Customizing the Composition of Web Services and Beyond

by

Shirin Sohrabi

A thesis submitted in conformity with the requirements
for the degree of Doctor of Philosophy
Graduate Department of Computer Science
University of Toronto

Copyright © 2012 by Shirin Sohrabi

Abstract

Customizing the Composition of Web Services and Beyond

Shirin Sohrabi
Doctor of Philosophy
Graduate Department of Computer Science
University of Toronto
2012

Web services provide a standardized means of publishing diverse, distributed applications. Increasingly, corporations are providing services or programs within and between organizations either on corporate intranets or on the cloud. Many of these services can be composed together, ideally automatically, to provide value-added service. Automated Web service composition is an example of such automation where given a specification of an objective to be realized and some knowledge of the state of the world, the problem is to automatically select, integrate, and invoke multiple services to achieve the specified objective. A popular approach to the Web service composition problem is to conceive it as an Artificial Intelligence planning task. This enables us to bring to bear many of the theoretical and computational advances in reasoning about actions to the task of Web service composition. However, Web service composition goes far beyond the reaches of classical planning, presenting a number of interesting challenges relevant to a large body of problems related to the composition of actions, programs, and services. Among these, an important challenge is generating not only a composition, but a high-quality composition tailored to user preferences.

In this thesis, we present an approach to the Web service composition problem with a particular focus on the *customization* of compositions. We claim that there is a correspondence between generating a customized composition of Web services and non-classical Artificial Intelligence planning where the objective of the planning problem is specified as a form of control knowledge, such as a workflow or template, together with a set of constraints to be optimized or enforced. We further claim that techniques in (preference-based) planning can provide a

computational basis for the development of effective, state-of-the-art techniques for generating customized compositions of Web services.

To evaluate our claim, we characterize the Web service composition problem with customization as a non-classical planning problem, exploit and advance preference specification languages and preference-based planning, develop algorithms tailored to the Web service composition problem, prove formal properties of these algorithms, implement proof-of-concept systems, and evaluate these systems experimentally. While our research has been motivated by Web services, the theory and techniques we have developed are amenable to analogous problems in such diverse sectors as multi-agent systems, business process modeling, component software composition, and social and computational behaviour modeling and verification.

Dedication

To my parents.

Acknowledgements

I owe sincere and earnest thankfulness to my Ph.D. supervisor, Sheila McIlraith. With her continuous support, understanding, mentorship, encouragement, and most importantly friendship she inspired me and gave me the courage to overcome the challenges in the completion of this research work. I will forever be grateful to her.

I am grateful to the members of my supervisory committee: Fahiem Bacchus, Michael Gruninger, Renée Miller, who provided me with continuous feedback throughout the thesis-writing process. I would also like to thank my external examiners Marsha Chechik and Paolo Traverso who also provided valuable suggestions on my thesis.

During my studies, I had the pleasure of working with several distinguished scientists. In particular, I would like to thank those whom I have collaborated with as a part of the work described in this thesis: Jorge Baier, Sotirios Liaskos, Sheila McIlraith, John Mylopoulos, Nataliya Prokoshyna, Anand Ranganathan, Anton Riabov, and Octavian Udrea.

I am thankful to the current and past members of the Knowledge Representation Group who provided me with feedback and constructive criticism on the early stages of my work as well as my practice talks.

I would like to thank the Department of Computer Science and also greatly acknowledge funding from the Natural Sciences and Engineering Research Council of Canada (NSERC).

Last but not least, I would like to thank my immediate family for their unconditional love and support, my mother Marjan Maha, my father Mansour Sohrabi, my brother Ali Sohrabi, and my future husband Oktie Hassanzadeh as well as my extended family, specially my grandfather Ebrahim Sohrabi who lived just a few days short of seeing the thesis finished. This thesis is dedicated to my family.

Contents

Glossary of Acronyms and Abbreviations

Notation	Description	Pages
BPEL	Business Process Execution Language for Web Services	**26**
BPM	Business Process Management	**1**
Golog	alGOl in LOGic. A high-level action-centric language for programming agents	**5**, **45**
HTN	Hierarchical Task Network	**6**, **51**
IPC	International Planning Competition	**98**, **108**, **112**
LTL	Linear Temporal Logic	**9**, **65**, **101**
OWL-S	Semantic Markup for Web Services	**43**
PDDL	Planning Domain Definition Language	**74**
PDDL3	Version of PDDL that supports preferences	**8**, **73**
QoS	Quality of Service	**34**
Soft Constraints	Preferences	**61**
WSC	Web Service Composition	**1**

Chapter 1

Introduction

1.1 Motivation

Web services are loosely-coupled, self-contained, Web-accessible programs that can be published, discovered (or located), composed, invoked, and executed. Web services provide a standardized means for diverse, distributed software applications to be published on the Web and to interoperate seamlessly. Increasingly, corporations are providing services or programs within and between organizations either on corporate intranets or on the cloud. Many of these programs can be composed together to achieve complex behaviour. Information-gathering services such as the weather service at www.weather.com and world-altering services such as the flight-booking service at www.aircanada.com, are examples of Web applications that can be described and composed as Web services. They might be coupled as part of a travel-booking service, for example.

While today's Web is designed primarily for human interpretation and use, the semantic Web proposes a vision for a next-generation Web that is computer interpretable [McIlraith et al., 2001]. Automated Web service composition (WSC) is one of many interesting challenges facing the semantic Web. WSC is an example of the more general task of composing processes or component-based programs that serves to provide value-added service. Given computer-interpretable descriptions of the task to be performed, the properties and capabilities of available Web services, and possibly some information about the client or user's specific constraints, automated composition of Web services requires a computer program to automatically select, integrate, and invoke multiple Web services in order to achieve the user-defined objective in accordance with any user, context, or instance-specific constraints.

Automated WSC problem is motivated by the need to improve the efficiency of composing

and integrating services. A number of Business Process Management (BPM) systems exist to help organizations optimize business performance by discovering, managing, composing, and integrating services represented as business processes. With the advent of cloud computing, an increasing number of small- and medium-sized businesses are attempting to blend cloud services from multiple providers. Performing such integration and interoperation manually is costly and time consuming. Further, generating a composition that is customizable with respect to additional constraints is often a challenge.

For the purpose of this thesis, we illustrate concepts in terms of a Travel domain; however, compelling examples exist in sectors such as Banking and Finance, Government, Healthcare and Life Sciences, Insurance, Retail, and Supply Chain Management. Many of these applications exploit extensive internet- or intranet-accessible data and will directly benefit from the work described here. In the Travel domain, one needs to arrange accommodations and various forms of transportation with varying options for their realization. We could additionally have the following soft constraints: *If destination is more than 500 km away, book a flight, otherwise I prefer to rent a car; I prefer to fly with a Star Alliance carrier; I prefer to book cars with Avis, and if not Budget; I prefer to book a Hilton hotel, and if not a Sheraton.* Further hard constraints can be imposed on the composition such as the following: *Never book business or first class flights; Get pre-approval for travel outside the US; Always pay for flights and hotels with your corporate credit card.*

Soft constraints (aka preferences) are a set of properties that define the quality of the composition while hard constraints (e.g., policies) are properties that need to be enforced by the composition. Preferences differ from hard constraints because their satisfaction is not mandatory, but desirable.

Hard constraints are a useful way of enforcing business rules and policies. Many customers are concerned with enforcement of hard constraints, often in the form of corporate policies and/or government regulations. *Policies* or *regulations* are a set of constraints imposed by an authority that define an acceptable behaviour or characteristic of an agent, person, or an organization. Policies or regulations hence are a set of constraints imposed on the composition that define an acceptable composition. If commerce is being performed across multiple governmental jurisdictions, there may be a need to ensure that laws and regulations pertaining to commerce are enforced appropriately. A company may wish to ensure that all transactions comply with company policies. For example, they might impose on their employees when traveling, to always use their corporate credit card for their travel expenses.

Software that is developed for use by a particular corporation or jurisdiction will have the

enforcement of such regulations built in. For Web services that are published for use by the masses this is not the case, and the onus is often on the customer to ensure that regulations are enforced when the composition is constructed from multiple service providers. For inter-jurisdictional or international business, different regulations may apply to different aspects of the composition. Hence, customizing the composition of Web service by imposing hard constraints and, to that end, providing a mechanism for generating compositions that adhere to such constraints, is a significant problem that we address in this thesis.

We also argue that customization with respect to soft constraints or preferences is an important problem and a critical and missing component of most existing approaches to the WSC problem or component-based compositions. User preferences are key for at least two reasons. First, the space of possible compositions is often under constrained; as such it induces an (often large) family of solutions. E.g., consider a user's objective to book a flight from Toronto to Atlanta on November 5. This alone could generate tens of solutions. Nevertheless, some solutions are more desirable, based on the user's preferences. User preferences enable a user to specify properties of solutions that make them more or less desirable. The composition system can use these to generate preferred solutions.

A second reason why user preferences are critical is with respect to *how* the composition is performed. A key component of Web service composition is the selection of specific services used to realize the composition. In AI planning, primitive actions (the analogue of services) are selected for composition based on their preconditions and effects, and there is often only one primitive action that realizes a particular effect. For many WSC problems, the task can be realized by a diversity of different services, offering comparable, but not identical services. Also unknown at the outset is the data that serves as choice points in a WSC – the availability of goods, their properties and pricing, etc.

Similar to actions, services are associated with their preconditions and effects. In addition, they are also associated with their inputs and outputs. These four properties (inputs, outputs, preconditions and effects) are called *functional properties* of services and can be found in semantic Web process model descriptions such as OWL-S [Martin et al., 2007]. The service profile associated with services in semantic Web representations such as OWL-S allows description of their *non-functional properties*. These properties are often used to describe the features of the service so as to facilitate their discovery and selection. The following is a list of non-functional example properties: service name, service author, service language, service trust, service subject, service reliability, and service cost.

Preferences enable specification of both functional and non-functional properties of ser-

vices. In the case of airline ticket booking, a book-flight service requires as its input specific times/dates of the flight and generates a seat number as its output indicating whether it is an aisle or a window, and as its effect the flight is booked. For this domain the preferences over functional properties could be whether the user prefers window-seated flights or prefers flying over weekends, the non-functional properties could be whether the user prefers trusted services or services that offer lower costs. By integrating preferences into the composition problem, preferences over services (the *how*) can be specified and considered alongside preferences over the solutions (the *what*).

1.2 Thesis Statement

In this thesis, we present an approach to the WSC problem with a particular focus on the *customization* of compositions with respect to both soft and hard constraints. We claim that there is a correspondence between generating a customized composition of Web services and non-classical AI planning where the objective of the planning problem is specified as a form of control knowledge, such as a workflow or template, together with a set of constraints to be optimized or enforced. We further claim that techniques in (preference-based) planning can provide a computational basis for the development of effective, state-of-the-art techniques for generating customized compositions of Web services. To that end, we formally characterize the customization of the WSC problem in such a way that allows us to view it as a preference-based planning problem, where actions (services, service parameters, and/or data) are selected to produce compositions that are of high quality. Our approach supports customization, optimization, and enforcement of constraints all within one reasoning framework by reflecting and advancing state-of-the-art techniques for preference-based planning.

1.3 Approach

A popular approach to WSC is to view it as an Artificial Intelligence (AI) planning task (e.g., [McIlraith et al., 2001, McDermott, 2002, McIlraith and Son, 2002]). Given an initial state, a set of actions, their precondition and effects, and a goal description, the AI planning task is to find a plan or a sequence of actions that achieves the goal. The WSC problem can be characterized as a planning task by specifying services as primitive and complex actions with precondition and effects and the user-defined objective as a composition template. The composition

is then the plan generated by the planner.

A number of researchers have advocated using AI planning techniques to address the WSC problem including planners that are based on model checking (e.g., [Traverso and Pistore, 2004]) and planners that use a template or workflow to ease and guide the task of composition (e.g., [McIlraith and Son, 2002, Sirin et al., 2005b]). A template-based composition is compelling for many applications in domains including e-government (e.g., [Chun et al., 2004]), e-science (e.g., [Cheung and Gil, 2007]), and Grid computing (e.g., [Gil et al., 2004]).

Exploiting recent advances in classical planning can provide great computational advantage, but the conception of the WSC problem we consider in this thesis goes far beyond a classical planning problem. In particular, unlike classical planning, we do not have a final state goal. Instead in the WSC problem we typically have a specification of a basic behaviour we wish to achieve – perhaps specified as a workflow or a template. This basic behaviour specification is often further augmented with hard constraints (e.g., policies or regulations) that need to be enforced. Further, while in classical planning, plan length is the only measure of quality, in the WSC problem, as in preference-based planning, compositions are plentiful and it is the generation of preferred compositions with respect to complex, temporally extended measures of quality or preference that we must realize. Another differentiating property of the WSC problem is that it can be data intensive resulting in planning domains with tens of thousands of ground actions, where each of which itself can be a program with non-determinism and intermediate state. Finally, unlike in classical planning, not all information required to generate the composition can be gathered in advance (i.e., we may be faced with incomplete information). Hence, we need to deliberate about what information to gather and when.

A composition template or composition workflow provides high-level guidance on how to perform a task while leaving enough flexibility and non-determinism for different possible realizations based upon the specific problem instance, context- and user-specific preferences, policies, and regulations, and upon the available services. For many WSC problems, the task can be realized by a diversity of different services, offering comparable, but not identical services. Also unknown at the outset is the data that serves as choice points in a WSC – the availability of goods, their properties and pricing, etc. A composition template streamlines the generation of a problem, and customer-specific WSC, while enabling the individual customer to *customize* the composition with respect to their soft constraints and hard constraints (e.g., constraints imposed by the corporation they work for, the laws of the countries in which they are doing business).

A composition template can be represented in a variety of different ways. In this thesis, we consider two different representations. In our first approach, we exploited the agent programming language Golog or its variant ConGolog (e.g., [Reiter, 2001]) to represent templates as generic procedures. (Con)Golog is a high-level logic programming language that augments the situation calculus with a set of Algol-inspired extralogical constructs for assembling primitive and complex actions into complex actions and programs. Its syntax contains conventional programming language constructs such as if-then-else and while-loops and allows for nondeterminism both in terms of parameterization of data and component selection. Among the appeals of ConGolog is its procedural specification which is very much like a textual specification of a flexible workflow; its ability to treat complex actions as first-class objects in the language; and the fact that it is first-order, allowing the easy specification of data (a pervasive element in Web services) as a parameterization of actions.

The second form of composition template representation that we have exploited is the use of Hierarchical Task Networks (HTNs) (e.g., [Ghallab et al., 2004]). HTN planning is a popular and widely used planning paradigm, and many domain-independent HTN planning systems exist (e.g., Nonlin [Tate, 1977], SIPE-2 [Wilkins, 1988], O-PLAN [Currie and Tate, 1991], UMCP [Erol et al., 1994], **SHOP2** [Nau et al., 2003]). Similar to Golog, HTNs provide useful control knowledge — advice on how to perform a composition – by specifying a template as a *task network* – a set of tasks that need to be performed and that can be repeatedly decomposed in various ways (using so-called *methods*) into finer grained subtasks, eventually culminating in primitive actions to be performed. This control knowledge can significantly reduce the search space for a plan while also ensuring that plans follow one of the stipulated courses of action. In addition to this hierarchical structure, ordering constraints can be imposed.

Returning to our Travel domain, the task of arranging travel can be decomposed into arranging transportation, accommodations, and local transportation. Each of these tasks can successively be decomposed into other subtasks based on alternative modes of transportation and accommodation, eventually reducing to primitive actions that can be executed in the world. Further constraints can be imposed to restrict decompositions. In planning, this decomposition and search is performed by an HTN planner. The merit of HTNs is the intuitive nature of task hierarchies and the extensive computational machinery. Interestingly, HTNs can be characterized as a special case of ConGolog programs [Gabaldon, 2002] and as such, while these different composition templates have their merits in terms of expressiveness, in terms of their theoretical analysis we have a unified formalism that we can and have employed.

We exploit preference-based planning together with the Golog or HTN specification of

composition templates to customize plans and optimize for high-quality plans. Preference-based planning augments a planning problem with a specification of properties that constitute a high-quality plan. For example, if one were generating an air travel plan, a high-quality plan might be one that minimizes cost, uses only direct flights, and flies with a preferred carrier. Preference-based planning attempts to optimize the satisfaction of these preferences while achieving the stipulated goals of the plan. To develop a preference-based HTN planner, we must develop a specification language that references HTN constructs, and a planning algorithm that computes a preferred plan while respecting the problem specification.

1.4 Challenges and Contributions

The general challenge we face in this thesis is to investigate principled techniques for composing Web services that support user customization. This manifests itself in a number of specific research challenges that we identify and address in this thesis.

In this thesis: we characterize the WSC problem with customization as a non-classical planning problem where the objective of the planning problem is specified as a composition template (either in HTN or Golog), together with a set of constraints to be optimized or enforced; exploit and advance preference specification languages and preference-based planning; develop algorithms tailored to the WSC problem; prove formal properties of these algorithms; implement (proof-of-concept) systems, and evaluate these systems experimentally. While our research has been motivated by Web services, the theory and techniques we have developed are amenable to analogous problems in such diverse sectors as multi-agent systems, business process modeling, stream processing, component software composition, and social and computational behaviour modeling and verification.

Challenge 1: Characterize the WSC Problem with Customization

The first and the foundational challenge of this thesis is defining a formal notion of the WSC problem with customization. Our characterization should articulate a notion of quality (optimizing) with respect to user, context, or instance-specific preferences (soft constraints) as well as a notion of enforcement with respect to hard constraints. Moreover, since we can employ different representations for specifying the constraints and the defined objective of the WSC problem, a unified formalism that can be adapted for different approaches is ideal.

Hence, our first contribution is to characterize the WSC problem with customization, where

the objective of the planning problem is represented in some form of control knowledge (either in Golog or HTNs) together with a set of constraints that need to be optimized or enforced. This characterization enables us to generate the customized composition of Web services through non-classical planning. To that end we explore the use of preference-based planning, where the customized composition (with respect to both soft and hard constraints) is the preferred plan generated by a preference-based planner that adheres to the hard constraints.

Challenge 2: Specify the Soft and Hard Constraints

Specifying the soft constraints (aka preferences) and/or hard constraints (e.g., policies) for the WSC problem is another important challenge addressed in this thesis. To that end we need to first design a set of desirable criteria that our specification languages can be evaluated against. We then need to explore the existing languages and possibly propose new specification languages that meet our set of desirable criteria. Similarly, we need to propose/employ a language for specifying hard constraints that can easily facilitate pruning strategies during the planning phase.

The second contribution of this thesis is to design a set of desirable criteria, evaluate the existing specification languages with respect to this set, and extend the existing specification languages to meet our set of desirable criteria. The result of this contribution was presented in [Sohrabi et al., 2006, Sohrabi and McIlraith, 2008, Sohrabi et al., 2009, Sohrabi and McIlraith, 2009]. We evaluate an existing preference languages proposed by Bienvenu et al. called $\mathcal{LPP}$ with respect to our set of desirable criteria and discuss how we can use $\mathcal{LPP}$ to express service selection preferences. We also extend $\mathcal{LPP}$ (propose a new language called $\mathcal{LPH}$) to be able to specify HTN-specific preferences. Among the HTN-specific properties that we add to our language is the ability to express preferences over how tasks in our HTN are decomposed into subtasks, preferences over the parameterizations of decomposed tasks, and a variety of temporal and nontemporal preferences over the task networks themselves.

Building on the development of $\mathcal{LPH}$ we extend the Planning Domain Definition Language, PDDL3 [Gerevini and Long, 2005, Gerevini et al., 2009], with HTN-specific preference constructs. PDDL3 preferences are highly expressive, however they are solely *state centric*, identifying preferred states along the plan trajectory. To develop a preference language for HTN we add *action-centric* constructs to PDDL3 that can express preferences over the occurrence of primitive actions within the plan trajectory, as well as expressing preferences over complex actions (tasks) and how they decompose into primitive actions. Moreover, we discuss

how we can specify preferences over service and data selection using our extension. For example, we are able to express preferences over which sets of subtasks are preferred in realizing a task (e.g., *When booking inter-city transportation, I prefer to book a flight*) and preferred parameters to use when choosing a set of subtasks to realize a task (e.g., *I prefer to book a flight with United*). In addition, we can express our preferences over the selection of trustworthy or more reliable services.

To specify hard constraints we need a specification language that can be easily integrated into our system and one that easily facilitates pruning strategies during the planning phase, similar to how temporal formulae provide powerful pruning in TLPlan [Bacchus and Kabanza, 2000] and TALPlan [Kvarnström and Doherty, 2000]. Hence, we specify our hard constraints in a subset of Linear Temporal Logic (LTL) that deals with safety or maintenance properties, considering for the most part the *never* and *always* modalities. This restriction was a design decision rather than a limitation in our ability to deal with arbitrary LTL formulae. With some sophisticated processing, we could extend our implementation to handle arbitrary LTL.

Challenge 3: Compute Optimized Compositions

We show in our first challenge that there is a correspondence between generating customized compositions of Web services and non-classical planning problem. Our third challenge is how to compute optimized compositions by reflecting and advancing state-of-the-art techniques for HTN planning with preferences. To that end we need to develop heuristics and algorithms that enable effective search for an optimal composition; a composition that has the best quality. Furthermore, we need to analyze and prove properties of these algorithms including correctness and optimality, implement (possibly proof-of-concept) systems that implement our approaches, and evaluate the implemented systems to show the applicability of our proposed approach.

Heuristic-guided search is an effective method for efficient plan generation (e.g., [Bonet and Geffner, 2001, Hoffmann and Nebel, 2001]), and many heuristic-based planners exists (e.g., FF [Hoffmann and Nebel, 2001], LPG [Gerevini et al., 2003], Fast Downward [Helmert, 2006], LAMA [Richter et al., 2008]). Our challenge is to find a heuristic that gives guidance towards optimal solutions without exhaustively searching the search space. We can use either an admissible or inadmissible set of heuristics. An admissible heuristic is a heuristic that never overestimates the cost of reaching the goal. If an admissible heuristic is used in an A*-like algorithm, then the first plan found would be an optimal plan. However, finding such

a plan in practice may not be feasible when the search space is large. Hence, we may instead consider using inadmissible heuristics with the hope of finding a good-quality plan quickly and finding a condition under which we can guarantee optimality despite the use of inadmissible heuristics.

Hence, the third contribution of this thesis is to propose algorithms that integrate preference-based reasoning, exploiting and advancing state-of-the-art preference-based heuristic search, proving properties of these algorithms, and implementing and evaluating systems that show the applicability of our proposed approach. The results of this contribution were presented in [Sohrabi et al., 2006, Sohrabi and McIlraith, 2008, Sohrabi et al., 2009, Sohrabi and McIlraith, 2009].

More specifically, we provide an algorithm that integrates preference-based reasoning into Golog, and prove the soundness of our approach and the optimality of our compositions with respect to the user's preferences. We provide a working implementation of our algorithm in a proof-of-concept system, **GOLOGPREF**, that can be used to select an optimal solution from among families of solutions that achieve the user's stated objective. A notable side effect of this work is the seamless integration of Web service selection with the composition process.

To compute optimal plans when the composition template is specified in HTN, we first need to develop effective techniques for HTN planning with preferences. To that end, we propose two HTN planners **HTNPLAN** and **HTNPLAN-P**. In **HTNPLAN**, we propose an approach based on forward-chaining heuristic search. Key to our approach is a means of evaluating the (partial) satisfaction of preferences during HTN plan generation based on progression. The optimistic evaluation of preferences yields an admissible evaluation function which we use to guide search. We implemented our planner, **HTNPLAN**, as an extension to the **SHOP2** HTN planner. Our empirical evaluation demonstrates the effectiveness of **HTNPLAN** heuristics in finding optimal plans. We provide a semantics for our preference language in the situation calculus [Reiter, 2001] and appeal to this semantics to prove the soundness and optimality of our planner with respect to the plans it generates.

To compute preferred plans (exploring inadmissible heuristics) when the composition template is specified in HTN, we propose a best-first, incremental search in the plan search space induced by the HTN initial task network. The search is performed in a series of *episodes*, each of which returns a sequence of ground primitive operators (i.e., a plan that satisfies the initial task network). During each episode, the search performs branch-and-bound pruning—a search node is pruned from the search space, if we can prove that it will not lead to a plan that is better than the one found in the previous episode. In the first episode no pruning is performed. In

each episode, search is guided by our *inadmissible heuristics*, designed specifically to guide the search quickly to a good decomposition. We show that under certain conditions, our heuristics can be used to prune suboptimal plan prefixes from the search space enabling the planner to return plans of increasing quality, culminating in an optimal plan. The experimental evaluations of our planner shows that our HTN preference-based planner, **HTNPLAN-P**, generates plans that, in all but a few cases, equal or exceed the best preference-based planners in terms of plan quality. As such, our result shows that our approach is viable and promising.

Given our advancements to HTN planning with preferences, we go back to address the WSC composition with customization problem. To that end, we build our system **HTNWSC-P** on top of our HTN planner with preferences **HTNPLAN-P**. We use pruning to eliminate those compositions that violate hard constraints (policies or regulations). That is, we ensure regulations, or policies are enforced by simply pruning partial plans that do not adhere to them. (I.e., upon violation of regulations, we immediately prune that part of the search space.) This allows enforcement of regulations or policies during composition construction time. Hence, our composition framework can now simultaneously optimize, at run time, the selection of services based on their functional and non-functional properties, while enforcing stated hard constraints. Experimental evaluation on our system, **HTNWSC-P**, shows that our approach can be scaled as we increase the number of preferences and the number of services.

Challenge 4: Execute and Optimize

Our fourth challenge is how to perform data-dependent optimization following online information gathering in order to gather the necessary information needed to produce high-quality compositions in the absence of complete information. Many planning-based characterizations of the WSC problem make an assumption that there is complete information about the initial state. This assumption is often violated in many real-world settings; it is impractical or impossible to have all the information necessary to generate a composition prior to the commencement of the search for a composition. A more compelling solution is to instead gather information as it becomes necessary in the generation and optimization of the composition.

Optimization requires considering all alternatives, at least implicitly. However, given the large volume of information available on the Web, evaluating the search space effectively is a challenging problem that has not been addressed in previous work. For example, assume there are three tasks, *rent-car*, *book-flight*, and *book-hotel* containing i, j, and k data items, respectively, and we need to compose these three tasks. In the worst case, the planner will

need to evaluate $i * j * k$ different car-flight-hotel combinations in order to identify an optimal combination. However, if the choice of hotel, car, and flight can be made independently of each other, then the search space is worst case $k + i + j$. More generally, if we identify the subset of the data that is relevant to the optimization of the composition, we can localize its optimization and significantly streamline our search.

Hence, our fourth contribution is to propose a way to address the information-gathering component of the WSC problem with customization. The need to actually execute services to gather data, as well as the potential size and nature of the resultant optimization problem truly distinguishes our approach to the WSC problem from previous work on preference-based planning. To this end we propose a notion of middle-ground execution system for the WSC problem with customization that interleaves online information gathering with offline search as deemed necessary. Our approach executes the information-gathering services as needed and only simulates the execution of the world-altering services in order to determine a customized composition of Web services for later execution.

By exploiting the structure in the preference specification and domain we propose a notion of what we call *localized data optimization* in which the optimization task can be decomposed into smaller, local optimization problems, while preserving global optimality. This notion comes from the observation that in many composition scenarios that involve preferences, most of the search time is spent on resolving the optimization that relates to the data (which flight, which car, which hotel). We propose to further improve the search by performing optimization of data choices locally, whenever possible, while still guaranteeing that the choice selected does not eliminate the globally optimal solution.

We modify our search algorithm for **HTNWSC-P** to perform information gathering as well as optimization. We prove the correctness of our approach and also identify a case where we could prove the optimality of resulting compositions. We showed that our approach to data optimization can greatly improve both the quality of compositions and the speed with which they are generated. This contribution was presented in [Sohrabi and McIlraith, 2010].

Challenge 5: Explore Applicability Beyond WSC

While the techniques we developed are motivated by the WSC problem, in our final challenge we explore the possibility of applying these techniques to analogous problems. To that end, we must address possible challenges that present themselves in these related, but different problems.

Hence, our final contribution is an investigation of how to use and adapt our framework to address two applications: requirements engineering and stream processing. We overview and evaluate the applicability of exploiting the techniques developed in this thesis in order to address these problems. The result of the requirements engineering application appears in [Liaskos et al., 2010, Liaskos et al., 2011]. Studying the stream processing application was conducted in collaboration with IBM T.J. Watson Research Center. This work involved addressing several unique and interesting challenges that are present when working with real large-scale application domains. The result of this research was presented in [Sohrabi et al., 2012]. Some of the key contributions of this work are proposing the use of HTN planning with preferences to address modeling, computing, and optimizing composition flows in the stream processing application; performing extensive experimentation with real-world patterns using IBM InfoSphere Streams `http://www-01.ibm.com/software/data/infosphere/streams/`; and developing an enhanced lookahead heuristic and showing that it improves the performance of our HTN planner by 65% on average.

1.5 Organization of this Thesis

This thesis is organized as follows:

- In Chapter 2 we describe some of the existing approaches that address the WSC problem. We begin this chapter with a discussion of the scope of the approaches we considered by describing several common criteria that we use to differentiate the different approaches. We also discuss a general framework for the WSC problem and update this framework for each approach that we discuss.

- In Chapter 3 we provide formal characterization of the WSC problem with customization. We characterize the WSC problem with customization as a non-classical planning problem where the objective of the planning problem is not represented as a final state goal, but rather as an objective in a workflow or a template (specified as HTNs or Golog) together with a set of constraints that need to be optimized or enforced. To that end, we articulate a notion of quality (optimizing) with respect to user, context, or instance-specific preferences (soft constraints) as well as a notion of enforcement with respect to hard constraints (e.g., policies or regulations).

- In Chapter 4 we describe a set of desirable criteria for constraints specification, explore the use of existing specification languages, and propose new specification languages that meet our set of desirable criteria. To that end we describe the syntax and the semantics of the preference languages we explore and propose. We also discuss how we specify the hard constraints.

- In Chapter 5 we address the problem of how to exploit the rich specification of the composition template (desired behaviour), specification of constraints (both soft and hard) described in Chapter 4 in order to generate high-quality compositions with enforced hard constraints. To that end, we describe our proposed algorithms, prove formal properties of these algorithms, present our (proof-of-concept) systems, and evaluate these systems experimentally.

- In Chapter 6 we discuss how we address the information-gathering component of the WSC problem with customization as well as optimization. We discuss how we can modify our existing framework to incorporate the proposed features. We also discuss our empirical evaluation and results.

- As mentioned earlier, although our research has been motivated by Web services, the theory and techniques we have developed go beyond WSC, and as such are amenable to many analogous problems. In Chapter 7 we overview two applications, one in requirements engineering and one in stream processing. We discuss how we can exploit the techniques and languages developed in this thesis to address these problems. In particular, we will show how to address the problem of automated composition of flow-based applications using HTN planning with preferences.

- Finally, Chapter 8 summarizes the contributions of this thesis and presents a number of possible future research directions.

Chapter 2

Background

2.1 Introduction

In this chapter, we describe some of the existing approaches that address the composition problem of Web services. We will discuss both planning approaches and non-planning approaches to this problem. We begin by describing several common criteria that we use to differentiate the different approaches. We will also provide a general framework that captures the different high-level components and update this framework for each approach that we discuss.

