first run. Either a plan actually exists but could not be found by the used planning algorithms (this

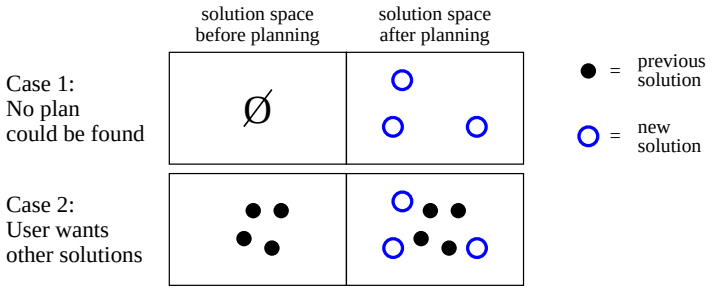


Figure 5.1: Enlargement of the solution space with imprecise planning

can happen with sub-optimal planners, for example a forward state space based algorithm could trap into a local minimum). Or, what is more probable, there exists no solution. This may occur because of one of the following reasons (there may be some other reasons not listed here, including those that originate from the listed):

- Reason $\mathcal{A}$: Mutual exclusions (see Section 3.2.3):
  The actions may not be assembled in a way (while complying with the precondition and effect constraints) such that all goals can be fulfilled.
- Reason $\mathcal{B}$: Unreachable goal atoms:
  There may be some goal atoms that can not be satisfied by any of the available actions.
- Reason $\mathcal{C}$: Unsatisfiable global constraints:
  Some global constraints, like for example state trajectory constraints or goal constraints with numerical fluents, may make the planning problem unsolvable.
- Reason $\mathcal{D}$: Incomplete knowledge:
  Missing information prevents the application of actions. It may be seen as a sub-reason for reason $\mathcal{B}$.

- **Case 2: User wants other solutions**

  The planner component has returned some alternative plan solutions, possibly these are even all existing solutions. But the user is not satisfied yet and wants to get some others. This may be the case when the user has an idea of another possible solution that has not shown up yet, or when he is interested to know any plausible solutions that are not visible on the first sight and wants to play around by switching on and off some constraints or goals to explore them.

Approaches

A first approach for imprecise planning addresses only the first case, whereas the consecutively presented approaches have a more general nature and can be applied in both

cases.

- **Problem analysis and partial results**

  First, we consider only the case *no plan could be found*, since it is a very specific problem. The user may want to get an explication why no plan could be found. The provided information about the reasons can at the same time be seen as hints for the user how he can adapt the domain in such a way that a replanning step would find a plan. On the other hand, the user might already be satisfied with a partial result, for example with a partial plan.

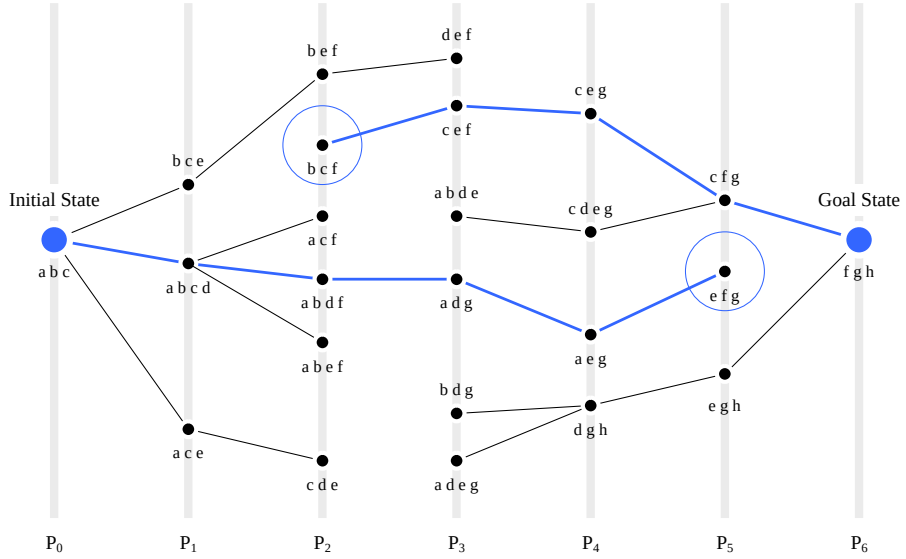
  - **Hints for the user**
    Some reasons have been listed above. For reason $\mathcal{A}$, we might collect all preconditions related to their actions that prevent the actions from being linked, or simply list the goals that can not be fulfilled. For reason $\mathcal{B}$, we can only list the unsatisfiable goals. With the global constraints (reason $\mathcal{C}$), we might intuitively search for all those that either block for sure or have a critical influence. The question is here whether these constraints can be sorted out so easily.
    A good starting point is to provide the user with a list of unsatisfiable or competing goals.
  - **Partial results**
    Partial results could be provided as partial plans, one (or several) starting at the initial state, and another (some others) backwards from the goal state (see Figure 5.2 for an illustration). State-space planning for example could be used to retrieve the two plan types. It would search forward and backwards, based on heuristics, until an abort condition holds.

  Hints for the user are regarded as more important than partial results.

- **Constraints relaxation**

  Another approach applicable in both cases is to relax some constraints. The decision on which constraints to relax mainly depends on whether a relaxation makes sense (for example the relaxation of an action's precondition could danger its proper execution) and on how complicated the definition of a relaxation constitutes to the user. Constraints coming into question are:

  - Goal state constraints: Some goal state constraints could be relaxed by changing them to soft goals that do not have to be fulfilled in either case. One could think of a penalty weight for a not satisfied soft goal.
  - Global constraints: They could be relaxed in the same way as the goal state constraints.
  - Temporal and resource constraints: These constraints may locally or globally be unsatisfiable, and may be relaxed separately.



The graph represents a solution space of a simple planning problem, consisting only of atomic propositions. The graph is layered by layers that stand for plan lengths (P_0 .. P_6). A maximum plan length of six steps is assumed (this can be calculated with a relaxed planning graph, for instance). The partial plans highlighted are the best found in terms of a sample heuristic which measures the distance between the outermost reached state and the state finally to be reached, i.e. either the goal state or the initial state.

Figure 5.2: Find best forwards chained and backward chained partial paths

- **Similarity matchings**

  A third approach is the idea of performing object matchings in a more generous kind, i.e. if the similarity of two objects reaches a certain threshold, they can be matched. Following, the application of similarity matchings in the domain of OWL-S is discussed.

  In ontologies we can measure the similarity between concepts [Bernstein, Kaufmann, Kiefer, & Bürki, 2005]. A concept is a RDFS/OWL class which has properties with either primitive typed or complex typed (with concepts) values, and is based on arbitrary super-concepts. The similarity between two concepts can intuitively be measured by their distance within a directed acyclic graph representing a multiple inheritance framework of ontologies. For querying data in a similar way as with data-mining, it may be sufficient to have a close common ancestor concept. Let us now analyze an OWL-S process definition. We have preconditions and effects consisting of atomic facts about the world state which in turn have variables that are bound to concepts during a process execution. The variable binding requires most probably exactly the declared types, and the application of similarity matchings to them may be critical for the proper execution of the service. Continuing, we have inputs and outputs which represent non-physical objects and whose content normally is not known

by a planner. I assume, by ignoring preconditions and effects: an input requirement with a given type must be satisfied by an object containing at least all data that is semantically expected and defined by the type, and an output object must at least produce all data that is semantically expected and defined by its type. A successful process execution can otherwise not be guaranteed. Having assumed this, let us now further focus on the types of inputs and outputs. Literal data types such as text can not be used for similarity matchings since we do not have any semantics about their content. With OWL concepts it looks different. We have all semantics about the object type and could thus say that the input object must be of the same concept or of an unrestricted sub-concept of the required concept. An issue arises when taking preconditions and effects into account: They may relate explicitly to input or output parameters (for example an effect could set the type of an instantiated output object). More analysis is therefore needed to find a proper appliance of similarity matchings to OWL-S processes in respect of planning.

Summarizing, imprecise planning should be applied with the techniques of relaxing constraints and in the case of no-solution by providing the user with information about the reasons.

5.2 Evaluation of a Main Concept

Before moving to the evaluation, let us briefly summarize the results of the goals analysis in the previous Section:

- User interaction turns out to be best addressed by an offline planning approach combined with iteratively consistency checking and replanning.
- Plan alternatives should be generated mainly by the usage of different AI planners. An AI planner developed by adapting some existing AI algorithms could be optionally plugged-in.
- Imprecise planning can be realized by relaxing constraints and, in the no-solution case, by providing the user with hints.

When we consider the sub-goals *plan alternatives* and *imprecise planning*, we come up with a mixture of different planning tasks necessary to realize them. These planning tasks are further supplemented by the techniques for diminishing the solution space, namely the setup of additional constraints and optimization.

Having the facts and all other requirements in mind, we set about evaluating concrete solution candidates for a main concept. Generally, it can be said that an adaptation of a quasi finished solution, like a Web service composition solution, requires less effort than a development with a mixture of some existing tools, and not to mention much less effort than a complete development from scratch. Therefore I start by evaluating first complete existing solutions. If they turn out to be not usable, I will focus on mixed solutions, i.e. self-developed solutions that make use of other techniques.

5.2.1 Evaluation of Existing Solutions

Pre-Evaluation

To make the evaluation process more effective, a pre-evaluation is inserted before the main evaluation of existing solutions. A pre-evaluation helps to sort out some alternative solutions. We do so by considering the following aspects:

Support for Atomic Processes

As assumed in Section 4.1, atomic processes must be supported. So all planners that rely on fully composed processes only, do not come into question as solutions. These are: pure HTN-based planners (e.g. SHOP2), and solutions that encode process flows, for example as Petri-Nets or similar networks, and extract some plans upon them.

Offline Planning

All solutions that are restricted to online planning can not be used (see Section 5.1.1). This concerns all ConGolog based planners.

Main Evalutation

We have a wide spectrum of different tasks (see Section 5.1) that have to be implemented in the main concept in any form, even if we reduce them to the smallest subset of the absolutely necessary tasks. It might thus be interesting to see whether any of the existing solutions could be adapted in a worthwhile manner.

- **OWLS-Xplan**

  OWLS-Xplan [Klusch et al., 2005] is a complete solution for transforming any OWL-S Web services to the typical AI planning language PDDL and for planning upon them. An advantage is the support for domain-specific (with composite processes) and domain-unspecific (with atomic processes) planning problems. In addition, it is possible to optimize a plan by means of QoS metrics. Unfortunately, the implementation was not finished at the time of writing, and the sources of the planning algorithm itself were not available so that the other planning tasks could not be integrated so easily. OWLS-Xplan thus does not come into question as a solution.

- **WSPlan**

  WSPlan [Peer, 2004] represents an overall solution for the Web services composition problem. It decouples a planning task by using different AI planners. This fantastic idea supported by the fact that all sources are available under an Open Source license makes WSPlan interesting to be used as a base architecture and to be extended by the missing planning tasks. A small problem could arise because WSPlan assumes to have the control over the service execution in order to combine the planning process by step-wisely executing and monitoring the planned Web services. Another flaw is that it is currently only compatible with WSDL as Web service description language. Whereas the semantics of Web services have been taken into account (they are actually retrieved from semantical annotations to WSDL), any compositional aspects are disregarded.

- **MBP-based planner**

  MBP-based planners in the domain of Web service composition, like [Traverso & Pistore, 2004], are nice solutions as they are able to deal with the peculiarities of the domain probably the best among the existing solutions. A MBP-based planner maps all relevant flows to one complex plan, so in order to generate alternative plans, this plan would have to be analyzed for extracting plans. It should be noted that the software MBP is currently only available for Linux systems and only as binary executables, and uses a PDDL dialect called *NuPDDL* which is compatible with PDDL version 2.1 but not with the newer ones. So additionally required planning features can not so easily be implemented but would rather have to be developed separately.

The presented solutions do not cover all planning tasks, especially those that generate alternative plans or address the no-solution case. Further, the preconditions for a guided user interaction completely miss, and for most solutions, an underlying application logic supporting the different planning tasks misses, too.

5.2.2 Evaluation of a Self-Developed Solution

Regarding the previous evaluation, it turns out that regardless of what alternative one chooses, each of them has to be extended by a sizable amount of features. For this reason, one might just as well develop a solution based on a mixture of different AI planning techniques. A development from scratch with the guide of one or several AI planning algorithms might however be too extensive in terms of time and complexity. The problems with it are that (1.) handling the different planning tasks with only one or two planning algorithms might not be sufficient in all likelihood since the tasks are simply too different, and (2.) trying to mix the algorithms into one compact solution might cause negative implications due to the incompatibilities of some algorithms when used in combination (extensions of a planning graph algorithm are an example). The development would spread out to the implementation of a number of different planning algorithms.

Therefore, rather than trying to defeat the complete problem solving of the different planning tasks by one compact solution, a solution should be considered which assigns the planning tasks to appropriate problem solvers, i.e. general or specialized AI planners.

- We have different planning tasks not solvable with just one compact solution. So allocate a planning task to the AI planner made for it, possibly to several of them.
- Adapt the planning component to requirements for planning features by adapting an existing AI planner or by adding another one.

Summarized, we have a decoupling of planning tasks from different implementations of their problem solving, i.e. different AI planners. Such a design highly contributes to the flexibility of the planning component and facilitates later extensions. Another advantage is the ability to include existing AI planners as problem solvers. The promising approach was thus chosen as a basic design principle in favor of a main concept for a self-developed solution.

I continue by analyzing the planning tasks with regard to possible solutions for their problem solving. The synthesis of plan alternatives can be established (1a) by the usage of several planners in parallel, or (1b) by a single AI planner that is capable of it. Looking at imprecise planning in the no-solution case, a usable solution in any form for it does not exist and since it corresponds to a highly specific problem, it is best addressed (2a) by an independent AI planner. Imprecise planning in all other cases can be achieved either (3a) by one or more specific AI planners which concentrate on that aspect, maybe amongst

others, or (3b) by formulating the problem solving of the required features as a problem for existing, well-known planning techniques, what would enable to let the planning problem be solved by any AI planner, provided it supports the used features. The latter approach is particularly interesting because it perfectly fits into the concept of using several AI planners as problem solvers.