There are several different problems that have to be considered in the end-to-end solution to the WSC problem. In particular, one needs to consider the problem of how to describe, discover, invoke, and execute the services. In addition, one needs to address both the high-level representation of message-exchange (communication) and data-flow among services, and the low-level details of the actual establishment of connection and communication to a service. There have been a lot of recent advances in addressing the above important problems, but discussing many of these is out of the scope of this thesis. In particular, there are several languages that have been proposed with the intention to provide a standard platform for Web service discovery, invocation, execution, and composition. The following are some of the most important languages that fall under the Service-Oriented Architecture (SOA) framework: Web Service Description Language (WSDL) [Chinnici and et al, 2001], Simple Object Access Protocol (SOAP) [Box and et al, 2003], and Universal Description, Discovery, and Integration (UDDI). WSDL is an XML-based language used to describe the Web services, SOAP is a message protocol used to establish connection to a service, and UDDI is a XML-based platform-independent registry that lists the services. In addition to those, the semantic Web community have proposed several ontologies with a well-defined semantics such as Semantic Markup

for Web Services (OWL-S) [Martin et al., 2007], Semantic Web Service Ontology (SWSO) [Battle et al., 2005], and Web Service Modeling Language (WSMO) [Bruijn et al., 2006], to enable reasoning about the Web services and their interactions with each other.

In addition, *orchestration* and *choreography* are two common terms used to describe the kind of interactions or communications that takes place between services [Peltz, 2003]. In choreography each party involved describes their own part; however, messages among multiple parties are tracked by the choreography. WS-CDL (Web Services Choreography Description Language) is an example of Web service choreography. Web Service orchestration is a less collaborative way of interaction. An orchestrator can activate, stop and resume any of the available services. Any message exchange sequence is controlled by the orchestration designer. Business Process Execution Language for Web Services (BPEL4WS) is an example of an orchestration [Peltz, 2003]. Discussing many of the languages and ontologies mentioned above as well as the details of execution and lower-level message exchanges is outside the scope of this thesis.

As discussed in Chapter 1, a popular approach to the WSC problem is to view it as an Artificial Intelligence (AI) planning task. Given an initial state, a set of actions, their precondition and effects, and a goal description, the AI planning task is to find a plan or a sequence of actions that achieves the goal. The WSC problem can be characterized as a planning problem by specifying services as primitive and complex actions with preconditions and effects and the desired behaviour or the objective as an (often temporally extended) goal description. The composition is then the plan generated by the planner. A number of researchers advocate using AI planning techniques to address the WSC problem. In particular, we will discuss the approach that uses Golog (e.g., [McIlraith and Son, 2002]) and Hierarchical Task Networks (HTNs) (e.g., [Sirin et al., 2005b]). We also discuss several different approaches, including those that are based on planning as model checking (e.g., [Traverso and Pistore, 2004]), rule-based planning (e.g., [Ponnekanti and Fox, 2002]), and theorem proving (e.g., [Waldinger, 2001]). We will also discuss how the work presented in this thesis (e.g., [Sohrabi et al., 2006, Sohrabi and McIlraith, 2009]) is different from the other planning based approaches.

In addition to the AI planning approaches to the WSC problem there are a number of non-AI planning approaches that we overview. In particular, we overview approaches that are based on workflow modeling of the composite service (e.g., [Schuster et al., 2000, Casati et al., 2000]) and several papers that also take a workflow or template-based approach, but have their focus explicitly on the optimization of Quality of Service (QoS) or the service selection problem

(e.g., [Zeng et al., 2003]). We also overview two automata-based approaches, the Petri Nets (e.g., [Hamadi and Benatallah, 2003, Narayanan and McIlraith, 2002]) and the Roman Model [Calvanese et al., 2008].

In order to help better differentiate the different approaches, we plan to further discuss each approach based on the following differentiating criteria.

Level of Automation: Composition of Web services can be achieved through a range of automation. At one end of the spectrum, the selection and coupling of Web services can be performed manually using a workflow or tools such as Business Process Execution Language for Web Services (BPEL4WS). Unfortunately, such compositions are brittle, requiring specification of the services to be coupled a priori and precluding most, if not all, user customization and generation of preferred compositions. At the other end of the spectrum, the WSC problem can be fully automated using, for example, a classical AI planning technique. This approach also has its drawbacks. In general, the search space for a composition is huge because of the large number of available services, which grow far larger with grounding for data creating scalability and expressivity problems [Srivastava and Koehler, 2003, Hoffmann et al., 2008]. While both extremes have their drawbacks, a reasonable middle ground is the "guided-automation" approach where the composition is guided through a composition template. The composition template provides the high-level knowledge of how to achieve the desired behaviour; hence, limits the ways in which services can be selected, while leaving enough flexibility for different possible realizations of the programs within the template, based upon the needs of the specific problem, the preferences of the customer, and the available services. Golog (e.g., [McIlraith and Son, 2002] and HTNs (e.g., [Sirin et al., 2005b]) are two of many ways to represent a composition template that we discuss in this thesis.

Optimality: In many WSC setting, akin to preference-based planning, while compositions are plentiful, it is the generation of preferred compositions with respect to some measures of quality or preference that is hard to realize. Regardless of the type of quality, preferences are a critical and often a missing component of most existing approaches to the WSC problem. The different approaches that handle optimization with respect to some notion of preferences (e.g., [Lin et al., 2008, Lécué, 2009]) define a way of how to specify preferences and also how to compute compositions that are optimal with respect to their quality measure.

Online vs. Offline: Many WSC approaches assume that all the information required to generate the composition is on hand at the outset, and as such, composition is done offline followed by subsequent execution of the composition, perhaps in association with execution monitoring. However, this is not realistic in many settings. Consider the task of travel planning or any other multi-step purchasing process on the Web. A good part of the compositions for these domains involves data gathering, followed by generation of an optimized composition with respect to that data and other criteria. Indeed many of the choice points relating to the composition require data acquired at execution time (i.e., online). To address this, most current WSC systems will acquire all the information required for the composition prior to initiating composition generation. This can result in a lot of unnecessary data access. Further, it results in an enormous search space. While this space may still be manageable for computing *a* composition, to compute *an optimal* composition, and to guarantee optimality, the entire search space must be searched, at least implicitly. This has the effect that most data-intensive WSC tasks that involve optimization of data (like picking preferred flights) will not scale using the conventional techniques. The problem of gathering information during composition has been examined in several research papers (e.g., [McIlraith and Son, 2002, Kuter et al., 2004]). McIlraith and Son in [McIlraith and Son, 2002] describe a middle-ground interpreter that collects relevant information, but only simulates the effects of world-altering actions. Kuter et al. in [Kuter et al., 2004] take a similar approach but their work focuses on dealing with services that do not return a result (if any) immediately. The techniques proposed in this thesis (Chapter 5) is among the few that attempts to balance the trade-off between offline composition and online information gathering with a view to producing high-quality compositions.

The Context (i.e., Semantic Web): While today the Web is designed for human interpretation and use, the semantic Web proposes a vision for a next-generation Web that is computer interpretable [McIlraith et al., 2001]. WSC is one of many interesting challenges facing the semantic Web. Several of the papers discussed in this report are in service of the semantic Web but we discuss both semantic Web and non-semantic Web related papers.

Stateful vs. Stateless: While many approaches keep track of the state of the world, some also consider services that keep track of their states. The state of a service keeps track of the history of this service in terms of its previous invocation, communications, and interactions. Much of the classical AI planning approaches (e.g., [McDermott, 2002]) consider stateless services, that is the atomic services that do not depend on the history or state of interactions, for exam-

ple, on the previous inputs to the service. Other approaches, including the approaches that use non-classical AI planning techniques (e.g., [McIlraith and Son, 2002, Sirin et al., 2005b]) or the model checking approaches (e.g., [Bertoli et al., 2010]), consider process-oriented services in which services are stateful. That is, the services are able to establish complex multi-phase interactions, possibly with the user or the client. Generally, the approaches that consider stateful services are more difficult, since services cannot be considered as atomic. Instead they must be represented as stateful processes that realize interaction protocols which may involve different sequential, conditional, and iterative steps. An example of a stateful service is a flight booking service that requires "a sequence of different operations including an authentication, a submission of a specific request for a flight, the possibility to submit iteratively different requests, acceptance (or refusal) of the offer, and finally, a payment procedure" [Bertoli et al., 2010].

2.2 Web Service Composition Framework

In this section, we describe a general framework for the Web service composition problem. In this framework we deliberately keep the different components high level, and hence, will not consider a particular language or algorithm used in the composition. The main purpose of this framework is to give a basic definition of the WSC problem that can be adapted by different approaches. The following definition is a generalized definition of WSC problem that originally appeared in [Sohrabi et al., 2006]. We will give a formal definition of the WSC problem we consider in this thesis in Chapter 3.

Definition 2.1 (Web Service Composition (WSC)) *A WSC problem is described as a 6-tuple* $(S_{init}, \mathcal{D}, O, \delta, \phi_{hard}, \phi_{soft})$ *where:*

- S_{init} *is the description of the initial state,*
- $\mathcal{D}$ *is a theory describing functional properties of the Web services,*
- O *is a theory describing the non-functional properties of the Web services,*
- δ *is a specification of the desired behaviour or objective,*
- ϕ_{hard} *is a specification of hard constraints (e.g., policies, regulations), and*
- ϕ_{soft} *is a specification of soft constraints (e.g., user, context, or instance-specific preferences).*

Web service composition determines a sequence of services (actions) whose execution starting from S_{init} *meets the objective* δ *while enforcing the* ϕ_{hard}*, and optimizing for a composition that satisfy* ϕ_{soft}*.*

Note, functional properties of a service include input, output, precondition, and effect of a service, while the non-functional properties of a service may include service trust, reliability, subject, cost, and language. Also, some approaches distinguish between *world-altering* services and *information-gathering* services. World-altering services are those that once executed have an effect on the world or change the state of the world. Information-gathering services, on the other hand, only provide an output or information, and their execution would not have an effect on the state of the world.

Several components in Definition 2.1 such as ϕ_{hard} or ϕ_{soft} will not be adapted in many of the WSC approaches we consider. Hence, we consider these optional arguments of the problem. In addition, in some approaches there is no need to explicitly specify the initial state as an extra argument to the problem because the initial state may be specified within $\mathcal{D}$ or δ. However, what many approaches have in common is that they will define the composition requirement or the objective, δ, discuss a language for specifying the functional and non-functional properties of the services, and discuss the way they find a (high-quality) composition, possibly by providing an algorithm for their computations (i.e., realizing what it means to meet the objective, enforce the hard constraints and optimize for a high-quality composition). Many of these elements of WSC problem are those we focus on when describing the different approaches.

2.3 Planning Approaches to the WSC Problem

In this section, we review some of the AI planning approaches to the WSC problem. We consider both classical and non-classical AI planning approaches to WSC. Exploiting recent theoretical and computational advances in AI planning can provide a great advantage for addressing the WSC problem. This is the primary reason for the popularity of the AI planning approaches to WSC. However, a classical AI planning approach is generally not sufficient to address the WSC problem, therefore, many non-classical approaches have emerged in the past few years that we will overview here.

2.3.1 Classical Planning

The classical AI planning approaches to the WSC problem generally translate OWL-S process models into internal representations such as Planning Domain Definition Language (PDDL) [McDermott, 1998] amenable to AI planning (e.g., [Klusch et al., 2005]). PDDL

is a popular and widely used planning input for many state-of-the-art planners. OWL-S [Martin et al., 2007] is a Web ontology [Horrocks et al., 2003] for Web services with a view to support automated discovery, enactment and composition of Web services. OWL-S will be discussed in more detail in Chapter 3.

Many classical planning approaches that we overview here consider an adapted version of the general framework or Definition 2.1 in which $\mathcal{D}$ and O are the description of functional and non-functional properties of a set of Web services mapped to some AI planning description language such as PDDL, and the objective is the goal description (usually a set of literals that need to hold in the final state. WSC determines a sequence of services (actions) whose execution starting from the initial state reaches a goal state. Hence, the WSC problem can be viewed as a classical planning problem by considering that the set of services are mapped to actions with precondition and effects, and the objective is mapped to a goal description. However, there are general assumptions that are made when considering this mapping. In particular, in this mapping it is assumed that each Web service can be specified by its precondition and effect in the planning context. Hence, specifying information-gathering services (those that have no effect), may not be possible. Furthermore, it is assumed that the desired functionality or objective can always be expressed as a final state goal condition. This is often not ideal, as goals can generally be temporally extended (i.e., not only over a final state). Furthermore, in WSC typically we have a specification of a basic behaviour we wish to achieve, specified as a workflow or a template, and a final state goal cannot capture this.

A classical AI planning approach to the WSC problem is generally a fully automated, offline approach that is optimal with respect to only plan length (shortest plans are preferred), and an approach that considers services that are stateless. In addition, a classical AI planner is incapable of handling rich, complex hard or soft constraints; hence policies, regulations, user, context, or instance-specific preferences are not handled in this approach. Furthermore, it is generally assumed that the initial state is complete (i.e., all information is given in advance), so planning is done offline with no extra online information-gathering step. This may create a scalability problem as WSC problems are generally data intensive resulting in planning domains with tens of thousands of ground actions.

Next, we consider several specific planning approaches to the WSC problem.

McDermott presents an approach based on PDDL to address the task of WSC and shows how a classical goal-regression planner can be extended to create conditional plans as needed [McDermott, 2002]. A regression-based approach starts from a goal rather than an initial state and searches backward until the initial state holds. McDermott argues that using a regression

planner is suited for the WSC problem and, in addition, he can relax the closed-world assumption normally made for classical planning in his formalism. In particular, he introduces a new type of knowledge called a *value of an action* which persists and can be given as input to the later steps of the planning phase. By using *value of an action* the planner is able to pass information from one plan step to another. For example, a *send message* action may generate a value *message id* that can be used in later communications to indicate which message it is referring to. He further proposes an approach that formalizes the *unknown*, using what he calls *learnable terms*. This feature enables differentiation of the information providing and world-altering services.

Another planning approach that attempts to address the incomplete initial state is the work by Hoffmann et al. (e.g., [Hoffmann et al., 2007]). This approach attempts to address the ramification problem (indirect effect of an action) that arises when WSC systems need to deal with background ontologies or derived information. Incorporating background ontologies (theory) into a planner is a computationally hard problem. This approach addresses the ramification problem by identifying a special case called *forward effects*, in order to limit the effect of an action. They argue that this special case is easier to deal with and yet it is relevant for many WSC scenarios. Furthermore, they provide a compilation into conformant planning, and therefore enabled the use of state-of-the-art conformant planner for the WSC problem.

Another notable work is the work by Klusch et al. [Klusch et al., 2005]. Klusch et al. present a PDDL-based approach to the WSC problem where the services described in OWL-S and the domain ontology described in OWL are mapped to the initial state, the goal description, and actions in PDDL2.2. Furthermore, they propose to represent the output of an information-gathering services by a special predicate "agentHasKnowledgeAbout(X)". Their planner XPlan, which uses the *heuristics* of the well-known FastForward (FF) planner [Hoffmann, 2001], is then used to generate a plan. They tested their approach in an e-Heath application scenario.

Finally, McIlraith and Fadel in [McIlraith and Fadel, 2002] formalize the notion of planning with complex actions by compiling complex, possibly non-deterministic actions into simple classical planning actions. This enables the use of the planning techniques for the WSC problem. Planning is a very active area of research and many fast and efficient planners exists. Hence, in general by having an operator-based planning domain encoding of the problem, one can experiment with state-of-the-art planners in order to address generation of a composition for the WSC problem. However, scalability may become a problem when a planner has to deal with large number of actions and search through a large search space. In order to deal with this

problem, some compact form of representing a domain is needed. We will see in Chapter 5 how using procedural domain control knowledge either in the form of Golog or HTNs, together with heuristic search, can help overcome the scalability problem.

2.3.2 Planning with Procedural Domain Control Knowledge

Many of the tasks performed on the Web or on the intranets are repeated routinely, while the basic steps to achieving these tasks are well understood, at least at an abstract level. These basic steps, often captured in a template or workflow, provide a compelling skeleton of a composition, generally referred to as procedural domain control knowledge. The composition template dictates the ways in which services can be composed, hence restricting the possible search space. The composition template is used to direct and provide high-level guidance on how to perform composition of Web services.

In this section, we overview approaches that extend classical planning to incorporate procedural domain control knowledge either in the form of Golog [Reiter, 2001] or Hierarchical Task Networks (HTNs) [Ghallab et al., 2004]. We overview relevant features of Golog, HTNs and how OWL-S is related to HTN in Chapter 3.

WSC with Procedural Control Knowledge Specified in Golog

In this section, we overview approaches that address the problem of WSC through the use of Golog. In this approach, Web services are viewed as actions in the situation calculus and are described as actions in terms of a situation calculus basic action theory. The details of the basic action theory are discussed in [Reiter, 2001, Narayanan and McIlraith, 2002, McIlraith and Son, 2002].

Golog [Reiter, 2001] is a high-level logic programming language for the specification and execution of complex actions in dynamical domains. It builds on top of the situation calculus by providing Algol-inspired extralogical constructs for assembling primitive situation calculus actions into complex actions (aka *programs*) δ. These complex actions simply serve as constraints upon the situation tree. Among the appeals of Golog are its procedural specification, which is very much like a textual specification of a flexible workflow; its ability to treat complex actions as first-class objects in the language; and the fact that it is first-order, allowing the easy specification of data (a pervasive element in Web services) as a parameterization of actions. The following is an example of a Golog program.

$$\text{bookAirTicket}(\boldsymbol{x})\ ;\ \textbf{if}\ \text{far}\ \textbf{then}\ \text{bookCar}(\boldsymbol{y})\ \textbf{else}\ \text{bookTaxi}(\boldsymbol{y})\ \textbf{endIf}$$

McIlraith et al. [McIlraith et al., 2001, McIlraith and Son, 2002] adapt and extend Golog for the task of WSC. The Golog procedures were combined with individual user constraints (e.g., *"I want to fly with a Star Alliance carrier"*) at run time, resulting in *dynamic binding of Web services*. However, the user constraints considered were hard constraints, that is, the realizations that did not satisfy those constraints were eliminated. Thus, this approach addressed hard constraints ϕ_{hard}, but did not deal with soft constraints ϕ_{soft}. Their modification to the Golog interpreter (written in Prolog) also had an ability to communicate with Web services via Open Agent Architecture (OAA) agent broker system. This approach finds a single WSC using the theorem proving capacity of their Prolog-based Golog interpreter.

The problem of gathering information during composition has been examined in [McIlraith and Son, 2002]. In particular, they describe a middle-ground interpreter that collects relevant information, but only simulates the effects of world-altering actions. Their interpreter works under the Invocation and Reasonable Persistence (IRP) assumption that (1) assumes all information gathering actions can be executed by the middle-ground interpreter and (2) assumes that the gathered information persists for a reasonable period of time, and none of the actions in the composition cause this assumption to be violated. Even though these assumptions may not hold for time-sensitive data (e.g., stock quotes), these assumptions are true for a large class of information available on the Web, in particular, for the type of information that interests us here (e.g., flight schedules, available hotels, tours, etc).

In this thesis, we extend the work in [McIlraith and Son, 2002] to be able to deal with *soft* user constraints. Our proposed preference language handles a wide variety of user constraints. It enables the synthesis of a composition of services, where the selection of services and service groundings (e.g., in the case of travel, the selection of the specific flight) can be customized to individual users at run time. The specification of the user preferences is discussed in Chapter 4 and the computation of optimized compositions is discussed in Chapter 5.

WSC with Procedural Control Knowledge Specified in HTN

An Hierarchical Task Network (HTN) planning problem can be viewed as a generalization of the classical planning paradigm [Ghallab et al., 2004]. An HTN domain contains, besides regular primitive actions, a set of *tasks* or high-level actions. Tasks can be successively refined or *decomposed* by the application of so-called *methods*. When this happens, the task is replaced by a new, intuitively more specific *task network* (of a set of tasks plus a set of restrictions, often ordering constraints among these tasks). The HTN planning problem consists of finding a

primitive decomposition of a given initial task network. The merit of HTNs is the intuitive nature of task hierarchies and the extensive computational machinery.

In this section, we discuss several papers that similarly use HTNs as a way to specify the procedural control knowledge. This approach relies on an existing OWL-S to HTN planning translation [Sirin et al., 2005b]. We discuss this translation in more detail in Chapter 3.

Sirin et al. [Sirin et al., 2005b] used **SHOP2**, a highly-optimized HTN planner for the WSC problem. The HTN induces a family of compositions and the if-then-else ordering of **SHOP2** provided a means of reflecting a preference for achieving a task one way over another. However, this limited form of preference was hard-coded into the **SHOP2** domain description and could not be customized by an individual user without recoding the HTN. In [Sirin et al., 2005a], an HTN-DL formalization was proposed in which they combined reasoning about Web service ontologies using a Description Logic (DL) reasoner with HTN planning. Other attempts have also been made to combine DL reasoning with Planning (e.g., [Lécué et al., 2008]). Unfortunately, many argue that DL-reasoner is not designed for the type of inference necessary for WSC and combining OWL-DL reasoning with planning can create significant performance challenges since one needs to call the reasoner many times during the planning phase, leading to very expensive computations [Sirin and Parsia, 2004]. Like their predecessor, they exploited **SHOP2** domain ordering to reflect preferences, but these were again not easily customizable to an individual user. They further provided a means of preferring services according to their class descriptions, but did not optimize the selection of service groundings.

Another notable work is by Kuter et al., in which they attempt to address the complete initial state assumption of a classical planner using an HTN-based planer ENQUIRER [Kuter et al., 2004]. ENQUIRER solves the WSC problem by gathering information during the composition process and is based on the **SHOP2** planning system. In particular, Kuter et al. take a similar approach to [McIlraith and Son, 2002] but their work focuses on dealing with services that do not return a result (if any) immediately. They provide a Query Manager that allows the planner to continue search without waiting for all of the information-gathering services to return data. They also assume that the information-gathering services are executable (similar to condition 1 of IRP), but they allow the planner itself to change the gathered information during planning (a variant of condition 2 of IRP). They prove soundness and completeness of their algorithm given sufficient conditions, and also test the efficiency of their algorithm.

Most recently, Lin et al. [Lin et al., 2008] proposed an algorithm for HTN planning with preferences described in the Planning Domain Definition Language PDDL3

[Gerevini and Long, 2005] that did allow for preferences over service groundings. They implemented a prototype of the algorithm in a planner, **SCUP**, tailored to the WSC problem. A merit of this work over previous HTN work is that it is not restricted to **SHOP2** syntax and, as such, provides the non-determinism (flexibility) necessary for preference-based planning. While this work seems closest to the work carried in this thesis (in fact it was carried out while this thesis was in progress), we believe the work described in this thesis has significant improvement over the work described in [Lin et al., 2008]. In particular, the ability of their planner to deal with preferences is somewhat limited, as it appears to be unable to handle conflicting user preferences. The authors indicate that conflicting preferences are removed (rather than resolved) during a preprocessing step prior to run time.

To conclude, the discussed approaches are all guided-automation approaches to address the WSC problem. Several of the papers overviewed consider some limited form of preferences and to some degree attempt to optimize for a preferred composition. However, these approaches do not consider handling policies or regulations. They also do not take advantage of state-of-the-art heuristic search techniques. In this thesis, we attempt to address their limitations with respect to both specification of soft and hard constraints (see Chapter 4), and computing optimized compositions (see Chapter 5) that not only optimizes for preferences, but also adheres to regulations and policies.

2.3.3 BPEL and Planning

In this section, we overview planning approaches that are based on Business Process Execution Language for Web Services (BPEL4WS). BPEL4WS is a collaborative efforts by Microsoft, IBM, BEA, SAP and Siebel, originally named BPEL4WS and later named WS-BPEL 2.0 (BPEL for short) [Andrews and et al, 2002]. We start this section by briefly overviewing BPEL, then we will overview several papers from the University of Trento researches that are based on the *planning as model checking* framework.

BPEL is a popular language for Web service orchestration that merges Microsoft and IBM languages XLANG and Web Services Flow Language (WSFL). BPEL models the specification of Executable and Abstract business processes. Executable business processes model the actual behaviour of a process that is generally internal to an organization, while Abstract business processes are more of a description, and only partially specify the behaviour of a progress. Abstract business processes are generally meant for external agents as a protocol to interact with a Web service. Consider the following purchasing protocol example from

[Andrews and et al, 2002]: "The seller has a service that receives a purchase order and responds with either acceptance or rejection based on a number of criteria, including availability of the goods and the credit of the buyer." While the actual decision is not clear, the decision process is a behaviour that must be represented. The abstract business process or the protocol must capture the different selections using non-determinism by enumerating the set of possibilities. This is done via the "switch" construct, which, similar to an if-then-else statement, effectively creates conditional branches. Besides the "switch" construct, there are several other constructs such as "reply", "receive", "partner-link", "sequence", "while", "pick", and "flow".

Given a logical specification of a goal, F, usually in a temporal logic formula CTL or LTL, and a formal model of a domain, M, usually represented as a Finite State Machine (FSM) or an automaton, model checking is a verification technique that solves the entailment problem $M \models F$. Planning as model checking (e.g., [Pistore and Traverso, 2001, Cimatti et al., 1997] is a technique to synthesize a plan, that is to automatically generate a plan from F and M. Planning as model checking generally allows for non-determinism, temporally extended goals, and incomplete knowledge. One way to address the planning as model checking approach is to use Binary Decision Diagrams (BDDs) model checking techniques [Burch et al., 1992]. This further allows for compact representation of the domain and the plan.

Several papers combine the BPEL specification of a process behaviour and the AI planning technique based on the *planning as model checking* approach for the task of WSC. Besides composition, monitoring and adaptation are two other key problems addressed by this approach, but those will not be discussed here.

EaGLe [Lago et al., 2002] is a goal specification language, inspired by LTL and CTL temporal logic specification languages, designed to express extended goals for planning with non-determinism. "Fail", "Repeat", "DoReach", and "TryReach" are among the EaGLe constructs. One of the features of this language is that some notion of preferences can be embodied in it. For example, one can specify a preference that states consider *goal* g_1 first, and only if it fails, consider *goal* g_2, by specifying the goal as g_1 *Fail* g_2. Also one can specify a goal of a different strength. For example, in *DoReach* p, p must be achieved, but in *TryReach*, it is recommended that p be achieved, but it is not forced. Hence, some notion of soft and hard constraints can be captured via the goal specification formula δ in EaGLe.

One of the first papers that attempts to combine the BPEL and AI planning technique based on the *planning as model checking* approach is the work by Pistore et al. [Pistore et al., 2004]. The BPEL abstract specification is expressed as a (non-deterministic) finite state automaton. The goal is specified in EaGLe language. To address partial observability, the executor or the

planner considers all the possible states, or the belief states when searching for the plan. Hence, at the "belief-level", produced by the power-set construction of the original domain, the search is fully observable. Their implemented planner is then used to generate a plan (an executable BPEL process) that meets the given specification. Their Model Based Planner (MBP), is able to handle several non-classical planning features. In particular, they take into account non-determinism, partial observability, and extended goals.

Traverso and Pistore take a slightly different approach in [Traverso and Pistore, 2004]. While they still represent the goals in the EaGle language, instead of describing services in BPEL abstract processes, they propose an approach in which Web services are described in the OWL-S process models. The OWL-S process models are then translated into non-deterministic and partially observable state transition systems. That is each OWL-S process model is encoded as a state transition system. The set of all transition systems is effectively the planning domain. Their proposed planning algorithm, which is implemented in their planner MBP, then generates plans. Note, they generate a family of plans, not just one, in a form of an automaton, and that their generated plan can be translated to BPEL executable processes. Below we give more details about the state transition system and the automata that represent their generated plan.

State transition systems describe the behaviour of a system. There may be more than one initial state, and the transitions indicate how occurrence of an action can generate new states (note that more than one state can be created due to non-determinism). The system's behaviour can be monitored by the *observations* which are defined for each state. Basically, observations model the output of the invoked process. The following definitions are taken from [Traverso and Pistore, 2004].

Definition 2.2 (State Transition System [Traverso and Pistore, 2004])
A (non-deterministic, partially observable) state transition system is a tuple $\Sigma_w = (S, A, O, I, T, X)$*, where:*

- S *is the set of states,*
- A *is the set of actions,*
- O *is the set of observations,*
- $I \subseteq S$ *is the set of initial states,*
- $T : S \times A \rightarrow 2^S$ *is the transition function; it associates to each current state* $s \in S$ *and to each action* $a \in A$ *the set* $T(s, a) \subseteq S$ *of next states, and*
- $X : S \rightarrow O$ *is the observation function.*

Definition 2.3 (Plan [Traverso and Pistore, 2004]) *Let Σ represents the domain (i.e., let Σ be $\Sigma = \Sigma_{w_1} \times ...\Sigma_{w_n}$). A plan for planning domain $\Sigma = (S, A, O, I, T, X)$ is a tuple $\pi = (C, c_0, \alpha, \epsilon)$, where:*

• *C is the set of plan contexts,*

• *$c_0 \in C$ is the initial context,*

• *$\alpha : C \times O \rightharpoonup A$ is the action function; it associates to a plan context c and an observation o an action a = (c, o) to be executed, and*

• *$\epsilon : C \times O \rightharpoonup C$ is the context evolutions function; it associates to a plan context c and an observation o a new plan context $c' = \epsilon(c, o)$.*

The plan is a deterministic automaton (both functions α and ϵ are deterministic). Note that the context is basically the internal states of the plans. It keeps track of the execution path by keeping into account for example, the knowledge gathered during previous execution steps.

Let Σ_π be the *execution structure* that represents the evolutions of the domain Σ controlled by the plan π. Note, the execution structure is a state transition system that compactly represents the execution of a plan in terms of transition between a configuration pair (s, c), where s is a state and c is a context. A plan is a **valid plan** for a goal G if $\Sigma_\pi \models G$. The plan π, which is an automaton, as claimed by the authors, can then be translated into an executable BPEL process [Traverso and Pistore, 2004].

Below we update our general framework definition for this approach.

Definition 2.4 (WSC via Planning as Model Checking) *A WSC problem via planning as model checking is described as $(\mathcal{D}, \delta)$ where: $\mathcal{D}$ is the abstract BPEL specifications of the Web services (in [Traverso and Pistore, 2004], they assume $\mathcal{D}$ is represented in OWL-S), and δ is a specification of the desired goal in EaGLe [Lago et al., 2002]. WSC generates a valid plan for the goal δ and the planning problem that results from representing $\mathcal{D}$ as a set of state transition systems.*

Pistore et al. in [Pistore et al., 2005] focus on the asynchronous feature of Web services and address the problem of WSC by appealing to planning in asynchronous domains. They argue that the interactions among Web services are asynchronous, that is each BPEL process is independent and may have unpredictable speed. Similar to their previous work, in which they translated the OWL-S process models to state transition systems, here they translate the BPEL abstract processes to state transition systems. The new state transition systems can change state either by "asynchronously" receiving message (input actions), sending messages (output

actions), or by evolving internally (internal actions). So in the state transition systems, they explicitly have a set of input actions, output actions, as well as the transition function, and the set of initial states. In addition, to capture the asynchronous interactions, when a message is sent it can be either received immediately or later after a sequence of internal action executions. Pistore et al. represent the goals as EaGle formula and again use the MBP planner to generate plans that can be translated to BPEL executable process. They summarize the body of the work that addresses WSC via planning in asynchronous domains in [Bertoli et al., 2010].

BPEL and AI planning techniques are combined in several papers (e.g., [Pistore et al., 2004]). Many of the papers we reviewed here consider planning under uncertainty and partial-observability. We consider this line of research to be a guided-automation approach, because the automated WSC problem is guided through a composition template, here expressed in the EaGLe specification language. Furthermore, the services considered in the BPEL-based approaches are generally stateful as BPEL represents the flow of interactions among Web services. In addition, the EaGLe goal specification language captures some limited notion of preferences. Finally, this line of research seems to be an online approach to the WSC problem, because information is gathered during the composition construction time.

2.3.4 Other Approaches

In this section, we briefly overview a few other approaches that use logic-based reasoning for the task of WSC.