For each planning task, I have outlined some solution variants about concrete AI planner designs. Following, they are summarized to more compact partial solutions:

Partial solution:	**Comments concerning design:**
1. Usage of several AI planners in parallel, no matter whether these planners are self-developed or are taken from existing. (1a)	Corresponds to the most general case regarding the principle of separating planning tasks from problem solvers. For the usage of existing AI planners, it may be necessary to convert an OWL-S planning problem to a planning language such as PDDL. See the discussion below.
2. A special imprecise planning based planner for the no-solution case. (2a)	Can be designed as described in Section 5.1.3.
3. Encoding the imprecise planning functionalities to be used with other planning techniques, i.e. with other AI planners. (3b)	This interesting approach is discussed below.
4. A compact self-developed solution. (1b, eventually 3a)	Solution variants 1b and 3a are combined because a development of two completely separate solutions is looked as too time consuming. The idea behind it is to concentrate primarily on 1b, and optionally on 3a, since the imprecise planning case is already treated by the above solution. Possible realizations are evaluated in Section 5.2.3.

As already mentioned, solution 3 should further be analyzed. In order to encode a planning problem, which should be solved by imprecise planning features such as constraints relaxation, as a planning problem that can be solved with existing and well-known planning techniques, a planning problem definition language is required which enables the specification of planning problems for the usage with the aimed features. Further, if one intends to forward planning problems to existing AI planners as in solution 1, such a language is needed ditto. PDDL offers itself as a language, since it has become a widely used

and accepted planning language and allows to express planning problems with sophisticated planning features. Considering imprecise planning, constraints relaxation for example could be encoded by the *Preferences* specification feature of PDDL 3.0.

A special compiler is needed additionally that not only converts OWL-S planning problems into PDDL, but also takes imprecise planning features into account. The converter used in OWLS-Xplan [Klusch et al., 2005], which is also available as a stand-alone component, extracts domain and problem information by parsing text lines of XML files, and converts them to PDDXML files. Since it does not quite seem to be suitable, I decided to develop a converter from scratch, but with the guide of the OWLS-Xplan converter.

Finally, regarding user interaction, the design decisions that have been made do not have any influence on it. User interaction is combined with offline planning. To enable the user to setup a planning problem to be solved with special planner features like imprecise planning according to his ideas, I decided a planning problem to be stateful, i.e. it should be session-like configurable and should accompany the user during a planning process.

5.2.3 Evaluation of a Self-Developed AI Planner

This section evaluates an AI planner that fits in the design of separating planning tasks from problem solvers. It should primarily generate alternative plans, and secondly support the imprecise planning features. The presented approaches would be self-developed solutions, possibly with support of other solutions.

- **State space planning:**

  State space planning provides a rather simple algorithmic base structure which is a good starting point for an own development and for extensions. It has proved, in conjunction with a heuristic function, to be one of the most successful AI planning techniques, and there are several planners (e.g. Metric-FF [Hoffmann, 2003]) whose source code is freely available and could be used as a guide. Unfortunately, state space planning is normally combined with depth-first search which is not compatible with the generation of plan alternatives. An adaptation of the A* heuristic search algorithm [Russell & Norvig, 1995, page 96] is a possible solution, but would have to be evaluated for feasibility. The algorithm is following shortly explained:

  The algorithm would be similar to the A* heuristic algorithm with beam search (keeping only a certain amount of nodes in the plan list Q). In contrast to A*, it would select the n best nodes of Q to branch, and in order to increase the diversity of resulting plan alternatives, the path valuation would take an additional positive weighted parameter, namely the degree of diversity between the path of the candidate node and the path of the first selected node, i.e. paths that are completely different to the best paths may get a higher chance to be selected. The algorithm would terminate not before n alternatives have been found, provided they exist.

- **Planning graph based planning:**

  A planning graph has the property to reveal parallel executable actions. This could be used to rapidly recognize the planning problem's potential to have plan alternatives. A plan extraction could generate alternative plans thanks to the locally parallel actions. The question is whether the plan alternatives, having rather minor differences, are enough decisive for the user. To gain more diverse alternatives, the plan extraction algorithm would have to be changed completely.
  A critical point is the difficulty of extending the Graphplan algorithm in a compatible way [Long & Fox, 2003].

- **Partial order planning:** (POP)

  POP is interesting due to the least commitment principle. It generates valid plans that are defined by only the absolutely necessary constraints to be valid, for example ordering constraints. So, a plan can then be interpreted in various ways, and in our scenario, could be used to extract several plan alternatives.
  The question is, as already discussed in Section 5.1.2, whether a plan extraction turns out to be as simple as one might guess, or whether it does not rather result in a complete re-analysis. Another aspect is, POP has recently become surprisingly competitive with other state-of-the-art planners [Nguyen & Kambhampati, 2001], but lags behind concerning new planning features. For example, there are no implementation propositions yet for the planning features introduced with PDDL 3.0, that would be necessary to encode constraint relaxations.

- **Planning as model checking**

  Planning as model checking mainly adapts the idea of verifying temporal formulas that formalize semantical domain properties to planning domains. It is especially effective in connection with the composition of Web services, since the underlying temporal logic like CTL allows to define non-deterministic behavior and partial observability. Another interesting aspect is the fact that the basic planning as model checking algorithm as presented in [Giunchiglia & Traverso, 1999] can be modified in order to generate automatically all potential solutions. However, the approach is relatively new in the planning research and thus lacks extensions to new planning features as proposed by the PDDL.

State space planning is chosen for a self-developed AI planner due to its advantages and in consideration of existing algorithms and extensions.

5.3 Evaluation of Sub-Concepts

Sub-concepts address features that are not directly subject of the thesis goals, but can greatly support the main concepts. Following, the sub-concept for a load balancing design is evaluated.

5.3.1 Load Balancing

The planning component is subject to time critical computations. Aside the usage of appropriate planning techniques, load balancing can help to slacken the problem. For an incorporation of a load balancing support, there are several architectural approaches, three are mentioned following, starting with the big picture and getting increasingly more fine grained:

1. Distribution of the planning component as a whole, i.e. the library is put on several machines and provides an access interface according to a common distributed communication framework like RMI.
2. Distribution of single parts of the planning component.
3. Considering even smaller code fragments, algorithms that support parallel processing and their distribution could increase the performance by means of a grid infrastructure.

The first approach mainly depends on whether the overlying application layer NExT supports remote interface access. And it has one big disadvantage: The API methods of the planning component have to be accessed and configured with complex objects as arguments, like ontology instances. For a distribution, those object structures would have to be serialized which causes additional time consumption as well as increased network load.

The second approach is interesting when regarding the AI planners as the library parts to be distributed, since these AI planners can easily be treated as atomic components and normally, several of them are executed anyway in parallel for a single planning run. So there is no difference for the planner engine. Only, the AI planners have to be prepared for their distribution by means of an encapsulation and an assignment with an interface. The serialization aspect is negligible since the AI planners are supplied with pre-compiled data which has a simple format structure such as plain text.

The third approach can be disregarded because most of the existing planning tools are only locally executable. There are of course exceptions (for example the GridSAT solver [Chrabakh & Wolski, 2004] or the SAT4SATIN solver[2] [Wrzesinska, Maassen, Verstoep, & Bal, 2005] based on SAT4J).

[2]SAT4SATIN is based on SAT4J, which is a satisfiability library for Java (see http://www.sat4j.org/).

The second approach was chosen due to the reasons mentioned. Nevertheless, a later implementation of the first approach is still possible by adapting the interface as well as the communication type between NExT and the planning component.

A design of a load balancing implementation has typically some characteristics, for example static versus dynamic: static load balancing is a pre-execution task distribution based on a prior knowledge, dynamic load balancing can adapt to changes in its target systems [Bustos, 2003]. On the spectrum between static and dynamic, different techniques line up and intermix. Whereas the former uses rather a centralized control for load measuring and resource allocation, the latter has autonomous node components acting upon their situated knowledge. A ground idea for a dynamic system is to have mobile components, i.e. time- and location-transparent components, so that they can be executed at the location providing the right amount of resources in terms of the global resource balance. An inferred requirement is the ability to serialize the components. Since most of the AI planners are native applications and a wrapping to JNI is not an alternative due to the additional time efforts that become necessary, and a dynamic load balancing in general would be an overkill for the requirements, a rather simple implementation with a centralized, static load balancing is preferred.

6

Concepts and Design

This chapter presents all concepts as a solution to the thesis goals and requirements. By starting with an overview, Section 6.1 that covers the main concept with its principles. I will then successively go into more detail in the following sections. Section 6.2 describes the most important elements of the architecture, Section 6.3 explains all aspects about the OWL-S process mapping to PDDL, Section 6.4 is all about AI planners and Section 6.5 about encoding planning features. Finally, Section 6.6 covers some sub-concepts that can be seen as separate but supporting concepts.

6.1 Main Concept

The main concept is based on two piles on which all other concepts build up. These are (1.) the decoupling of planning problem solving by the usage of different AI planners and (2.) cooperative planning. Figure 6.1 illustrates them from a static perspective.

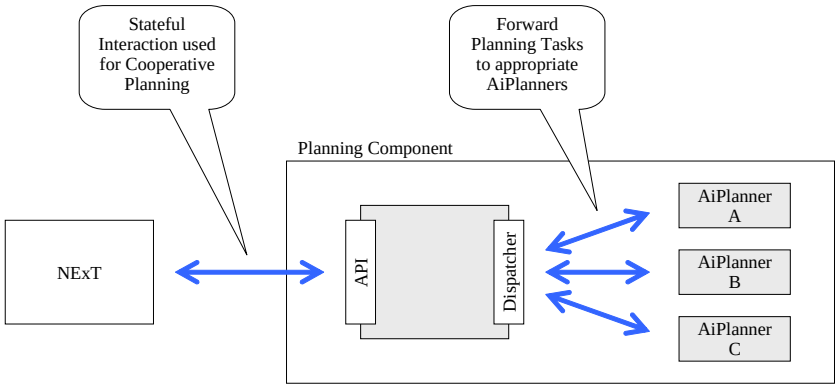


Figure 6.1: Architectural overview from an abstract, static perspective

- **Decoupling the planning problem solving by using different AI planners:**

  As described in Section 5.1 we have plenty of different planning tasks. And in order to gain most flexibility, their problem solving is decoupled by concrete solutions: A planning problem is given to the planning component by accessing a unique interface, what results in a planning task. Behind the scenes, there is a pool of different AI planners, and the planning task is allocated to a selection of appropriate planners. Of course the user has the possibility to influence the choice of planners by configuring a planning problem, nevertheless, it is mainly the responsibility of the planning component to have a strategy for making the right choice.
  The usage of different AI planners makes it possible in a simple way to generate several alternative plan solutions by executing a mixture of planners with diverse underlying planning techniques.

- **Cooperative planning:**

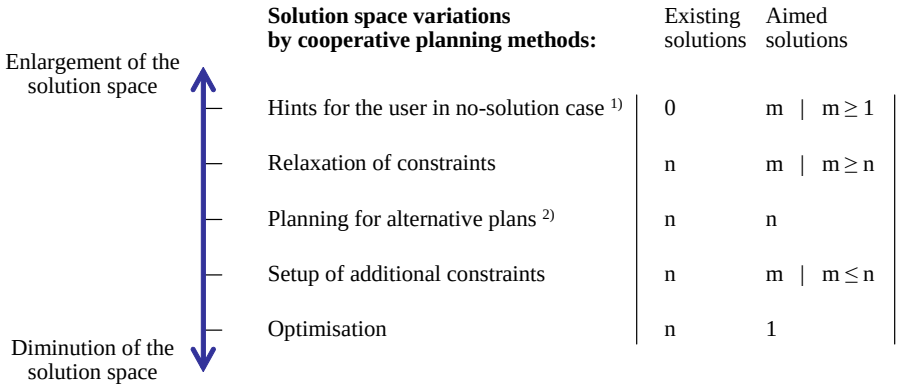
  It is about letting the user as well as the planning component cooperate in regard to the problem solving of a planning problem. It may often be the case that one party needs the support from the other (for example the planning component was not able to find a solution, so it might provide the user with information about the reasons), or one party can help to find a more accurate and satisfying solution (for example if the user has a picture of a solution in his mind the planning component does not find, he will then be able to configure an existing planning problem according to his ideas). Cooperative planning is a superordinate idea for particular other concepts:

  - **Stateful interaction:**
    In order to provide an infrastructure for the interaction between the user and the planning component that supports cooperative planning, the interaction type should be stateful, i.e. an object representing a planning problem (we call it a *OwlsPlanningProblem* instance) accompanies the user during the planning process and holds all ontologies and configurations that have been set. A planning process does not necessarily consist of only one planning run, but can have several runs in order to optimize the planning problem until the planning results correspond to the users expectations. For that purpose, the OwlsPlanningProblem instance can be updated and configured.
  - **Imprecise planning:**
    Imprecise planning is required for cases where the solution space has to be enlarged (see Section 5.1.3 for that issue). It should help the user to find other solutions. In case of no potential solution, the user is provided with hints. In all other cases in contrast, the user should give some hints by means of the configuration, for example what constraints to relax.
  - **Setup of additional constraints:**
    If a planning problem has too many potential or irrelevant plan solutions, the

setup of additional constraints will help to diminish the solution space (see Section 5.1.2 for details). It is again the user who knows which constraints should be set.

– **Optimization:**
The user might want to retrieve one optimal plan, so the planning component optimizes a planning problem according to pre-defined optimization criteria and returns the found solution.

The last three presented methods, namely imprecise planning, setup of additional constraints, and optimization, can all be arranged on a linear spectrum expressing their degrees of variation of the solution space, ranging from the reduction of the solution space to one solution (optimization), to the enlargement (imprecise planning) of it. See Figure 6.2 for the arrangement of all methods.



[1] Hints for the user are actually not plan solutions, but the planner intends with them the achievement of several solutions.

[2] Planning for alternative plans corresponds to the normal case, and it aims at generating n solutions.

Figure 6.2: Spectrum of solution space variations

- **Compilation of planning problems and cooperative planning features:**

A compilation of a planning problem into an intermediate planning problem definition language like PDDL supports the usage of different AI planners and cooperative planning features in an optimal way because of following reasons:

– An intermediate language (PDDL) standardizes the exchange of planning problems between the planning component and the AI planners, and enables the simple integration of existing AI planners.

– It allows to encode features of certain cooperative planning methods (imprecise planning in the general case, setup of additional constraints and optimization) as problem specified in the intermediate language.

6.2 Architectural Elements

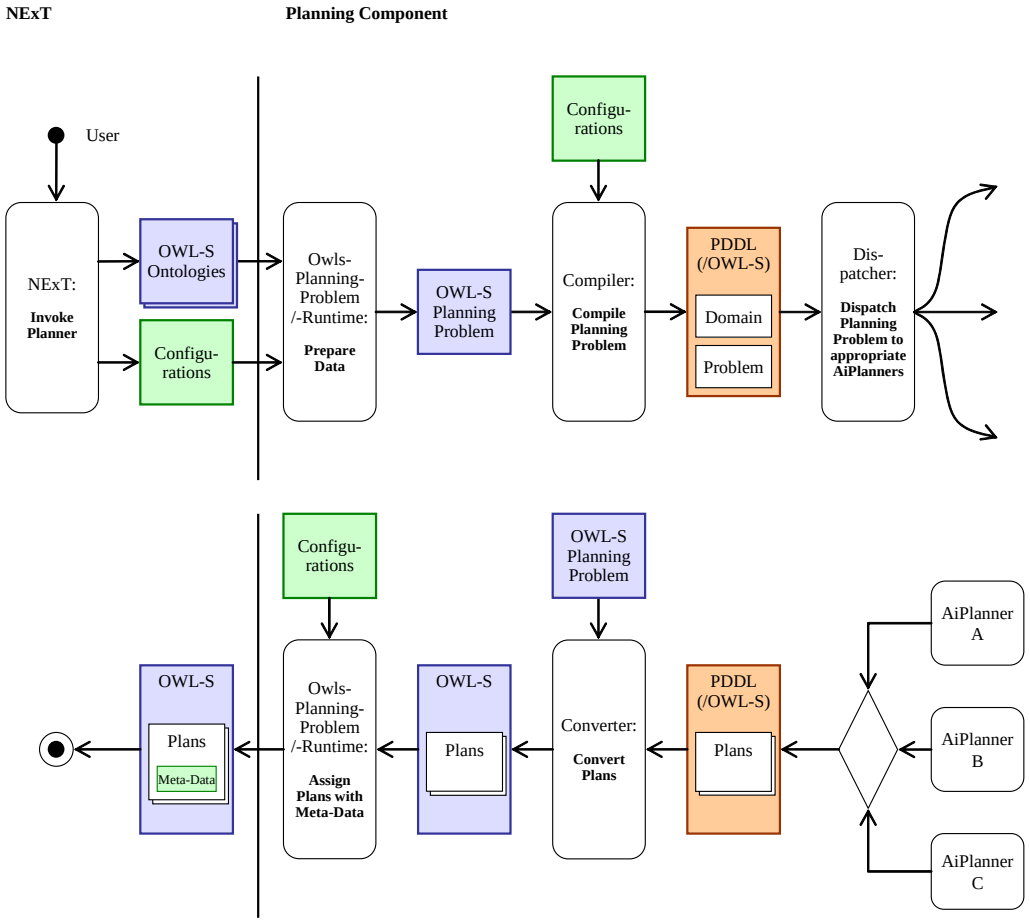
The concept intends a flexible call and data flow structure in order to optimally support the principles described above. The structure of elements involved in a planning process call is similar to that of the data flow. A basic flow could be summarized as following:

1. NExT: invokes the planning component to solve a planning problem after having set the necessary OWL-S ontologies and configurations,
2. OwlsPlanningProblem: collects all necessary data and prepares them according to the configuration,
3. Compiler: compiles a prepared planning problem into a planning specific description language (PDDL), taking into account cooperative planning features,
4. Dispatcher: dispatches the compiled planning problem to appropriate AI planners,
5. AiPlanners: solve the planning problem,
6. Converter: translates plans described in a planning specific language (PDDL) back to OWL-S, and
7. OwlsPlanningProblem: prepares plans by adding meta-data such as plan quality values.

Asynchronous message exchange is used for the planning component's interface access, especially for the invocation of the planning runs. The elements and their functionalities are described in detail below, starting from the invoker side with NExT. Various AI planners are covered separately in Section 6.4. Figure 6.3 gives an overview about the processing units and the intermediate data flows.

OwlsPlanningProblem and OwlsPlanningRuntime

An OwlsPlanningProblem instance represents the stateful object for the user interaction and can be opened at a OwlsPlanner instance (the entry point of the planning component). It is designed to let the user set the different OWL-S ontologies, namely arbitrary domain, initial and goal ontologies, and to let him configure the planning problem. If he starts a planning run on the planning problem, an OwlsPlanningRuntime instance will be created and will be run as a separate thread. With the configuration, the user is able to setup information about: general properties (time-out behavior), constraints relaxation, additional constraints, all kind of optimization information (QoS variables and weights, grounding weights), and others.



Both diagrams are UML activity diagrams and are additionally used for the illustration of data flows. The flow is separated into two diagrams which can be thought as connected on the AI planners' side.

Figure 6.3: Planning process: involved components and intermediate data flows

During a planning process, the components have two functions:

- **Preparation of planning domain and problem data:**
  This step collects all relevant processes, all types and properties dependent of the processes, and finally all individuals with their property values. These data are stored in tables for fast lookup. Another important task is the elaboration of a list of planning technique requirements: Based on the current state of the planning problem, i.e. whether this is the first planning run or not, and – if not – whether some plans could already be retrieved, and based on the configurations, different requirements for the planning problem solving arise. For example, if the planning problem was supposed to be optimized for certain aspects, the planning problem solving might require support for numeric fluents.

- **Preparation of plans:**
  As soon as some plans have been produced and converted back to OWL-S, the components calculate QoS values as defined in the configuration. They help the user to decide which plan alternative he wants to use. These values are then assigned to the plans among other information.

Compiler / Converter

They are both components that divide the entire data flow into two areas concerning description languages, namely one in which the planning problems and plans are described in OWL-S, and the other containing the compiled planning problems or plans described in an intermediate language (PDDL).

- **Compiler:**
  It compiles domain and problem representations. The most probable case is the compilation to PDDL. Since planning problem description languages are normally less expressive than OWL-S and some important information may be lost, a compilation to such a language brings with it a sophisticated translation mechanism that helps to translate all relevant parts and to annotate elements for a later reverse translation of corresponding plans. In addition, cooperative planning features are encoded as well. A sample compilation to PDDL, which can be used for other planning specific languages as a guide, is explained in Section 6.3.

- **Converter:**
  The converter translates plans back to adequate plans described in OWL-S with the help of annotations and the original OWL-S planning problem.

Dispatcher

The Dispatcher mainly forwards a prepared planning problem to appropriate AI planners according to the list of required planning techniques. In more details, it performs

following steps:

1. based on the requirements list for planning techniques, it collects all suitable AI planners, i.e. AI planners that support the required features,
2. since the collected AI planners may expect different languages for the planning problem definition, it runs the Compiler to retrieve all necessary planning problem formats, and
3. it dispatches the compiled planning problems to the AI planners in an asynchronous way.

6.3 Compilation of an OWL-S Planning Problem to PDDL

This Section addresses the sample conversion of a planning problem defined in OWL-S to PDDL. The compilation of cooperative planning features is treated separately in Section 6.5.

First of all, compositional aspects cannot be represented in PDDL. Therefore if the Compiler is supposed to find some composite processes, they will have to be mapped in some way to atomic PDDL actions. And this is done in two ways:

1. a composite process is mapped as-is to a PDDL action,
2. a composite process is decomposed in order to be able to use the resulting subprocesses for further mappings.

These two steps are recursively performed until only atomic processes result which can be mapped anyway. Simple processes are expanded to either an atomic or a composite process. The idea behind the described procedure is taking into account the QoS annotations of processes of both types. It might namely be possible that a composite process has proved to perform very well (e.g. it is reliable and fast) and to represent a good combination of other services such that it may be further treated as a template process.

The behavior of this decomposition procedure can be setup by means of configurations, i.e. one can set a maximum recursion depth or advice that composite processes should not be mapped directly to PDDL actions. Or it is possible to let the processes be annotated with QoS values in order to express a preference for atomic processes over composite process, or vice-versa.

Now let us look at the mappings of the various elements of an OWL-S planning problem definition to PDDL elements. To illustrate the mappings, the "Babel Fish Translator" Web service from the Mindswap website[1] is taken as example. A summary of all mappings is shown in Table 6.1.

[1]See http://www.mindswap.org/2004/owl-s/services.shtml for the source file. The Babel Fish Translator, known as the AltaVista Babel Fish Translator, is a simple text translator Web service. It was lightly

OWL-S / SWRL	➔	PDDL domain PDDL problem
Class(ID, subClassOf)	➔	type(name=ID, type=subClassOf)
Property(ID, domain, range)	➔	predicate(name=ID, variables={ {name="?domain", type=domain}, {name="?range", type=range}})
Process(ID)	➔	action(name=ID)
– hasInput(ID, parameterType)		– precondition: literal(name="agentKnows", {?p}) – ?p = parameter(name=ID, type=parameterType)
– hasPrecondition(ID, SWRL-Condition)		– precondition: formula=SWRL-Condition – parameters for all SWRL-variables
– hasOutput(ID, parameterType)		– effect: literal(name="agentKnows", {?p}) – ?p = parameter(name=ID, type=paramterType)
– hasEffect(ID, SWRL-Expression)		– effect: formula=SWRL-Expression – parameters for all SWRL-variables
Individual(ID, class)	➔	object(name=ID, type=class)
Description(class, property, value)	➔	fact for init state: literal(name=class, {property, value})

Table 6.1: Mapping of OWL-S elements to PDDL

We start with the domain description file in PDDL. For that purpose, we first map relevant OWL Classes to PDDL types. The class hierarchy is therewhile copied, too:

OWL-S	PDDL domain

XMLSchema "&xsd;#string"
Class "#SharedObject" subClassOf "#Thing"
Class "#Language" subClassOf "#SharedObject"
Class "#SupportedLanguage" subClassOf "#Language"
	`(:types string SharedObject - object`
	`Language - SharedObject`
	`SupportedLanguage - Language)`

OWL Properties (object properties and datatype properties) are triples (class, property, value) and are mapped to PDDL predicates (atomic propositions):

extended to be used as example.

OWL-S

```
ObjectProperty canBeTranslatedTo
      domain="#SupportedLanguage"
      range="#SupportedLanguage"
DatatypeProperty isLanguageOfText
      domain="#Language"
      range="&xsd;#string"
```

PDDL domain

```
(:predicates
    (canBeTranslatedTo
        ?domain - SupportedLanguage
        ?range - SupportedLanguage)
    (isLanguageOfText
        ?domain - Language
        ?range - string))
```

Things are getting more delicate when we move to the mapping of processes. We have two distinct types of parameters, namely:

1. parameters representing facts of the world, that are required to satisfy a precondition or that can be altered by effects, and
2. inputs and outputs which represent non-physical knowledge such as a translated text produced as an output.

The elements of the former type can directly be mapped to its equivalent elements in PDDL since they are described in SWRL as well-formed first-order like formulae. For the latter type, we should keep in mind that PDDL does not allow for describing non-physical knowledge. What we can say is: a planner knows an input. Inputs are semantically treated as knowledge preconditions and outputs as knowledge effects. For that reason, a predicate *agentKnows* is introduced with one variable that can either be bound to an input or an output parameter. The explained differentiation was used in [Narayanan & McIlraith, 2002] in the context of the analysis of Situation Calculus aspects, and is successfully applied in [Klusch et al., 2005].

OWL-S

```
AtomicProcess "BabelFishTranslatorProcess"
- hasInput "#InputString" type="&xsd;#string"
- hasInput "#InputLanguage" type="#Language"
- hasInput "#OutputLanguage" type="#Language"
- hasOutput "#OutputString" type="&xsd;#string"
- hasPrecondition "#SupportedLanguagePair"
      SWRL-Condition:
          propertyPredicate="#canBeTranslatedTo"
          args="#InputLanguage", "#OutputLanguage"
- hasPrecondition "#TextInInputLanguage"
      SWRL-Condition:
          propertyPredicate="#isLanguageOfText"
          args="#InputLanguage", "#InputString"
- hasEffect "#TextInOutputLanguage"
      SWRL-Expression:
          propertyPredicate="#isLanguageOfText"
          args="#OutputLanguage", "#OutputString"
```

PDDL domain

```
(:action BabelFishTranslator
 :parameters (
    ?InputString ?OutputString - string
    ?InputLanguage ?OutputLanguage
                                - Language)
 :precondition (and
    (agentKnows ?InputString)
    (agentKnows ?InputLanguage)
    (agentKnows ?OutputLanguage)
    (canBeTranslatedTo
        ?InputLanguage ?OutputLanguage)
    (isLanguageOfText
        ?InputLanguage ?InputString))
 :effect (and
    (agentKnows ?OutputString)
    (isLanguageOfText
        ?OutputLanguage ?OutputString)))
```

Moving forward to the mapping of PDDL problem relevant elements, we have OWL Individuals that can be described as PDDL objects.

OWL-S	**PDDL problem**

Language ”#English”
Language ”#Dutch”
..
	`(:objects`
	`English Dutch .. - Language`
	`.. )`

And finally all property values defined by means of OWL Descriptions may be used to describe the initial fact base, i.e. predicates are instantiated by assigning initial values to them.