Waldinger proposes an approach based on automated deduction and program synthesis [Waldinger, 2001]. Program synthesis refers to the automatic derivation of a program to meet a given formal specification of its behaviour [Manna and Waldinger, 1980]. More formally, given a valid input $P(x)$ and specification $R(x, y)$, the goal is to find a program $f(x)$ such that for an input a, if $P(a)$ holds, then the output $f(a)$ satisfies $R(a, f(a))$. In the deductive approach, the domain axioms and the program specification are represented by sequents, together with a mathematical induction rule, in a three-column table that represent assertion, goals, and outputs. Using the program synthesis approach, Waldinger [Waldinger, 2001] describes both the service description and the user desired functionality in a first-order language. The service description is the specification and the desired goal is the input or the query which is phrased as a theorem. Waldinger then uses the SNARK theorem prover to construct a proof for this problem. Finally, the answer (the composition) is extracted from a particular proof.

Another similar approach is by Rao et al., in which they use Linear Logic (LL) theorem proving for the WSC problem (e.g., [Rao et al., 2004]). They use OWL-S for the external representation of the description of Web services and use LL to internally represent axioms and proofs; the service functional (and non-functional) attributes are represented as proposition in the logical axioms. One of the key advantages of using LL is that they are able to define and distinguish the non-functional attributes of Web services through LL; they do so by translating the OWL-S service profiles to LL axioms. They then use a LL theorem prover to address the problem of WSC. In particular, the LL theorem prover is used to prove if the user's objective can be achieved by composition of available services. If so, the process model for the composite service is automatically extracted from the proof.

Another notable work is by Ponnekanti and Fox [Ponnekanti and Fox, 2002], which describes a set of tools for building composition of Web services using rule-based planning. Each service is represented by a Horn rule that, given a certain input, produces a particular output. The resulting toolkit, called SWORD, uses Prolog to reason about the encoded rule-based rules to generate a plan. One of the merits of this work is that it can compose information-gathering services. The objective or the goal description is represented as desired inputs and outputs to the system. SWORD then determines if the described behaviour can be generated using the rule engine, if so it generates a composition plan for it using the execution trace of Prolog. The current prototype version of the system cannot deal with services that have side-effects (world-altering services), and seems to only compose information-gathering services.

Finally, Petrick and Bacchus propose a planner that is able to address incomplete knowledge and sensing [Petrick and Bacchus, 2002, Petrick and Bacchus, 2004]. The key notion of their knowledge-level planner **PKS** is that it is able to reason with multiple databases that correspond to its knowledge (i.e., the agent's knowledge) rather than the state of the world and that the actions change the agent's knowledge, rather than the state of the world. This planner is able to construct conditional plans in a forward-search manner to deal with the incomplete knowledge. This planner is not used within the context of the WSC problem, but the key notions of this work can be applied to address incomplete information within the WSC problem.

2.4 Non-planning Approaches to the WSC Problem

In this section, we overview some of the non-planning approaches to the WSC problem. In particular, we first overview two approaches that are based on workflow modeling of the composite service [Schuster et al., 2000, Casati et al., 2000]. We then overview several pa-

pers (e.g., [Zeng et al., 2003]) that also take a workflow or template-based approach, but allow explicitly for optimization of Quality of Service (QoS). We call these QoS-aware approaches. We then overview two automata approaches, one using the Petri Nets (e.g., [Hamadi and Benatallah, 2003, Zhovtobryukh, 2007, Valero et al., 2009]) and one using the Roman Model [Calvanese et al., 2008]. We conclude this section by briefly discussing a few other papers that did not fall into any of the categories mentioned above.

2.4.1 Workflows

In this section, we overview two workflow-based approaches, eFlow [Casati et al., 2000] and Polymorphic Process Model (PPM) [Schuster et al., 2000]. Similar to how a composition template specified in a HTN or Golog can provide guidance in meeting a specific objective, workflows too can provide such guidance. However, the workflow-based approaches considered here (i.e., the traditional workflow-based approaches) usually provide more than just a guidance, as they rely on some form of manual coupling of the services, and may allow for limited flexibility or user customization. The workflow generally specifies how the abstract specifications of a service functionality are coupled, and at run time these will be binded to concrete available services, usually through service providers.

One example of such a system is eFLOW [Casati et al., 2000]. In eFlow each composite process is modeled by a graph (the flow structure or *process schema*) that includes service, decision and event nodes. The service node represents the abstract specification of an atomic or a composite service. The arcs in the graph specify how one service is connected to another. The transition is only possible upon successful execution of a service that has the outgoing arc. A successful *instantiation* of the process schema is called a service process instance. Note, eFlows allow for non-determinism, hence, there may be multiple possible instantiation of the same process schema.

The process schema for each composite service must be specified manually, but eFlow automatically binds the nodes with concrete services at run time. This is done via the eFlow engine's communication with the service broker that has a repository of all service description, and its job is to discover the actual service (and the service provider) that can meet the functionality specified in the service node by looking at the input/output/parameter of the service node. Also, eFlow performs online information gathering, as well as online execution of world-altering actions. In particular, there is an event monitor whose job is to fire *compensating nodes* in the case of failure to undo the effect of actions. A set of compensating nodes have

to be defined for each service to guarantee that all services executed successfully, or none did.

According to the authors, two forms of customizations are possible. The first is in the selection of services from the service broker's pool of services. This can be done by providing a service selection rule in a language understandable to a service broker. The actual form of these service selection rules and the kind of specification possible are not discussed. The second is done using generic nodes. Generic nodes allow more flexibility with respect to service selection. That is, instead of a service node one might use generic nodes to allow multiple abstract functionality or more dynamic bounding of a service (i.e., generic nodes provide a set of service nodes). For example, instead of a service node that provides air-transportation, one might use a generic node that allows the selection of both air-transportation and city-transportation. Our general framework can be updated for eFlow as follows.

Definition 2.5 (WSC as Workflow) *A WSC as workflow is described as a 3-tuple* $(\mathcal{D}, \delta, \phi_{soft})$ *where:* $\mathcal{D}$ *is a repository of process schema,* δ *is a specification of the desired behaviour in a process schema, and* ϕ_{soft} *is a specification of service-selection rule specified in a language understandable to a service broker. WSC engine determines an instantiation of the desired process schema or the* δ*.*

Similar to eFlow, Polymorphic Process Model (PPM) [Schuster et al., 2000] also binds the abstract services at run time. However, instead of just a single service provider or an enterprise in business terms, there could be a number of providers, called multi-enterprise processes (MEPs), that are themselves workflows of activities modeled as state machines implemented by different enterprises (activities are an abstract description of services). Similar to a generic node, if an activity and its implementation are coupled, then this precludes another potential implementation of the activity and, hence, limits the flexibility of the workflow. To capture all potential implementations of an activity (i.e., instantiation of an abstract service), Schuster et al. proposes to expand each activity to a subprocess that captures all alternatives as separate activities, and leaves the choice of implementation at run time. By decoupling activity interface from activity implementation their model allows individual enterprises (service providers) to maintain some flexibility in choosing how to implement the activities, while still providing guidance through activity interface, modeled as the state machines.

To conclude, workflow-based approaches considered here allow for limited form of optimization. In the next section, we overview a body of work that focuses specifically on how services can be selected based on their Quality of Service (QoS), using optimization techniques.

2.4.2 QoS-Aware

Several WSC approaches focus on the actual service selection problem and argue that the selection of services based on their quality is an important problem that must be addressed as a separate problem. In general they argue that there are four components to the WSC problem: Planning, Discovery, Selection and Optimization, and Execution. The planning phase determines the execution order of the tasks in a template or workflow. Note a task is the abstract specification of a services based on their functional property. The discovery process returns a set of candidate services that meet the functionality of the tasks specified in the composition. This line of research addresses the problem of service selection and optimization at the composition-construction-time based on both the functional (e.g., input and output matching) and non-functional (e.g., cost, availability, and reputation) properties of a service. This is addressed by encoding the problem as an optimization problem that can be solved using for example: Integer Programming (e.g., [Zeng et al., 2003]), Mixed Integer Programming (e.g., [Alrifai and Risse, 2009]) or Genetic Algorithms (e.g., [Lécué, 2009]).

Next, we overview the required terminology taken from [Zeng et al., 2003]. Let s_i, $i \in [1, ..., n]$ be a set of candidate services, and t_j, $j \in [1, ..., m]$ be set of tasks. The following are some non-functional properties related to a service (local constraints):

- **Cost** c_{ij} is the amount that a service requester needs to pay in order to execute service i using task j. This value is undetermined when service i cannot execute task j.
- **Time** t_{ij} measures the execution time between the moment the request is sent and the moment the results are received.
- **Availability** a_{ij} is the probability that the service can be accessed and used. It can be defined as the ratio of number of successful requests over the total number of invocations.
- **Reputation** r_{ij} is the measure of its trustworthiness. It can be defined as the average ranking given to the service by the end users.

In addition, there could also be some global constraints, which can be thought of policies, regulations, or hard constraints that are over the whole selection, such as the user's budget [Alrifai and Risse, 2009]. This problem can be characterized as an optimization problem whose objective is to minimize cost and time, and maximize availability and reputation while adhering to the global constraints.

Lécué [Lécué, 2009] proposes a QoS-aware approach in the context of the semantic Web. In particular, Lécué proposes to consider quality semantic links as additional optimization criteria.

The semantic link between two services is defined by the semantic similarities between output and input parameters of the two. Lécué argues that Genetic Algorithms provide a more scalable solution to the optimization problem at hand, and his experiments support this claim.

We will not update our general framework for this approach, as the problem addressed in this line of work is independent of the theory that describes the functional or non-functional properties of services. For example, Lécué [Lécué, 2009] assumes that the functional properties of a service is specified in an ontology (for example OWL-S or BPEL), but again the addressed problem is largely independent. Also, the line of work discussed here assumes an existence of some Web service discovery engine (e.g., UDDI (Universal, Description, Discovery, Integration)) to locate the available services for each task. So going back to our differentiating criteria, the QoS-aware approaches are guided-automation, offline approaches that may or may not serve the semantic Web. The main focus of the QoS-aware approaches is optimizing for the quality of services selected at the composition time.

2.4.3 The Petri Nets

Petri Nets is a formal tool for study and modeling of events and states in a distributed system [Murata, 1989]. A Petri Net is a directed connected bipartite graph containing a finite set of *places* (drawn as circles), finite set of *transitions* (drawn as rectangles), and transition input and output function (drawn as arcs). *Tokens* occupy a place and they can move using transitions from one place to another. A transition is enabled to fire (becomes potentially firable) whenever there is a token in all of the places that have an arc into this transition. When a transition fires, it removes one token from every place that has an arc into this transition (consumes the token) and puts one token in all of the places that have an arc out of this transition.

The following definitions are taken from [Narayanan and McIlraith, 2002].

Definition 2.6 (Petri Nets [Narayanan and McIlraith, 2002]) *A Petri Net is an algebraic structure* (P, T, I, O) *composed of:*

- *Finite set of places,* $P = \{p_1, p_2, ..., p_n\}$.
- *Finite set of transitions,* $T = \{t_1, t_2, ..., t_m\}$.
- *Transition input function,* I. I *maps each transition* t_i *to a multiset of* P.
- *Transition output function,* O. O *maps each transition* t_i *to a multiset of* P.

Definition 2.7 (Marking [Narayanan and McIlraith, 2002]) *A marking in a Petri Net* (P, T, I, O) *is a function* μ*, that maps every place into a natural number. If for a given mark-*

ing μ, $\mu(p_i) = x$, then it is said that the place p_i holds x tokens at the marking μ. A special marking, denoted by μ_0, is called the initial marking.

Petri Net execution can continue until a deadlock marking is reached, meaning that it reaches a marking in which no transitions can be fired. A sequence of firings $(t_1, ..., t_n)$ that take an initial marking μ_0 to a new marking μ_N is called an *occurrence sequence*. A marking is *reachable* if it is the marking reached by some occurrence sequence.

Petri Nets have many advantages that make them suitable for the WSC task. They have a formal semantic, can be visualized, and have the ability to model sequentiality (e.g., precedence constraint t_2 after t_1), concurrency and non-determinism. Petri Nets model concurrently because it allows non-determinism, meaning that when there is more than one transition that is enabled to fire, the Petri Nets fire a transition non-deterministically.

There are several Petri Nets approaches that address the task of WSC problem (e.g., [Narayanan and McIlraith, 2002, Hamadi and Benatallah, 2003, Zhovtobryukh, 2007]). In these approaches, the functional properties of a service is specified as a Petri Net with one input place and one output place. At any given time, a Web service can be in one of the following states: *not instantiated*, *ready*, *running*, *suspended*, or *completed*. When a service is *ready* that means a token is in its input place. Similarly, a service is completed when there is a token in its output place. Also note that several control constructs such as *sequence*, *parallel*, *condition*, *choice*, and the *iterate* can be represented in Petri Nets.

Below is an updated version of our general framework (i.e., Definition 2.1).

Definition 2.8 (WSC via Petri Nets) *A WSC problem via Petri Nets is described as $(S_{init}, \mathcal{D}, \delta)$ where: S_{init} is the initial markings for the set of Petri Nets; $\mathcal{D}$ is the set of Petri Nets, representing functional properties of a set of Web services; and δ is representation of the desired behaviour via a marking for $\mathcal{D}$. WSC determines a sequence of atomic services whose execution achieves the goal such that this sequence is an occurrence sequence in the reachability analysis of the marking δ for the set of Petri Nets $\mathcal{D}$.*

Narayanan and McIlraith in [Narayanan and McIlraith, 2002] automatically translate the OWL-S description of Web services into Petri Nets. In particular, they translate each of the OWL-S control constructs such as *iterate*, *if-then-else*, and *sequence*, into a Petri Net model representation. In addition, they provide a set of computational analysis tools that enable automation of Web service simulation (i.e., simulation of the evolution of a Web service under different conditions), validation (i.e., testing whether a Web service behaves as expected), verification and composition.

Hamadi and Benatallah in [Hamadi and Benatallah, 2003] provide an algebra that allows composition of services (i.e., creation of a new-value added service). They represent the control flow constructs such as the sequence, alternative, and iteration, using algebraic operations defined by their proposed algebra, hence, they provide a semantic to the Web service composition constructs in terms of Petri Nets. Similarly, Zhovtobryukh in [Zhovtobryukh, 2007] proposes another approach based on algebraic operators. However, this approach focuses on the automatic goal-driven creation of the composite Web services via HTN-like structure. This work also addresses limited user customization by allowing the user to specify the composition goal (objective); so the composition goal takes into account possible limitations and constraints of the user. However, the user constraints are treated as hard constraints rather than preferences.

Finally, in a more recent work, Valero et al. [Valero et al., 2009] developed a WSC approach based on Web Services Choreography Description Language (WS-CDL), while representing the WS-CDL behaviour in terms of Petri Nets. Using the formal semantics of Petri Nets, they further validate and verify their solution for the WSC problem.

To conclude, the Petri Nets approach to the WSC problem is a guided-automation, offline approach that can be used in the context of the semantic Web services (e.g., [Narayanan and McIlraith, 2002]). Also, the services considered in this approach are stateful. This approach also does not optimize for quality with respect to preferences or constraints.

2.4.4 The Roman Model

The Roman Model originally referred to by Rich Hull in [Hull, 2005] is another automata-based approach to the WSC problem. In the Roman Model approach, services are formally specified as transition systems and the desired specification (functionality) is a target service that is itself described as a transition system. The objective then is to synthesize an orchestrator that realizes the target service by executing the available services [Calvanese et al., 2008].

Services in the Roman Model are stateful software components, capable of performing operations. They are stateful because, depending on their state, they may offer a different choice of operations to their client. The client (which could be an automated system) will choose from one of the possible operations based on the state of the service, then the service executes it and accordingly changes its current state. So in other words, when a service is being used by a client the following three steps happen in sequence: (1) the client is given a set of operations to choose from, (2) the system waits for the client to choose exactly one element from the set, (3) the chosen operation is executed (this could result to termination or going

back to step (1)) [Berardi et al., 2005, Hull and Su, 2005]. The following is a formal definition of how a service is represented in a transition system [Calvanese et al., 2008].

Definition 2.9 (Service [Calvanese et al., 2008]) *A service is a transition system* $\mathcal{S} = (O, S, s^0, S^f, \rho)$*, where:*

- O *is the set of possible operations that the service recognizes,*
- S *is the finite set of service's states,*
- $s^0 \in S$ *is the initial state,*
- $S^f \subseteq S$ *is the set of final states, and*
- $\rho \subseteq S \times O \times S$ *is the service's transition relation, which accounts for its state changes.*

An operation o is said to be *executable* if there exists a transition $s \longrightarrow^o s'$ in $\mathcal{S}$ (s' is a possible successor state of s). The following is an updated version of Definition 2.1.

Definition 2.10 (WSC via Roman Model) *A WSC via Roman Model problem is described as* $(\mathcal{D}, \delta)$ *where:* $\mathcal{D}$ *is a set of available services specified in transition systems and* δ *is a specification of the target service as a transition system. WSC synthesizes an orchestrator that realizes the target service,* δ*, by exploiting the available services,* $\mathcal{D}$*.*

Note that the transition system associated with the available services allows non-determinism, but the transition system of the target service does not (because the target service is fully controllable by the clients).

The orchestrator considers the operation chosen by the client (as specified by the target service) and delegates it to one of the available services that is able to execute it. In other words, the orchestrator *realizes* a target service, if it is able to delegate all the operators that are executable by the target service to one of the available services. The goal of the WSC problem for this approach is proving the existence of such an orchestrator for the problem. There are several techniques to address the composition problem for this approach, including an approach based on model checking of game structures, a simulation-based approach, and an approach based on exploiting a reduction to Satisfiability (SAT) [Calvanese et al., 2008].

To summarize, the Roman Model is an offline, guided-automation approach to the WSC problem. One of the drawbacks of the Roman Model approach is that it does not support data (i.e., there is no way to represent the data and the data flow). However, since the client or the user can choose an operation as specified by the target service, this model can handle some limited notion of user preferences.

Category	**Level of Automation**	**Optimality**	**Context**	**Offline vs. online**	**Stateless vs. Stateful**
Classical Planning	fully-automated	limited optimality	some within semantic Web	offline	stateless
Golog	guided-automation	limited optimality	semantic Web	both	stateful
HTN	guided-automation	some handle optimality	semantic Web	both	stateful
BPEL	guided-automation	limited optimality	not within semantic Web	both	stateful
Logic-Based & Theorem Proving	guided-automation	optimality not handled	not within semantic Web	offline	stateless

Figure 2.1: Summary of the AI planning approaches to the WSC problem.

Category	**Level of Automation**	**Optimality**	**Context**	**Offline vs. online**	**Stateless vs. Stateful**
Workflow-Based	limited automation	limited optimality	not within semantic Web	online	stateless
QoS-Aware	guided-automation	handles optimality	some within semantic Web	offline	stateless
The Petri Nets	guided-automation	optimality not handled	some within semantic Web	offline	stateful
The Roman Model	guided-automation	limited optimality	could be within semantic Web	offline	stateful

Figure 2.2: Summary of the non-planning approaches to the WSC problem.

2.5 Summary

In this chapter, we overviewed some of the most popular approaches to the WSC problem with a view to address their limitations in this thesis. The problem addressed in the approaches we considered is an important piece of a larger set of problems that make up the Web service composition problem as a whole. We have divided the approaches into two main categories namely the AI planning and the non-planning approaches. The AI planning techniques focused on solving the composition problem by means of planning, whereas the non-planning approaches focused on either the non-planning component of the composition or proposed a non-planning approach to the problem. Figures 2.1 and 2.2 summarize the planning and non-planning approaches (respectively) with respect to our differentiating criteria.

Chapter 3

Characterizing Web Service Composition

3.1 Introduction

In Chapter 2 we described a general framework for the WSC problem while deliberately keeping the different components high level. The purpose of this framework is to give a basic definition of the WSC problem that can be adapted for the different approaches we overview. In this chapter, following previous work (discussed in Section 2.3.2), we adapt this general framework to provide a formal characterization of the WSC problem with customization. To that end, we articulate a notion of quality (optimizing) with respect to user, context, or instance-specific preferences (soft constraints) as well as a notion of enforcement with respect to hard constraints (e.g., policies or regulations). We give two closely-related characterizations based on how the WSC objective is specified. Given this characterization, we show how generating customized compositions of Web services is related to a non-classical planning problem.

As mentioned previously there are many reasons why using the classical planning approach to the WSC problem is not sufficient. Below we reiterate some of these reasons:

- **Non-Final Goal**: unlike classical planning, in the WSC problem we do not have a final-state goal. Instead we are given a specification of a basic behaviour or an objective we wish to achieve via a composition template.

- **Plan with Complex Actions**: in the WSC problem, Web services are specified as primitive and/or complex actions in an action formalism. Therefore, the complex actions as well as the primitive actions are the building blocks to constructing the composition and in order to compose services, one needs to plan with complex actions. The classical

planning approach may face scalability problems if the complex actions are compiled and represented as primitive [McIlraith and Fadel, 2002].

- **Many Actions**: the WSC problem can be data intensive resulting in planning domains with tens of thousands of ground actions. The composition template dictates the ways in which services can be composed, hence it can significantly reduce the search space for a composition. This can overcome the scalability problem of the classical planning approach to the WSC problem.

- **Optimization with Respect to Soft Constraints**: often, plan length is not the only measure of quality, and while compositions are plentiful, it is the generation of a high-quality composition with respect to rich measures of quality that is appealing.

- **Enforcement of Hard Constraints**: the composition templates are often further augmented with hard constraints such as policies or regulations that need to be enforced on the composition.

- **Incomplete Knowledge**: in the WSC problem, we often do not have complete information about the initial state, hence information-gathering services have to be executed to collect relevant information.

We start this chapter with a basic overview of OWL-S[1] [Martin et al., 2007], an ontology for Web services. We then discuss how OWL-S process models can be translated into an action/planning formalism so that they are more amenable to AI planning. We also establish a correspondence between the WSC problem and non-classical planning problem. Finally, we overview the OWL-S to HTN translation with a view of modifying it to support customization.

The main contributions of this chapter is the characterization of the WSC problem with customization. In the case where the composition template is specified as a Golog program, we provide a characterization of customized composition in the situation calculus. Given this characterization, we show how generating a composition is related to deductive planning. Furthermore, we show how generating a preferred composition (customized with respect to both soft and hard constraints) is related to preference-based planning.

Following previous work on the use of HTN planning for the WSC problem [Sirin et al., 2005b], we provide a characterization of customized composition where the composition template is specified as an HTN. Given this characterization, we show how generating

[1]W3C Recommendation. Latest version is available at `http://www.w3.org/Submission/OWL-S/`

a composition is related to generating an HTN plan, and how generating a preferred composition is related to generating a preferred HTN plan.

Interestingly, HTNs can be characterized as a special case of Golog programs (i.e., an HTN planning problem can be encoded in ConGolog) [Gabaldon, 2002]. In the final section of this chapter, we review salient features of this translation. Furthermore, we augment this translation to provide a situation calculus encoding of preference-based HTN planning. We use this translation to provide a semantics for our preference specification languages in Chapter 4. Furthermore, this provides a unifying framework for our characterization of the WSC problem with customization because it shows how these two forms of composition templates are related.

3.1.1 Contributions

The following are the main contributions of this chapter.

- Characterized the WSC problem with customization based on the specification of the objective as a Golog generic procedures together with a set of constraints. Our characterization is in the situation calculus and enables the use of deductive planning to generate a composition

- Defined a notion of preference-based planning within the HTN planning formalism

- Augmented the translation of OWL-S services into HTN planning to support customization. This translation translates OWL-S process models as well as OWL-S service profiles into corresponding HTN planning elements

- Characterized the WSC problem with customization based on the HTN-specification of the objective together with a set of constraints. This characterization enables the use of HTN planning for generating the composition (i.e., a plan returned by an HTN planner is the composition)

- Augmented the translation of HTN planning in the situation calculus to provide a situation calculus encoding of preference-based HTN planning. This characterization is used to provide the semantics of our preference languages. This characterization also provides a unifying framework for characterizing the WSC problem with customization because it shows how these two forms of composition templates are related.

3.2 OWL-S: From Services to Actions

OWL-S [Martin et al., 2007] is a Web ontology [Horrocks et al., 2003] for Web services that was developed with a view to supporting automated discovery, enactment and composition of Web services. The OWL-S ontology has three major components: service profile, process model, service grounding. Service profile indicates "what the service does". It can be used for service discovery (or the requester) to determine if the service meets the required needs. The service profile is used to advertise the service by describing its functional properties (e.g., input, output, precondition, and effects) and non-functional properties (e.g., service trust, reliability, subject, cost, etc). The process model describes how the service works and how to use the service. It describes how to request the service and furthermore, it explains what happens when the service is executed. Finally, service grounding explains how to interact with the service. Often service grounding will specify a communication protocol or service-specific details of how to contact the service.

OWL-S defines three classes of processes: atomic, composite and simple processes. Each process has input, output, precondition and effects. Atomic processes have no subprocesses and can be executed in a single step. Simple processes provide an abstract view for an existing process. However, unlike atomic processes a simple process is not associated with a grounding. A composite process is composed of other processes via control constructs such as Sequence, Split, Split-Join, Any-Order, Choice, If-Then-Else, Repeat-While, Repeat-Until, and Iterate.

Web service composition systems generally translate OWL-S process models into internal representations such as HTN, PDDL, or Golog that are more amenable to AI planning (e.g., [Narayanan and McIlraith, 2002, McIlraith and Fadel, 2002, McDermott, 2002, Sirin et al., 2005b]). These translations generally translate OWL-S process models as primitive and/or complex actions in an action formalism such as the situation calculus[Reiter, 2001]. Through these translations OWL-S is given a situation calculus semantics.

3.3 The Customization of the WSC Problem via Golog

The situation calculus and first-order logic can be used to describe the functional and non-functional properties of Web services. Next, we review the essentials of the situation calculus and Golog. We then provide a characterization of the WSC problem with customization.

3.3.1 Preliminaries

Situation Calculus

The situation calculus is a sorted logical language for specifying and reasoning about dynamical systems [Reiter, 2001]. The sorts of the language are *situation*, *action*, and a catch-all *object* sort. Situations are sequences of actions that represent a history of the world from an initial situation, which is denoted by S_0. The distinguished function $do(a, s)$ maps a situation and an action into a new situation, thus inducing a tree of situations rooted in S_0. Relational fluents (or simply fluents) are situation-dependent predicates that describe the properties that hold true in a particular situation[2]. Thus, a fluent is a predicate with a situation argument, e.g., $F(x, s)$. Finally the atomic expression $Poss(a, s)$ is true if action a is possible in situation s.

Web services such as the Web-exposed application at www.weather.com are viewed as actions in the situation calculus and are described as actions in terms of a situation calculus basic action theory, $\mathcal{D}$. A *basic action theory* in the situation calculus $\mathcal{D}$ includes *domain independent foundational axioms*, and *domain dependent axioms*. A situation s' precedes a situation s, i.e., $s' \sqsubset s$, means that the sequence s' is a proper prefix of sequence s. More details can be found in [Reiter, 2001, Narayanan and McIlraith, 2002, McIlraith and Son, 2002].

In the situation calculus, the planning problem is characterized as deductive planning (e.g., [Green, 1969, Reiter, 2001]). That is plans are generated as a side-effect of theorem proving. Given a planning problem and a goal formula, the planning problem is to find an action sequence that will lead to a state that satisfies the goal. More formally, given the **planning problem** $(\mathcal{D}, G)$ where $\mathcal{D}$ is the situation calculus basic action theory, and G is the specification of the goal formula, the deductive planning task is to prove that there exists a situation s such that the goal G is satisfied. I.e.,

$$\mathcal{D} \models (\exists s).executable(s) \wedge G(s) \tag{3.1}$$

where $executable(s) \stackrel{\text{def}}{=} (\forall a, s^*).do(a, s^*) \sqsubseteq s \supset Poss(a, s^*)$. Note, s is a situation that results from executing the sequence of actions (i.e., the plan) $a_1, ..., a_n$ in S_0.

[2]We do not consider functional fluents in this chapter and thus omit their description.

Golog

Golog[3] [Reiter, 2001] is a high-level logic programming language for the specification and execution of complex actions in dynamical domains. It builds on top of the situation calculus by providing Algol-inspired extralogical constructs for assembling primitive situation calculus actions into complex actions (aka *programs*), δ. These complex actions simply serve as constraints upon the situation tree. Complex action constructs include the following:

nil – the empty program

a – primitive action

$\phi?$ – test action

$\pi x.\, \delta$ – non-deterministic choice of argument

$\delta_1; \delta_2$ – sequences (δ_1 is followed by δ_2)

$\delta_1 | \delta_2$ – non-deterministic choice between δ_1 and δ_2

$\mathbf{if}\ \phi\ \mathbf{then}\ \delta_1\ \mathbf{else}\ \delta_2\ \mathbf{endif}$ – conditional

$\mathbf{while}\ \phi\ \mathbf{do}\ \delta\ \mathbf{endW}$ – loop

proc $P(\boldsymbol{v})$ δ **endProc** – procedure

In this thesis (originally proposed in [Sohrabi et al., 2006]), we also include the construct **anyorder**$[\delta_1, \ldots, \delta_n]$ which denotes the non-deterministic choice of all possible permutations of the sequencing of $\delta_1, \ldots, \delta_n$. This construct supports the specification of very flexible generic procedures. For example, one can define an anyorder construct over the following programs: book accommodations, book city-to-city transportation and book local transportation. This indicates that there does not exists a pre-defined ordering of these procedures. However, users may then wish to add constraints or preferences on top of these. In Chapter 4 we discuss how preferences can be specified over these generic procedures.

The conditional and while-loop constructs are defined in terms of other constructs. For the purposes of WSC we generally treat iteration as finitely bounded by a parameter k. Such finitely bounded programs are called *tree programs*. That is we assume that every process or program is going to terminate eventually. And in fact many services either are assigned a timeout (e.g., Air Canada allows ten minutes to complete a purchase), or allow a predefined number of attempts (e.g., your account will get suspended if you unsuccessfully try to login more than three times).