OWL-S	**PDDL problem**

Description ”#English”
canBeTranslatedTo=”#Dutch”
Description ”#English”
canBeTranslatedTo=”#French”
..
	`(:init`
	`(canBeTranslatedTo English Dutch)`
	`(canBeTranslatedTo English French)`
	`.. )`

Having outlined a general example mapping, it should be mentioned that there are several other elements not existing in all planning problems:

- Conditional Outputs:
  If actions have conditional outputs and/or effects, each conditional output/effect group is separately expressed in PDDL by means of the effect clause *when* which defines that some effects are only applied if a certain condition holds.

- Time and Resource Constraints:
  The mapping of time and resource constraints requires that the Converter knows how they are defined for OWL-S processes. That is why the technique used to define them (e.g. with OWL annotations) has to be declared in the configurations. Resource constraints and other numeric constraints are mapped to numeric expressions, conditions and effects (PDDL 2.1 level 2). Time constraints in turn are mapped to durative actions (PDDL 2.1 level 3). Durative actions might not be supported by all planners.

- Optimization:
  Any numeric fluents, for example resource variables, that should be optimized, i.e. either minimized or maximized, are assembled to one variable which is then globally optimized. In PDDL, an optimization is described by metrics (PDDL 2.1 level 2). A weighting of the respective variables is therefore necessary and can be established by means of the configurations.

6.4 AI Planners

After having discussed how a planning problem is preprocessed, we will throw a glance at the engine of the planning component, at the AI planners. We need a potpourri of planners in order to solve the various planning tasks. Based on the organization of useful AI planner solutions as proposed in Section 5.2.2, I differentiate between following four types:

1. usage of existing AI planners,
2. a special imprecise planning based planner for the no-solution case,
3. encoding of cooperative planning features to be used with other AI planners, and
4. a compact self-developed solution.

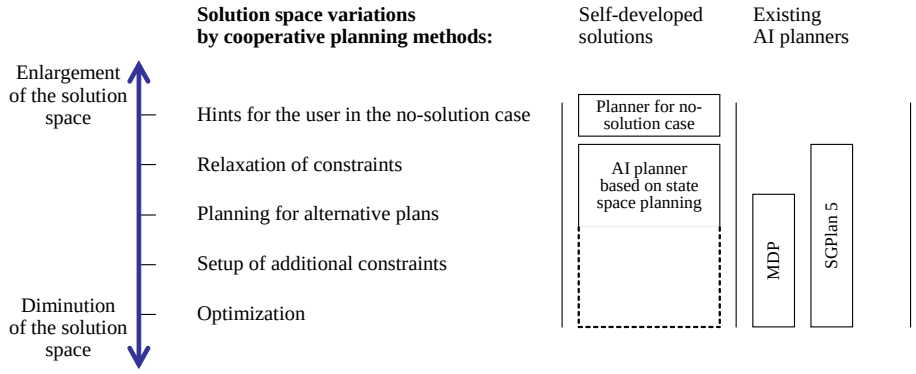


Figure 6.4: AI planner solutions regarding cooperative planning methods

Every solution has its own domain of application. And thanks to the flexible base architecture, several of them can used in combination. For example solution 3 might encode a planning problem by taking into account some constraint relaxations. The compiled planning problem might then be dispatched to two existing AI planners (solution 1) and to a solution of type 4. Figure 6.4 clarifies the application domains of the listed solution types. The four solutions theoretically allow, when considering the fact that solution 1 enables to integrate any imaginable planner, to implement an interesting amount of problem solving. This thought has to be seen in relation because some combinations may be incompatible. Taking the above example, a plugged-in existing AI planner that does not support the features of PDDL 3.0 might only be partially usable with solution 3.

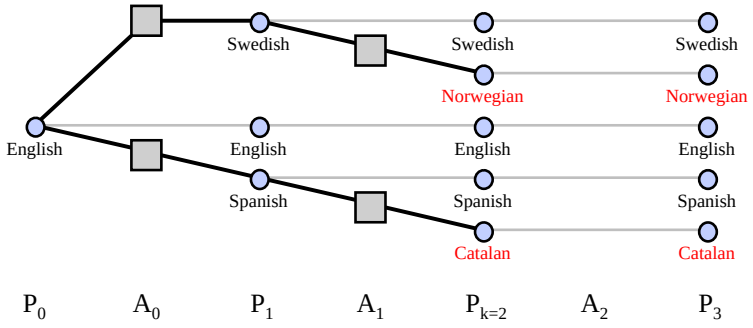
For solution 1 it should be mentioned that a good mixture of AI planners, i.e. planner based on different planning techniques, should be used in order produce plan alternatives. Solutions 2 and 4 are presented in the following subsections. Solution 3 is separately explained in Section 6.5.

6.4.1 Planner for No-Solution Case

The planner designed for the no-solution case (see Section 5.1.3) mainly concentrates on providing the user with useful information as hints. The solution has insofar a rather analytic nature. It is based on the creation of a relaxation planning graph and on the subsequent extraction of information. Following, two cases of no solution and the extraction of the relevant reasons (*mutual exclusions* and *unreachable goal atoms*) are explained by means of the "Babel Fish Translator" Web service.

For illustration purposes, the BabelFishTranslator planning problem is simplified to only have propositions for the languages the texts are written in or translated to. Further, the translator is capable of translating the language pairs: (English Swedish) (English Spanish) (Swedish Norwegian) (Spanish Catalan). Given a text in English, we search for a plan that incorporates the translation of the text to Norwegian, Catalan and Greenlandic.

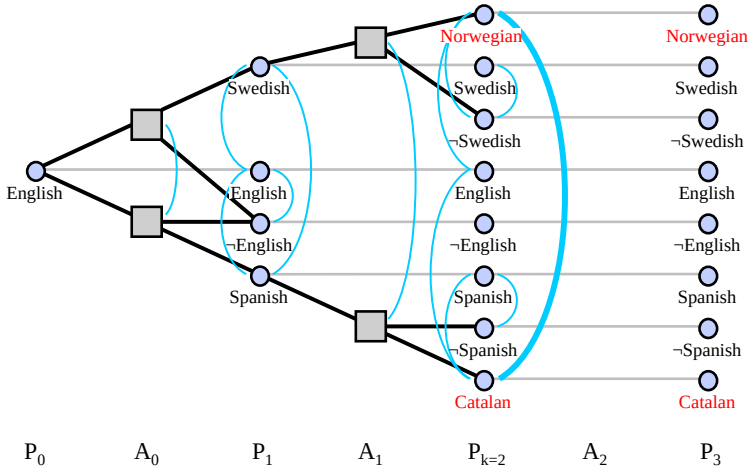
In order to find unreachable goal atoms, one simply takes the fix point layer of the planning graph and lists all missing goal propositions. In Figure 6.5, as shown with our example, there is one missing, namely Greenlandic, i.e. the translation to Greenlandic. The planner would therefore return the information that the goal Greenlandic can not be satisfied.



Layer P_2 is the planning graph's fix point. The goal Greenlandic has not appeared up to this layer.
Note: Actions are shown only the first time they are applicable to a proposition layer, but are in fact applicable to all subsequent proposition layers.

Figure 6.5: Extraction of unreachable goal atoms

In order to find competing goals, the planner analyzes the mutual exclusions of the planning graph. In the example planning problem, we assume that a translation eliminates the original text (the documents might be secret and there must be only one exemplar for each content). It turns out, as highlighted in Figure 6.6, that the translations to Norwegian and Catalan are mutex, i.e. they can not be performed at the same time within a plan. So the planner would return the information that Norwegian and Catalan are competing goals.



In Layer P_2, the goal propositions Norwegian and Catalan are mutex, because all ways to achieve them are mutex.
Note: Actions are shown only the first time they are applicable to a proposition layer, but are in fact applicable to all subsequent proposition layers.

Figure 6.6: Extraction of competing goals

Unsatisfiable global constraints can be determined in a similar way, but the original planning graph has to be extended to handle the respective features. The planning graph technique was chosen because – thanks to the relaxation principle – it clarifies reachability of propositions and other constraints and shows mutual exclusive relations between actions or between propositions. The relaxation can be applied to other aspects as well. [Long & Fox, 2003] provides a planning graph extension for temporal aspects while being compatible to the original planning graph to a maximum amount, and thus can be used to analyze temporal constraints.

6.4.2 AI Planner Based on State Space Planning

The feature of synthesizing plan alternatives is mainly addressed by the usage of several AI planners in parallel. By using a good mixture of AI planners that have different underlying planning techniques, it should be possible to synthesize a couple of different plans. However, it does not guarantee for it. For example, if there are two potential plan solutions, one of which is a optimal solution in terms of plan length and one of which is significantly worse, all AI planners may come up with the first plan. In this case, the sub-optimal solution is ignored though it might correspond to an interesting solution in the eyes of the user. In order to have a solution which for sure returns some alternative plans, a solution is following presented which does so.

```
Q ← initial node
C ← empty
A ← empty
repeat
    if Q is empty, return failure
    b ← first element of Q, Q ← rest(Q)
    T ← Q
    T ← sort(T, f − w_sim ∗ f_sim(n, b))
    B ← first k elements of T, or T
    for i := 1 to length(B)
        n ← first element of B, B ← rest(B)
        if n is a final node then
            A ← A ∪ {n}
            if length(A) = k, return A
        endif
        if n ∉ C, or has lower cost that its copy in C then
            add n to C
            S ← succ(n)
            S ← sort(S,f)
            Q ← merge(Q,S,f)
                (Q is ordered in increasing order of f(n) = g(n) + h(n))
        endif
    endfor
endrepeat
```

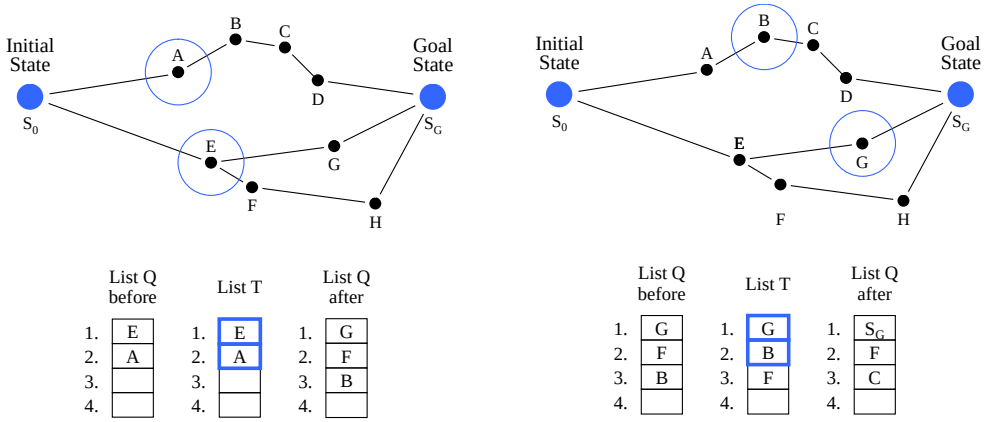
Figure 6.7: Modified A* search algorithm

The proposed solution is based on state space planning with heuristics. State space planning actually does not directly suggest to be used for generating plan alternatives, since common state space planners mostly use a depth-first search algorithm. This algorithm concentrates on exploring rapidly promising paths while loosing track of the big picture of the solution space. Aside depth-first search, there are many others, for example breadth-first search which first explores all neighboring nodes and makes fast planning more difficult. In contrast, the A* algorithm [Russell & Norvig, 1995, page 96] is lying somewhere in between the mentioned search algorithms, since it maintains a global list of partial solutions. However, by always taking the best node of the list, some potential solutions might easily be overlooked. I therefore propose a slight modification of the A* algorithm with beam search (limited list of partial plans) in three ways (the complete algorithm is shown in Figure 6.7):

1. for branching, select the k best or all nodes of Q, whereas k corresponds to the aimed number of plan alternatives,

2. for the above selection, copy a temporal list T from Q, and reorder it by taking an additional negative weighted parameter which is defined as a function measuring the degree of similarity between the path of the first selected node b and the path of the node n to be valued, and

3. terminate not before k solutions were found or Q is empty.



Iteration 2: List Q contains already two paths from iteration 1. The path to node E has a better heuristic value than the path to node A, because a plan through E is shorter than a plan through A. The Figure illustrates the branching upon the two nodes E and A.

Iteration 3: Without the usage of the specially ordered list T, the nodes G and F would have been chosen to branch and the plan through node N would have been ignored. The figure illustrates the purpose of the re-ordered list T, namely it considers plan solutions with different and possibly sub-optimal paths.

Figure 6.8: Two sample iterations of the modified A* algorithm

The first extension tries to conquer more paths at the same time in regard of solution alternatives. The second extension addresses the issue of overlooking some branches. Figure 6.8 illustrates both aspects with two sample iterations of the algorithm's repeat loop. The algorithm uses a heuristic function $f_{sim}(n, b)$ that calculates the similarity of two paths, namely those belonging to the first best chosen node b from Q and to the candidate node n, by comparing their path elements. The paths to be compared can be represented as vectors $\vec{x_n}$ and $\vec{x_b}$ containing either the nodes or the actions of the paths. One simple method to measure the similarity regarding computational simplicity is to count the number of identical nodes or actions and set the resulting value in proportion to the averaged total number of nodes or actions, respectively. Another possibility is to use the Jaccard measure which measures the ratio of the number of shared elements.

Using a similarity measure function, which has too much impact on the node selection in comparison to the heuristic function, might make the algorithm inefficient. The reason is that the algorithm would concentrate on finding paths, the main thing they are different, instead of trying to achieve the goal state. On the other side, a too low impact could disregard the idea of exploring all possible paths. The similarity measure function is thus additionally weighted to adjust its impact.

The algorithm presented is flexible in terms of extensions. Regarding the heuristic function, it is untouched and thus can be exchanged by any existing heuristic function, for

example one which is based on relaxed planning graphs.

6.5 Encoding of Cooperative Planning Features

Cooperative planning is linked to the formulation of respective planning features as planning problems in a planner specific language such as PDDL. The encoding for several cooperative planning methods is described in this Section. Cooperative planning makes intense use of configurations, for example when the user wants to relax a planning problem, wants to set-up additional constraints or wants to optimize it. Until now, the configurations were treated as black box and thus are first briefly explained.

Planning Problem Configuration

The configuration is an important element regarding the support for user interaction and cooperative planning. The shown configuration fragments should not be seen as implementational artifacts, but rather as a way of illustrating how the gap between the planning requirements of the user and the problem solving by AI planners is bridged.

Configurations are simple, textual key-value-pairs. Definitions of particular configuration elements, for example a definition of an OWL-S process annotation mapping, are separated from their activations. This allows the user to separately define configuration elements and switch them on and off. The listing below illustrates the mapping definitions of OWL-S process annotation. "processes.annotation.duration" is thereby one configuration element, whereas "processes.annotations" activates it.

The mapping definition of OWL-S process annotations have additional parameters that describe in details how an annotation should be used for planning. Different types of annotations exist. Most common types are QoS values which split into utility (type="utility") and costs (type="costs") values. Their relative QoS importance can be defined separately as QoS weight ("qos_weight"). They are used for to cases:

1. for calculating quality information of a plan like the total utility and costs of all processes of the plan, and
2. to optimize a planning problem for an optimal plan solution.

```
processes.annotations = duration, accessibility, energy-consume
processes.annotation.duration.declaration = owl_annotation
processes.annotation.duration.ref_name = "#ProcessDuration"
processes.annotation.duration.type = costs
processes.annotation.duration.qos_weight = 0.5
processes.annotation.accessibility.declaration = owl_annotation
processes.annotation.accessibility.ref_name = "#QoS_Accessibility"
processes.annotation.accessibility.type = utility
processes.annotation.accessibility.qos_weight = 2.0
processes.annotation.energy-consume.declaration = owl_annotation
```

```
processes.annotation.energy-consume.ref_name = "#EnergyConsume"
processes.annotation.energy-consume.type = costs
```

Setup of Additional Constraints

Additional constraints help to diminish the solution space. Possible constraints are function constraints, constraints for the grounding types of OWL-S processes, or state trajectory constraints. State trajectory constraints can be defined according to the declaration of plan constraints in PDDL 3.0. The configurations may be extended to support other constraint types.

```
constraints = energy-consume-constr, grounding-constr
constraint.energy-consume-constr.type = function_constraint
constraint.energy-consume-constr.ref_name = energy-consume
constraint.energy-consume-constr.value = "< 10"
constraint.grounding-constr.type = owls_grounding_constraint
constraint.grounding-constr.value = java
```

Relaxation of Constraints

Relaxation of constraints helps to enlarge the solution space. The relaxations presented here concentrates mainly on the relaxation of goal predicates. They can be assigned with an additional penalty cost, in order to express their relative importance. Apart from goal predicates, it is of course possible to relax all kind of constraints that have been setup explicitly by means of configurations, e.g. state trajectory constraints. This would allow to take into account the relative importance of these constraints.

```
relaxations = soft-goals1, soft-goals2
relaxation.soft-goals1.type = goal_predicate
relaxation.soft-goals1.value = Greendlandic
relaxation.soft-goals1.penalty = 3
relaxation.soft-goals2.type = goal_predicate
relaxation.soft-goals2.value = Norwegian, Catalan
relaxation.soft-goals2.penalty = 5, 6
```

Optimization

An optimization strategy is defined by means of several optimization elements and a composition of arbitrary of them. In PDDL, an optimization for a planning problem is defined with the metrics feature (PDDL 2.1 level 2).

```
optimizations = qos
optimization.qos.metrics = annotations(duration, accessibility)
```

6.6 Sub-Concepts

6.6.1 Information Gathering

Information gathering was primarily stated as a useful feature in conjunction with imprecise planning. It should help to complement a knowledge pool with missing information and thus higher the chance of finding some solutions. However, information gathering is a feature that can be used more generally in nearly all cases and anytime during a planning procedure. The feature is following explained in details.

We assume to have a planning problem, including (1) processes, some of them which are of the information-gathering type only and any others that might alter the world state, and (2) an initial knowledge base consisting of facts and of untyped data the software only knows. Further, we do not care about the completeness of the knowledge base. The information gathering feature is organized to be either optionally switched on for planning runs or to be invoked explicitly. So the user might switch it on and do a planning run. Plans are normally synthesized, but the information gathering feature additionally collects all information retrieval processes that are for sure or may be relevant for the execution of the resulted plans or for further planning. NExT can then execute them in order to retrieve the information. This could have several consequences: (a) Some information has missed and its acquisition allows now for finding some plan solutions, (b) the obtained information may indicate an inconsistency of the plans, or (c) the user may given the opportunity to profit from the planners' increased reasoning capabilities and from better plan solutions. In such cases a replanning step would be necessary.

6.6.2 Load Balancing

The concept of using several planners in parallel gives the impression that the computational load would rise enormously. On the first sight this is correct, and thus load balancing makes a contribution to making the software more scalable. On the other hand, the parallelism of computations just leverages the concept in respect of load balancing, since it allows to distribute the computational load.

In order to explain the load balancing architecture, the AiPlannerManager helper class is first introduced.

AiPlannerManager

Since a plant of different planners possibly on several platforms may be used to tackle a solving of a planning problem, somebody is needed who manages all these diligent helpers. And this is exactly the task of the AiPlannerManager. It has the overview about installed planners any time, it instantiates them to let them plan, and is responsible to provide information about the planners such as supported features or required domain and problem

description formats. The Dispatcher does actually not address AI planners directly, but rather asks the AiPlannerManager for suitable planners.

Load Balancing Concept

A load balancing mechanism is incorporated to make the application more scalable. A major reason for it is the usage of several AI planners in parallel. Among different possibilities, load balancing was decided (see Section 5.3.1)

1. to be applied to the execution of AI planners corresponding to the most time critical part and
2. to be realized in a rather simple centralized and static fashion.

The *Active Object* design pattern [Lavender & Schmidt, 1995] lends itself for a design. Looking at an object that should be executed distributed and concurrently, the pattern decouples the method's execution from the method's invocation, so that execution and invocation can take place at different locations, and introduces concurrency by using asynchronous method invocation and a scheduler for handling requests. An active component consists of the following components:

- a *proxy* which provides an interface towards clients with publicly accessible methods
- a *method request* representing a method invocation from the client
- an *activation queue* where the methods requests are stored
- a *scheduler* which decides which pending request to execute next
- a *servant* being the implementation of the active object
- a *future* with whom clients can obtain the results of a method's execution

AiPlannerManagers and AI planners are designed as active objects and can be freely distributed. A special Dispatcher takes care of all available AiPlanningManagers, no matter on which computer they are located. It acts as a proxy when the conventional Dispatcher searches for appropriate AI planners.

7

Implementation

A prototype of a design helps to recognize whether the assumptions and decisions about a design are correct or not. Further, it clarifies critical aspects which could danger a design decision. And this is exactly what the prototype of the planning component architecture targets at. A complete prototype was not developed, but rather some critical parts. This chapter first gives an overview about the implementation, and then focuses on specific aspects and gives feedback on whether the design is feasible or has to be further evaluated.

7.1 Overview

The global organization is basically done according to the design (see Figure 7.2 for the dynamic aspects). All components and sub-components were organized in four package groups, and three abstraction layers (Figure 7.1 illustrates the packages and their dependencies):

- **Component Interface / OwlsPlanner**
  This part can be seen as the heart of the planning component. On one side, the whole communication between NExT and the planning component is transacted by an API with interfaces. On the other side, this software part is a concrete implementation of the API, controls all planning processes and makes use of the other packages. In more detail, it maintains planning problems, prepares them for a plan finding and executes appropriate AI planners.
- **Planning Engine (AI Planners)**
  The planning engine is responsible for all concrete planning jobs. It manages all AI planners and allocates the necessary infrastructure for planning.
- **Models**
  We have one input model (OWL-S), and several intermediate models (e.g. PDDL) to be used for the AI planners. This package group provides a set of needed model and basic converters.

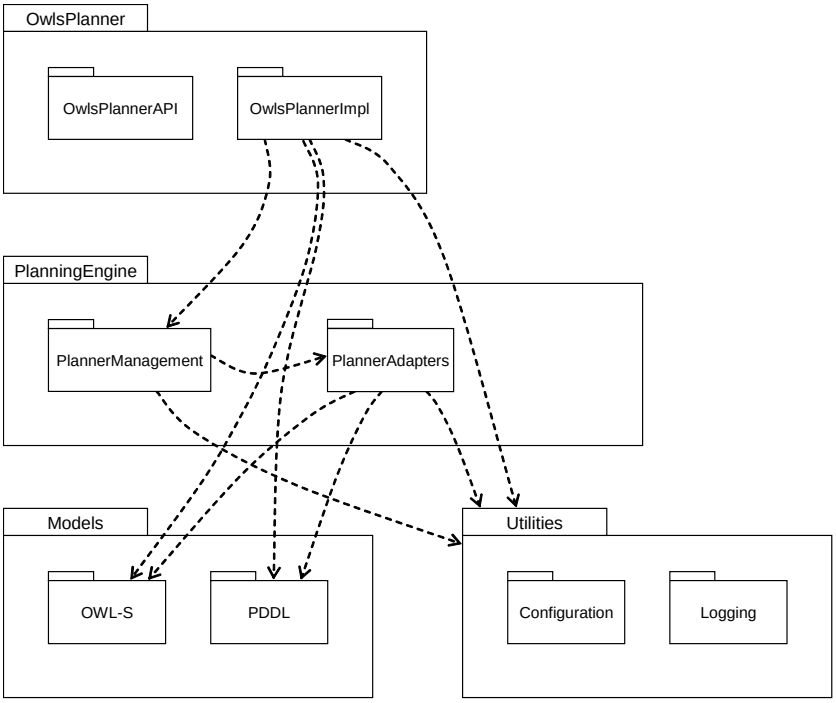


Figure 7.1: Package overview and dependencies

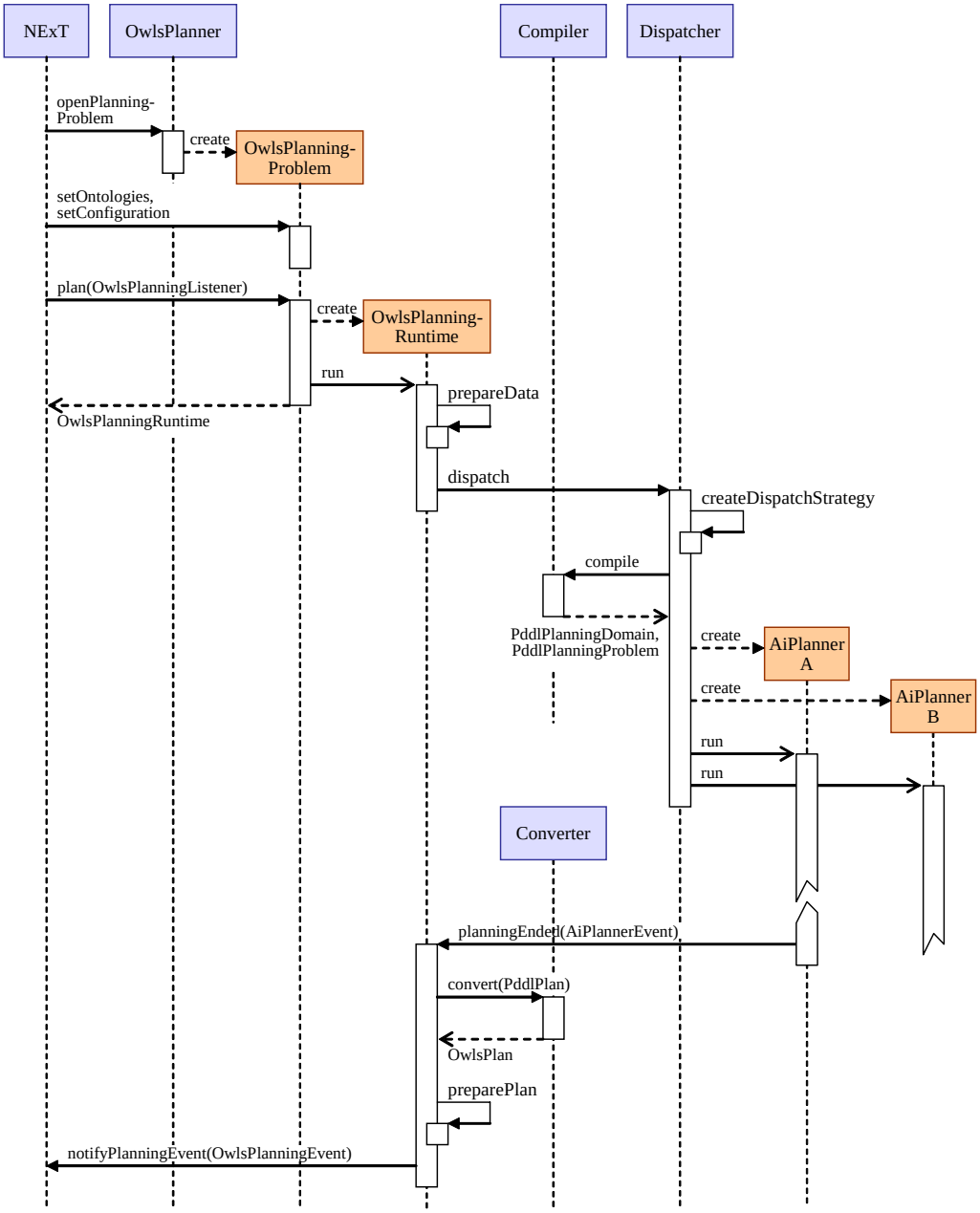
- **Utilities**
  The utilities packages contain components used by all software parts.