[3]We refer to Golog with its more recent semantics of ConGolog.

$$\textbf{if } \phi \textbf{ then } \delta_1 \textbf{ else } \delta_2 \textbf{ endIf} \stackrel{\text{def}}{=} [\phi?; \delta_1] \mid [\neg\phi?; \delta_2]$$

$$\textbf{while}_1(\phi)\ \delta\ \textbf{endWhile} \stackrel{\text{def}}{=} \textbf{if } \phi \textbf{ then } \delta \textbf{ endIf}$$

$$\textbf{while}_k(\phi)\ \delta \textbf{ endWhile} \stackrel{\text{def}}{=} \textbf{if } \phi \textbf{ then } [\delta;\ \textbf{while}_{k-1}(\phi)\delta \textbf{ endWhile}]\ \textbf{endIf}$$

These constructs can be used to write programs in the language of the domain theory, or more specifically, they can be used to specify both composite Web services and also generic procedures for WSC. Among the appeals of Golog is its procedural specification which is very much like a textual specification of a flexible workflow; its ability to treat complex actions as first-class objects in the language; and the fact that it is first-order, allowing the easy specification of data (a pervasive element in Web services) as a parameterization of actions. E.g.,[4]

$$bookAirTicket(\boldsymbol{x})\ ;\ \textbf{if far then } bookRentalCar(\boldsymbol{y})\ \textbf{else } bookTaxi(\boldsymbol{y})\ \textbf{endIf}$$
$$bookRentalCar(\boldsymbol{x})\ ; bookHotel(\boldsymbol{y}).$$

There are two popular semantics for Golog programs: the original evaluation semantics [Reiter, 2001] and a related single-step successor transition semantics that was proposed for on-line execution of concurrent Golog (ConGolog) programs [De Giacomo et al., 2000]. The transition semantics is axiomatized through two predicates $\mathit{Trans}(\delta, s, \delta', s')$ and $\mathit{Final}(\delta, s)$. Given an action theory $\mathcal{D}$, a program δ and a situation s, *Trans* defines the set of possible successor configurations (δ', s') according to the action theory. *Final* defines whether a program successfully terminated, in a given situation. *Trans* and *Final* are defined for every complex action. A few examples follow. (See [De Giacomo et al., 2000] for details):

$$\begin{aligned}
\mathit{Trans}(nil, s, \delta', s') &\equiv False \\
\mathit{Trans}(a, s, \delta', s') &\equiv Poss(a[s], s) \wedge \delta' = nil \wedge s' = do(a[s], s) \\
\mathit{Trans}(\phi?, s, \delta', s') &\equiv \phi[s] \wedge \delta' = nil \wedge s' = s \\
\mathit{Trans}([\delta_1; \delta_2], s, \delta', s') &\equiv \mathit{Final}(\delta_1, s) \wedge \mathit{Trans}(\delta_2, s, \delta', s') \\
&\quad \vee \exists \delta''.\delta' = (\delta''; \delta_2) \wedge \mathit{Trans}(\delta_1, s, \delta'', s')
\end{aligned}$$

[4]Following convention we will generally refer to fluents in situation-suppressed form, e.g., $at(Toronto)$ rather than $at(Toronto, s)$. Reintroduction of the situation term is denoted by $[s]$. Variables are universally quantified unless otherwise noted.

$$
\begin{aligned}
\textit{Trans}([\delta_1 \mid \delta_2], s, \delta', s') &\equiv \textit{Trans}(\delta_1, s, \delta', s') \vee \textit{Trans}(\delta_2, s, \delta', s') \\
\textit{Trans}(\pi(x)\delta, s, \delta', s') &\equiv \exists x. Trans(\delta_x^v, s, \delta', s') \\
\textit{Final}(nil, s) &\equiv \text{TRUE} \\
\textit{Final}(a, s) &\equiv \text{FALSE} \\
\textit{Final}([\delta_1; \delta_2], s) &\equiv Final(\delta_1, s) \wedge \textit{Final}(\delta_2, s)
\end{aligned}
$$

Thus, given the program $bookCar(\boldsymbol{x}); bookHotel(\boldsymbol{y})$, if the action $bookCar(\boldsymbol{x})$ is possible in situation s, then

$$\textit{Trans}([bookCar(\boldsymbol{x}); bookHotel(\boldsymbol{y})], s, bookHotel(\boldsymbol{y}), do(bookCar(\boldsymbol{x}), s))$$

describes the only possible transition according to the action theory. $do(bookCar(\boldsymbol{x}), s)$ is the transition and $bookHotel(\boldsymbol{y})$ is the remaining program to be executed. Using the transitive closure of *Trans*, denoted $\textit{Trans}^*$, one can define a Do predicate as follows. This Do is equivalent to the original evaluation semantics Do [De Giacomo et al., 2000].

$$Do(\delta, s, s') \stackrel{\text{def}}{=} \exists \delta'. \textit{Trans}^*(\delta, s, \delta', s') \wedge \textit{Final}(\delta', s'). \tag{3.2}$$

Given a domain theory, $\mathcal{D}$ and Golog program δ, program execution must find a sequence of actions $\boldsymbol{a}$ (where $\boldsymbol{a}$ is a vector of actions) such that: $\mathcal{D} \models Do(\delta, S_0, do(\boldsymbol{a}, S_0))$. $Do(\delta, S_0, do(\boldsymbol{a}, S_0))$ denotes that the Golog program δ, starting execution in S_0 will legally terminate in situation $do(\boldsymbol{a}, S_0)$, where $do(\boldsymbol{a}, S_0)$ abbreviates $do(a_n, do(a_{n-1}, \ldots, do(a_1, S_0)))$. Thus, given a generic procedure, described as a Golog program δ, and an initial situation S_0, we would like to infer a terminating situation $do(\boldsymbol{a}, S_0)$ such that the vector $\boldsymbol{a}$ denotes a sequence of Web services that can be performed to realize the generic procedure.

From Services to Golog

As mentioned earlier, WSC systems generally translate description of services provided in the OWL-S process models into primitive and/or complex actions in an action formalism such as the situation calculus (e.g., [Narayanan and McIlraith, 2002, McIlraith and Son, 2002]). In particular, the atomic process in OWL-S is translated into a situation calculus action. Special care is given to encode the input, output, precondition, and effect of the atomic process (its functional properties) in the situation calculus terms. The OWL-S composite process is repre-

sented as complex action in the situation calculus using Golog. Further details can be found in [Narayanan and McIlraith, 2002].

Following this translation, the following definition taken from [Sirin et al., 2005b] defines the so-called OWL-S WSC problem in the situation calculus using Golog.

Definition 3.1 (Adapted from [Sirin et al., 2005b]) *The OWL-S WSC problem $\mathcal{P}^{\mathcal{W}}$ is a 3-tuple (s_0, C, K), where s_0 is the initial state, K is a collection of OWL-S process models, and C is a possibly composite OWL-S process defined in K. Then $\pi = p_1...p_n$ (a sequence of atomic processes defined in K) is a solution/composition for $\mathcal{P}^{\mathcal{W}}$ if and only if:*

$$\mathcal{D} \models Do(\delta, S_0, do(\boldsymbol{a}, S_0)) \tag{3.3}$$

where $\boldsymbol{a} = a_1, ...a_n$, $\mathcal{D}$ is the situation calculus basic action theory axiomatizing K and s_0, δ is the Golog specification of the complex action defined for C as defined by the action theory (following the above translation), and a_i are the primitive actions that correspond to atomic process p_i as defined by the action theory (following the above translation).

3.3.2 Customized Composition of Web services via Golog

Following the form of our general framework, we now give a definition of the WSC problem with customization where all the elements are defined either in the situation calculus or have a situation calculus semantics. We assume the existing relationship between OWL-S WSC problem $\mathcal{P}^{\mathcal{W}}$ and Golog as defined in Definition 3.1.

Definition 3.2 (Customization of the WSC Problem via Golog) *A WSC problem with customization is described as a 5-tuple $(\mathcal{D}, O, \delta, \phi_{hard}, \phi_{soft})$ where:*

- *$\mathcal{D}$ is a situation calculus basic action theory describing functional properties of the services,*
- *O is a first-order logic theory describing the non-functional properties of the Web services,*
- *δ is a generic procedure described in Golog,*
- *ϕ_{hard} is a specification of hard constraints either in Golog or in LTL,*
- *ϕ_{soft} is a formula expressing soft constraints. This specification language should have its semantics defined in the situation calculus (e.g., $\mathcal{LPP}$ [Bienvenu et al., 2011]).*

A composition $\boldsymbol{a}$, customized with respect to the hard constraints, ϕ_{hard}, is a solution to $(\mathcal{D}, O, \delta, \phi_{hard})$ if and only if

$$\mathcal{D} \cup O \models \exists s. Do(\delta, S_0, s) \wedge s = do(\boldsymbol{a}, S_0) \wedge \phi_{hard}(s) \tag{3.4}$$

A composition $\boldsymbol{a}$, customized with respect to both the soft and hard constraints, ϕ_{soft} and ϕ_{hard}, is a solution to $(\mathcal{D}, O, \delta, \phi_{hard}, \phi_{soft})$ if and only if

$$\begin{aligned} \mathcal{D} \cup O \;\models\; & \exists s.Do(\delta, S_0, s) \wedge s = do(\boldsymbol{a}, S_0) \wedge \phi_{hard}(s) \\ & \wedge \;\nexists s'.[Do(\delta, S_0, s') \wedge \phi_{hard}(s') \wedge pref(s', s, \phi_{soft})] \end{aligned} \tag{3.5}$$

where $pref(s', s, \phi_{soft})$ is defined in the situation calculus stating that a situation s' is at least as preferred as a situation s with respect to a preference formula ϕ_{soft}. Further $Do(\delta, S_0, do(\boldsymbol{a}, S_0))$ denotes that the Golog program δ, starting execution in S_0 will legally terminate in situation $do(\boldsymbol{a}, S_0)$, where $do(\boldsymbol{a}, S_0)$ abbreviates $do(a_n, do(a_{n-1}, \ldots, do(a_1, S_0)))$.

Hence, a composition is a sequence of Web services, $\boldsymbol{a}$, whose execution starting in the initial situation (note, S_0 is described within the situation calculus basic action theory) enforces the generic procedure and hard constraints terminating successfully in $do(\boldsymbol{a}, S_0)$. Thus, given a generic procedure, described as a Golog program δ, and an initial situation S_0, we would like to infer a terminating situation $do(\boldsymbol{a}, S_0)$ such that $\boldsymbol{a}$ denotes a sequence of Web services that can be performed to realize the generic procedure. A customized composition with respect to both soft and hard constraints, $\boldsymbol{a}$, is a composition that yields a most preferred terminating situation (i.e., $\boldsymbol{a}$ is optimal).

In the next two propositions, we show how generating a customized composition is related to non-classical planning. Note, we will refer to goals that express properties that must hold throughout the execution of the plan as temporally extended goals.

Proposition 3.1 *Composition $\boldsymbol{a}$, customized with respect to the hard constraints, ϕ_{hard}, is a solution to the WSC problem with customization $(\mathcal{D}, O, \delta, \phi_{hard})$ as defined in Definition 3.2, if and only if $\boldsymbol{a}$ is a solution to (i.e., a plan for) the planning problem $(\mathcal{D} \cup O, G)$ with temporally extended goal G, where δ and ϕ_{hard} constitute a conjunctive temporally extended goal G.*

Proposition 3.2 *Composition $\boldsymbol{a}$, customized with respect to both the soft and hard constraints, ϕ_{soft} and ϕ_{hard}, is a solution to the WSC problem with customization $(\mathcal{D}, O, \delta, \phi_{hard}, \phi_{soft})$ as defined in Definition 3.2, if and only if $\boldsymbol{a}$ is a solution to (i.e., an optimal plan for) the preference-based planning problem $(\mathcal{D} \cup O, G, \phi_{soft})$ with temporally extended goal G, where δ and ϕ_{hard} constitute a conjunctive temporally extended goal G and ϕ_{soft} constitute the property to be optimized.*

It can be seen by inspection that the above propositions follow directly from the definition of deductive planning in the situation calculus, Equation 3.1. In particular, $s = do(\boldsymbol{a}, S_0)$ is a situation that results from executing the sequence of actions $\boldsymbol{a}$, starting in the initial situation S_0, such that the goal G is satisfied. This is analogous to the successful terminating situation s that results from executing the sequence of actions $\boldsymbol{a}$, in the initial situation S_0, while enforcing the Golog generic procedure and the hard constraints.

Given, the above propositions, we can use non-classical planning for the WSC problem with customization. In particular, we can use preference-based planning to generate a customized composition. In this thesis, we generate compositions using a planner that builds on top of a Prolog implementation of a Golog interpreter. In Chapter 5, Section 5.2, we discuss how our planner **GOLOGPREF**, computes the preferred composition, and prove its correctness.

3.4 The Customization of the WSC Problem via HTNs

In this section, we briefly overview HTN planning, provide a translation of OWL-S based WSC problem to HTN planning, and provide a characterization of the WSC problem with customization via HTNs.

3.4.1 Preliminaries

HTN Planning

An Hierarchical Task Network (HTN) planning problem can be viewed as a generalization of the classical planning paradigm [Ghallab et al., 2004]. An HTN domain contains, besides regular primitive actions, a set of *tasks* or high-level actions. Tasks can be successively refined or *decomposed* by the application of so-called *methods*. When this happens, the task is replaced by a new, intuitively more specific *task network* which a set of tasks plus a set of restrictions (often ordering constraints) that its tasks should satisfy. The HTN planning problem consists of finding a primitive decomposition of a given (initial) task network.

Returning to our Travel example the task of arranging travel can be decomposed into arranging transportation, accommodations, and local transportation. Each of these tasks can successively be decomposed into other subtasks based on alternative modes of transportation and accommodations, eventually reducing to primitive actions that can be executed in the world. In planning, this decomposition and search is performed by an HTN planner. The merit of

HTNs is the intuitive nature of task hierarchies and the extensive computational machinery. The following is a definition of HTN planning taken from [Ghallab et al., 2004].

Definition 3.3 (HTN Planning Problem. Adapted from [Ghallab et al., 2004])
An HTN planning problem is a 3-tuple $\mathcal{P} = (s_0, w_0, D)$ where s_0 is the initial state, w_0 is a task network called the initial task network, and D is the HTN planning domain which consists of a set of operators and methods.

A domain is a pair $D = (O, M)$ where O is a set of operators and M is a set of methods. An operator is a primitive action, described by a triple *o =(name(o), pre(o), eff(o))*, corresponding to the operator's name, preconditions and effects. In our example, ignoring the parameters, operators might include: *book-train, book-hotel,* and *book-flight*.

A *task* consists of a task symbol and a list of arguments. A task is primitive if its task symbol is an operator name and its parameters match, otherwise it is *nonprimitive*. In our example, *arrange-trans* and *arrange-acc* are nonprimitive tasks, while *book-flight* and *book-car* are primitive tasks.

A method, m, is a 4-tuple (*name(m), task(m),subtasks(m), constr(m))* corresponding to the method's name, a nonprimitive task and the method's task network, comprising subtasks and constraints. Method m is relevant for a task t if there is a substitution σ such that $\sigma(t)$ =*task*(m). Several methods can be relevant to a particular nonprimitive task t, leading to different decompositions of t. In our example, the method with *name by-flight-trans* can be used to decompose the *task arrange-trans* into the *subtasks* of booking a flight and paying, with the constraint (*constr*) that the booking precede payment. An operator o may also accomplish a ground primitive task t if their names match.

Definition 3.4 (Task Network. Adapted from [Ghallab et al., 2004])
A task network is a pair w = (U, C) where U is a set of task nodes and C is a set of constraints. Each task node u ∈ U contains a task t_u. If all of the tasks are primitive, then w is called primitive; otherwise it is called nonprimitive.

In our example, we could have a task network (U, C) where $U = \{u_1, u_2\}$, u_1 =*book-car*, and u_2= *pay*, and C is a precedence constraint such that u_1 must occur before u_2 and a before-constraint such that at least one car is available for rent before u_1.

Definition 3.5 (Plan. Adapted from [Ghallab et al., 2004])
$\boldsymbol{a} = o_1 o_2 \ldots o_k$ is a plan for HTN planning program $\mathcal{P} = (s_0, w_0, D)$ if there is a primitive decomposition of w_0, w, of which $\boldsymbol{a}$ is an instance.

Finally, we define the notion of *preference-based* planning (published in [Sohrabi et al., 2009]). To do so we assume the existence of a reflexive and transitive relation $\preceq$ between plans. If $\boldsymbol{a}_1$ and $\boldsymbol{a}_2$ are plans for $\mathcal{P}$ and $\boldsymbol{a}_1 \preceq \boldsymbol{a}_2$ we say that $\boldsymbol{a}_1$ is *at least as preferred as* $\boldsymbol{a}_2$. We use $\boldsymbol{a}_1 \prec \boldsymbol{a}_2$ as an abbreviation for $\boldsymbol{a}_1 \preceq \boldsymbol{a}_2$ and $\boldsymbol{a}_2 \not\preceq \boldsymbol{a}_1$.

Definition 3.6 (Preference-based HTN Planning) *An HTN planning problem with user preferences is described as a 4-tuple $\mathcal{P} = (s_0, w_0, D, \preceq)$ where $\preceq$ is a preorder between plans. A plan $\boldsymbol{a}$ is a solution to $\mathcal{P}$ if and only if: $\boldsymbol{a}$ is a plan for $\mathcal{P}' = (s_0, w_0, D)$ ($\mathcal{P}$ without the preferences) and there does not exists a plan $\boldsymbol{a}'$ for $\mathcal{P}'$ such that $\boldsymbol{a}' \prec \boldsymbol{a}$.*

The $\preceq$ relation can be defined in many ways. In Chapter 4 we describe PDDL3, which defines $\preceq$ quantitatively through a metric function.

From Services to HTN

In this thesis, we rely on the translation of the OWL-S based WSC problem to HTN planning. Our translation is similar to that in [Sirin et al., 2005b]. We augment this translation in order to be able to specify compelling soft constraints to support customization. We first describe how to encode an OWL-S process model as elements of HTN planning (i.e., operators and methods). Then we describe how to encode the service profile. Encoding the service profile as a component of HTN planning will enable users to specify preferences over how to select services based on their non-functional properties (i.e., those specified in the service profile).

Our translation, similar to that in [Sirin et al., 2005b], also encodes each atomic process as an HTN operator. We also encode each composite and simple process as an HTN method. Where our translation differs is that we associate each method with a unique name. Having a name for a method allows preferences to refer to methods by their name. This is particularly important in preferences that describe how to decompose a particular task. Since a task can be realized by more than one method, being able to distinguish each method by its name allows the user to express preferences over which methods they prefer, or in other words, how they prefer the task to be realized. In Chapter 4, we will give examples of such preferences. Below we show how to translate the Sequence and Choice construct. The translations for the rest of the constructs is similar. See [Sirin et al., 2005b] for more details of the translation.

Translate-Sequence(Q)
Input: an OWL-S definition of a composite process Q in the form Q_1;Q_2;...;Q_k with Sequence

control construct.

Output: an HTN method M.

Procedure:

(1) let $\boldsymbol{v}$ = the list of input parameters defined for Q

(2) let Pre = conjunct of all preconditions of Q

(3) **for all** $i : 1 \leq i \leq k$: let n_i be a task node for Q_i

(4) let C= $\{before(n_1, Pre), (n_i, n_{i+1}) | 1 \leq i < k\}$

(5) **Return** M = (N_m, $Q(\boldsymbol{v})$, $\{n_1, n_2, ..., n_k\}$, C), where N_m is a unique method name.

Translate-Choice(Q)

Input: an OWL-S definition of a composite process Q in the form Q_1;?Q_2;?...;?Q_k with Choice control construct.

Output: a collection of HTN methods M.

Procedure:

(1) let $\boldsymbol{v}$ = the list of input parameters defined for Q

(2) let Pre = conjunct of all preconditions of Q

(3) **for all** $i : 1 \leq i \leq k$

(4) let n_i be a task node for Q_i

(5) let M_i = (N_{m_i}, $Q(\boldsymbol{v})$, $\{n_i\}$, $\{before(n_i, Pre)\}$), where N_{m_i} is a unique method name.

(6) **Return** M = $\{M_1, ..., M_2\}$.

In addition, for every process and subprocesses in the process model that is associated with a service (i.e., is executable on the Web), we compile its service profile as extra properties of their corresponding HTN element. Hence, if an atomic/composite process is associated with a service, its corresponding HTN operator/method will be associated with that service profile. We capture this extra property using a predicate *isAssociatedWith*.

For example, let us assume that the Air Canada service can be described by an atomic process *AP* and service profile *SP*. In addition, assume that the service profile *SP has-name* AirCanada, *has-url* www.aircanada.com, *has-language* English, *has-trust* high, *has-reliability* high. Then we will encode the atomic process *AP* into an HTN operator with the same name as described above. Next, we would capture the service profile of the service Air Canada associated with the atomic process *AP* by the binary predicate *isAssociatedWith(AP, SP)*. Note, *AP* is the name of the encoded HTN operator. In the case of composite process we would have the name of the corresponding HTN method. The profile information of the service profile

SP would now be described by predicates *has-language(SP, English)*, *has-trust(SP, High)*, and *has-reliability(SP, High)*.

The following definition establishes the relationship between the plans found for an HTN planning problem and for compositions of the OWL-S WSC problem. This definition is based on the correctness theorem of the OWL-S to HTN translation given in [Sirin et al., 2005b, Theorem 5].

Definition 3.7 (Adapted from [Sirin et al., 2005b]) *Let $\mathcal{P}^{\mathcal{W}} = (s_0, C, K)$ be the OWL-S WSC problem, where s_0 is the initial state, K is a collection of OWL-S process models, and C is a possibly composite OWL-S process defined in K. Then $\pi = p_1...p_n$ (a sequence of atomic processes defined in K) is a solution to (i.e., a composition for) $\mathcal{P}^{\mathcal{W}}$ if and only if $\boldsymbol{a} = a_1...a_n$ is a solution to (i.e., a plan for) $\mathcal{P} = (s_0, w_0, D)$, where*

- *w_0 is generated by the OWL-S to HTN translation for the OWL-S process C,*
- *D is generated by the OWL-S to HTN translation for the OWL-S process models K, and*
- *a_i are the primitive actions that correspond to atomic process p_i as defined by the OWL-S to HTN translation.*

3.4.2 Customized Composition of Web services via HTNs

We now give a definition of the WSC problem with customization, following the form of our general framework definition given in Chapter 2. Note, we now assume the existing relationship between OWL-S WSC problem and HTN planning problem as defined in Definition 3.7.

Definition 3.8 (Customization of the WSC Problem via HTNs) *Let $\mathcal{P} = (s_0, w_0, D)$ be an HTN planning problem whose solution is also a solution to an OWL-S WSC problem as defined in Definition 3.7. A WSC problem with customization is then described as a 6-tuple $(S_{init}, \mathcal{D}, O, \delta, \phi_{hard}, \phi_{soft})$ where:*

- *S_{init} is the initial state s_0,*
- *$\mathcal{D} \cup O$ is the HTN domain description D, describing functional and non-functional properties of the Web services[5],*
- *δ is the HTN initial task network w_0 specifying the objective,*
- *ϕ_{hard} is a specification of the hard constraints specified in (a subset of) LTL,*
- *ϕ_{soft} is a specification of soft constraints.*

[5] O is specified as an HTN special predicate that encodes the OWL-S service profiles

A composition $\mathbf{a}$, customized with respect to the hard constraints, ϕ_{hard}, is a solution to $(S_{init}, \mathcal{D}, O, \delta, \phi_{hard})$ if and only if $\mathbf{a}$ is a solution to (i.e., a plan for) the HTN planning problem (s_0, w_0, D) that adheres to ϕ_{hard}.

A composition $\mathbf{a}$, customized with respect to both the soft and hard constraints, ϕ_{soft} and ϕ_{hard}, is a solution to $(S_{init}, \mathcal{D}, O, \delta, \phi_{hard}, \phi_{soft})$ if and only if $\mathbf{a}$ is a solution to (i.e., a plan for) the preference-based HTN planning problem $(s_0, w_0, D, \preceq)$ that adheres to ϕ_{hard}, where $\preceq$ is a preorder between plans as defined by ϕ_{soft}.

The above definition shows how generating a customized composition is related to generating a plan using HTN planning. This shows that we can use preference-based planning to generate customized compositions. We show in Chapter 5, how we generate customized compositions using preference-based planning.

3.5 Situation Calculus Specification of HTN Planning

Next, we describe an encoding of HTN planning in Golog/ConGolog. In particular, we describe an existing translation of the HTN planning into the situation calculus entailment of ConGolog (an extension of Golog with concurrency and interrupts), which we augment and extend to provide an encoding of preference-based HTN planning. The situation calculus encoding of the HTN planning problem and the preference-based HTN planning problem provides a unifying framework for our characterization of the WSC problem with customization because it shows how these the two forms of composition templates we use, HTNs and Golog, are related. In this section, we review the salient features of this translation. The situation calculus[6] encoding of HTN planning also helps us with the semantics of our preference languages.

A number of researchers have pointed out the connection between HTN and ConGolog. In this thesis, we appeal to the Gabaldon [Gabaldon, 2002] encoding of HTN planning problem in Golog/ConGolog. In short, the translation defines a way to construct a logical theory and formula $\Psi(s)$ such that $\Psi(s)$ is entailed by the logical theory if and only if the sequence of actions encoded by s is a solution to the original HTN planning problem.

More specifically, the initial HTN state s_0 is encoded as the initial situation, S_0. Each literal l is mapped to a fluent or non-fluent relation in the situation calculus, as appropriate. The HTN domain description maps to a corresponding situation calculus domain description, $\mathcal{D}$, where

[6]By situation calculus we mean the situation calculus entailment of ConGolog

for every operator o there is a corresponding primitive action a, such that the preconditions and the effects of o are axiomatized in $\mathcal{D}$ as action precondition axioms and successor state axioms respectively. In addition, $\mathcal{D}$ has the unique name axioms, axioms describing the initial situations, and domain-independent foundational axioms for the situation calculus. Every method and nonprimitive task together with constraints is encoded as a ConGolog procedure. $\mathcal{R}$ is the set of procedures in the ConGolog domain theory.

To deal with partially ordered task networks, following Gabaldon's translation, we add two new primitive actions *start*$(P(\boldsymbol{v}))$ and *end*$(P(\boldsymbol{v}))$, to each procedure P that corresponds to an HTN task or method. In addition, we add the fluents *executing*$(P(\boldsymbol{v}), s)$ and *terminated*(X, s), where $P(\boldsymbol{v})$ is a ConGolog procedure and X is either $P(\boldsymbol{v})$ or a primitive action a. *executing*$(P(\boldsymbol{v}), s)$ states that $P(\boldsymbol{v})$ is executing in situation s, *terminated*(X, s) states that X has terminated in s. *executing*(a, s) where a is a primitive action, is defined to be false. The successor state axioms for these fluents follow. They show how the actions *start*$(P(\boldsymbol{v}))$ and *end*$(P(\boldsymbol{v}))$ change the truth value of these fluents:

$$\textit{executing}(P(\boldsymbol{v}), do(a, s)) \equiv a = \textit{start}(P(\boldsymbol{v})) \vee \textit{executing}(P(\boldsymbol{v}), s) \wedge a \neq \textit{end}(P(\boldsymbol{v}))$$
$$\textit{terminated}(X, do(a, s)) \equiv X = a \vee (X \in \mathcal{R} \wedge a = \textit{end}(X)) \vee \textit{terminated}(X, s)$$

Note, the two primitive actions *start*$(P(\boldsymbol{v}))$, *end*$(P(\boldsymbol{v}))$ actions are helper actions (they are auxiliary); hence, they are not returned as part of the plan.

Definition 3.9 (HTN Planning in the Situation Calculus) *An HTN planning problem with preferences described as a 4-tuple $(s_0, w, D, \preceq)$, where s_0 is the initial state, w is the initial task network, D is the HTN planning domain, and $\preceq$ is a preorder between plans, is encoded in the situation calculus as a 5-tuple $(\mathcal{D}, \mathcal{C}, \Delta, \delta_0, \Phi_{sc})$ where $\mathcal{D}$ is the basic action theory, $\mathcal{C}$ is the set of ConGolog axioms, Δ is the sequence of procedure declarations for all ConGolog procedures in $\mathcal{R}$, δ_0 is an encoding of the initial task network in ConGolog, and Φ_{sc} is a mapping of the preference relation $\preceq$ in the situation calculus. A plan $\boldsymbol{a}$ is a solution to the HTN problem (s_0, w, D) if and only if:*

$$\mathcal{D} \cup \mathcal{C} \models (\exists s) Do(\Delta; \delta_0, S_0, s) \wedge s = do(\boldsymbol{a}, S_0) \tag{3.6}$$

A plan $\boldsymbol{a}$ is a solution to preference-based HTN problem $(s_0, w, D, \preceq)$ if and only if:

$$\begin{aligned} \mathcal{D} \cup \mathcal{C} \models (\exists s) Do(\Delta; \delta_0, S_0, s) \wedge s = do(\boldsymbol{a}, S_0) \\ \wedge \nexists s'.[Do(\Delta; \delta_0, S_0, s') \wedge pref(s', s, \Phi_{sc})] \end{aligned} \tag{3.7}$$

where $pref(s', s, \Phi_{sc})$ (defined in Definition 4.7) denotes that the situation s' is preferred to situation s with respect to the preference formula Φ_{sc}, and $Do(\delta, S_0, do(\boldsymbol{a}, S_0))$ denotes that the ConGolog program δ, starting execution in S_0 will legally terminate in situation $do(\boldsymbol{a}, S_0)$.

3.6 Summary and Discussion

In this chapter we gave two closely-related characterizations of WSC with customization, based on how the WSC objective is specified. Given this characterization, we showed how generating customized compositions of Web services is related to a non-classical planning problem. With this characterization of the WSC problem with customization in hand, we now move to the problem of how to specify the soft and hard constraints.

In this chapter, we assume for the most part that relevant information is gathered offline, before composition is constructed. Additionally, we also assume the planning domain is deterministic, meaning that an action or service's behaviour is deterministic (i.e., an action transforms a state into a single known successor state). At face value, many exposed Web services may appear to behave non-deterministically, but this apparent non-determinism is often predicated on a lack of information. For example, one cannot predict in advance whether a particular book can be purchased at amazon.com, so the outcome of the action *buy-book(X)* may appear non-deterministic, but once it is determined whether or not the book is in stock, the outcome is determined. Similarly, other sources of apparent non-determinism are predicated on unknown information that *cannot* be observed, or that cannot be observed until execution time. Such forms of non-determinism can be modeled away by enumerating a set of outcomes, conditioned on (possibly unobservable) aspects of state. For example, while from an applicant's perspective, whether or not a mortgage approval service will grant a mortgage appears non-deterministic, but it is actually a deterministic process, predicated on some aspect of state that is unknown to the applicant. Thus, the service can be modeled as a deterministic action. This latter form of non-determinism relates to the notion of model-lite planning that has emerged in recent years (e.g., [Kambhampati, 2007]).

In Chapter 6 we remove the assumption that information is gathered offline, and address the information-gathering problem with a view to producing high-quality compositions. In so doing, we are able to address the class of non-deterministic Web service behaviour that originates from a lack of information that is observable and not predicated on the context at execution time. Gathering such information, as we do in Chapter 6, allows us to effectively determinize the action outcome. We do so via a middle-ground execution engine that simulates world-

altering services but executes information-gathering services to obtain relevant information as it becomes necessary to do so in order to plan. This addresses a large class of, what appear to be, non-deterministic services.

Nevertheless, there are other classes of non-determinism in the domain that are due to lack of predictability of the outcome of a service or run-time failures, that our approach currently may not be able to handle. There are a number of different ways that our work can be extended in order to address these classes of problems.

For example, we can exploit the planning as model checking approach (e.g., [Bertoli et al., 2010]) and in particular the recent work by Kuter et al. [Kuter et al., 2009] in which they describe an approach that combines the non-deterministic version of **SHOP2**, **ND-SHOP2** [Kuter and Nau, 2004] with the MBP planner (e.g., [Traverso and Pistore, 2004]) in order to address the non-determinism. Another interesting approach would be to model this non-determinism as deterministic actions predicated on unobservable state, and then to characterize the composition task as a conformant planning task. We could then, for example, exploit the recent reformulation approaches to conformant or contingent planning (e.g., [Palacios and Geffner, 2007]) to address the composition problem. Regardless of how we choose to extend our work, an important problem is to ensure some notion of guaranteed optimality. To that end, we can exploit the recent work by Shaparau et al. [Shaparau et al., 2006] in order to deal with the issue of optimality and non-determinism, in some measure. This is an interesting area for future work.

Chapter 4

Specifying Soft and Hard Constraints

4.1 Introduction

In the previous chapter, we characterized the WSC problem with customization with respect to soft and hard constraints. In this chapter, we describe how to specify these constraints. To that end, we first design a set of desirable criteria, evaluate the existing specification languages with respect to this set, and extend the existing languages to meet our set of desirable criteria.

As discussed in the introduction, we argue that customization is an important problem and a critical and missing component of most existing approaches to the WSC problem. Hard constraints are a useful way of enforcing business rules and policies. Many customers are concerned with enforcement of hard constraints, often in the form of corporate policies and/or government regulations. *Policies* or *regulations* are a set of constraints imposed by an authority that define an acceptable behaviour or characteristic of an agent, person, or an organization. Policies or regulations hence are a set of constraints imposed on the composition that define an acceptable composition. If commerce is being performed across multiple governmental jurisdictions, there may be a need to ensure that laws and regulations pertaining to commerce are enforced appropriately. A company may wish to ensure that all transactions comply with company policies. For example, they might impose on their employees when traveling, to always use their corporate credit card for their travel expenses.

Software that is developed for use by a particular corporation or jurisdiction will have the enforcement of such regulations built in. For Web services that are published for use by the masses this is not the case, and the onus is often on the customer to ensure that regulations are enforced when the composition is constructed from multiple service providers. For inter-jurisdictional or international business, different regulations may apply to different aspects

of the composition. Hence, customizing the composition of Web service by imposing hard constraints and to that end, providing a mechanism for generating compositions that adhere to such constraints is an important problem and one that we address in this thesis.