7.2 Components

7.2.1 API / User Interaction

A major goal of the concept was the support for a continuous user interaction. It was therefor interesting to see how the user interactivity could be realized by an API.

Technically, the user interactivity is mainly established by providing a stateful class `OwlsPlanningProblem` (see Figure 7.3), which accompanies the user during the whole planning process. An instance of it can be created by the `OwlsPlanner`, which in turn is created by the `OwlsPlannerFactory` class. It lets the user set all domain and problem ontologies, and configure the planning problem. The configuration is implemented as a simple hash table of key-value text pairs. It is possible to import or export the complete configuration.



The execution of a planning process is twofold asynchronously decoupled from its invocation by creating and running an OwlsPlanningRuntime instance and AiPlanner instances as separate threads.

Figure 7.2: UML sequence diagram of a planning process

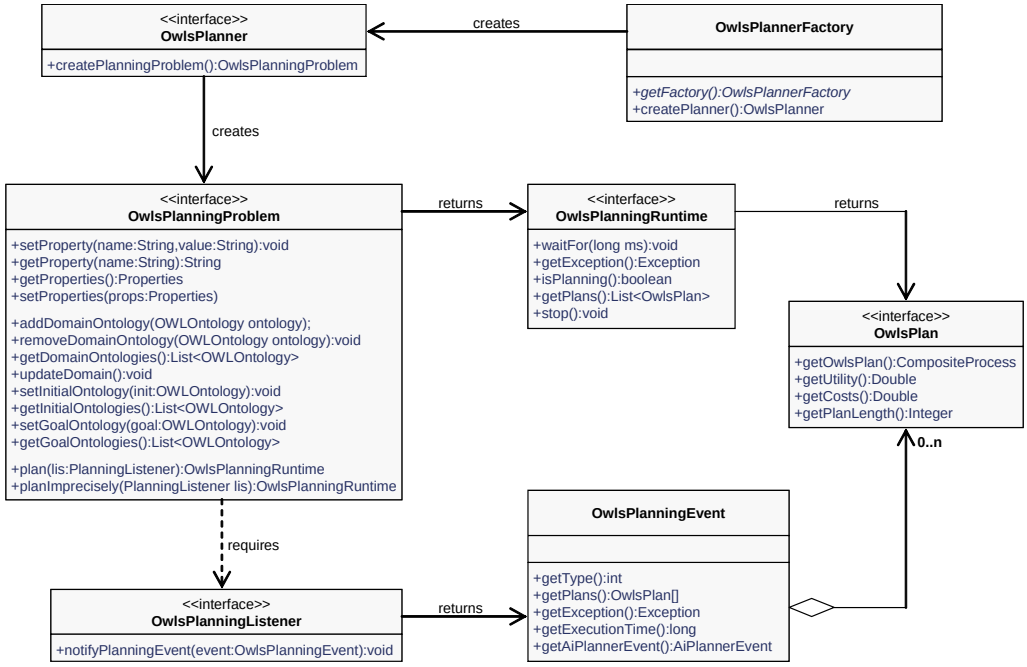


Figure 7.3: API classes and interfaces used for the user interaction

User interaction is supported by asynchronous communication as proposed in the design. If the user starts a planning run, an `OwlsPlanningRuntime` instance is returned. It is a runtime object, i.e. it provides all up-to-date information from the planning run and allows to immediately stop the process. The user is notified by an `OwlsPlanningEvent` as soon as a AI planner is has found some plans, was unable to or if its execution has failed due to another reason.

7.2.2 Models

Models of planning problems or solutions are intensively used for data exchange and conversion. An implementation based on the principles of simplicity and reusability thus assigns each model with a unique identifier and enables their automatic conversion to other models. All models are serializable.

The PDDL model was completely implemented as tree of different classes and the hierarchy structure was adapted from the PDDL version 3.0 specification. Figure 7.4 shows the model part `Action`. The lexer and parser tool ANTLR was used to translate textual model representations to PDDL models.

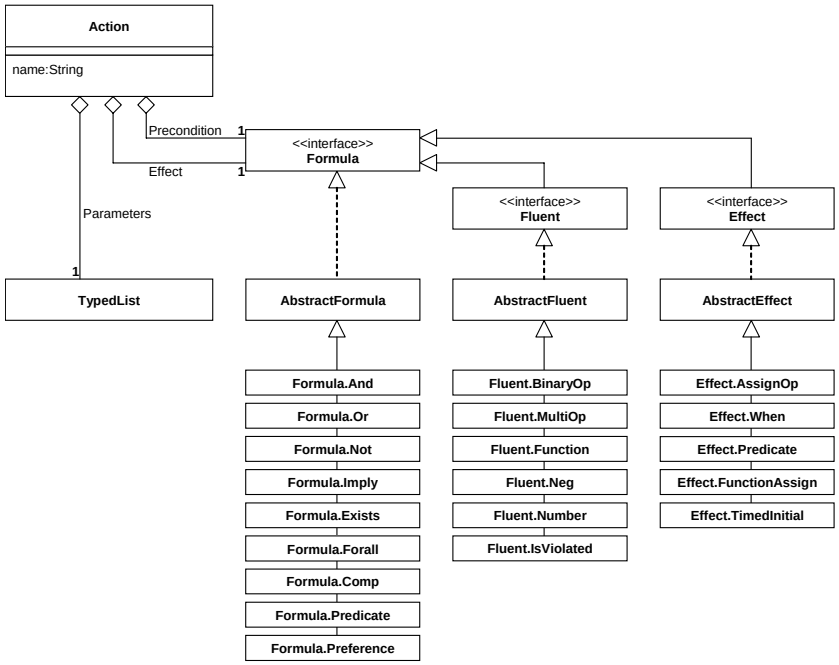


Figure 7.4: Model implementation for a PDDL action specification

7.2.3 AI Planner Infrastructure

AI planners need a sub-layered infrastructure to perform successfully. The infrastructure should enable to plug-in them as external components, also if they are native compiled, and to separately configure them. Since AI planners are a central element of the software and are part of multiple-planners idea, it was important to realize such an infrastructure.

An instantiated planning component library has a planners directory for placing in the different AI planners, each in separate directory. The planners directory is either specified relatively to the planning component's library file (owls-planner.jar) or absolutely. Each planner specific directory consists of a mandatory configuration file and optional libraries that are automatically loaded by an extra class loader. The configuration file provides information about the name of the AI planner, the supported features and models, and can at the same time be used to configure the planner, for example set-up some command line options.

Every AI planner has to implement the `AiPlannerFactory` and the `AiPlanner` interface. In order to create an AI planner instance, `AiPlannerManager` invokes the factory's `createPlanner` method with the properties of the configuration file as argument. Figure 7.5 shows all involved classes and interfaces, including the `AiPlannerEvent`s which have to be created by the AI planner for a proper interaction with planning component.

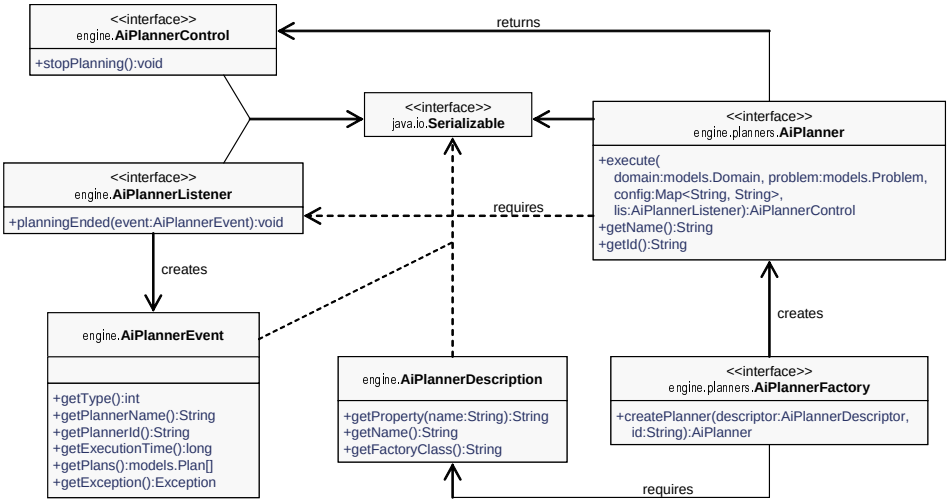


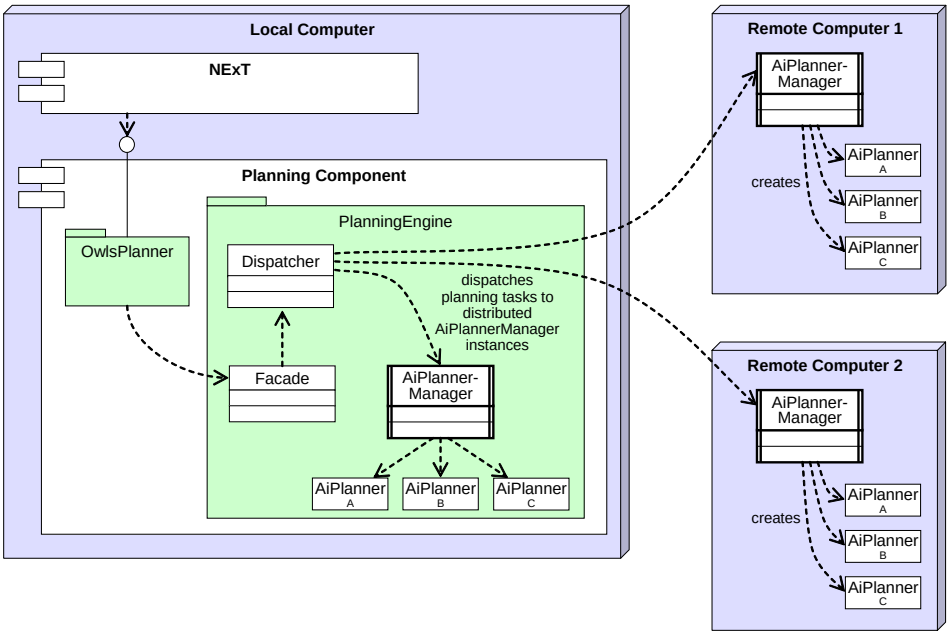
Figure 7.5: AI planner adapter classes and interfaces

7.2.4 Load Balancing

Local balancing, corresponding to a sub-concept, attracts the interest because its feasibility would emphasize the usage of several planners in parallel. Following, the load balancing implementation is explained in detail.

Two classes were implemented as active objects (AOs) to be used in conjunction with the framework ProActive [Baduel et al., 2006]: It is firstly every AI planner implementation (i.e. the different implementations of the `AiPlanner` interface), and secondly the class `AiPlannerManager`. Every `AiPlannerManager` singleton instance of an involved active computer is registered by means of a registry service (e.g. Jini registry service) and bound to an absolute address. The `Dispatcher` instance looks them up, i.e. those which are available, and forwards a planning request not only to the local `AiPlannerManager` instance but to all AOs found. A dispatch strategy defines to which `AiPlannerManager` AOs it should dispatch. A dispatch decision depends on whether they support the asked planning features and whether their system has sufficient free memory and CPU resources. The `Dispatcher` instance obtains `AiPlanner` instances as AOs from the `AiPlannerManager` AOs. It then has full control over the AI planners: it can execute them and since the execution is decoupled due to the asynchronous invocation, it can also stop them anytime.

The `AiPlannerManager` AO does not have a *scheduler* implementation since the methods executed remotely consume practically no processor time and the default FIFO based scheduler is sufficient. A `AiPlanner` implementation does not have a scheduler implementation either because it has mainly one method. Instead, it has to take care to immediately return a `AiPlannerControl` object which allows the planner caller to stop it.



The `AiPlanningManager` class and implementations of `AiPlanner` interface are all active objects.

Figure 7.6: Distribution of the AI planning load: a static view of the involved components

The implementation was successfully tested. The application of a load balancing strategy turned out to be a relatively uncritical implementation job.

7.3 Discussion

Some parts of the software could already be implemented and can emphasize the correctness of the design until now. On the other side, a testing of a full planning run is currently not possible. Several aspects are discussed following in more detail:

- **User interaction**
  An API was implemented according to state-of-the-art patterns to enable the user interaction. Stateful objects and asynchronous communication with the listener-event mechanism are quality arguments of the API. The configuration is deliberately implemented as primitive key-value text pairs to enable ease of usage, but it could be useful to add more intuitive methods for the setting of configurations.
  A final statement can not be made, until the user interaction was intensively tested from within NExT.

- **Plan alternatives**
  The generation of plan alternatives mainly builds on the multiple-planners idea. Whereas

this idea allows to plug-in any imaginable AI planner, it must be evaluated whether the current usage of only existing AI planners is sufficient to produce satisfying alternatives.

- **Cooperative planning**
  Apart from the no-solution case, most cases correspond to known and solvable problems, e.g. optimization or relaxation of constraints by means of PDDL planning features. On the other hand, some example planning problems were used to theoretically test the correctness of the concepts. But more testing is needed, especially with process examples from the NMR experiment domain.

- **Usage of several AI planners in parallel**
  In this respect, some important effort has already been made, namely the implementation of relevant parts. Factoring out the aspect of plan alternatives, one can say that the design was feasible and effective.

8

Conclusions

8.1 Summary

The goal of the thesis was the concept elaboration for an architecture of a planning component that semi-automatically creates plans based on pre-defined OWL-S processes. Great store was set on:

- focus on *user interaction*,
- generation of *plan alternatives*, and
- *imprecise planning* in case of no or too few solutions.

Aside, different design alternatives should have been evaluated first, and a prototype should confirm the design decisions.

The concept was developed for two different scopes, namely primarily to be used as a planning component for the process management system NExT in the NMR experiments domain, and additionally as an overall solution for the Web service composition problem.

A detailed analysis of the goals has shown that a set of different planning features is necessary to deal with all goal aspects and that an incorporation of corresponding planning techniques into a compact solution turns out to be an undertaking since specialized planning algorithms tend to be incompatible when being integrated with each other. It was not surprisingly to see existing Web service composition solutions concentrating on some of the required planning aspects only. An extension of one of them would amount to a self-developed solution.