In contrast, *preferences* or soft constraints are a set of properties that define the quality of the composition. They differ from hard constraints because their satisfaction is not mandatory but desirable. User preferences are important, because they enable a user to specify properties of solutions that make them more or less desirable. The composition system can use preferences to find preferred solutions among (often large) families of solutions.

User preferences are also critical because they can express a preference for *how* the composition is performed. A key component of Web service composition is the selection of specific services that are used to realize the composition. In AI planning, primitive actions (the analogue of services) are selected for composition based on their preconditions and effects, and there is often only one primitive action that realizes a particular effect. For many WSC problems, the task can be realized by a diversity of different services, offering comparable, but not identical services. By integrating user preferences into the composition problem, preferences over services (the *how*) can be specified and considered alongside preferences over the solutions (the *what*).

4.1.1 Contributions

The following are the main contributions of this chapter.

- Designed a set of desirable criteria for specification of preferences, tailored the task of the WSC problem.
- Evaluated the existing preference language $\mathcal{LPP}$ [Bienvenu et al., 2006, Bienvenu et al., 2011] with respect to our set of desirable criteria. To that end, we describe $\mathcal{LPP}$, a rich qualitative preference language proposed by Bienvenu et al. We also discuss how we can specify service selection preferences within the $\mathcal{LPP}$ language. $\mathcal{LPP}$, unlike many other preference languages, provides a facility to stipulate the relative strength of preferences. Furthermore, $\mathcal{LPP}$ is qualitative in nature, facilitating preference specification. We use this language to specify preferences in our system **GOLOGPREF** which we discuss in Chapter 5.
- Extended the preference language $\mathcal{LPP}$ to address our set of desirable criteria. To that end, we extend $\mathcal{LPP}$ and we name the updated language $\mathcal{LPH}$. The proposed prefer-

ence language augments the preference language $\mathcal{LPP}$ with HTN-specific constructs. Among the HTN-specific properties that we add to $\mathcal{LPH}$, is the ability to express preferences over how tasks in our HTN are decomposed into subtasks, preferences over the parameterizations of decomposed tasks, and a variety of temporal and non-temporal preferences over the task networks themselves. We use $\mathcal{LPH}$ in our system **HTNPLAN** which we discuss in Chapter 5.

- Evaluated of the existing preference language, PDDL3 [Gerevini and Long, 2005] with respect to our set of desirable criteria. To that end, we show how PDDL3 preferences can be specified and aggregated. We also show how PDDL3 enables the expression of temporally extended preferences.

- Extended PDDL3 to meet our set of desirable criteria. Our extension to PDDL3 is similar to our extension for $\mathcal{LPP}$ as it allows users to specify preferences over HTN constructs. This was done by augmenting PDDL3 with three new constructs that indicate the occurrence, initiation, and termination of tasks. Our extension also enables expression of action-centric preferences, which in turn allows users to specify preferences over the functional as well as non-functional properties of services. We discuss how we can specify these non-functional properties of services using PDDL3. Our PDDL3 extension is used in our system **HTNPLAN-P** which we discuss in Chapter 5.

- Designed a set of desirable criteria for specification of hard constraints (e.g., policies and regulations) and discussed how to specify the hard constraints using the existing languages.

In Chapter 5, we will discuss how to compute preferred plans/compositions, given the specification of preferences and policies discussed in this chapter.

4.2 Set of Desirable Criteria for Constraint Specification

In order to evaluate our preference specification language, we design a set of desirable criteria that we wish our languages to satisfy. We start this section with a set of desirable criteria for preference specification. The following is our proposed list.

- **State-Centric and Action-Centric**: often the preference language specifies state related constraints. That is they identify preferred states along the plan trajectory. While this

is interesting we also wish to be able to express preferences over the occurrence of both primitive and complex actions within the plan trajectory. Preferences over complex actions can specify a preference over how they decompose into primitive actions.

- **Functional and Non-Functional Properties of Services**: services are associated with four properties (inputs, outputs, preconditions and effects), called *functional properties* of services. Services are also associated with their *non-functional properties*. These properties are often used to describe the features of the service so as to ease their discovery and selection. The following is a possible list of non-functional properties: service name, service author, service language, service trust, service subject, service reliability, and service cost. An ideal preferences language should enable preferences over both functional and non-functional properties of services. That is, the preference language should support the expression of preferences over the selection of services.

- **Distinct from Domain Specification**: the specification of a preference formula should be distinct from the specification of the domain. Separating the two enables users to share HTN method definitions within the domain specification while individualizing preferences that relate to those methods.

- **Conditional**: the preference language should be able to express conditional preferences. The conditional preferences make contextualization of preferences possible.

- **Temporally Extended**: the preference language should be able to express temporally extended preferences. That is it should be able to express preferences that involve multiple states within a plan trajectory.

- **Handle Relative Strength and Inconsistencies**: the preference language should provide a facility to stipulate the relative strength of preferences. It should also be able to encode conflicting constraints or inconsistencies that may arise.

- **Can Benefit from and Benefit the Planning Community**: while it may be possible to design a preference language that satisfies our set of desirable criteria, it would be ideal to leverage (reuse and/or extend) an existing, broadly adopted preference language. This makes it possible to both benefit from existing research in planning and to benefit the planning community.

Our preference language should be well balanced with respect to expressivity and tractability. That is the preference language, while rich, and expressive, should be easy to reason with.

The following is a set of desirable criteria for the specification of hard constraints. Similarly, the hard constraints specification language should be balanced with respect to expressivity and tractability.

- **Easy to Reason and Integrate**: it should be easy to reason and integrate the hard constraints into our system.
- **Pruning**: the hard constraints should facilitate pruning.
- **Benefit from Existing Languages**: the hard constraints can be specified in an existing language, which makes it possible to benefit from the existing research within the community.

4.3 Specifying Preferences in $\mathcal{LPP}$

In this section, we describe a first-order language, called $\mathcal{LPP}$ that we use for specifying user preferences in our system **GOLOGPREF**. We will also discuss how one can specify service selection preferences in $\mathcal{LPP}$. $\mathcal{LPP}$ was proposed in [Bienvenu et al., 2006] for preference-based planning. It is richly expressive, enabling the expression of static as well as the temporal preferences. That is preferences can talk about the temporal relationship between different aspect of the plan and not just about the final state. Furthermore, $\mathcal{LPP}$ is qualitative in nature, facilitating elicitation. Unlike many ordinal preference languages, this language provides a facility to stipulate the relative strength of preferences. $\mathcal{LPP}$ is an extension of the $\mathcal{PP}$ language proposed by Son and Pontelli [Son and Pontelli, 2006] capable of representing temporal and aggregation of preferences. However, $\mathcal{LPP}$ has a different semantics from $\mathcal{PP}$. In addition, the $\mathcal{LPP}$ language provides a total order on preferences. That is not the case with the $\mathcal{PP}$ language, because $\mathcal{PP}$ allows for incomparability.

To help illustrate the expressive power of the $\mathcal{LPP}$ language, we go back to our Travel example. A generic procedure, easily specified in Golog, might say: *In any order, book inter-city transportation, book local accommodations and book local transportation.* With this generic procedure in hand an individual user can specify their hard constraints (e.g., *Lara needs to be in Chicago July 29-Aug 5, 2006.*) together with a list of preferences described in $\mathcal{LPP}$.

To understand the preference language, consider the composition we are trying to generate to be a situation – a sequence of actions or Web services executed from the initial situation. A user specifies his or her preferences in terms of a single, so-called *General Preference Formula*.

This formula is an aggregation of preferences over constituent properties of situations (i.e., compositions). The basic building block of the preference formula is a *Trajectory Property Formula* which describes properties of (partial) situations (i.e., compositions). Furthermore, the so-called *Atomic Preference Formulae* can be used to describe the relative strength or importance of different preferences. The definitions that follow in this section are taken from [Bienvenu et al., 2006, Bienvenu et al., 2011].

Note, in this section we distinguish between the set of fluent predicates, $\mathcal{F}$, and the set of non-fluent predicates, $\mathcal{N}$, representing properties that do not change over time.

Definition 4.1 (Trajectory Property Formula (TPF) [Bienvenu et al., 2006])
A trajectory property formula is a sentence drawn from the smallest set $\mathcal{B}$ where:

1. $\mathcal{F} \subset \mathcal{B}$
2. $\mathcal{N} \subset \mathcal{B}$
3. *If* $f \in \mathcal{F}$, *then* ***final***$(f) \in \mathcal{B}$
4. *If* $a \in \mathcal{A}$, *then* ***occ*** $(a) \in \mathcal{B}$
5. *If* φ_1 *and* φ_2 *are in* $\mathcal{B}$, *then so are* $\neg\varphi_1$, $\varphi_1 \wedge \varphi_2$, $\varphi_1 \vee \varphi_2$, $(\exists x)\varphi_1$, $(\forall x)\varphi_1$, ***next***(φ_1), ***always***(φ_1), ***eventually***(φ_1), *and* ***until***(φ_1, φ_2).

final(f) states that fluent f holds in the final situation, **occ** (a) states that action a occurs in the present situation, and **next**(φ_1), **always**(φ_1), **eventually**(φ_1), and **until**(φ_1, φ_2) are basic Linear Temporal Logic (LTL) [Emerson, 1990] constructs.

TPFs establish properties of preferred situations. By combining TPFs using boolean connectives we are able to express a wide variety of properties of situations. Some examples follow. Note, to simplify the examples many parameters have been suppressed. For legibility, variables are bold faced, constants start with uppercase, we abbreviate **eventually**(**occ**(φ)) by **occ**$'(\varphi)$, and we refer to the preference formulae by their labels.

$$\textbf{final}(at(Home)) \tag{P1}$$

$$(\exists\, \boldsymbol{c}).\textbf{occ}'(bookAir(\boldsymbol{c}, Economy, Direct)) \wedge member(\boldsymbol{c}, StarAlliance) \tag{P2}$$

$$\textbf{always}(\neg((\exists\, \boldsymbol{h}).hotelBooked(\boldsymbol{h}) \wedge hilton(\boldsymbol{h}))) \tag{P3}$$

$$(\exists\, \boldsymbol{h}, \boldsymbol{r}).(\textbf{occ}'(bookHotel(\boldsymbol{h}, \boldsymbol{r})) \wedge paymentOption(\boldsymbol{h}, Visa) \wedge starsGE(\boldsymbol{r}, 3) \tag{P4}$$

P1 states that the user is at home in the final situation. P2 states that at some point the user books a direct economy flight with a Star Alliance carrier. Recall there was no stipulation in the generic procedure regarding the mode of transportation between cities or locally. P3 states

that a Hilton hotel never be booked while P4 states that at some point the user books a hotel that accept Visa credit cards and has a rating of 3 or more.

To define a preference ordering over alternative properties of situations, $\mathcal{LPP}$ defines *Atomic Preference Formulae* (APFs). Each alternative being ordered comprises two components: the property of the situation, specified by a TPF, and a *value* term which stipulates the relative strength of the preference. The incorporation of the value terms reduces the incomparability of user preferences and better captures users preferences.

Definition 4.2 (Atomic Preference Formula (APF) [Bienvenu et al., 2006])
Let $\mathcal{V}$ be a totally ordered set with minimal element v_{min} and maximal element v_{max}. An atomic preference formula is a formula $\varphi_0[v_0] \gg \varphi_1[v_1] \gg ... \gg \varphi_n[v_n]$, where each φ_i is a TPF, each $v_i \in \mathcal{V}$, $v_i < v_j$ for $i < j$, and $v_0 = v_{min}$. When $n = 0$, atomic preference formulae correspond to TPFs.

An APF expresses a preference over alternatives. An APF explicitly refers to a value from the total set $\mathcal{V}$ which is the level of satisfaction described the user. Note that v_{min} is the most preferred and v_{max} is the least preferred. In what follows, we let $\mathcal{V} = [0, 1]$, but we could instead choose a strictly qualitative set like $\{$*best* $<$ *good* $<$ *indifferent* $<$ *bad* $<$ *worst*$\}$ since the operations on these values are limited to max and min. Returning to our example, the following APFs express an ordering over Lara's preferences.

$$
\begin{aligned}
&P2[0] \\
&\quad \gg\ (\exists\, \boldsymbol{c}, \boldsymbol{w}).\mathbf{occ}'(bookAir(\boldsymbol{c}, Economy, \boldsymbol{w}) \wedge member(\boldsymbol{c}, StarAlliance))[0.2] \\
&\qquad \gg \mathbf{occ}'(bookAir(Delta, Economy, Direct))[0.5] && \text{(P5)} \\
&(\exists\, \boldsymbol{t}).\mathbf{occ}'(bookCar(National, \boldsymbol{t}))[0] \\
&\quad \gg (\exists\, \boldsymbol{t}).\mathbf{occ}'(bookCar(Alamo, \boldsymbol{t}))[0.2] \\
&\qquad \gg\ (\exists\, \boldsymbol{t}).\mathbf{occ}'(bookCar(Avis, \boldsymbol{t}))[0.8] && \text{(P6)} \\
&(\exists\, \boldsymbol{c}).\mathbf{occ}'(bookCar(\boldsymbol{c}, SUV))[0] \gg\ (\exists\, \boldsymbol{c}).\mathbf{occ}'(bookCar(\boldsymbol{c}, Compact))[0.2] && \text{(P7)}
\end{aligned}
$$

P5 states that Lara prefers direct economy flights with a Star Alliance carrier, followed by economy flights with a Star Alliance carrier, followed by direct economy flights with Delta airlines. P6 and P7 are preference over cars. Lara strongly prefers National and then Alamo over Avis, followed by all other car-rental companies. Finally she slightly prefers an SUV over a compact with any other type of car a distant third.

To allow the user to specify more complex preferences and to aggregate preferences, *General Preference Formulae* (GFPs) extend the language to conditional, conjunctive, and disjunctive preferences.

Definition 4.3 (General Preference Formula (GPF) [Bienvenu et al., 2006])
A formula Φ is a general preference formula if one of the following holds:

- Φ *is an APF*
- Φ *is* $\gamma : \Psi$*, where* γ *is a TPF and* Ψ *is a GPF [Conditional]*
- Φ *is one of*
  - $\Psi_0 \,\&\, \Psi_1 \,\&\, ... \,\&\, \Psi_n$ *[General Conjunction]*
  - $\Psi_0 \mid \Psi_1 \mid ... \mid \Psi_n$ *[General Disjunction]*

where $n \geq 1$ *and each* Ψ_i *is a GPF.*

Continuing our example:

$$(\forall \boldsymbol{h}, \boldsymbol{c}, \boldsymbol{e}, \boldsymbol{w}).\textbf{always}(\neg hotelBooked(\boldsymbol{h}) : \neg \textbf{occ}'(bookAir(\boldsymbol{c}, \boldsymbol{e}, \boldsymbol{w}))) \quad \text{(P8)}$$

$$far \ : \ P5 \quad \text{(P9)}$$

$$P3 \,\&\, P4 \,\&\, P6 \,\&\, P7 \,\&\, P8 \,\&\, P9 \quad \text{(P10)}$$

P8 states that Lara prefers not to book her air ticket until she has a hotel booked. P9 conditions Lara's airline preferences on her destination being far away. (If it is not far, she will not fly and the preferences are irrelevant.) Finally, P10 aggregates previous preferences into one formula.

4.3.1 The Semantics of $\mathcal{LPP}$

In this section, we briefly overview the semantics of $\mathcal{LPP}$. Understanding the semantics of this language is important because our preference language, $\mathcal{LPH}$ builds on the $\mathcal{LPP}$ language and therefore inherits its semantics for the constructs that they share. In Section 4.4.1 we will discuss the semantics of $\mathcal{LPP}$ while focusing only on the semantics of our extension to $\mathcal{LPP}$.

The semantics of the preference language $\mathcal{LPP}$ is achieved through assigning a weight to a situation s with respect to a GPF, Φ, written $w_s(\Phi)$. This weight is a composition of its constituents. For TPFs, a situation s is assigned the value v_{min} if the TPF is satisfied in s, v_{max} otherwise. Recall that in our example above $v_{min} = 0$ and $v_{max} = 1$, though they could equally well have been a qualitative e.g., [excellent, abysmal]. Similarly, given an APF, and a situation s, s is assigned the weight of the best TPF that it satisfies within the defined APF. Returning

to our example above, for P6 if a situation (composition) booked a car from Alamo rental car, it would get a weight of 0.2. Finally GPF semantics follow the natural semantics of boolean connectives. As such General Conjunction yields the maximum of its constituent GPF weights and General Disjunction yields the minimum of its constituent GPF weights. The definitions that follow are taken from [Bienvenu et al., 2006, Bienvenu et al., 2011]. For a full explanation of the situation calculus semantics, please see [Bienvenu et al., 2006].

Definition 4.4 (Basic Desire Satisfaction [Bienvenu et al., 2006]) *Let $\mathcal{D}$ be an action theory, and let s' and s be situations such that $s' \sqsubseteq s$. The situations beginning in s' and terminating in s satisfy φ just in the case that $\mathcal{D} \models \varphi[s', s]$. We define $w_{s',s}(\varphi)$ to be the weight of the situations originating in s' and ending in s with respect to TPF φ. $w_{s',s}(\varphi) = v_{min}$ if φ is satisfied, otherwise $w_{s',s}(\varphi) = v_{max}$.*

Note that for readability we are going to drop s' from the index, i.e., $w_s(\varphi) = w_{s',s}(\varphi)$ in the special case of $s' = S_0$.

Definition 4.5 (Atomic Preference Satisfaction [Bienvenu et al., 2006]) *Let s be a situation and $\Phi = \varphi_0[v_0] \gg \varphi_1[v_1] \gg ... \gg \varphi_n[v_n]$ be an atomic preference formula. Then $w_s(\Phi) = v_i$ if $\mathcal{D} \models \varphi_i[S_0, s]$ and $\mathcal{D} \not\models \varphi_j[S_0, s]$ for all $0 \leq j \leq i$, and $w_s(\Phi) = v_{max}$ if no such i exists.*

Definition 4.6 (General Preference Satisfaction [Bienvenu et al., 2006]) *Let s be a situation and Φ be a general preference formula. Then $w_s(\Phi)$ is defined as follows:*

- $w_s(\varphi_0[v_0] \gg \varphi_1[v_1] \gg ... \gg \varphi_n[v_n])$ *is defined above*
- $w_s(\gamma : \Psi) = \begin{cases} v_{min} & \text{if } w_s(\gamma) = v_{max} \\ w_s(\Psi) & \text{otherwise} \end{cases}$
- $w_s(\Psi_0 \,\&\, \Psi_1 \,\&\, ... \,\&\, \Psi_n) = \max\{w_s(\Psi_i) : 1 \leq i \leq n\}$
- $w_s(\Psi_0 \mid \Psi_1 \mid ... \mid \Psi_n) = \min\{w_s(\Psi_i) : 1 \leq i \leq n\}$

We conclude this section with the following definition which shows us how to compare two situations (and thus two compositions) with respect to a GPF:

Definition 4.7 (Preferred Situations [Bienvenu et al., 2006]) *A situation s_1 is at least as preferred as a situation s_2 with respect to a GPF Φ, written $pref(s_1, s_2, \Phi)$ if $w_{s_1}(\Phi) \leq w_{s_2}(\Phi)$.*

4.3.2 Integrated Optimal Web Service Selection

Most WSC systems use AI planning techniques and as such generally ignore the important problem of Web service selection or discovery, assuming it will be done by a separate matchmaker. The work presented in this thesis is significant because it enables the selection of services for composition based, not only on their inputs, outputs, preconditions and effects but also based on other non-functional properties. As such, users are able to specify properties of services that they desire alongside other properties of their preferred solution, and services are selected that optimize for the users preferences in the context of the overall composition.

To see how the selection of services can be encoded, we reintroduce the service parameter $\boldsymbol{u}$ which was suppressed from the example in the previous section (discussion of $\mathcal{LPP}$). Revisiting P2, and P4, we see how the selection of a service $\boldsymbol{u}$ is easily realized within our preference framework with preference P2' and P3'.

$$(\exists\, \boldsymbol{c}, \boldsymbol{u}).\mathbf{occ}'(bookAir(\boldsymbol{c}, Economy, Direct, \boldsymbol{u})) \wedge member(\boldsymbol{c}, StarAlliance) \\ \wedge serviceType(\boldsymbol{u}, AirTicketVendor) \wedge sellsTickets(\boldsymbol{u}, \boldsymbol{c}) \quad \text{(P2')}$$

$$(\exists\, \boldsymbol{h}, \boldsymbol{r}, \boldsymbol{u}).\mathbf{occ}'(bookHotel(\boldsymbol{h}, \boldsymbol{r}, \boldsymbol{u})) \wedge paymentOption(\boldsymbol{h}, Visa) \\ \wedge serviceType(\boldsymbol{u}, HotelRoomVendor) \wedge sellsRooms(\boldsymbol{u}, \boldsymbol{h}) \\ \wedge starsGE(\boldsymbol{r}, 3) \quad \text{(P3')}$$

P2' causes **GOLOGPREF** to prefer booking air tickets with an air ticket vendor that sells the tickets of a carrier that is a member of Star Alliance. Similarly, P3' causes **GOLOGPREF** to prefer booking hotel with a hotel vendor that has hotels that accept visa and that have a rating of 3 or more.

4.4 Specifying Preferences in $\mathcal{LPH}$

In this section, we propose the language $\mathcal{LPH}$ that modifies and extends the $\mathcal{LPP}$ qualitative preference language proposed in [Bienvenu et al., 2006] to capture HTN-specific preferences. We use this language to specify the formula ϕ_{soft} that appears in Definition 3.4 in Chapter 3.

In designing a preference specification language over HTNs, we made a number of strategic design decisions. We first considered adding our preference specifications directly to the definitions of HTN methods. This seemed like a natural extension to the hard constraints that are already part of method definitions. Unfortunately, this precludes easy contextualization of

methods relative to the task the method is realizing. For example, in the Travel domain, many methods may eventually involve the primitive operation of *paying*, but a user may prefer different methods of payment dependent upon the high-level task being realized (e.g., *When booking a car, pay with amex to exploit amex's free collision coverage, when booking a flight, pay with my Aeroplan-visa to collect travel bonus points*, etc.). We also found the option of including preferences in method definitions unappealing because we wished to separate domain-specific, but user-independent knowledge, such as method definitions, from user-specific preferences. Separating the two, enables users to share method definitions but individualizes the preferences. This led us to propose $\mathcal{LPH}$, a declarative specification language for qualitative HTN-tailored preferences.

Our $\mathcal{LPH}$ language has the ability to express preferences over certain parameterization of a task (e.g., preferring one task grounding to another), over a particular decomposition of nonprimitive tasks (i.e., prefer to apply a certain method over another), and a soft version of the before, after, and in between constraints. A soft constraint is defined via a preference formula whose evaluation determines when a plan is *more preferred* than another. However, unlike the task network constraints which will prune or eliminate those plans that have not satisfied them, not meeting a soft constraint simplify deems a plan to be of poorer quality.

Below we give an updated definition for Trajectory Property Formula in $\mathcal{LPH}$.

Definition 4.8 (Trajectory Property Formula (TPF)) *A basic desire formula is a sentence drawn from the smallest set $\mathcal{B}$ where:*

1. *If l is a literal, then $l \in \mathcal{B}$ and **final**$(l) \in \mathcal{B}$*
2. *If t is a task, then **occ**$(t) \in \mathcal{B}$*
3. *If m is a method, and $n = name(m)$, then **apply**$(n) \in \mathcal{B}$*
4. *If t_1, and t_2 are tasks, and l is a literal, then **before**(t_1, t_2), **holdBefore**(t_1, l), **holdAfter**(t_1, l), **holdBetween**(t_1, l, t_2) are in $\mathcal{B}$.*
5. *If φ_1 and φ_2 are in $\mathcal{B}$, then so are $\neg\varphi_1$, $\varphi_1 \wedge \varphi_2$, $\varphi_1 \vee \varphi_2$, $(\exists x)\varphi_1$, $(\forall x)\varphi_1$, **next**(φ_1), **always**(φ_1), **eventually**(φ_1), and **until**(φ_1, φ_2).*

final(l) states that the literal l holds in the final state, **occ** (t) states that the task t occurs in the present state, and **next**(φ_1), **always**(φ_1), **eventually**(φ_1), and **until**(φ_1, φ_2) are basic LTL constructs. **apply** (n) states that a method whose name is n is applied to decompose a nonprimitive task. **before**(t_1, t_2) states a precedence ordering between two tasks. **holdBefore**(t_1, l), **holdAfter**(t_1, l), **holdBetween**(t_1, l, t_2) state a soft constraint over when the fluent l is preferred to hold. (i.e., **holdBefore**(t_1, l) state that l must be true right before the last operator descender

of t_1 occurs). Combining $\mathbf{occ}(t)$ with the rest of $\mathcal{LPH}$ language enables the construction of preference statements over parameterizations of tasks.

TPFs establish properties of different states within a plan. By combining TPFs using boolean and temporal connectives, we are able to express other properties of state. The following are a few examples from our Travel domain[1].

$$\mathbf{occ}'(\textit{book-flight}(Economy, Aircanada)) \tag{P1}$$

$$(\exists \boldsymbol{c}).\mathbf{occ}'(\textit{book-car}(\boldsymbol{c}, Enterprise)) \tag{P2}$$

$$\mathbf{apply}'(\textit{by-car-local(SUV, Avis)}) \tag{P3}$$

$$\mathbf{before}(\textit{arrange-trans}, \textit{arrange-acc}) \tag{P4}$$

$$\mathbf{holdBefore}(hotelReservation, \textit{arrange-trans}) \tag{P5}$$

$$\mathbf{always}(\neg(\mathbf{occ}'(\textit{pay}(Mastercard)))) \tag{P6}$$

$$(\exists \boldsymbol{h}, \boldsymbol{r}).\mathbf{occ}'(\textit{book-hotel}(\boldsymbol{h}, \boldsymbol{r})) \wedge \textit{starsGE}(\boldsymbol{r}, 3) \tag{P7}$$

$$(\exists \boldsymbol{c}).\mathbf{occ}'(\textit{book-flight}(\boldsymbol{c}, \textit{Economy}, \textit{Direct}, \textit{WindowSeat})) \wedge \textit{member}(\boldsymbol{c}, \textit{StarAlliance}) \tag{P8}$$

P1 states that the user eventually books an economy flight from Air Canada. P2 states that at some point the user books a car with Enterprise. P3 states that at some point, the *by-car-local* method is applied to book an SUV from Avis. P4 states that the *arrange-trans* task occurs before the *arrange-acc* task. P5 states that the hotel is reserved before transportation is arranged. P6 states that the user never pays by Mastercard. P7 states that at some point the user books a hotel that has a rating of 3 or more. P8 states that at some point the user books a direct economy window-seated flight with a Star Alliance carrier.

These examples show how the language gives users the ability to express their preferences over a particular paramaterization of tasks and also some temporal relation among them. To define a preference ordering over alternative properties of states, *Atomic Preference Formulae* (APFs) are defined. Each alternative comprises two components: the property of the state, specified by a TPF, and a *value* term which stipulates the relative strength of the preference. The definition of APF is the same as in $\mathcal{LPP}$. Here are a few APF examples.

$$P3[0] \gg \mathbf{apply}'(\textit{by-car-local(SUV, National)})[0.3] \tag{P9}$$

$$\mathbf{apply}'(\textit{by-car-trans})[0] \gg \mathbf{apply}'(\textit{by-flight})[0.4] \tag{P10}$$

$$\mathbf{occ}'(\textit{book-train})[0] \gg \mathbf{occ}'(\textit{book-car})[0.4] \tag{P11}$$

[1]Again to simplify the examples many parameters have been suppressed. For legibility, variables are bold faced, constants start with uppercase, we abbreviate **eventually**($\mathbf{occ}(\varphi)$) by $\mathbf{occ}'(\varphi)$, **eventually**(**apply** (φ)) by **apply** $'$, and we refer to the preference formulae by their labels.

P9 states that the user prefers that the *by-car-local* method rents an SUV and that the rental car company Avis is preferred to National. P10 states that the user prefers to decompose the *arrange-trans* task by the method *by-car-trans* rather than the *by-flight* method. Note that the task is implicit in the definition of the method. P11 states that the user prefers traveling by train over renting a car.

As in $\mathcal{LPP}$, *General Preference Formulae* (GPFs) extend the language to conditional, conjunctive, and disjunctive preferences, allowing the user to specify more complex preferences. The definition of GPF in $\mathcal{LPH}$ is the same as in $\mathcal{LPP}$. General conjunction (resp.general disjunction) refines the ordering defined by $\Psi_0 \,\&\, \Psi_1 \,\&\, ... \,\&\, \Psi_n$ (resp. $\Psi_0|\Psi_1|...|\Psi_n$) by sorting indistinguishable states using the lexicograping ordering. Continuing our example:

$$\textbf{occ}(\textit{arrange-trans}) : (\exists \boldsymbol{c}).\textbf{occ}'(\textit{book-car}(\boldsymbol{c}, Avis)) \quad \text{(P12)}$$

$$\textbf{occ}(\textit{arrange-local-trans}) : P2 \quad \text{(P13)}$$

$$drivable : P11[0] \gg \textbf{occ}'(\textit{book-flight})[0.3] \quad \text{(P14)}$$

$$P5 \,\&\, P7 \,\&\, P1 \,\&\, P9 \,\&\, P10 \,\&\, P11 \,\&\, P13 \,\&\, P14 \quad \text{(P15)}$$

P12 states that if inter-city transportation is being arranged then the user prefers to rent a car from Avis. P13 states that if local transportation is being arranged the user prefers Enterprise. P14 states that if the distance between the origin and the destination is drivable then the user prefers to book a train over booking a car over booking a flight. P15 aggregates preferences into one formula.

4.4.1 The Semantics of $\mathcal{LPH}$

To define the semantics of $\mathcal{LPH}$ and help us prove the correctness and optimality of our algorithm, we appeal to Gabaldon's translation of HTN planning into the situation calculus entailment of ConGolog programs discussed in Section 3.5.

Similar to $\mathcal{LPP}$, the semantics of $\mathcal{LPH}$ is achieved through assigning a weight to a situation s with respect to a GPF, Φ, written $w_s(\Phi)$. This weight is a composition of its constituents. For TPFs, a situation s is assigned the value v_{min} if the TPF is satisfied in s, v_{max} otherwise. Similarly, given an APF, and a situation s, s is assigned the weight of the best TPF that it satisfies within the defined APF. Finally GPF semantics follow the natural semantics of boolean connectives. As such General Conjunction yields the minimum of its constituent GPF weights

and General Disjunction yields the maximum.

Similar to Gabaldon [Gabaldon, 2004] and following $\mathcal{LPP}$, we use the notation $\varphi[s', s]$ to denote that φ holds in the sequence of situations starting from s' and terminating in s. Also we use the notation $s' \sqsubset s$ to denote that the sequence s' precedes a situation s, i.e., the sequence s' is a proper prefix of sequence s. Next, we will show how to interpret TPFs in the situation calculus.

If f is a fluent, we will write $f[s', s] = f[s']$ since fluents are represented in situation-suppressed form. If r is a non-fluent, we will have $r[s', s] = r$ since r is already a situation calculus formula. Furthermore, we will write **final**$(f)[s', s] = f[s]$ since **final**(f) means that the fluent f must hold in the final situation.

The TPF **occ**(X) states the occurrence of X which can be either an action or a procedure. written as:

$$\textbf{occ}(X)[s', s] = \begin{cases} do(X, s') \sqsubseteq s & \text{if X} \in \mathcal{A} \\ do(\textit{start}(X), s') \sqsubseteq s & \text{if X} \in \mathcal{R} \end{cases}$$

The TPF **apply**$(P(\boldsymbol{v}))$ will be interpreted as follows:

$$\textbf{apply}(P(\boldsymbol{v}))[s', s] = do(\textit{start}(P(\boldsymbol{v})), s') \sqsubseteq s$$

Boolean connectives and quantifiers are already part of the situation calculus and require no further explanation here. The LTL constructs are interpreted in the same way as in [Gabaldon, 2004]. We interpret the rest of the connectives as follows[2].

$$\textbf{before}(X_1, X_2)[s', s] = (\exists s_1, s_2 : s' \sqsubseteq s_1 \sqsubseteq s_2 \sqsubseteq s)$$
$$\{\textit{terminated}(X_1)[s_1] \wedge \neg \textit{executing}(X_2)[s_1] \wedge \neg \textit{terminated}(X_2)[s_1] \wedge \textbf{occ}(X_2)[s_2, s]\}$$

$$\textbf{holdBefore}(X, f)[s', s] = (\exists s_1 : s' \sqsubseteq s_1 \sqsubseteq s)\{f[s_1] \wedge \textbf{occ}(X)[s_1, s]\}$$

$$\textbf{holdAfter}(X, f)[s', s] = (\exists s_1 : s' \sqsubseteq s_1 \sqsubseteq s)\{\textit{terminated}(X)[s_1] \wedge f[s_1]\}$$

$$\textbf{holdBetween}(X_1, f, X_2)[s', s] = (\exists s_1, s_2 : s' \sqsubseteq s_1 \sqsubseteq s_2 \sqsubseteq s)$$
$$\{\textit{terminated}(X_1)[s_1] \wedge \neg \textit{executing}(X_2)[s_1] \wedge \neg \textit{terminated}(X_2)[s_1] \wedge \textbf{occ}(X_2)[s_2, s]\}$$
$$\wedge (\forall s_i : s_1 \sqsubseteq s_i \sqsubseteq s_2) f[s_i]$$

From here, the semantics follows that of $\mathcal{LPP}$.

[2] We use the following $(\exists s_1 : s' \sqsubseteq s_1 \sqsubseteq s)\Phi = (\exists s_1)\{s' \sqsubseteq s_1 \wedge s_1 \sqsubseteq s \wedge \Phi\}$
$(\forall s_1 : s' \sqsubseteq s_1 \sqsubseteq s)\Phi = (\forall s_1)\{[s' \sqsubseteq s_1 \wedge s_1 \sqsubseteq s] \subset \Phi\}$

4.5 Specifying Preferences in our PDDL3 Extension

In this section, we extend the popular PDDL3 preference language to meet our set of desirable criteria. As argued earlier, supporting preferences over how tasks are decomposed, their preferred parameterizations, and the conditions underwhich these preferences hold, is compelling. It goes beyond the traditional specification of preferences over the properties of states within plan trajectories to provide preferences over non-functional properties of the planning problem including *how* some planning objective is accomplished. This is particularly useful when HTN methods are realized using Web service software components, because these services have many non-functional properties that distinguish them (e.g., credit cards accepted, country of origin, trustworthiness, etc.) and that influence user preferences.

PDDL3 preferences are highly expressive, however they are solely state centric, identifying preferred states along the plan trajectory. To develop a preference language for HTN we add action-centric constructs to PDDL3 that can express preferences over the occurrence of primitive actions (operators) within the plan trajectory, as well as expressing preferences over complex actions (tasks) and how they decompose into primitive actions. For example, we are able to express preferences over which sets of subtasks are preferred in realizing a task (e.g., When booking inter-city transportation, I prefer to book a flight) and preferred parameters to use when choosing a set of subtasks to realize a task (e.g., I prefer to book a flight with United).

4.5.1 Overview of PDDL3

The Planning Domain Definition Language (PDDL) is the de facto standard input language for many planning systems. PDDL3 [Gerevini and Long, 2005, Gerevini et al., 2009] extends PDDL2.2 to support the specification of preferences and hard constraints over *state* properties of a trajectory. These preferences form the building blocks for definition of a PDDL3 *metric function* that defines the quality of a plan. In this context, preference-based planning necessitates maximization (or minimization) of the metric function. In what follows, we describe those elements of PDDL3 that are most relevant to our work. In particular, the current implementation of our preference-based HTN planner does not support the numeric and temporal subset of PDDL3. Namely, the PDDL3 constructs such as *within*, *always-within*, *hold-during*, and *hold-after* are not supported as they explicitly mention time.

Temporally extended preferences/constraints PDDL3 specifies temporally extended preferences and temporally extended hard constraints in a subset of LTL [Emerson, 1990]. Both

are declared using the `:constraints` construct and are interpreted over a sequence of states that result from legal execution of a sequence of actions. *always*, *sometime*, *at-most-once*, *sometime-after*, *at-end sometime-before* are among the constructs allowed in PDDL3. Note that unlike $\mathcal{LPP}$, the temporal operators cannot be nested in PDDL3.

Preferences are given names in their declaration, which are used elsewhere to refer to the preference. The following PDDL3 code illustrates one preference and one hard constraint[3].

```
(forall (?l - light)(preference p-light (sometime (turn-off ?l))))
(always (forall ?x - explosive)(not (holding ?x)))
```

The `p-light` preference advocates that the agent eventually turns off all the lights. The unnamed hard constraint states that an explosive object cannot be held by the agent anywhere in a valid plan.

If a preference is *externally* universally quantified, it defines a family of preferences, comprising an individual preference for each binding of the variables in the quantifier. Thus, preference `p-light` defines an individual preference for each object of type `light` in the domain.

Precondition Preferences Precondition preferences are atemporal formulae that express conditions that ideally should hold in the state in which the action is performed. Precondition preferences are defined as part of the action's precondition.

Simple Preferences Simple preferences express a preference for certain conditions to hold in the final state of the plan. Simple preferences are atemporal formulae and are declared as part of the goal. For example, the following PDDL3 code:

```
(:goal (preference p-truck (at Truck Depot1)))
```

specifies a simple preference (that `truck` is at `Depot1`). Simple preferences can also be quantified.

Metric Function The PDDL3 metric function defines the quality of the plan. PDDL3 defines an `is-violated` function that takes as input a preference `name` and records the number of individual preferences in the `name` family of preferences that have been violated by the plan. Although preferences are boolean formulae, they can be violated *numerous times* if they are scoped by a universal quantifier in their definition. Furthermore, the quality of the plan can also depend on the function `total-time`, which returns the plan length. In our implementation, since we generate sequential plans, this corresponds to the number of actions in the plan.

[3]Note, variables start with "?". Constants start with uppercase.

$\sigma \models$ (*at-end* ϕ)	iff $s_n \models \phi$
$\sigma \models$ (*always* ϕ)	iff $\forall i : 0 \leq i \leq n,\ s_i \models \phi$
$\sigma \models$ (*sometime* ϕ)	iff $\exists i : 0 \leq i \leq n,\ s_i \models \phi$
$\sigma \models$ (*sometime-after* $\phi\ \psi$)	iff $\forall i$ if $s_i \models \phi$ then $\exists j : i \leq j \leq n,\ s_j \models \psi$
$\sigma \models$ (*sometime-before* $\phi\ \psi$)	iff $\forall i$ if $s_i \models \phi$ then $\exists j : 0 \leq j < i,\ s_j \models \psi$

Figure 4.1: Semantics of a subset of PDDL3's temporal formulae. We assume $\sigma = \langle (s_0, t_0) \cdots (s_n, t_n) \rangle$.

Figure 4.1, defines when ϕ is satisfied in a trajectory σ (abbreviated $\sigma \models \phi$) for a subset of the temporal language of PDDL3. Trajectory $\sigma = \langle (s_0, t_0) \cdots (s_n, t_n) \rangle$ represents a sequence of state-time pairs that results from the execution of the sequence of actions in the plan. More details can be found in [Gerevini and Long, 2005, Gerevini et al., 2009].

Finally, it is also possible to define whether we want to maximize or minimize the metric, and how we want to weigh its different components. For example, the PDDL3 metric function:

```
(:metric minimize (+
                  (* 40 (is-violated p-light))
                  (* 20 (is-violated p-truck))))
```

specifies that it is twice as important to satisfy preference `p-light` as to satisfy preference `p-truck`.

Since it is always possible to transform a metric that requires maximization into one that requires minimization, we will assume that the metric is always being *minimized*. Further note that inconsistent preferences are handled automatically using the PDDL metric function as discussed above. The metric function is a weighted sum of individual preference formulae. This function is then minimized by our planning approach. In doing so, it makes an appropriate trade off between inconsistent preferences so that it can optimize the metric function.

Finally, we now complete the formal definition for HTN planning with PDDL3 preferences. Given a PDDL3 metric function M the *HTN preference-based planning problem with PDDL3 preferences* is defined by Definition 3.6, where the relation $\preceq$ is such that $\pi_1 \preceq \pi_2$ if and only if $M(\pi_1) \leq M(\pi_2)$.

4.5.2 PDDL3 Extension for HTN Planning

One of our desirable criteria was adapting and/or extending a specification language that is broadly adopted within the planning community. To that end, we decided to leverage the popularity of PDDL3 as a language for our specifying our preferences.

PDDL3 extends PDDL2.2 to support the specification of preferences and hard constraints over *state* properties of a trajectory. Here, we extend PDDL3 to incorporate complex action-centric preferences over HTN tasks. This gives users the ability to express preferences over certain parameterization of a task (e.g., preferring one task grounding to another or constraints over *action* properties) and over a particular decomposition of nonprimitive tasks (i.e., prefer to apply a certain method over another). In the context of the WSC problem, following the translation of the OWL-S based WSC problem to HTN planning, how to decompose a tasks is analogous to realizing a service using its process model. This is particularly important when the process model is constructed using the Choice construct and users may prefer one choice over another.

As in PDDL3 each preference formula is given a name and a metric value (i.e., penalty if the preference formula is not satisfied). The quality of a plan is defined using a *metric function*. A lower metric value indicates higher satisfaction of the preferences, and vice versa. PDDL3 supports specification of preferences that are temporally extended in a subset of LTL. We extended PDDL3 to give users the ability to express preferences over how to decompose tasks as well as expressing preferences over the preferred parameterization of a task

To support preferences over task occurrences (primitive and nonprimitive) and task decompositions, we added three new constructs to PDDL3: **occ**(a), **initiate**(X) and **terminate**(X), where a is a primitive task (i.e., an operator or an atomic process), and X is either a task (i.e., a composite process' name and its input parameters) or a name of method (i.e., the unique method name assigned for each method during the translation). **occ**(a) states that the primitive task a occurs in the present state. On the other hand **initiate**(t) and **terminate**(t) state, respectively, that the task t is initiated or terminated in the current state. Similarly **initiate**(n) (resp. **terminate**(n)) states that the application of method named n is initiated (resp. terminated) in the current state. These new constructs can be used within simple and temporally extended preferences and constraints, but not within precondition preferences.

The following are a few temporally extended preferences from our Travel domain that use the above extension[4].

```
(preference p1 (always (not (occ (pay MasterCard)))))
(preference p2 (sometime (occ (book-flight SA Eco Direct Window))))
(preference p3 (imply (close origin dest)
                      (sometime (initiate (by-rail-trans)))))
```

[4]For simplicity many parameters have been suppressed. Variables are existentially quantified and start with "?". Constants start with uppercase.

```
(preference p4 (sometime-after (terminate (arrange-trans))
                               (initiate (arrange-acc))))
(preference p5 (sometime-after (terminate arrange-acc)
                               (initiate get-insurance)))
(preference p6 (always (imply (and (hasBookedFlight ?y)
                                   (hasAirline ?y ?x)
                                   (member ?x SA))
                              (sometime (occ (pay ?y CIBC))))))
(preference p7 (always (imply (hasBookedCar ?z)
                              (sometime (occ (pay ?z AE))))))
```

The **p1** preference states that the user never pays by Mastercard. Note here that payment with MasterCard is thought of as an atomic process. The **p2** preference states that at some point the user books a direct economy window-seated flight with a Star Alliance (SA) carrier. Here, booking a flight is believed to be a composite process. The **p3** preference states that the *by-rail-trans* method is applied when origin is close to destination (i.e., the user prefer the train if origin is close to destination). The **p4** states that *arrange-trans* task is terminated before the *arrange-acc* task begins (for example: finish arranging your transportation before booking a hotel). In other words, the user prefers booking their accommodations after their transportation. Similarly, **p5** states that the task *arrange-acc* is terminated before the task *get-insurance* begins. That is, the user prefers getting their insurance after their accommodations. The **p6** preference states that if a flight is booked with a Star Alliance (SA) carrier, pay using the user's CIBC credit card. Finally **p7** preference states that if a car is booked, the user prefers to pay with their American Express (AE) credit card.

4.5.3 The Semantics

The semantics of the preference language (our HTN extension to PDDL3) comprises two parts: (1) a formal definition of the satisfaction of individual preference formulae, and (2) a formal definition of the aggregation of preferences through an objective function. The satisfaction of individual preference formulae is defined by mapping HTN decompositions and LTL formulae into the situation calculus. In so doing, satisfaction of a preference formula is reduced to entailment of the formula in a logical theory. Preference formulae are composed into a metric function. The semantics of the metric function, including the aggregation of quantified preferences via the `is-violated` function, is defined in the same way as in PDDL3, following [Gerevini and Long, 2005].

The satisfaction of the preference formulae is defined by a translation of formulae into the situation calculus. Formulae are satisfied if their translations are entailed by the situation calculus logical theory representing the HTN planning problem and plan. The translation of our HTN constructs are more complex, so we begin with the original elements of PDDL3.

The translation to the situation calculus proceeds as follows. Since we are operating over finite domains, all universally quantified PDDL3 formulae are translated into individual grounded instances of the formulae. Simple preferences (resp. constraints) are translated into the corresponding situation calculus formulae. Temporally extended preferences (resp. constraints) are translated into situation calculus formulae following the translation of LTL formulae into the situation calculus by [Gabaldon, 2004] and [Bienvenu et al., 2006].

To define the semantics of our HTN extension, we again appeal to Gabaldon's translation of HTN planning into the situation calculus entailment of a ConGolog program [Gabaldon, 2002] discussed in Section 3.5. ConGolog is a logic programming language built on top of the situation calculus that supports the expression of complex actions.

In addition to this translation, we need to deal with the new elements of PDDL3 that we introduced: **occ**(a), **initiate**(X), and **terminate**(X). To this end, again following Gabaldon's translation we add two new primitive actions *start*$(P(\boldsymbol{v}))$, *end*$(P(\boldsymbol{v}))$, to each procedure P that corresponds to an HTN task or method. In addition, we add the fluents *executing*$(P(\boldsymbol{v}), s)$ and *terminated*(X, s), where $P(\boldsymbol{v})$ is a ConGolog procedure and X is either $P(\boldsymbol{v})$ or a primitive action a. *executing*$(P(\boldsymbol{v}), s)$ states that $P(\boldsymbol{v})$ is executing in situation s, *terminated*(X, s) states that X has terminated in s. *executing*(a, s) where a is a primitive action, is defined to be false.

occ(a), **initiate**(X), and **terminate**(X) are translated into the situation calculus by building situation calculus formulae that are evaluated when they appear in a preference formula. Below we define these formulae, using a notation compatible with Gabaldon's translation, in which $\varphi[s', s]$ denotes that the (temporal) expression φ holds over the situation fragment s, that starts in situation s'.

occ(a) tells us the first action executed is a: **occ**$(a)[s', s] = do(a, s') \sqsubseteq s$

initiate(X) and **terminate**(X) are interpreted as follows:

$$\textbf{initiate}(X)[s', s] = \begin{cases} do(X, s') \sqsubseteq s & \text{if X} \in \mathcal{A} \\ do(\textit{start}(X), s') \sqsubseteq s & \text{if X} \in \mathcal{R} \end{cases}$$

$$\textbf{terminate}(X)[s', s] = \begin{cases} do(X, s') \sqsubseteq s & \text{if X} \in \mathcal{A} \\ do(\textit{end}(X), s') \sqsubseteq s & \text{if X} \in \mathcal{R}, \end{cases}$$

where $s' \sqsubseteq s$ denotes that situation s' is a predecessor of situation s, and $\mathcal{A}$ is a set containing all primitive actions. The satisfaction of our HTN constructs is now reduced to an entailment.

Definition 4.9 *Given a preference formula φ in one of these forms **occ**(X) **initiate**(X), **terminate**(X), and $\mathcal{D}$, $\mathcal{C}$, Δ, δ_0, the elements of the situation calculus encoding of the preference-based planning problem (see Section 3.5), φ is satisfied if and only if*

$$\mathcal{D} \cup \mathcal{C} \models Do(\Delta; \delta_0, S_0, s) \wedge \varphi[S_0, s]$$

For further details (semantics of other constructs), please see Section 4.4.1, [Gabaldon, 2002] and [Bienvenu et al., 2006].

4.5.4 Service Selection Preferences

Service selection or discovery is a key component of the WSC problem. However, the only other approaches, to our knowledge, that treat this as a preference optimization task *integrated with* actual composition are [Sirin et al., 2005a] and our Golog related work discussed in Section 4.3.2. In [Sirin et al., 2005a], they rely on extending an OWL-S ontology to include abstract processes that refer to service profiles. These descriptions also need to be represented as assertions in an OWL ontology, and an OWL-DL reasoner needs to undertake the task of matching and ranking services based on their service selection preferences. Unfortunately, combining OWL-DL reasoning with planning can create significant performance challenges since one needs to call the OWL-DL reasoner many times during the planning phase, leading to very expensive computations.

We took a different approach to selecting preferred services in our Golog related work discussed in Section 4.3.2. In our approach, we added a service parameter $\boldsymbol{u}$ to each action in the planning problem. Occurrence of this action during the plan with a preferred service parameter argument ensured that a particular service was used. However, we did not discuss how the service parameter was encoded nor we discussed a translation of OWL-S to Golog or to the actions in the planning problem.

Our approach for HTNs is different. Following discussion in the translation of the OWL-S based WSC problem to HTN planning Section 3.4.1, during the translation phase we compile each service profile as an extra property of its corresponding HTN element. Note that not all processes will be associated to a service since a process can correspond to an internal subprocess of the service. We only associate profiles with Web-accessible processes. We cap-

ture the profile property using a binary predicate *isAssociatedWith(process, service-profile*). The service-profile serves as an index for the profile information and is encoded as additional predicates (e.g., *has-language(service-profile, language)*, *has-trust(service-profile, trust)*, *has-reliability(service-profile, reliability)*, etc). Below are some service selection preferences for our Travel domain.

```
(preference p8 (always (imply (and (initiate ?x)
                                   (isAssociatedWith ?x ?y))
                              (has-trust ?y High)))
(preference p9 (sometime (and (initiate ?z)
                              (isAssociatedWith ?z ?y)
                              (has-name ?y AirCanada))))
(preference p10 (always (not (and (initiate ?z)
                                  (isAssociatedWith ?z ?y)
                                  (has-reliability ?y Low)))))
```

p8 states that the user prefers selecting services that have high trust values. **p9** states that a user prefers to invoke the AirCanada service. Lastly, **p10** states that the user prefers to never select low reliability services.

4.6 Specifying Hard Constraints

In addition to preferences, a particular context, user, or problem may necessitate the enforcement of hard constraints. Hard constraints are a useful way of enforcing business rules and policies. We consider and focus specifically on hard constraints in the form of policies or regulations. *Policies* or *regulations* are a set of constraints imposed by an authority that define an acceptable behaviour or characteristic of an agent, person, or an organization. Policies or regulations, hence are a set of constraints imposed on the composition that define an acceptable composition. For example, if commerce is being performed across multiple governmental jurisdictions, there may be a need to ensure that laws and regulations pertaining to commerce are enforced appropriately. In a smart-building setting, we may wish to enforce temperature or lighting policies. To realize this, we must have a means of specifying these further hard constraints.

Policies and regulations are an important aspect of semantic Web services. A number of researchers have proposed approaches to both regulation representation and regulation enforcement as part of semantic Web service tasks (e.g., [Tonti et al., 2003]). Kolovski et al.

[Kolovski et al., 2005] proposed a formal semantics for the WS-policy [WS-Policy, 2006] language by providing a mapping to a Web ontology language OWL and describing how an OWL-DL reasoner could be used to enforce policies. They provided two translations of WS-Policy to OWL-DL by treating policies as instances and classes in the DL framework. Chun et al. [Chun et al., 2005] considered policies imposed on both service selection and on the entire composition, expressed using RuleML [RuleML, 2008]. In their work, policies take the form of condition-action pairs providing an action-centric approach to policy enforcement. There has also been work on compliance checking, using a constraint-based approach that is similar in spirit to regulation enforcement (e.g., [Hoffmann et al., 2009b]).

Also noteworthy is the work of [McIlraith and Son, 2002] in which they represent and reason with hard constraints. However, they provide a way of precompiling these constraints into action preconditions rather than algorithmic pruning at run time.

There has also been work on compliance checking using a constraint-based approach that is similar in spirit to regulation enforcement (e.g., [Hoffmann et al., 2009b]). Also, recent work [Hoffmann et al., 2009a] has considered integrity constraints, and proposed various ways to solve the ramification problem.

There are at least two types of regulations, inter-organizational and intra-organizational [Chun et al., 2004]. Inter-organizational is a type of regulation that is imposed by one organization or a corporation, for example, on their employees or clients. An intra-organizational regulation involves more than one organization and is usually on the whole composition of services. Hence, it is important to have a reasoning framework that can impose regulations not only at design time of the templates but also at the composition construction time.

Regulations are traditionally enforced at design time by verifying that a workflow, composition, or software adheres to the regulations. For example, software developed for use by a particular corporation will have the enforcement of such regulations built in. However, for Web services that are designed for use in multiple scenarios, policies and regulations must be enforced and properties verified in a context-specific manner as new compositions are constructed. Hence, in this thesis, we ensure hard constraints are enforced by simply pruning partial plans that do not adhere to them. This allows enforcement of regulations during composition construction. In Chapter 6, we provide an algorithm that specifies exactly how this pruning occurs within the HTN composition algorithm.

During our regulation enforcement phase, we ensure that the computed composition preserves certain properties of the world. These types of regulations can be specified potentially by state conditions that must hold during the composition. Hence, rather than having action-

centric rules in the form of RuleML or rule-based languages, we are interested in assertions that must be enforced during the composition. Classically this form of verification has been represented in LTL [Emerson, 1990] or some combination of first-order logic with temporal logic (e.g., [Gerth et al., 1995]).

In this thesis, the form of hard constraints we are concerned with does not necessarily provide guidance towards a solution in the same sense as preferences; however, they provide pruning strategies during the composition phase. That is, upon violation of hard constraints, we immediately prune the part of the search space that is disallowed by the hard constraints. This is similar to how temporal formulae provide pruning in TLPlan [Bacchus and Kabanza, 2000] and TALPlan [Kvarnström and Doherty, 2000]. However, the composition template (here the HTN structure) should still be feasible and the imposed regulations should not be in contradiction with or invalidate the HTN structure. For example, in the Travel example, where the desired behaviour is to book a flight and to book an accommodation only, it should not be possible to impose (temporal) policies that mention booking of any other task not mentioned in the HTN structure. For example, imposing a policy that forces booking a car from Avis should not be possible since booking a car is not part of the desired behaviour.

To specify hard constraints, we have restricted our policy and regulation specifications to a subset of LTL that deals with *safety* or *maintenance* properties combined with conditionals. This restriction was a design decision rather than a limitation in our ability to deal with arbitrary LTL formulae. Hence, we represent regulations in a subset of LTL considering for the most part the *never* and *always* modalities. While specifying the hard constraints in this subset could still invalidate the HTN structure, it is easier to specify valid policies with respect to the HTN structure while providing powerful pruning.

Below are some example regulations that corporations might impose on their employees when traveling:

1. Always book flights with US-carriers.
2. Never book business or first-class flights.
3. Get pre-approval for travel outside the US.
4. Always pay for flights and hotels with your corporate credit card.

As an example, the first regulation above can be written in LTL as follows[5]:

$\Box$ `[((hasBookedFlight ?y)` $\wedge$ `(hasAirline ?y ?x))` $\Rightarrow$ `(USCarrier ?x)]`

[5] $\Box$ is a symbol for *always*.

4.7 Summary and Discussion

We now go back to our preference languages and reiterate how they satisfy our set of desirable criteria. Our proposed language $\mathcal{LPH}$ is both action- and state-centric and has the ability to express conditional and temporally extended preferences. $\mathcal{LPH}$ is action-centric because it includes constructs **occ** and **apply**. $\mathcal{LPH}$ has the ability to express preferences over certain parameterization of a task (e.g., preferring one task grounding to another), over a particular decomposition of nonprimitive tasks (i.e., prefer to apply a certain method over another), and a soft version of the *before*, *after*, and *in-between* constraints. Furthermore, through assigning weights $\mathcal{LPH}$ has the ability to express relative strength of preferences. It has a well defined semantics in the situation calculus and is easy to reason with. Finally, the specification of $\mathcal{LPH}$ preferences are disjoint from the specification of the domain.

Our PDDL3 extension for HTN planning is also both action- and state-centric and enables expression of both conditional and temporally extended preferences. Specification of a preference over this language is also done separately from the specification of the domain description. We defined a semantics for our preference language in the situation calculus. Our preference language particularly handles preferences over non-functional properties of services (or service selection preferences). Since our language is an extension to a popular and acceptable preference language we are able to use many of the features already in place. For example, because PDDL3 already handles inconsistencies, inconsistencies for our preference language is automatically handled. Furthermore, we can reuse existing research including an existing compilation technique [Baier et al., 2009] to encode the satisfaction of temporally extended preferences into predicates of the domain.

Finally, we go back to our proposed hard constraints specification language and reiterate how they satisfy our set of desirable criteria. We chose to specify the hard constraints in a subset of LTL. This means that the hard constraints specification language already has a defined semantics, and we are able to benefit from the already existing and acceptable language. Furthermore, we specifically chose this subset of LTL constructs because they easily facilitate pruning techniques.

Chapter 5

Computing Optimized Compositions

5.1 Introduction

In Chapter 3 we established that there is a correspondence between generating a customized composition of Web services and a non-classical planning problem, where the objective of the planning problem can be specified as a composition template together with a set of constraints to be optimized or enforced. While a composition template can be represented in a variety of different ways, in this thesis, we consider two representations: Golog and HTNs. Also in Chapter 4 we discussed different forms of specifying soft and hard constraints. Reflecting on the state-of-the-art techniques for planning with preferences, in this chapter, we address the problem of computing optimal compositions. The approaches we describe differ with respect to the specification of the composition template, Golog or HTNs, the specification of preferences, $\mathcal{LPP}$, $\mathcal{LPH}$, or PDDL3, and the specification of hard constraints. We propose algorithms that integrate preference-based reasoning, exploit and advance state-of-the-art heuristic search, prove properties of these algorithms, implement and evaluate our systems, and ultimately illustrate the applicability of our proposed approaches.

Figure 5.1 summarizes the four systems we discuss in this chapter: **GOLOGPREF**, **HTNPLAN**, **HTNPLAN-P**, and **HTNWSC-P**. Each of these systems uses heuristic search but differ with respect to the forms of the composition templates, specification of soft and hard constraints, and the nature of the heuristics. Heuristic-guided search is an effective method for efficient plan generation. While an admissible heuristic, a heuristic that never overestimates the cost of reaching the goal if used in an A*-like algorithm, guarantees optimality, inadmissible heuristics lead to finding plans quickly, but do not guarantee optimality. **GOLOGPREF** and **HTNPLAN** which use an admissible heuristic, are provably optimal preference-based systems.

System	Composition Template	Soft Constraints	Hard Constraints	Optimal?
GOLOGPREF	Golog	$\mathcal{LPP}$	Golog (or LTL)	Yes
HTNPLAN	HTN	$\mathcal{LPH}$	-	Yes
HTNPLAN-P	HTN	Extended PDDL3	-	Under certain conditions
HTNWSC-P	HTN	Extended PDDL3	LTL	Under certain conditions

Figure 5.1: Summary of our systems.

Indeed, the first solution they generate is the best solution with respect to the given preferences. However, **HTNPLAN-P** and **HTNWSC-P** exploit inadmissible heuristics to compute preferred solutions in a best-first, incremental approach. **HTNPLAN-P** and **HTNWSC-P** perform the search in a series of episodes, returning solutions with increasing quality. We show that despite use of inadmissible heuristics, under some condition we can guarantee optimality.

Furthermore, while **GOLOGPREF** exploits Golog to specify the composition template, the other systems use HTNs to specify the composition template. Unfortunately, the state of the art in HTN planning was not capable of reasoning about preferences. Hence, before addressing the WSC composition with preferences, we first propose incorporating preferences into HTN planning and as a result develop our two preference-based HTN planners, **HTNPLAN** and **HTNPLAN-P**. Both of our preference-based planners are built as an extension to the **SHOP2** planner [Nau et al., 2003].

SHOP2 is a highly optimized domain-independent HTN planning system that received an award for its performance in the 2002 International Planning Competition (IPC-2002). **SHOP2** is different from other HTN planning systems because it plans for tasks in the same order that they later can be executed. To do so **SHOP2** keeps track of the current state of the world at every step of the planning phase and plans forward from the initial state. **SHOP2** allows the task networks to be partially ordered and generates a plan, which is a sequence of actions (instantiated operators) that achieve the initial task network. **SHOP2** has additional features that make it ideal for the WSC problem. Namely, it is able to reason with axioms and has the ability to call external procedures. Nevertheless, **SHOP2** has limited ability to perform preference reasoning. In particular, if there is more than one way of decomposing a task using different methods, **SHOP2** sorts the alternatives based on the order written in the domain description, then it selects the first method whose precondition is satisfied. This if-then-else ordering of **SHOP2** provides a means of reflecting a preference for achieving a task one way over another.

This limited form of preference has to be hard-coded into the **SHOP2** domain description and cannot be customized by an individual user without recoding the HTN domain.

Our implemented preference-based HTN planners are able to take advantage of many of **SHOP2**'s features. Nevertheless, in order to reason with rich expressive preferences, our extension ensures true non-determinism. To do so, our extension to **SHOP2** disables **SHOP2**'s interpretation of the if-then-else statement, enabling all task decompositions to be included in the generation of a plan that addresses to the soft and hard constraints. That is, if a task can be decomposed using two different methods, both of these methods are considered, not just the first applicable one. This broadening of the search space provides the flexibility to support reasoning with soft and hard constraints.

Given our advancements to the state of the art in HTN planning with preferences, we then go back to address the generation of customized composition through planning. Our system **HTNWSC-P** builds on **HTNPLAN-P** and further advances it by being able to enforce hard constraints. We use pruning to eliminate those compositions that violate hard constraints. That is, we ensure the hard constraints are enforced by simply pruning partial plans that do not adhere to them. This allows enforcement of regulations or policies during composition construction time. **HTNWSC-P** is also capable of handling service selection preferences. Hence, our composition framework implemented in **HTNWSC-P** can simultaneously optimize, at run time, the selection of services based on their functional and non-functional properties, while enforcing stated hard constraints. Experimental evaluation supports our claim that our approach can indeed provide a computational basis for the development of effective, state-of-the-art techniques for generating customized compositions of Web services.

5.1.1 Contributions

The following are the main contributions of this chapter.

- Proposed algorithms that can compute optimal solutions both for when the composition template is specified as Golog and as HTNs
- Analyzed and proved properties of these algorithms including correctness and optimality (again for both the Golog and the HTN approach)
- Proposed preference-based heuristic search techniques that are tailored to HTN planning
- Implemented proof-of-concept systems that implement our approaches

- Evaluated the implemented systems

5.2 GOLOGPREF: Computing Optimal Compositions

In this section, we will discuss our system **GOLOGPREF**. **GOLOGPREF** addresses the WSC problem with customization, Definition 3.2, where the composition template is specified in Golog, the soft constraints, ϕ_{soft}, is specified in $\mathcal{LPP}$ [Bienvenu et al., 2006] and the hard constraints, ϕ_{hard}, is specified as a Golog program (or as LTL). Our system can be used to select an optimal solution from among families of solutions that achieve the user's stated objective. **GOLOGPREF**, is implemented in Prolog and for the purposes of evaluation, integrated with a selection of Web services that were appropriate to our test domain.