Therefore, using low coupling and reusability as fundamental principles, a concept was chosen that separates different planning tasks from concrete planning implementations, i.e. from specialized planning techniques. It enables on one side the generation of several plan alternatives quasi by nature, and on the other side the integration of arbitrary specific planners that can be selected to solve a given planning task.

User interaction is supported by asynchronous communication, i.e. by planning offline and by immediately returning the control to the user, and by a stateful object that represents a planning problem of an NMR experiment case the user is working at. The user configures the planning problem and plans with it, but it is the planning component's task to forward the planning problem to appropriate planners.

The concept intends a flexible compilation of a planning problem into the planning language PDDL or any other compatible language, in order to encode different planning task functionalities and in order to optimally support the planning problem solving by the subsequent planners.

Taking a planning result of a couple of plan alternatives as the normal case, we may have the case of no or too few solutions, so the user's demand would be to get more solutions what corresponds to the idea of imprecise planning, or we may have too many or imperfect solutions and the user might wish a containment. All these variations of the solution space are tightly coupled to the user's preferences and were thus realized as the concept of cooperative planning.

Imprecise planning is addressed in two ways: (1) If a planning problem has no potential solution, an analysis of it with the help of an extensible planning graph yields reasons, (2) generally, it is possible to configure constraints relaxations on all imaginable constraints.

The diminution of the solution space can be attained by setting up additional constraints such as state trajectory constraints, and by configuring optimization criteria based on weighted QoS values of processes and on other values.

Considering the typical problems of automated Web service composition, most of them could be addressed thanks to the concept. Non-deterministic behavior is handled by iteratively consistency checking and replanning after a process execution of plan step. Incomplete knowledge can be complemented by the information gathering feature that searches for all relevant information retrieval processes in order to query the missing world facts. Complex plans can be defined by the setup of additional constraints.

Finally, a load balancing concept was developed to make the software, especially with regard to the parallel execution of different planners, more scalable.

The prototype was not completely implemented, but rather parts of it to proof some concept aspects. The API was realized to see how the user interaction can be established by interfaces. A complete PDDL model was created based on a flexible interface/class hierarchy to be able to compile planning problems into it. Concerning the parallel execution of different planners, a plug-in system like infrastructure for planners and a load balancing mechanism was implemented. The listed parts could successfully be implemented and confirm the concept ideas so far.

8.2 Future Work

Having developed and presented a complete concept for a planning component according to specific goals, I consider some ideas for future work that might complement or extend the concept. Though the planning component can not be seen as a complete software system with several sub-components, it offers the plugging-in of other components, i.e. AI planners which fit into the architecture. Whereas the basic concepts look very promising and does not primarily have to be enhanced, various additional AI planners solutions could be considered separately. This Section thus concentrates mainly on the later type of extensions.

8.2.1 AI Planners

As already mentioned, the architecture of the planning component enables the easy integration of other AI planners. Considering cooperative planning and the spectrum of different solutions space variations, we may think about planners that support any subset of the various cooperative planning methods. Following, some plausible extensions are described. In Figure 8.1, these extensions are arranged on the solution space variation spectrum.

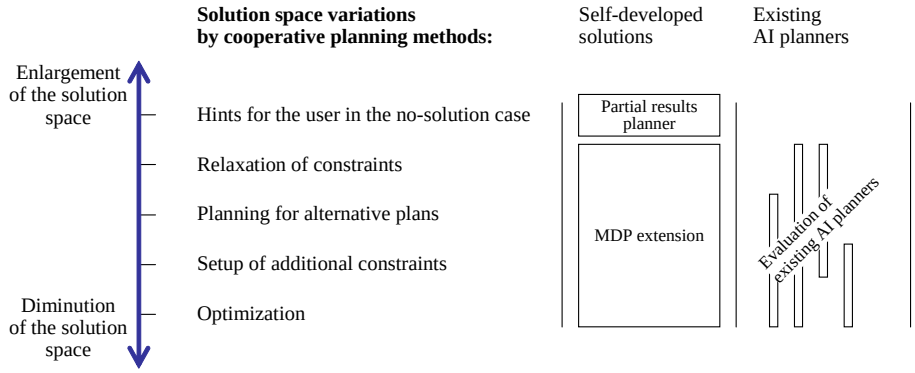


Figure 8.1: Additional AI planner solutions regarding cooperative planning methods

Partial Results Planner

It may often be the case that a given planning problem has no potential solution. In such a case, the user should be provided with some reason which he can interpret as hints. If this solution turned out to be not sufficient, i.e. users might wish to obtain more results, one could envisage the development of a partial results planner. It may return partial results in the form as described in Section 5.1.3 or of any other type.

Evaluation of Existing AI Planners

A major goal of the thesis is the synthesis of plan alternatives. The presented concepts intend several ways to address that problem, including the usage of several planners. A simple way to extend the overall capability of the planning component is to plug-in existing AI planners. Therefore, one might evaluate all plausible planners, eventually with minor adaptations. In connection with the International Planning Competitions (IPC) researchers are stimulated to develop outstanding new planners. This is a promising fact: Since these planners may have additional features, they could optimally be integrated and would highly contribute to planning component usability.

One point should be noted. When intending the integration of existing AI planners in order to mainly produce plan alternatives, one should keep in mind that they possibly output the same solutions. An evaluation for the purpose of plan alternatives should thus rather be seen as an exact analysis of the planners concerning their planning techniques and supported planning features. On the other hand, if one looked for a specific feature such as the generation of complex plans, this would not have to be taken into account.

Extension of the MDP

The Model Driven Planner (MDP) [Bertoli et al., 2001] is based on planning as model checking and would theoretically enable the production of plan alternatives [Giunchiglia & Traverso, 1999]. Currently it does not. The planner supports all features of PDDL 2.1 and some additional features for uncertainty. The latter features could be used to produce complex plans if a user demands them. Anyway, the planner has interesting properties and one could therefore think about an developing an extension in regard to the alternative plans feature. Since the planner is not available under Open Source license, the extension should be considered to be developed in cooperation with the authors.

8.2.2 Load Balancing

Load balancing is a sub-concept that was enabled by the separation of planning tasks from different problem solvers, and that in turn supports the planning component regarding performance. An interesting fact is that in the environment of NExT, i.e. the computational chemistry science environment, specialized computer programs are used to calculate the structures and properties of molecules and solids. The programs have high computational complexity and thus are normally run on grids to distribute the load. This theoretically permits the planning component to distribute the AI planners' load on such grids. The load balancing architecture would have to be modified, possibly in such a way that the planning component could be adapted to some grid environment on-the-fly.

A

Formalization of a Planning Problem

A planning domain has to be formalized in order to be understandable by a planner. Domain theories address that issue by defining the rules of the domain operations (for example preconditions). There exist several methods to represent domains. Most of them follow the model of the state-transition system, as described in Section 3.1.1. They have different levels of expressiveness, and range from simple add- and delete-lists of propositions (for example with STRIPS) to logical formulations in first-order logic (for example Situation Calculus or Event Calculus) (see Figure A.1). Pure logic based approaches provide precise semantics and the ability to reason about the domain, but are expensive in terms of computational complexity.

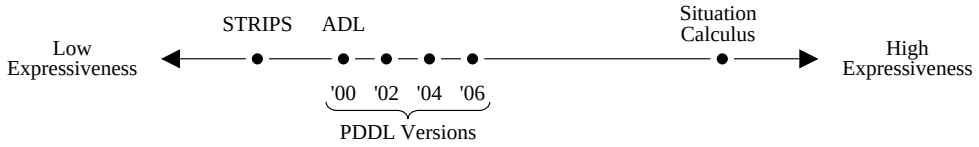


Figure A.1: The Expressiveness of Description Languages

The choice of a formal planning problem representation is thus always a trade-off between the expressiveness of general logical formalism and computation complexity of reasoning with that representation. A logical consequence for AI planners which use an expressive description language (like for example PDDL in the version used for the IPC 2006) is the tendency towards a specialization of the planners on specific subsets of the language features.

In this section, some common representation methods are captured.

A.1 STRIPS

STRIPS [Nilsson, 1982] corresponds to the classical representation scheme and has taken over the name from an early automated planning system STRIPS (Stanford Research Institute Problem Solver), where it originally has been used. It takes all restricted assumptions listed in Section 3.1.2.

STRIPS uses a language $\mathcal{L}$ with notations derived from first-order language: $\mathcal{L}$ has a finite set of predicate and constant symbols, and some additional symbols and expressions. It is based on logical atoms, that can have a true or a false value within some interpretation. Operators change the truth value of the atoms.

A state s is a set of ground atoms and is represented by:

- atomic propositions, that are either positive (an atom p holds in s iff $p \in s$) or negative. An example of a state:
  $On(A, B)$, $On(C, Table)$, $Clear(B)$, $Handempty()$, $Holding(C)$

The elements of an operator representation are:

- preconditions: a conjunctive list of literals (positive or negative atoms). The operator o is applicable in state s if the set of literals $precond(o)$ can be satisfied by s ($s \models precond(o)$).
- effects: a conjunctive ADD-list ($effects^+(o)$) of literals to be added to the world state and a conjunctive DELETE-list ($effects^-(o)$) for those to be removed.

Example of an operator description in STRIPS:

```
(def-strips-operator (pickup ?x)
  (pre (handempty) (clear ?x) (ontable ?x))
  (add (holding ?x))
  (del (handempty) (clear ?x) (ontable ?x)))
```

A.2 ADL

ADL (Action Definition Language) [Pednault, 1989] generalizes STRIPS by extending the restricted set of formulas supported by STRIPS. The extensions include the formalization of preconditions and effects as first-order formulae, and the support of conditional effects. In addition, the language $\mathcal{L}$ is extended by functions symbols, so functions can be used to specify fluents on a numeric basis. Functions have the drawback that a planning problem might become undecidable.

Example for negated and quantified formulas, and for conditional effects:

```
(def-adl-operator (move ?x ?old ?new)
  (pre (and (on ?x ?old)
            (not (?old = ?new))
            (not (exists (?z) (on ?z ?x)))
            (not (exists (?z) (on ?z ?new)))))
  (add (on ?x ?new))
  (del (on ?x ?old))
  (forall (?z) (implies (above ?x ?z)   (del (above ?x ?z))))
  (forall (?z) (implies (above ?new ?z) (add (above ?x ?z)))))
```

Example for functions (fluents):

```
(def-adl-operator (store-in-room ?x ?r)
  (pre (> (load-capacity (floor-material ?r)) (weight ?x)))
  (update (storage-capacity ?r)
            (- (storage-capacity ?r) (base-area ?x))))
```

A.3 Situation Calculus

Situation Calculus [McCarthy & Hayes, 1969] is a first-order language for representing states and actions that change states. It introduces the notion of *situation* in order to provide one logical theory for all domain states , in contrast to most other representation techniques which use a logical theory for every state. Thereby the whole expressive power of a first-order language can be used, for example the usage of axioms describing general behaviours of the domain. On the other side, it has a critical weakness: Situation Calculus reasons explicitly about changes in a domain, but it can not deduce the non-effects (things that remain unchanged) of actions. A way out are successor state axioms [Reiter, 2001].

Each atom takes an extra situation argument for the state s in which it is true. For example:

$on(A, B, s)$	A is on B in situation s
$weight(Fred, s) = 100$	Freds weight in situation s

Actions are represented as terms in the same first-order language the states are described in. If α is a variable denoting an action, $do(\alpha, s)$ is the situation resulting from the execution of α in situation s. Action preconditions are represented as *action precondition axioms* by means of a predicate $Poss$. For example:

$$\forall r \forall l \forall l' \forall s(Poss(move(r, l, l'), s) \leftrightarrow at(r, l, s))$$

Action effects are represented as *action effect axioms*. For example:

$$\forall r \forall l \forall l' \forall s(Poss(move(r, l, l'), s) \rightarrow at(r, l', do(move(r, l, l'), s))$$

A.4 PDDL

PDDL (Planning Domain Definition Language) [Ghallab et al., 1998] introduced at the first IPC 1998 for standardization purposes, takes a middle way between high-expressiveness associated with computation complexity and unrealistic, simple representations[1]. The language was built on ADL and since continuously extended to enable more realistic domain descriptions (see Appendix B for more details).

[1] Actually ADL already targeted at offering a description language in the middle between STRIPS and the Situation Calculus concerning expressiveness. But PDDL goes one step further and incorporates the ideas of the modern planning approach (see Section 3.1.3).

B

PDDL - The Planning Domain Description Language

B.1 History

The *Planning Domain Description Language* [Ghallab et al., 1998] has been developed in the context of the first International Planning Competition (IPC) 1998, originally from Drew McDermott. The language was necessary to have a standardized planning domain and problem description language for the competition. The IPC 2000 specification was significantly reduced, and closer to what most planning systems actually support. Since then it was continuously extended for the IPCs. The language is based on ADL (for propositions) and UMCP (for actions). The language is in the meantime widely accepted and used for the exchange of planning problems.

The IPC 2002 has changed in a way the previous objectives due to the complaint in some research communities that the competitions have focused too much on artificial benchmark problems instead on tackling real problems that might have longer term interest [Fox & Long, 2002]. The organizers have realized the danger of putting off participants with potential good technologies and have pointed on meeting real challenges.

B.2 Elements of a PDDL Planning Problem Definition

The description is separated into two parts, 1. one for the description of parameterized actions that characterize domain behaviors (domain description), and 2. another for the descriptions of specific objects, initial conditions and goals that characterize a problem instance (problem description). A domain description can be used for several problem descriptions, but not vice-versa.

The notation is Lisp like. The grammar of PDDL is defined in extended BackusNaur form (EBNF) and is available under http://zeus.ing.unibs.it/ipc-5/bnf.pdf for PDDL version 3.0.

Actions may have parameters (syntax: `?parametername`). These are variables standing for terms and are bind during the grounding of actions to object instances of the problem instance. The pre- and post-conditions of actions are expressed as logical propositions constructed from predicates and logical connectives.