5.2.1 Algorithm and its Properties

In this section, we present an algorithm for computing optimal compositions and prove properties of our algorithm. **GOLOGPREF**'s algorithm relies on our defined notion of WSC problem with customization (see Definition 3.2). In the discussion of the algorithm, we use C instead of ϕ_{hard} and Φ instead of ϕ_{soft}. Note again that Φ is specified in $\mathcal{LPP}$. In particular, Φ is a GPF that describes the given preferences.

A Golog program places constraints on the situation tree that evolves from S_0. As mentioned in Section 3.3.1 for the purposes of WSC we generally treat iteration as finitely bounded by a parameter k. As such, any implementation of Golog is effectively doing planning in a constrained search space, searching for a legal termination of the Golog program. The actions that define this terminating situation are the plan. In the case of composing Web services, this plan is a composition as shown in Chapter 3.

To compute a composition that satisfies the preferences while meeting the hard constraints, we search through this same constrained search space to find the *best* terminating situation. Our approach, embodied in a system called **GOLOGPREF**, searches for this optimal terminating situation by modifying the **PPLAN** approach to planning with preferences proposed in [Bienvenu et al., 2006].

Note, in **GOLOGPREF** we specify the hard constraint C as a Golog program but in previous work Golog has been extended to deal with temporally extended goals [McIlraith and Son, 2002]. For the rest of this section, we assume C is specified in Golog. In particular, **GOLOGPREF** performs best-first search through the constrained search space re-

sulting from the Golog program, $\delta; C$. Note, this is a constrained search space where the hard constraint C is of type final state goal. The search is guided by an admissible evaluation function that evaluates partial plans with respect to whether they satisfy the preference formula, Φ. The admissible evaluation function is the optimistic evaluation of the preference formula, with the pessimistic evaluation and the plan length used as tie breakers where necessary, in that order. Optimistic (resp. pessimistic) weights are defined based on optimistic (resp. pessimistic) satisfaction of preferences. Optimistic satisfaction assumes that any part of the formula not yet satisfied will eventually be satisfied. Pessimistic satisfaction assumes the opposite.

The preference formula Φ is evaluated over intermediate situations (partial compositions) by exploiting *progression* as described in [Bienvenu et al., 2006]. Informally, progression takes a situation and a temporal logic formula (TLF), evaluates the TLF with respect to the state of the situation, and generates a new formula representing those aspects of the TLF that remain to be satisfied in subsequent situations. Note again that we assume that C is specified in Golog, but if it was specified in LTL, we can similarly use progression to evaluate it.

In **GOLOGPREF**, we likewise transition through the Golog program, now using *Trans*, however with the generation of each new situation term, **GOLOGPREF** progresses the preference formula and computes the weight of the situation (which represents a partial composition) relative to the progressed preference formula, ordering situations on the frontier based on **PPLAN**'s [Bienvenu et al., 2006] admissible evaluation function. **GOLOGPREF** expands situations on the frontier based on this best-first search heuristic until it finds a solution.

Figure 5.2 provides a sketch of the basic **GOLOGPREF** algorithm following from **PPLAN** . The full **GOLOGPREF** algorithm takes as input a 5-tuple $(\mathcal{D}, O, \delta, C, \Phi)$. For ease of explication, our algorithm sketch in Figure 5.2 explicitly identifies the initial situation of $\mathcal{D}$, *init*, the Golog program, $\delta; C$ which we refer to as *pgm* and Φ, which we refer to as *pref*. **GOLOGPREF** returns a sequence of Web services, i.e. a plan, and the weight of that plan. The *frontier* is a list of nodes of the form [*optW*, *pessW*, *pgm*, *partialPlan*, *state*, *pref*], sorted by optimistic weight, pessimistic weight, and then by length. The frontier is initialized to the input program and the empty partial plan, its *optW*, *pessW* corresponds to evaluation of the input preference formula in the initial state and *pref* corresponds to the progression of the preference formula.

On each iteration of the **while** loop, **GOLOGPREF** removes the first node from the frontier and places it in *current*. If the Golog program of *current* is *nil* then the situation associated with this node is a terminating situation. If it is also the case that *optW*=*pessW*, then **GOLOGPREF** returns *current*'s partial plan and weight. Otherwise, it calls the function **EXPAND** with *current's* node as input.

```
GOLOGPREF(init, pgm, pref)
frontier ← INITFRONTIER(init, pgm, pref)
while frontier ≠ ∅
    current ← REMOVEFIRST(frontier)
    % establishes current values for progPgm, partialPlan, state, progPref
    if progPgm=nil and optW=pessW
        return partialPlan, optW
    end if
    neighbours ← EXPAND(progPgm, partialPlan, state, progPref)
    frontier ← SORTNMERGE(neighbours, frontier)
end while
return [], ∞

EXPAND(progPgm, partialPlan, state, progPref) returns a list of new nodes to add to the
frontier. If partialPlan=nil then EXPAND returns [ ]. Otherwise, EXPAND uses Golog's
Trans to determine all the executable actions that are legal transitions of progPgm in state
and to compute the remaining program for each.
It returns a list which contains, for each of these executable actions a a node
  (optW, pessW,newProgPgm, newPartialPlan, newState, newProgPref)
and for each action a leading to a terminating state, a second node
  (realW, realW, nil, newPartialPlan, newState, newProgPref).
```

Figure 5.2: A sketch of the **GOLOGPREF** algorithm.

EXPAND returns a new list of nodes to add to the frontier. If *progPgm* is *nil* then no new nodes are added to the frontier. Otherwise, **EXPAND** generates a new set of nodes of the form [*optW*, *pessW*, *prog*, *partialPlan*, *state*, *pref*]. These nodes are generated for each action that is a legal Golog transition of *pgm* in *state*. For actions leading to terminating states, **EXPAND** also generates a second node of the same form with replacing *optW* and *pessW* with their actual weights. The generated nodes by **EXPAND** are then sorted and merged with the frontier. The nodes are sorted by their *optW*, *pessW*, and then by their length. The while loop terminates if an empty frontier is reached, and in that case we return the empty plan indicating that no solution was found. We next prove the correctness of our algorithm.

Theorem 5.1 (Soundness and Optimality) *Let $\mathcal{P}$=$(\mathcal{D}, O, \delta, C, \Phi)$ be a WSC problem with customization, where δ is specified in Golog. Given as input $\mathcal{P}$,* **GOLOGPREF** *returns a plan $\mathbf{a}$ that is a solution to $(\mathcal{D}, O, \delta, C, \Phi)$, and returns [] (i.e., no solution) otherwise.*

Proof: We first prove that the algorithm terminates[1]. There are two ways that **GOLOGPREF** terminates. First, either the frontier is empty, in which case **GOLOGPREF** returns "[]" (i.e., no solution). Second, if *progPgm* = *nil* (i.e., the progressed program is empty) and *optW* = *pessW*, in which case **GOLOGPREF** returns this plan. We appeal to the fact that the program δ is a tree program. Hence, the nodes generated by expand will eventually hit bottom, and we will eventually run out of nodes and reach the empty frontier. Thus, the algorithm always terminates. Next, we observe that $\boldsymbol{a}$ is a solution to $(\mathcal{D}, O, \delta, C)$. This is a simple proof by cases over *Trans* and *Final*. Next, we prove that $\boldsymbol{a}$ is optimal, by assuming and exploiting the correctness of progression of preference formulae proven in [Bienvenu et al., 2006], the admissibility of our evaluation function, and the bounded size of the search space generated by the Golog program $\delta; C$. Suppose for contradiction that $\boldsymbol{a}$ is not optimal. This mean that there exists a plan $\boldsymbol{a}'$ which is more preferred than $\boldsymbol{a}$ (i.e., has a better weight). That means either (1) a node corresponding to $\boldsymbol{a}'$ was generated and was placed behind the node that corresponds to $\boldsymbol{a}$. But this is a contradiction, as we keep the frontier sorted at all times (i.e., if weight of $\boldsymbol{a}'$ was better, the node that corresponds to it should have been placed in front of the node that corresponds to $\boldsymbol{a}$ not behind). Or (2) there is an ancestor node that corresponds to $\boldsymbol{a}'$ that is behind the node that corresponds to $\boldsymbol{a}$. But this is not possible because of the admissibility of our evaluation function. This complete the proof. ■

5.2.2 Implementation and Evaluation

We have implemented a system that generates a solution to the WSC problem with customization. Our implementation, **GOLOGPREF**, builds on an implementation of **PPLAN** [Bienvenu et al., 2006] and an implementation of IndiGolog [Reiter, 2001] both in SWI Prolog[2]. **GOLOGPREF** interfaces with Web services through the implementation of domain-specific scrapers developed using AgentBuilder 3.2, and AgentRunner 3.2, Web agent design applications developed by Fetch Technologies ©. Among the sites we have scraped are Mapquest, and several air, car and hotel services. The information gathered is collected in XML and then processed by **GOLOGPREF**.

We tested **GOLOGPREF** in the Travel domain. Our tests serve predominantly as a proof of the concept and to illustrate the utility of **GOLOGPREF**. Our generic procedure which is represented in Golog was very simple, allowing flexibility in how it could be instantiated.

[1] This proof is similar to the correctness proof of **PPLAN** [Bienvenu et al., 2006, Bienvenu et al., 2011]

[2] See [Reiter, 2001] for a description of the translation of $\mathcal{D}$ to Prolog.

What follows is an example of the Prolog encoding of a **GOLOGPREF** generic procedure.

```
anyorder[bookAcc, bookCityToCityTranspo, bookLocalTranspo]

proc(bookAcc(Location, Day, Num),
[ stayWithFriends(Location) | bookHotel(Location, Day, Num) ]).

proc(bookLocalTranspo(Location, StartDay, ReturnDay),
[       getRide(Location, StartDay, ReturnDay)  |
        walk(Location)   |   bookCar(Location, StartDay, ReturnDay) ]).

proc(bookCityToCityTranspo(Location, Des, StartDay, ReturnDay),
[       getRide(Location, Des, StartDay, ReturnDay) |
        bookAir(Location, Des, StartDay, ReturnDay) |
        bookCar(Location, Des, StartDay, ReturnDay) ]).
```

We tested our **GOLOGPREF** generic procedure with 3 different user profiles: Jack the impoverished university student, Lara the picky frequent flyer, and Conrad the corporate executive who likes timely luxury travel. Each user lived in Toronto and wanted to be in Chicago for specific days. A set of rich user preferences were defined for each user along the lines of those illustrated in Section 3. These preferences often required access to different Web information, such as driving distances.

Not surprisingly, in all cases, **GOLOGPREF** found the solution to the WSC problem with customization (i.e., an optimal solution) for the user. Compositions varied greatly ranging from Jack who arranged accommodations with friends; checked out the distance to his local destinations and then arranged his local transportation (walking since his local destination was close to where he was staying); then once his accommodations were confirmed, booking an economy air ticket Toronto-Chicago with one stop on US Airways with Expedia. Lara on the other hand, booked a hotel (not Hilton), booked an intermediate-sized car with National, and a direct economy air ticket with Star Alliance partner Air Canada via the Air Canada Web site. The optimality and the diversity of the compositions, all from the same generic procedure, illustrate the flexibility afforded by our approach.

Figure 5.3 shows the number of nodes expanded relative to the search space size for 6 test scenarios. The full search space represents all possible combinations of city-to-city transportation, accommodations and local transportation available to the users which could have

CASE NUMBER	NODES EXPANDED	NODES CONSIDERED	TIME (SEC)	NODES IN FULL SEARCH SPACE
1	104	1700	14.38	28,512
2	102	1647	13.71	28,512
3	27	371	2.06	28,512
4	27	368	2.09	28,512
5	99	1692	14.92	28,512
6	108	1761	14.97	28,512

Figure 5.3: Test results for 6 scenarios run under 64bit Ubuntu Linux with 2.66 GHz CPU.

been considered. These results illustrate the effectiveness of the heuristic used to find optimal compositions.

To conclude, highlights of the **GOLOGPREF** implementation include the use of progression for evaluation of preference formulae and an admissible heuristic to guide best-first search, guaranteeing optimality. We tested GologPref on diverse scenarios applied to the same generic procedure. Our tests serve predominantly as a proof of the concept and to illustrate the utility of **GOLOGPREF**. Results illustrated the diversity of compositions that could be generated from the same generic procedure. The number of nodes expanded by the heuristic search was several orders of magnitude smaller than the grounded search space, illustrating the effectiveness of the heuristic and the Golog program in guiding search.

5.3 HTNPLAN: Computing Optimal Plans

As previously mentioned, we also exploit the use of HTNs to specify the composition template. Unfortunately, the state of the art in HTN planning was not capable of reasoning about preferences. Hence, before addressing the WSC problem with customization where the composition template is specified as HTNs, we first propose two HTN planners with preferences, **HTNPLAN** and **HTNPLAN-P**. Hence, in this section and the next we discuss HTN planning with preferences and in Section 5.5 we return to the WSC composition with customization problem using our advancement in HTN planning with preferences.

In this section, we present our preference-based HTN planner, **HTNPLAN**. In particular, we discuss how **HTNPLAN** addresses the preference-based HTN planning problem in Definition 3.6, where the preference relation, $\preceq$, is defined using our preference language $\mathcal{LPH}$. To compute preferred plans, we propose an approach based on forward-chaining heuristic search. Our heuristic uses an admissible evaluation function measuring the satisfaction of preferences over

partial plans. We use *progression* to evaluate the preference formula satisfaction over partial plans. Our empirical evaluation demonstrates the effectiveness of our **HTNPLAN** heuristics. Recall that we appeal to a situation calculus semantics of our preference language and of HTN planning (see Section 4.4.1 and Section 3.5).

5.3.1 Progression

Given a situation and a temporal formula, progression evaluates the temporal formula with respect to the state of a situation to generate a new formula representing parts of the formula that remain to be satisfied. Progression has been used previously in forward chaining planners such as TLPlan [Bacchus and Kabanza, 2000] and TALPlan [Kvarnström and Doherty, 2000], where the hard constraints in the form of domain control knowledge are efficiently evaluated using progression. In this section, we use progression to evaluate the satisfaction of our preference formula. In particular, we define the progression of the constructs we added/modified from $\mathcal{LPP}$ for our preference language $\mathcal{LPH}$.

To define the progression, similar to [Bienvenu et al., 2006] we add the propositional constants TRUE and FALSE to both the situation calculus and to our set of TPFs, where $\mathcal{D} \models$ TRUE and $\mathcal{D} \not\models$ FALSE for every action theory $\mathcal{D}$. We also add the TPF **occNext**(X), and **applyNext**$(P(\boldsymbol{v}))$ to capture the progression of **occ**(X) and **apply**$(P(\boldsymbol{v}))$. Below we show the progression of the added constructs. Recall $\mathcal{A}$ is a set of primitive actions and $\mathcal{R}$ is the set of ConGolog procedures that correspond to the set of methods.

Definition 5.1 (Progression) *Let s be a situation, and let φ be a TPF. The progression of φ through s, written $\rho_s(\varphi)$, is given by:*

- *If φ=**occ**(X) then $\rho_s(\varphi)$ = **occNext**$(X) \wedge$ **eventually**$(terminated(X))$*
- *If $\varphi =$ **occNext**(X) , then* $\rho_s(\varphi) = \begin{cases} TRUE & \text{if } X \in \mathcal{A} \wedge \mathcal{D} \models \exists s'.s = do(X, s') \\ TRUE & \text{if } X \in \mathcal{R} \wedge \mathcal{D} \models \exists s'.s = do(start(X), s') \\ FALSE & otherwise \end{cases}$
- *If $\varphi =$ **apply**$(P(\boldsymbol{v}))$, then $\rho_s(\varphi) =$ **applyNext**$(P(\boldsymbol{v})) \wedge$ **eventually**$(terminated(P(\boldsymbol{v})))$*
- *If $\varphi =$ **applyNext**$(P(\boldsymbol{v}))$, then* $\rho_s(\varphi) = \begin{cases} TRUE & \text{if } \mathcal{D} \models \exists s'.s = do(start(P(\boldsymbol{v})), s') \\ FALSE & otherwise \end{cases}$
- *If $\varphi =$ **before**(X_1, X_2), **holdBefore**(X, f), **holdAfter**(X, f), or **holdBetween**(X_1, f, X_2),*

$$\textit{then } \rho_s(\varphi) = \begin{cases} \textit{TRUE} & \textit{if } w_s(\varphi) = v_{min} \\ \textit{FALSE} & \textit{otherwise} \end{cases}$$

To see how the other constructs are progressed please refer to [Bienvenu et al., 2006].

5.3.2 Admissible Evaluation Function

In this section, we describe an admissible evaluation function using the notion of *optimistic* and *pessimistic* weights proposed in [Bienvenu et al., 2006] that provide a bound on the best and worst weights of any successor situation with respect to a GPF Φ. Optimistic weights, $w_s^{opt}(\Phi)$ are defined based on optimistic satisfaction of a preference formula while pessimistic weights $w_s^{pess}(\Phi)$) are defined based on pessimistic satisfaction of a preference formula. As before we use the notation $\varphi[s', s]$ to denote that φ holds in the sequence of situations starting from s' and terminating in s. Optimistic satisfaction ($\varphi[s', s]^{opt}$) views optimistically the satisfaction of the preference formula and hence assumes that any parts of the preference formula not yet proven to be false will eventually be satisfied. Assuming the opposite, pessimistic satisfaction ($\varphi[s', s]^{pess}$) views pessimistically the satisfaction of the preference formula and hence assumes that any formula not yet falsified will never get satisfied. The following definitions highlight the key differences between our work (shows how our added constructs are evaluated) and the definitions in [Bienvenu et al., 2006].

$$\textbf{occ}(X)[s', s]^{opt} \stackrel{\text{def}}{=} \begin{cases} do(X, s') \sqsubseteq s \vee s' = s & \text{if X} \in \mathcal{A} \\ do(\textit{start}(X), s') \sqsubseteq s \vee s' = s & \text{if X} \in \mathcal{R} \end{cases}$$

$$\textbf{occ}(X)[s', s]^{pess} \stackrel{\text{def}}{=} \begin{cases} do(X, s') \sqsubseteq s & \text{if X} \in \mathcal{A} \\ do(\textit{start}(X), s') \sqsubseteq s & \text{if X} \in \mathcal{R} \end{cases}$$

$$\textbf{apply}(P(\boldsymbol{v}))[s', s]^{opt} \stackrel{\text{def}}{=} do(\textit{start}(P(\boldsymbol{v})), s') \sqsubseteq s \vee s' = s$$

$$\textbf{apply}(P(\boldsymbol{v}))[s', s]^{pess} \stackrel{\text{def}}{=} do(\textit{start}(P(\boldsymbol{v})), s') \sqsubseteq s$$

If $\varphi = \textbf{before}(X_1, X_2), \textbf{holdBefore}(X, f), \textbf{holdAfter}(X, f)$ $\textbf{holdBetween}(X_1, f, X_2)$,

then $\varphi[s', s]^{opt} \stackrel{\text{def}}{=} \varphi[s', s]^{pess} \stackrel{\text{def}}{=} w_{s',s}(\varphi)$

The following theorem describes some important properties of the optimistic and pes-

simistic weight functions. It has been taken from [Bienvenu et al., 2006].

Theorem 5.2 ([Bienvenu et al., 2006]) *Let $s_n = do([a_1, ..., a_n], S_0), n \geq 0$ be a collection of situations, φ be a TPF, Φ a general preference formula, and $w_s^{opt}(\Phi)$, $w_s^{pess}(\Phi)$ be the optimistic and pessimistic weights of Φ with respect to s. Then for any $0 \leq i \leq j \leq k \leq n$,*
1. $\mathcal{D} \models \varphi[s_i]^{pess} \Rightarrow \mathcal{D} \models \varphi[s_j]$, $\mathcal{D} \not\models \varphi[s_i]^{opt} \Rightarrow \mathcal{D} \not\models \varphi[s_j]$,
2. $\left(w_{s_i}^{opt}(\Phi) = w_{s_i}^{pess}(\Phi)\right) \Rightarrow w_{s_j}(\Phi) = w_{s_i}^{opt}(\Phi) = w_{s_i}^{pess}(\Phi)$,
3. $w_{s_i}^{opt}(\Phi) \leq w_{s_j}^{opt}(\Phi) \leq w_{s_k}(\Phi)$, $w_{s_i}^{pess}(\Phi) \geq w_{s_j}^{pess}(\Phi) \geq w_{s_k}(\Phi)$

Definition 5.2 (Evaluation function. Adapted from [Bienvenu et al., 2006])
Let $s = do(\boldsymbol{a}, S_0)$ be a situation and let Φ be a general preference formula. Then $f_\Phi(s) \stackrel{\text{def}}{=} w_s(\Phi)$ if $\boldsymbol{a}$ is a plan, otherwise $f_\Phi(s) \stackrel{\text{def}}{=} w_s^{opt}(\Phi)$.

Theorem 5.2 states that the optimistic weight is non-decreasing and never over-estimates the real weight. Thus, f_Φ is admissible and when used in best-first search, the search is optimal.

5.3.3 Implementation and Evaluation

In this section, we describe our best-first search, preference-based HTN planner. Figure 5.4 outlines the algorithm. **HTNPLAN** takes as input $\mathcal{P} = (s_0, w, D, \Phi)$ where s_0 is the initial state, w the initial task network (not to be confused by weights), D is the HTN planning domain, and Φ the general preference formula, and returns a sequence of ground primitive operators, i.e., a plan, and the weight of that plan. The *frontier* is a list of nodes of the form [*optW*, *pessW*, w, *partialP*, *s*, *pref*], sorted by optimistic weight, pessimistic weight, and then by plan length. The frontier is initialized to the initial task network w, the empty partial plan, its *optW*, *pessW*, and *pref* corresponding to the progression and evaluation of the input preference formula in the initial state.

On each iteration of the **while** loop, **HTNPLAN** removes the first node from the frontier and places it in *current*. If w is empty, the situation associated with this node is a terminating situation. Then **HTNPLAN** returns *current*'s partial plan and weight. Otherwise, it calls the function **EXPAND** with *current's* node as input.

EXPAND returns a new list of nodes that need to be added to the frontier. The new nodes are sorted by *optW*, *pessW*, and merged with the remainder of the frontier. If *w* is *nil* then the frontier is left as is. Otherwise, it generates a new set of nodes of the form [*optW*, *pessW*, *newW*, *newPartialP*, *newS*, *newProgPref*], one for each legal ground operator that can be reached by performing w using a partial-order forward decomposition procedure (PFD) [Ghallab et al., 2004].

Currently **HTNPLAN** uses **SHOP2** [Nau et al., 2003] as its PFD. Hence, the current implementation of **HTNPLAN** is an implementation of **SHOP2** with user preferences. For each primitive task leading to terminating states, **EXPAND** generates a node of the same form but with *optW* and *pessW* replaced by the actual weight. If we reach the empty frontier, we return the empty plan (i.e., no solution is found).

Theorem 5.3 (Soundness and Optimality) *Let $\mathcal{P}$=(s_0, w, D, Φ) be a HTN planning problem with user preferences. Given as input $\mathcal{P}$, **HTNPLAN** returns a plan π that is a solution to (s_0, w, D, Φ), and returns [] (i.e., no solution) otherwise.*

Proof: We first prove that the algorithm terminates. There are two ways that **HTNPLAN** terminates. First, either the frontier is empty, in which case **HTNPLAN** returns "[]" (i.e., no solution). Second, if $w = \emptyset$ (i.e., the initial task network is empty) and *optW* = *pessW*, in which case **HTNPLAN** returns this plan. We appeal to the fact that PFD procedure is sound and complete. Hence, the nodes generated by expand will eventually hit bottom, and we will eventually run out of nodes and reach the empty frontier. Thus, the algorithm always terminates. Next we prove that π is an optimal plan, by assuming and exploiting the correctness of progression of preference formula, and admissibility of our evaluation function. Suppose for contradiction that π is not optimal. This mean that there exists a plan π' which is more preferred than π (i.e., has a better weight). That means either (1) a node corresponding to π' was generated and was placed behind the node that corresponds to π. But this is a contradiction, as we keep the frontier sorted at all times (i.e., if weight of π' was better the node that corresponds to it should have been placed in front of the node that corresponds to π not behind). Or (2) there is an ancestor node that corresponds to π' that is behind the node that corresponds to π. But this is not possible because of the admissibility of our evaluation function. ■

HTNPLAN is an HTN planner with preferences that computes an optimal plan (solution to the HTN planning problem with preferences). In Section 5.5 we will go back to addressing the WSC problem with customization given our advancement in HTN planning with preferences.

Experiments

We implemented our preference-based HTN planner, **HTNPLAN**, on top of the LISP implementation of **SHOP2** [Nau et al., 2003]. Note again that we ensure true non-determinism. That is if a task can be decomposed using two different methods, then both of these methods are considered, not just the first applicable one. All experiments were run on a Pentium 4 HT,

```
HTNPLAN(s_0, w, D, pref)
frontier ← INITFRONTIER(s_0, w, pref)
while frontier ≠ ∅
    current ← REMOVEFIRST(frontier)
    % establishes values of w, partialP, s, progPref
    if w= ∅ and optW=pessW then
        return partialP, optW
    neighbours ← EXPAND(w, D, partialP, s, progPref)
    frontier ← SORTNMERGE(neighbours, frontier)
return [], ∞
```

Figure 5.4: A sketch of the **HTNPLAN** algorithm.

3GHZ CPU, and 1 GB RAM, with a time limit of 1800 seconds (30 min). Since the optimality of **HTNPLAN**-generated plans was established in Theorem 5.3, our objective was to evaluate the effectiveness of our heuristics in guiding search towards an optimal plan.

Since there are no benchmarks for HTN preference-based planning we tested **HTNPLAN** with ZenoTravel and Logistics domains, which were adapted from the International Planning Competition (IPC). In particular, we augmented these domains with $\mathcal{LPH}$ preference test suites. ZenoTravel domain is taken from the IPC-2002 [Fox and Long, 2002] and the Logistics domain from IPC-2000 [Nau et al., 2001]. The planning competition is a bi-annual event, hosted at the ICAPS planning venue with the objective of providing benchmarks for comparison and evaluation of planning systems, highlighting challenges in the planning community, and proposing new directions for research.

The ZenoTravel domain involves transporting people on aircrafts that can fly at two alternative speeds between locations. In the numeric variant the planes consume fuel at different rates according to the speed of travel, and distances between locations vary. The simple-time variant combines the speed of travel with the associated costs. We used both. The Logistics domain involves transporting packages to different destinations using trucks for delivery within cities and planes for between cities. Some of the preferences we used in the evaluations are as follows: we prefer that the high priority packages be delivered first, we prefer to use trucks with lower gas consumptions, and we prefer certain truck routes to another. The problems become harder as the number of objects and/or number of tasks in the domain increases.

Prob	SHOP2			HTNPLAN			
	# Plan	NE	Time	NE	NC	Time	PL
1	12	172	0.61	78	88	1.19	22
2	155	1628	8.60	448	547	9.45	26
3	230	2234	11.15	76	97	1.05	23
4	230	2234	11.10	361	413	4.67	23
5	485	6331	74.10	240	276	8.14	38
6	487	6226	113.20	1084	1218	63.60	46
7	720	6724	50.46	211	250	4.63	31
8	720	6724	50.90	699	808	13.63	28
9	851	9152	165.22	2689	3066	142.7	40
10	2069	23200	205.10	2290	2733	91.25	34
11	2875	27022	369.20	609	704	17.20	30
12	3956	35789	275.30	304	361	5.10	22
13	>8K	>104K	>1800	150	167	5.64	63
14	>13K	>143K	>1800	2153	2922	80.01	35
15	>13K	>136K	>1800	1624	1910	36.02	29
16	>31K	>293K	>1800	1510	1848	24.80	21

(a) ZenoTravel domain

Prob	SHOP2			HTNPLAN			
	# Plan	NE	Time	NE	NC	Time	PL
1	80	1297	1.27	73	93	0.64	14
2	90	540	0.28	19	24	0.20	12
3	808	4597	4.00	301	404	2.22	18
4	1024	10665	79.95	1626	1820	49.56	42
5	1024	10665	79.95	98	115	2.30	42
6	1260	6320	4.66	130	172	1.04	14
7	2178	15104	17.20	27	32	0.22	20
8	2520	14728	12.47	29	40	0.33	16
9	21776	114548	119.1	866	1163	9.44	15
10	>28K	>264K	>1800	1062	1437	13.21	19
11	>28K	>239K	>1800	1767	2417	32.76	14
12	>30K	>118K	>1800	1417	1925	21.07	20
13	>42K	>368K	>1800	2398	2968	82.62	42
14	>54K	>407K	>1800	858	1088	19.26	33
15	>65K	>428K	>1800	37	48	0.46	24
16	>67K	>376K	>1800	451	618	5.14	22

(b) Logistic domain

Figure 5.5: Our criteria for comparisons are number of Nodes Expanded (NE), number of applied operators; number of Nodes Considered (NC), the number of nodes that were added to the frontier, and time measured in seconds. Note, NC is equal to NE for **SHOP2**. PL is the Plan Length and # Plan is the total number of plans.

In order to evaluate the effectiveness of **HTNPLAN** it would have been appealing to evaluate our planner with a preference-based planner that also makes use of procedural control knowledge. But since no comparable planner exists, and it would not have been fair to compare **HTNPLAN** with a preference-based planner that does not use control knowledge, we compared **HTNPLAN** with **SHOP2**, using a brute-force technique for **SHOP2** to determine an optimal plan. In particular, **SHOP2** generated all plans that satisfied the HTN specification and then evaluated each to find an optimal plan. Note that the times reported for **SHOP2** do not actually include the time for posthoc preference evaluation, so they are lower bounds on the time to compute an optimal plan.

Figure 5.5 reports our experimental results for ZenoTravel and the Logistics domain. The problems varied in preference difficulty and are shown in the order of difficulty with respect to number of possible plans (# Plan) that satisfy the HTN control.

The results show that, in all but the first two cases of the ZenoTravel domain, **SHOP2** required more time to find an optimal plan, and expanded more nodes. In particular, note that in a number of problems, for example problems 13 and 14 **SHOP2** ran out of time (1800 seconds) while **HTNPLAN** found an optimal plan well within the time limit. Also note that **HTNPLAN** expands far fewer nodes in comparison to **SHOP2**, illustrating the effectiveness of our evaluation function in guiding search.

5.4 HTNPLAN-P: Computing High-Quality Plans

In the previous section, we discussed **HTNPLAN**, a provably optimal preference-based planner; however, with large search spaces, finding this optimal plan may not be feasible. As an alternative, to compute preferred plans quickly by exploring inadmissible heuristics, we propose a best-first, incremental search in the plan search space induced by the HTN initial task network. We implemented a system called **HTNPLAN-P** which takes as input, specification of preferences in our PDDL3 extension and computes plans with increasing quality. That is, **HTNPLAN-P** addresses the preference-based HTN planning problem in Definition 3.6, where the preference relation, $\preceq$, is defined through PDDL3 metric function. The two important heuristics we use are the Optimistic Metric Function (OM) and the Lookahead Metric Function (LA). The OM function estimates optimistically the metric value resulting from the current node. LA function estimate the metric of the *best successor* to the current node.

In this section, we first discuss the preprocessing step required before search. Then, we discuss our algorithm, heuristics, and prove the properties of our algorithm. Our evaluation shows

that our implemented HTN planner with preferences, **HTNPLAN-P**, is competitive with the state of the art in preference-based reasoning. In the next section, putting everything together, we go back to address the WSC problem with customization.

5.4.1 Preprocessing HTN problems

Before searching for a most preferred plan, we preprocess the original problem. This is needed in order to make the planning problem more easily manageable by standard planning techniques. We accomplish this objective by removing all of the modal operators appearing in the preferences. The resulting domain, has only final-state preferences, and all preferences refer to state properties.