Requirements
Since many planners may support only a subset of the features describable in PDDL, a list of requirements defines the needed features for a planning problem. Some commons are listed below:

`:strips`	only the most basic features of STRIPS
`:typing`	support for typing (of action parameters and functions)
`:fluents`	using functions definitions: arithmetic preconditions and effects with assignment operators
`:adl`	all ADL features (STRIPS/typing plus negative/disjunctive/quantified preconditions, equality, conditional effects)

B.2.1 Domain Description

The domain description contains the definition of all language constructs referenced in the actions (for example types, predicates, functions) and the definition of the actions itself. The structure of a simple domain description looks as following:

```
(define (domain DOMAIN_NAME)
  (:requirements [:strips] [:typing] [:adl] ...)
  [(:types TYPE_A1 ... TYPE_AN - object
           TYPE_B1 ... TYPE_BN - SUPERTYPE
           ...)]
  (:predicates (PREDICATE_1_NAME [?A1 ?A2 ... ?AN])
               (PREDICATE_2_NAME [?A1 ?A2 ... ?AN])
               ...)

  (:action ACTION_1_NAME
    [:parameters (?P1 ?P2 ... ?PN)]
    [:precondition PRECOND_FORMULA]
    [:effect EFFECT_FORMULA])
```

```
  (:action ACTION_2_NAME ...)

  ...)
```

B.2.2 Problem Description

The problem definition contains mainly the objects of the problem, the initial state and the goals. See following basic structure:

```
(define (problem PROBLEM_NAME)
  (:domain DOMAIN_NAME)
  (:objects OBJ_1 OBJ_2 ... OBJ_N)
  (:init ATOM_1 ATOM_2 ... ATOM_N)
  (:goal CONDITION_FORMULA)

  ...)
```

B.3 Extensions

B.3.1 Version 2.1 (IPC-3 2002)

Of the five levels proposed, levels 1-3 have been accepted and were affiliated as new features for the IPC 2002 [Fox & Long, 2002]. The three levels are:

- **Level 1**
  - **ADL planning**
    Corresponds the the STRIPS/ADL fragment of PDDL IPC-2 2000.
- **Level 2**
  - **Numeric Expressions, Conditions and Effects**
    Numeric expressions are constructed from primitive numeric expressions using arithmetic operations.
  - **Plan Metrics**
    Plan metrics define for what a planning problem is evaluated. They are defined by means of a numeric expression. Example:
    `(:metric minimize (total-fuel-used))`
- **Level 3**

– **Durative Actions**
Durative actions explicitly represent time and durations. Whereas logical change is considered to be instantaneous (discretized durative actions), numeric values may change over the interval of an action (continuous durative action).
The modeling of temporal relationships in a discretized durative action is done by means of temporally annotated conditions and effects. An annotation of a condition makes explicit whether a condition must hold at the beginning, at the end, during the action, or any combination of them. An annotation of an effect makes explicit whether an effect is immediate (at the action's beginning) or delayed (at the action's end).

B.3.2 Version 2.2 (IPC-4 2004)

PDDL version 2.2 [Edelkamp & Hoffmann, 2004] introduces derived predicates and timed initial literals.

- **Derived Predicates**
Derived predicates enable the handling of domain axioms. The instance of a derived predicate is true if the rule of the form *if formula(x) then predicate(x)* can be derived. They are not affected by any of the actions. For example:

```
(:derived (above ?x ?y)
  (or (on ?x ?y)
      (exists (?z) (and (on ?x ?z)
                        (above ?z ?y)))))
```

- **Timed Initial Literals**
Timed initial literals are syntactically a very simple way of expressing deterministic unconditional exogenous events: They represent facts that will become true or false at time points that are known to the planner in advance. For example:

```
(:init
  (at 9 (shop-open))
  (at 20 (not (shop-open)))
)
```

B.3.3 Version 3.0 (IPC-5 2006)

The extensions introduced in PDDL version 3.0 [Gerevini & Long, 2006] address mainly the definition of more sophisticated goals.

- **Plan Constraints**
  Plan constraints are state trajectory constraints, which are constraints on the structure of the plans. They can be hard, to be used to express control knowledge or restrictions on the valid plans for a planning domain and/or for a specific planning problem, or soft, to be used to express preferences that affect the plan quality, without restricting the set of the valid plans. The following example shows some hard trajectory constraints:

```
(:constraints (and (sometime (at coffee-room))
                   (sometime (and (at coffee-room)
                                  (coffee-time)))) )
```

- **Preferences**
  Soft goals and soft constraints are preferences, i.e. conditions marked as such, whose opportunity cost is expressed by a penalty weight. Normally, not all specified preferences can be satisfied, and identifying the best subset of preferences that can be achieved is an extra difficulty to deal with in the planning progress. The violation penalties are defined by means of the metrics to be optimized. Following example illustrates the combination of preference marks and the metric construct.

```
(:goal (and (finished job1)
            (finished job2)
            (finished job3))
            (preference reviewing (reviewed paper1)) )

(:constraints (and (preference break
                     (sometime (at coffee-room)))
                   (preference social
                     (sometime (and (at coffee-room)
                                    (coffee-time))))) )

(:metric minimize (+ (* 5 (total-time))
                     (* 4 (is-violated social))
                     (* 2 (is-violated break))
                     (is-violated reviewing)) )
```

List of Figures

List of Tables

References

Baduel, L., Baude, F., Caromel, D., Contes, A., Huet, F., Morel, M., et al. [2006, January]. Grid Computing: Software Environments and Tools. In (chap. Programming, Deploying, Composing, for the Grid). Springer-Verlag.

Bernstein, A. [2000]. How can cooperative work tools support dynamic group processes? bridging the specificity frontier. In *Proc. of the 2000 acm conference on computer supported cooperative work (cscw 2000)* (p. 279-288). ACM Press.

Bernstein, A., Kaufmann, E., Kiefer, C., Bürki, C. [2005]. *SimPack: A Generic Java Library for Similiarity Measures in Ontologies* (Tech. Rep.). Department of Informatics, University of Zurich.

Bertoli, P., Cimatti, A., Pistore, M., Roveri, M., Traverso, P. [2001]. Mbp: a model based planner.

Blum, A., Furst, M. [1995]. Fast planning through planning graph analysis. In *Proceedings of the 14th international joint conference on artificial intelligence (ijcai 95)* (p. 1636-1642).

Bustos, J. [2003]. Robin hood: An active objects load balancing mechanism for intranet. In *Proc. of the workshop de sistemas distribuidos (wsdp'2003).*

Chrabakh, W., Wolski, R. [2004]. Solving hard satisfiability problems using gridsat. In *Proc. of the fourteenth global grid forum (ggf14).*

Cockburn, A. [2000]. *Writing effective use cases*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.

Dänzer, M. [2005]. *Next - the nmr experiment toolbox*. Unpublished master's thesis, University of Zurich.

Edelkamp, S., Hoffmann, J. [2004, January]. *Pddl2.2: The language for the classical part of the 4th international planning competition* (Tech. Rep. No. 195).

Fox, M., Long, D. [2002]. The third international planning competition: Temporal and metric planning. In *Proc. of the sixth international conference on ai planning & scheduling (aips'02).*

Gerevini, A., Long, D. [2006]. Preferences and soft constraints in pddl3. In *Proc. of icaps workshop on planning with preferences and soft constraints* (p. 46-53).

Ghallab, M., Howe, A., Knoblock, C., McDermott, D., Ram, A., Veloso, M., et al. [1998].

Pddl - the planning domain definition language (Technical Report CVC TR-98-003/DCS TR-1165). Yale Center for Computational Vision and Control.

Ghallab, M., Nau, D., Traverso, P. [2004]. *Automated planning (theory and practice)*. Morgan Kaufmann Publishers.

Giunchiglia, F., Traverso, P. [1999]. Planning as model checking. In *Sixth international conference on ai planning & scheduling (aips'02)* (p. 1-20).

Glinz, M. [2005]. Rethinking the notion of non-functional requirements. In *Proc. of the 3rd world congress for software quality (wcsq 2005).*

Hoffmann, J. [2003]. The metric-ff planning system: Translating ïgnoring delete listsẗo numeric state variables. *Journal of Artificial Intelligence Research (JAIR)*, *20*, 291-341.

Hoffmann, J., Nebel, B. [2001, 5]. The ff planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research (JAIR)*, *14*, 253-302.

Hsu, W., Wah, B. W., Huang, R., Chen, Y. X. [n.d.].

Klusch, M., Gerber, A., Schmidt, M. [2005]. Semantic web service composition planning with owls-xplan. In *Proc. of the 1st international aaai fall symposium on agents and the semantic web (aaai-fss 2005), arlington va, usa* (p. 55-62). AAAI Press.

Koehler, J., Nebel, B., Hoffmann, J., Dimopoulos, Y. [1997, 1,]. *Extending planning graphs to an ADL subset* (Tech. Rep. No. report00088).

Kuter, U., Sirin, E., Nau, D., Parsia, B., Hendler, J. [2006]. Information gathering during planning for web service composition. In *Proc. of the 3rd international semantic web conference (iswc2004), 7-11 november 2004, hiroshima, japan.*

Lavender, R. G., Schmidt, D. C. [1995]. Active object - an object behavioral pattern for concurrent programming. In *Proc. of the second pattern languages of programs conference in monticello, illinois, september 6-8, 1995.*

Levesque, H. J., Reiter, R., Lesperance, Y., Lin, F., Scherl, R. B. [1997]. Golog: A logic programming language for dynamic domains. *Journal of Logic Programming*, *31*(1-3), 59-83.

Long, D., Fox, M. [2003]. Exploiting a graphplan framework in temporal planning. In *Proc. of icaps'03* (p. 51-62).

McCarthy, J., Hayes, P. J. [1969]. Some philosophical problems from the standpoint of artificial intelligence. In B. Meltzer D. Michie (Eds.), *Machine intelligence 4* (pp. 463–

502). Edinburgh University Press. (reprinted in McC90)

McDermott, D. [1996]. A heuristic estimator for means ends analysis in planning. In *Proc. of the 3rd international conference on artificial intelligence planning systems (AIPS-96)* (pp. 142–149). AAAI Press.

McIlraith, S., Son, T. C. [2002]. Adapting golog for composition of semantic web services.

Narayanan, S., McIlraith, S. A. [2002]. Simulation, verification and automated composition of web services. In *Proc. of the 11th international conference on world wide web (www'02)* (p. 77-88). New York, NY, USA: ACM Press.

Nguyen, X., Kambhampati, S. [2001]. Reviving partial order planning. In *Proc. of the 17th international joint conference on artificial intelligence (ijcai'01)* (p. 459-466).

Nilsson, N. J. [1982]. *Principles of artificial intelligence*. Berlin, Heidelberg: Springer.

Pednault, E. P. D. [1989]. Adl: Exploring the middle ground between strips and the situation calculus. In *Proceedings of the first international conference on principles of knowledge representation and reasoning* (pp. 324–332). San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

Peer, J. [2004, 9]. A pddl based tool for automatic web service composition. In *Second workshop on principles and practice of semantic web reasoning (ppswr 2004) at the 20th international conference on logic programming (st. malo, france)* (p. 149-163). Springer (Berlin).

Peer, J. [2005a]. A pop-based replanning agent for automatic web service composition. In *Proc. of the 2nd european semantic web conference (eswc 2005)* (p. 47-61).

Peer, J. [2005b, 3]. *Web service composition as ai planning - a survey* (Tech. Rep.). University of St.Gallen.

Peer, J. [2006]. *Description and automated processing of web services*. Unpublished doctoral dissertation, University of St. Gallen.

Phan, M., Hattori, F. [2006]. Automatic web service composition using congolog. In *Proc. of the 26th ieee international conference on distributed computing systems (icdcs'06) workshops* (p. 17). Washington, DC, USA: IEEE Computer Society.

Rao, J., Su, X. [2004, 6]. A survey of automated web service composition methods. In *Proc. of the first international workshop on semantic web services and web process composition (swswpc 2004), san diego, california, usa, july 6th, 2004.* Springer-Verlag.

Reiter, R. [2001]. *Knowledge in action: Logical foundations for specifying and implementing dynamical systems*. Cambridge, Massachusetts: The MIT Press.

Russell, S. J., Norvig, P. [1995]. *Artificial intelligence: A modern approach*. Prentice Hall.

Sirin, E., Parsia, B., Wu, D., Hendler, J., Nau, D. [2004]. Htn planning for web service composition using shop2. *Journal of Web Semantics*, *1*(4), 377-396.

Spalzzi, L. [2001]. A survey on case-based planning. *Artificial Intelligence Review*, *16*(1), 3–36.

Srivastava, B., Koehler, J. [2003]. Web service composition current solutions and open problems. In *Proc. of the 13th international conference on automated planning & scheduling (icaps 2003).*

Srivastava, B., Koehler, J. [2004]. Planning with workflows - an emerging paradigm for web service composition. In *Proc. of the the 14th international conference on automated planning and scheduling (icaps 2004).*

Traverso, P., Pistore, M. [2004]. Automated composition of semantic web services into executable processes. In *Proc. of the 4th international semantic web conference (iswc 2004).*

Veloso, M. M., Mulvehill, A. M., Cox, M. T. [1997]. Rationale-supported mixed-initiative case-based planning. In *Proc. of the fourteenth national conference on artificial intelligence (aaai97) / the ninth conference on innovative applications of artificial intelligence (iaai97)* (p. 1072-1077).

Weld, D. S. [1994]. An introduction to least commitment planning. *Artificial Intelligence Magazine*, *15*(4), 27-61.

Weld, D. S., Anderson, C. R., Smith, D. E. [1998]. Extending graphplan to handle uncertainty and sensing actions. In *AAAI/IAAI* (p. 897-904).

Wrzesinska, G., Maassen, J., Verstoep, K., Bal, H. E. [2005, December]. *Satin++: Divide-and-share on the grid.* (Submitted for publication)