By converting the temporally extended preferences into final-state preferences, our heuristic functions are only defined in terms of domain predicates, rather than being based on non-standard evaluations of an LTL formula, such as the ones used by other approaches [e.g. [Bienvenu et al., 2006], [Bienvenu et al., 2006]]. Nor do we need to implement specialized algorithms to reason about LTL formulae such as the progression algorithm used by **TLPLAN** [Bacchus and Kabanza, 1998].

Further, by removing the modal operators **occ**, **initiate**, and **terminate** we provide a way to refer to these operators via state predicates. This allows us to use standard HTN planning software as modules of our planner, without needing special modifications such as a mechanism to keep track of the tasks that have been decomposed or the methods that have been applied.

Preprocessing Tasks and Methods

Our preferences can refer to the occurrence of tasks and the application of methods. In order to reason about task occurrences and method applications, we preprocess the methods of our HTN problem. In the compiled problem, for each nonprimitive task t that occurs in some preference of the original problem, there are two new predicates: $executing$-t and $terminated$-t. If $a_0 a_1 \cdots a_n$ is a plan for the problem, and a_i and a_j are respectively the first and last primitive actions that resulted from decomposing t, then $executing$-t is true in all the states in between the application of a_i and a_j, and $terminated$-t is true in all states after a_j. This is accomplished by adding new actions at the beginning and end of each task network in the methods that decompose t. Further, for each primitive task (i.e., operator) t occurring in the preferences, we extend the compiled problem with a new occ-t predicate, such that occ-t is true iff t has just been performed.

Finally, we modify each method m whose name n (i.e., $n = name(m)$) that occurs in some preference. We use two predicates $executing\text{-}n$ and $terminated\text{-}n$, whose updates are realized analogously to their task versions described above.

Preprocessing the Modal Operators

We replace each occurrence of **occ**(t), **initiate**(t), and **terminate**(t) by $occ\text{-}t$ when t is primitive. We replace the occurrence of **initiate**(t) by $executing\text{-}t$, and **terminate**(t) by $terminated\text{-}t$ when t is nonprimitive. Occurrences of **initiate**(n) are replaced by $executing\text{-}n$, and **terminate**(n) by $terminated\text{-}n$.

Up to this point all our preferences exclusively reference predicates of the HTN problem, enabling us to apply standard techniques to simplify the problem further.

Temporally Extended and Precondition Preferences

We use an existing compilation technique [Baier et al., 2009] to encode the satisfaction of temporally extended preferences into predicates of the domain. For each LTL preference φ in the original problem, we generate additional predicates for the compiled domain that encode the various ways in which φ can become true. Indeed, the additional predicates represent a finite-state automaton for φ, where the accepting state of the automaton represents satisfaction of the preference. In our resulting domains, we axiomatically define an *accepting predicate for* φ, which represents the accepting condition of φ's automaton. The accepting predicate is true at a state s if and only if φ is satisfied at s. Quantified preferences are compiled into parametric automata for efficiency.

Precondition preferences, preferences that should ideally hold in the state in which the action is performed, are compiled away as conditional action costs. For each precondition preference, we associate a counter function in the compiled domain, that is incremented whenever an action has been performed violating some of its precondition preferences. This process is exactly the same as the one used in the **HPLAN-P** planner [Baier et al., 2009].

This compilation technique enables us to reason about LTL formulae, something we could have also done using **TLPLAN**'s *progression* algorithm [Bacchus and Kabanza, 1998]. However, because it basically encodes in the new predicates all the possible ways in which the formula can be progressed, it allows us to avoid defining a progression mechanism in our planner. It also allows us to define heuristics that are only be based on the evaluation of domain predicates; this is beneficial, since we can compute these heuristics quickly.

```
1: function HTNPBP(s_0, w_0, D, METRICFN, HEURISTICFN)
2:     frontier ← ⟨s_0, w_0, ∅⟩                                  ▷ initialize frontier
3:     bestMetric ← worst case upper bound
4:     while frontier is not empty do
5:         current ← Extract best element from frontier
6:         ⟨s, w, partialP⟩ ← current
7:         lbound ← METRICBOUNDFN(s)
8:         if lbound < bestMetric then                            ▷ pruning by bounding
9:             if w = ∅ and current's metric < bestMetric then
10:                Output plan partialP
11:                bestMetric ← METRICFN(s)
12:            succ ← successors of current
13:            frontier ← sort and merge succ into frontier
```

Figure 5.6: A sketch of our HTN preference-based planning algorithm.

5.4.2 Algorithm

We address the problem of finding a most preferred decomposition of an HTN by performing a best-first, incremental search in the plan search space induced by the initial task network. The search is performed in a series of *episodes*, each of which returns a sequence of ground primitive operators (i.e., a plan that satisfies the initial task network). During each episode, the search performs branch-and-bound pruning—a search node is pruned from the search space, if we can prove that it will not lead to a plan that is better than the one found in the previous episode. In the first episode no pruning is performed. In each episode, search is guided by *inadmissible heuristics*, designed specifically to guide the search quickly to a good decomposition. The remainder of this section describes the heuristics we use, and the planning algorithm.

Our HTN preference-based planning algorithm outlined in Figure 5.6 performs a best-first, incremental search in the space of decompositions of a given initial task network. It takes as input a planning problem (s_0, w_0, D), a metric function METRICFN, and a heuristic function HEURISTICFN.

The main variables kept by the algorithm are $frontier$ and $bestMetric$. $frontier$ contains the nodes in the search frontier. Each of these nodes is of the form $\langle s, w, partialP\rangle$, where s is a plan state, w is a task network, and $partialP$ is a partial plan. Intuitively, a search node $\langle s, w, partialP\rangle$ indicates that task network w remains to be decomposed in state s, and that state s is reached from the initial state of the planning problem s_0 by performing the sequence of actions $partialP$. $frontier$ is initialized with a single node $\langle s_0, w_0, \emptyset\rangle$, where $\emptyset$ represents the empty plan. Its elements are always sorted according to the function HEURISTICFN. On

the other hand, $bestMetric$ is a variable that stores the metric value of the best plan found so far, and it is initialized to a high value representing a worst case upper bound.

Search is carried out in the main **while** loop. In each iteration, **HTNPLAN-P** extracts the best element from the $frontier$ and places it in $current$. Then, an estimation of a lowerbound of the metric value that can be achieved by decomposing w – $current$'s task network – is computed (Line 7) using the function METRICBOUNDFN. Function METRICBOUNDFN will be computed using the *optimistic metric* function described in the next subsection.

The algorithm *prunes* $current$ from the search space if $lbound$ is greater than or equal to $bestMetric$ (line 8). Otherwise, **HTNPLAN-P** checks whether or not $current$ corresponds to a plan (this happens when its task network is empty). If $current$ corresponds to a plan, the sequence of actions in its tuple is returned and the value of $bestMetric$ is updated.

Finally, all successors to $current$ are computed using the Partial-order Forward Decomposition procedure (PFD) [Ghallab et al., 2004]. However, note that we compute all the successors, that is, instead of choosing one applicable ground instance of an operator or a method, we consider them all, in order to generate all possible successors. The new nodes are then sorted and merged into the $frontier$ based on the HEURISTICFN function. The algorithm terminates when $frontier$ is empty.

5.4.3 Heuristics

Our algorithm searches for a plan in the space of all possible decompositions of the initial task network. HTNs that have been designed specifically to be customizable by user preferences may contain tasks that could be decomposed by a fairly large number of methods. In this scenario, it is essential for the algorithm to be able to evaluate which methods to use to decompose a task in order to get to a reasonably good solution quickly. The heuristics we propose in this section are specifically designed to address this problem. All heuristics are evaluated in a search node $\langle s, w, partialP \rangle$.

Optimistic Metric Function (OM) This function is an estimate of the best metric value achievable by any plan that can result from the decomposition of the current task network w. Its value is computed by evaluating the metric function in s, but assuming that (1) no further precondition preferences will be violated in the future, (2) temporally extended preference that are violated and that can be proved to be unachievable from s are regarded as false, (3) all remaining preferences are regarded as satisfied. To prove that a temporally extended preference p is unachievable from s, OM uses a sufficient condition: it checks whether or not the

automaton for p is currently in a state from which there is no path to an accepting state. Recall that an accepting state is reached when the preference formula is satisfied.

OM provides a lower bound on the best plan extending the partial plan $partialP$ assuming that the metric function is non-decreasing in the number of violated preferences. This is the function used as METRICBOUNDFN in our planner. OM is a variant of "optimistic weight" [Bienvenu et al., 2006].

Pessimistic Metric Function (PM) This function is the dual of OM. While OM regards anything that is not provably violated (regardless of future actions) as satisfied, PM regards anything that is not provably satisfied (regardless of future actions) as violated. Its value is computed by evaluating the metric function in s, but assuming that (1) no further precondition preferences will be violated in the future, (2) temporally extended preferences that are satisfied and that can be proved to be true in any successor of s are regarded as satisfied, (3) all remaining preferences are regarded as violated. To prove that a temporally extended preference p is true in any successor of s, we check whether in the current state of the world the automaton for p would be in an accepting state that is also a sink state, i.e., from which it is not possible to escape, regardless of the actions performed in the future.

For reasonable metric functions (e.g., those that are non-decreasing in the number of violated preferences), PM is monotonically decreasing as more actions are added to $partialP$. PM provides good guidance because it is a measure of assured progress towards the satisfaction of the preferences.

Lookahead Metric Function (LA) This function is an estimate of the metric of the *best successor* to the current node. It is computed by conducting a two-phase search. In the first phase, a search for all possible decompositions of w is performed, up to a certain depth k, where k depends on the domain. In the second phase, for each of the resulting nodes, a single primitive decomposition is computed, using depth-first search. The result of LA is the best metric value among all the fully decomposed nodes. Intuitively, LA estimates the metric value of a node by first performing an exhaustive search for decompositions of the current node, and then by approximating the metric value of the resulting nodes by the metric value of the first primitive decomposition that can be found, a form of sampling of the remainder of the search space.

Depth (D) We use the depth as another heuristic to guide the search. This heuristic does not take into account the preferences. Rather, it encourages the planner to find a decomposition soon. Since the search is guided by the HTN structure, guiding the search toward finding a

Strategy	Check whether	If tied	If tied
No-LA	$OM_1 < OM_2$	$PM_1 < PM_2$	-
LA	$LA_1 < LA_2$	$OM_1 < OM_2$	$PM_1 < PM_2$

Figure 5.7: Strategies to determine whether a node n_1 is better than a node n_2. OM is the optimistic-metric, PM is the pessimistic-metric, and LA is the look-ahead heuristic.

plan using depth is natural. Other HTN planners such as **SHOP2** also use depth or depth-first search to guide the search to find a plan quickly.

The HEURISTICFN function we use in our algorithm corresponds to a *prioritized sequence* of the above heuristics, in which D is always considered first. As such, when comparing two nodes we look at their depths, returning the one that has a higher depth value. If the depths are equal, we use the other heuristics in sequence to break ties. Figure 5.7 outlines the sequences we have used in our experiments. For example, LA breaks a tie between two nodes at equal depth by first comparing the LA function of those nodes, then by comparing the value of OM, and if still tied, by comparing the value of PM.

5.4.4 Optimality and Pruning

Since we are using inadmissible heuristics, we cannot guarantee that the plans we generate are optimal. The only way to do this is to run our algorithm until the space is exhausted. In this case, we prove that the final plan returned is guaranteed to be optimal.

The first property required in the proof is that the pruning of the algorithm is *sound*.

Definition 5.3 (Sound Pruning) *Pruning strategy is sound if and only if whenever a node is pruned (line 8) the metric value of any plan extending this node will exceed the current bound* $bestMetric$*. This means that no state will be incorrectly pruned from the search space.*

Exhaustively searching the search space is not reasonable in most planning domains, however here we are able to exploit properties of our planning problem to make this achievable some of the time. Specifically, most HTN specifications severely restrict the search space so that, relative to a classical planning problem, the search space is exhaustively searchable. Further, in the case where our preference metric function is additive, our OM heuristic function enables us to soundly prune partial plans from our search space (since we use the OM function as METRICBOUNDFN in our planner).

The following is taken from [Baier et al., 2009].

		HTNPLAN-P							
		No-LA		*LA*		**SGPlan$_5$**		**HPLAN-P**	
	#Prb	#S	#Best	#S	#Best	#S	#Best	#S	#Best
Travel	41	41	3	41	**37**	41	1	41	17
Rovers	20	20	4	20	**19**	20	1	11	2
Trucks	20	20	6	20	**15**	20	11	4	2

Figure 5.8: Comparison between two configurations of **HTNPLAN-P**, **HPLAN-P**, and **SGPlan$_5$** on Rovers, Trucks, and Travel domains. Entries show number of problems in each domain (#Prb), number of solved instances in each domain (#S) by each planner, and number of times each planner found a plan of equal or better quality to those found by all other planners (#Best). All planners were run for 60 minutes, and with a limit of 2GB per process.

Proposition 5.1 ([Baier et al., 2009]) *The OM function provides sounds pruning if the metric function is non-decreasing in the number of satisfied preferences, non-decreasing in plan length, and independent of other state properties.*

A metric is non-decreasing in plan length if one cannot make a plan better by increasing its length only (without satisfying additional preferences).

Theorem 5.4 *If the algorithm performs sound pruning, then the last plan returned, if any, is optimal.*

Proof: The proof follows the proof of optimality for the **HPLAN-P** planner [Baier et al., 2009]. We know each planning episode returns a better plan. We also know that the algorithm stops only when the final planning episode has rejected all possible plans. Moreover, because of sound pruning, the algorithm never prunes states incorrectly from the search space. Therefore, no better plan than the last returned plan exists. ■

5.4.5 Implementation and Evaluation

Our implemented HTN preference-based planner, **HTNPLAN-P**, has two modules: a preprocessor and a preference-based HTN planner. The preprocessor reads PDDL3 problems and generates a **SHOP2**-style planning problem with only simple (final-state) preferences. The planner itself is a modification of the LISP version of **SHOP2** [Nau et al., 2003] that implements the algorithm and heuristics described above. Our modification ensures true non-determinism that is if a task can be decomposed using two different methods, then both of these methods are considered, not just the first applicable one.

We had three objectives in performing our experimental evaluation: to evaluate the relative effectiveness of our heuristics, to compare our planner with state-of-the-art preference-based planners, and to compare our planner with other HTN preference-based planners. As shown in Figure 5.7 we evaluated the effectiveness of our various heuristics in obtaining plans with good quality (i.e., with low metric value). We compared **HTNPLAN-P** with **SGPlan**$_5$ and **HPLAN-P**– the top performers in the IPC-2006 preferences track. Unfortunately, we were unable to achieve our third objective, since we could not obtain a copy of **SCUP**, the only HTN preference-based planner we know of [Lin et al., 2008] (see Chapter 2).

We used three domains for the evaluation: the Rovers domain, the Trucks domain, both standard IPC benchmark domains; and the Travel domain, which is a domain of our own making. In the Rovers domain the goal is to sample scientific data such as rock and soil, or take images from different objects by navigating rovers between the different surfaces. However, a rover can only traverse between different waypoints if there is a visible path from the source to the destination. In addition, each rover is equipped for either soil, rock, image, or some combination of them. In the Trucks domain, similar to the Logistic domain, the goal is to move packages between locations by trucks. However, the packages need to be delivered under certain constraints (deadlines).

Both the Rovers and Trucks domains comprised the preferences from IPC-2006, qualitative preferences track, where preferences were specified in PDDL3. In Rovers domain we used the HTN designed by the developers of **SHOP2** for IPC-2000 and in Trucks we created our own HTN. We modified the HTN in Rovers very slightly to reflect the true non-determinism in our **HTNPLAN-P** planner: i.e., if a task could be decomposed using two different methods, then both methods would be considered, not just the first applicable one. We also modified the IPC-2006 preferences slightly to ensure fair comparison between planners. In particular, we removed preferences that could never be achieved because of the constraints imposed by the HTN. Such preferences would always cause the planner to exhaustively search the entire search space before termination. For example, if the HTN task is to collect rock from location *waypoint3*, a preference for collecting rocks from other locations will never be met because the HTN structure will never consider sampling rock from other locations. The Rovers and Trucks problems sets comprised 20 problems. The number of preferences in these problem sets ranged in size, with several having over 100 preferences per problem instance.

The Travel domain is a PDDL3 formulation of the domain used throughout this thesis. Its problem set was designed in order to evaluate the preference-based planning approaches based on two dimensions: (1) scalability, which we achieved by increasing the branching factor and

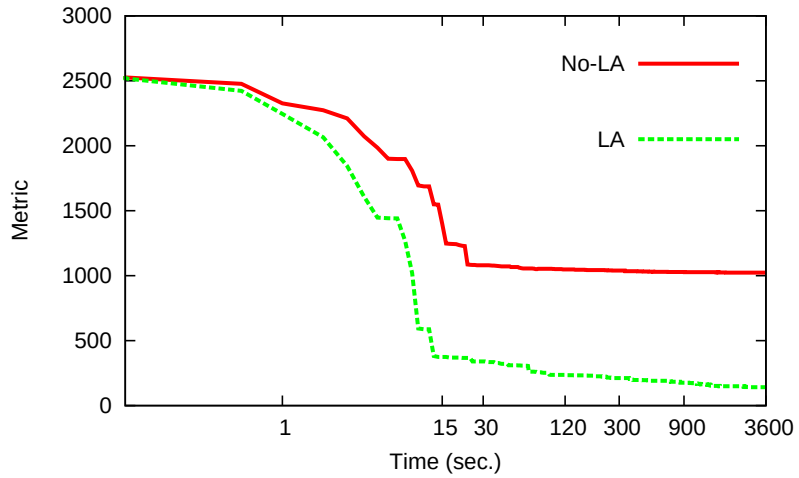

Figure 5.9: Added metric vs. time for the two strategies in the Trucks domain. Recall that a low metric value means higher quality plan. When a problem is not solved at time t, we add its worst possible metric value (i.e. we assume no satisfied preferences).

grounding options of the domain, and (2) the complexity of the preferences, which we achieved by injecting inconsistencies (i.e., conflicts) among the preferences. In particular, we created 41 problems with preferences generated automatically with increasing complexity. For example problem 3 has 27 preferences with 8 conflicts in the choice of transportation while problem 40 has 134 preferences with 54 conflicts in the choice of transportation.

Our experiments evaluated the performance of four planners: **HTNPLAN-P** with the *No-LA* heuristic, and **HTNPLAN-P** with the *LA* heuristic, **SGPlan**$_5$ [Hsu et al., 2007], and **HPLAN-P**– the latter two being the top preference-based planners at IPC-2006. Each planner was run on 41 problems in the Travel domain, and 20 problems in the Rovers domain. Results are summarized in Figure 5.8, and show that **HTNPLAN-P** generated plans that in all but a few cases equalled or exceeded the quality of plans returned by **HPLAN-P** and **SGPlan**$_5$. The results also show that **HTNPLAN-P** performs better on the three domains with the *LA* heuristic.

Conducting the search in a series of episodes does help in finding better-quality plans. To evaluate this, we calculated the *percent metric improvement* (PMI), i.e., the percent difference between the metric of the first and the last plan returned by our planner (relative to the first plan). The average PMI is 40% in Rovers, 72% in Trucks, and 8% in Travel.

To compare the relative performance between *LA* and *No-LA*, we averaged the percent metric difference (relative to the worst plan) in problems in which the configurations found a different plan. This difference is 45% in Rovers, 60% in Trucks, and 3% in Travel, all in favour

of *LA*. We also created 18 instances of the Travel domain where we tested the performance between *LA* and *No-LA* on problems with preferences represented in our HTN extension to PDDL3. The average PMI for these problems is 13%, and the relative performance between the two is 5%.

Finally, Figure 5.9 shows the decrease of the sum of the metric value of all instances of the Trucks domain relative to solving time. We observe a rapid improvement during the first seconds of search, followed by a marginal one after 900 seconds. Other domains exhibit similar behaviour.

5.5 HTNWSC-P: Computing High-Quality Compositions

In the previous sections we addressed the problem of HTN planning with preferences by proposing two preference-based HTN planners. In this section, putting all the different required pieces together we go back to address the problem of how to compute a preferred composition while enforcing the hard constraints. In particular, we present our system, **HTNWSC-P**, that addresses the WSC composition with customization, Definition 3.8, where the composition template is specified as HTNs, the soft constraints, ϕ_{soft}, is specified in our PDDL3 extension and the hard constraints, ϕ_{hard}, is specified in a subset of LTL. Our system builds on **HTNPLAN-P**'s algorithm and is able to use pruning to eliminate those compositions that violate the hard constraints. That is, we ensure the hard constraints (e.g., regulations or policies) are enforced by simply pruning partial plans that do not adhere to them. (I.e., upon violation of the hard constraints, we immediately prune that part of the search space). This allows enforcement of the hard constraints during composition construction time.

Our composition framework, embodied in our system **HTNWSC-P**, can simultaneously optimize, at run time, the selection of services based on functional and non-functional properties and their groundings, while enforcing stated hard constraints. It also explores the use of the heuristic search discussed in Section 5.4.3. Our implementation now combines the HTN representation of composition template, the optimization of rich user preferences specified in our PDDL3 extension, and adherence to LTL hard constraints all within one system. Experimental evaluation on our system, **HTNWSC-P**, shows that our approach can be scaled as we increase the number of preferences and the number of services.

```
1: function HTNWSC(s_0, w_0, D, METRICFN, HEURISTICFN, HARDCONSTRAINTS)
2:     frontier ← ⟨s_0, w_0, ∅⟩                                  ▷ initialize frontier
3:     bestMetric ← worst case upper bound
4:     while frontier is not empty do
5:         current ← Extract best element from frontier
6:         ⟨s, w, partialP⟩ ← current
7:         if SATISFIESHARDCONSTRAINTS(S) then                   ▷ pruning to enforce hard constraints
8:             lbound ← METRICBOUNDFN(s)
9:             if lbound < bestMetric then                       ▷ pruning suboptimal partial plans
10:                if w = ∅ and current's metric < bestMetric then
11:                    Output plan partialP
12:                    bestMetric ← METRICFN(s)
13:                succ ← successors of current
14:                frontier ← sort and merge succ into frontier
```

Figure 5.10: A sketch of our HTNWSC algorithm.

5.5.1 Algorithm

Our algorithm is outlined in Figure 5.10. Our HTNWSC planner performs best-first, incremental search (i.e., always improves on the quality of the plans returned). This algorithm is an extension to the Algorithm 5.6, where we added line 7. If the state violates the hard constraints (i.e., SATISFIESHARDCONSTRAINTS(S) returns false), this node will be pruned from the search space. In the case that hard constraints are expressed in LTL, they are enforced by progressing the formula as the plan is constructed (e.g., [Bacchus and Kabanza, 2000]). The rest of the algorithm is the same as before.

Although the HTN representation of the composition template greatly reduces the search space, a task can be decomposed by a fairly large number of methods corresponding to a large number of services that can carry out the same task. Hence, we use the heuristics proposed in Section 5.4.3 to guide the search towards finding a high-quality composition quickly. The HEURISTICFN function we use in our algorithm is a *prioritized sequence* of our heuristics. However, as shown in the previous section the best combination is to use D, LA, OM, and PM, in that order, when comparing two nodes. Hence, if the depths are equal, we use the other heuristics in sequence to break ties. We will use this prioritized sequence in our evaluations.

The search space for a composition is reduced by the HTN specification of the composition template, imposing the hard constraints, and by further sound pruning that results from the incremental search. In particular, the OM function provides sound pruning if the metric function is non-decreasing in the number of satisfied preferences, non-decreasing in plan length, and

independent of other state properties. A metric is non-decreasing in plan length if one cannot make a plan better by increasing its length only (without satisfying additional preferences).

Using inadmissible heuristics does not guarantee generation of an optimal plan. However, we have shown in the previous section that in the case the search is exhausted, the last plan returned is guaranteed to be optimal. In our algorithm we are pruning those states that violate the hard constraints, so optimality is with respect to the subset of plans that adhere to the hard constraints.

Proposition 5.2 *If the algorithm performs sound pruning, then the last plan returned, if any, is optimal.*

Proof: The proof follows the proof of optimality for the **HPLAN-P** planner [Baier et al., 2009] and is similar to the proof of Theorem 5.4. We know each planning episode returns a better plan. We also know that the algorithm stops only when the final planning episode has rejected all possible plans. Moreover, because of sound pruning, pruning by the OM heuristic and by the enforcement of the hard constraints, the algorithm never prunes states incorrectly from the search space. Therefore, no better plan than the last returned plan exists. ■

5.5.2 Implementation and Evaluation

We implemented our WSC system using the HTN representation of the composition template, preferences specified in PDDL3 syntax, hard constraints specified as LTL formulae, and the user's initial task specified as HTN's initial task network. Our implementation, **HTNWSC-P**, builds on our earlier work **HTNPLAN-P** and implements the algorithm and heuristic described above. We used a 15 minute time out and a limit of 1 GB per process in all our experiments.

HTNWSC-P builds on the effective search techniques for **HTNPLAN-P**, which was shown to generate better quality plans faster than the leading planners from the IPC-2006 planning competition. We had three main objectives in performing our experimental evaluation. We wanted to evaluate the performance of our implementation as we increased the number of preferences and the number of services. We also wanted to compare our work with other WSC planners that use HTNs. Unfortunately, we were unable to achieve our third objective, since we could not obtain a copy of **SCUP** [Lin et al., 2008], the only other HTN planner with preferences we know of (see Chapter 2).

We used the Travel domain described in this thesis as our benchmark. Note that **HTNPLAN-P** was additionally evaluated with IPC-2006 planning domains as discussed in

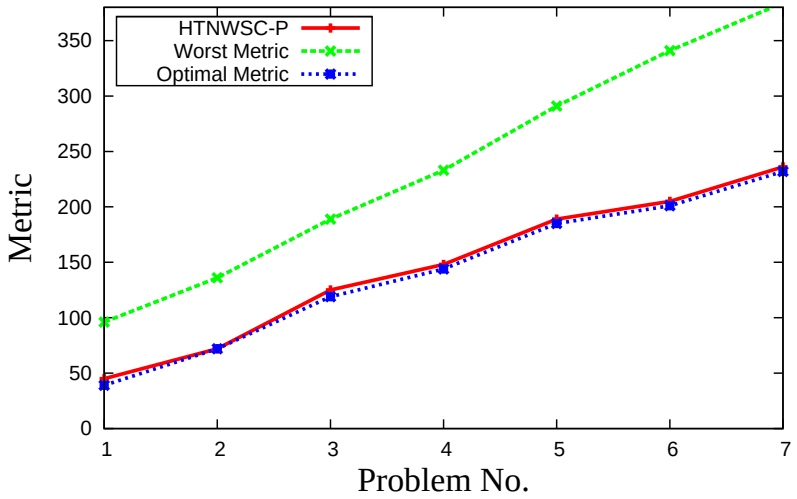

Figure 5.11: Evaluating the quality of the last plan as the number of preferences increases. A low metric value means higher quality plan. Worst Metric is a metric value if none of the preferences are satisfied.

Section 5.4.5. The problem sets we used were designed to help us achieve our first and second objectives. We achieved this by adding more preferences some of which could potentially be conflicting with each other, and by increasing the number of services, achieved by increasing the branching factor and grounding options of the domain. To this end, we automatically generated 7 problems where the number of services was kept constant and the number of preferences was increased. We similarly generated 10 problems with increasing number of services, keeping the number of preferences constant. The preferences were rich, temporally extended preferences over task groundings and task decompositions. Note that we used a constant number of hard constraints in each problem.

Figure 5.11 shows the last metric value returned by **HTNWSC-P** for the 7 problems with increasing number of preferences and constant number of services. It also shows the Worst and Optimal Metric value for these problems. Worst Metric is the metric value of the problem if none of the preferences are satisfied while Optimal Metric is the best possible metric value achievable. The result shows that **HTNWSC-P** finds a very close to optimal solution within the time limit. Furthermore, similar to the **HTNPLAN-P**'s performance in Section 5.4.5, we observe a rapid improvement during the first seconds of search, followed by a marginal one after that.

Next, we evaluated the performance of **HTNWSC-P** by increasing the number of available services. This results in having more methods and operators in the HTN description, hence,

Problem #	Service #	First Plan Time(s)	Last Plan Time(s)
1	10	0.22	580.00
2	30	0.23	610.00
3	50	0.21	636.00
4	70	0.22	640.00
5	110	0.23	643.00
6	130	0.24	656.00
7	150	0.24	660.00
8	170	0.26	668.00
9	190	0.24	671.00
10	210	0.25	673.00

Figure 5.12: Time comparison between the first and last plan returned as we increase the number of services in the problem.

the number of possible ways to decompose a single task increases. This causes the number of nodes in the $frontier$ to blow up according to the algorithm described in Section 5, and the planner to run out of stack. There are two common ways HTN planners solve this problem. Combining the advantages of both, we propose a middle-ground solution to the problem.

One way to avoid the problem is to have a limit on the size of the $frontier$ as in [Lin et al., 2008]. However, this approach only works if the size is relatively small. Moreover, many possible decompositions and high-quality solutions could potentially be removed from the search space. Another approach is to use the if-then-else non-determinism semantics taken for example by **SHOP2**. In this semantics, if there are several methods m_1 to m_k that can be used to decompose a task t, method $m1$ should be used if it is applicable, else method m_2, else method m_3, and so forth. Hence, the order in which the methods are written in the domain description can influence the quality of the results. This simple ordering is considered a form of user preferences in [Nau et al., 2003]. Hence, users must write different versions of a domain description to specify their preferences. However, this form of preferences is very limited and is analogous to writing different composition templates for different users as opposed to customizing one fixed, generic composition template to meet users' differing needs.

In this experiment, we employed a combination of the above two approaches, modifying our algorithm to place a limit on the number of applicable methods for a task. Our search considered *all* tasks by considering all of their corresponding nodes in the $frontier$, but we limited the number of applicable methods for each task. Note that with this approach we might also

potentially prune good-quality plans, but the likelihood of this is small compared to limiting the size of the frontier. Nevertheless, our optimality result does not hold for this experiment. Our results are summarized in Figure 5.12. In Chapter 6 we propose another solution, within the context of our middle-ground execution that allows us to soundly perform what we call localized data optimization, whenever possible. Doing so will ensure that optimality results hold while we improve on the performance of **HTNWSC-P**.

Figure 5.12 shows the time to find the first and the last plan within the time-out. The experiments are run on the 10 problem sets with constant preferences and increasing service numbers. Note that the metric value of all the first and last plans is equal since all 10 problems use the same sets of preferences. The result shows that as the number of services increases, the time to solve the problem increases only slightly.

Finally, recall that our implementation is incremental, performing search in a series, each one returning a better-quality plan. To see how effective this approach is, we calculated the *percent metric improvement* (PMI), i.e., the percent difference between the metric of the first and the last plan returned by our planner (relative to the first plan). The average PMI for the problems used in our experiments is 23%.

5.6 Summary and Discussion

In this chapter, we address the problem of computing optimal compositions. The approaches we describe differ with respect to the specification of the composition template, Golog or HTNs, the specification of preferences, $\mathcal{LPP}$, $\mathcal{LPH}$, or PDDL3, the specification of hard constraints, and the nature of their heuristics. Going back to Figure 5.1 we now summarize the four systems discussed in this chapter.

We first explored the use of Golog to represent the composition template in our system **GOLOGPREF**. In **GOLOGPREF** preferences were in $\mathcal{LPP}$ and we explored the use of admissible heuristics in forward search. **GOLOGPREF** is proven to generate an optimal solution. Unfortunately, the implementation of the system was not optimized and **GOLOGPREF** served as a proof-of-concept system. We then explored the use of HTNs to represent the composition template and to that end advanced state-of-the-art techniques in HTN planning with preferences. We implemented two systems **HTNPLAN** and **HTNPLAN-P** both of which are built as extensions of a modified **SHOP2** planner. **HTNPLAN** is a provably optimal planner that takes as input the specification of preferences in $\mathcal{LPH}$, and uses progression to evaluate the satisfaction of the given preferences.

We also implemented a system, **HTNPLAN-P**, that takes as input specification of preferences in our PDDL3 extension and computes plans with increasing quality. **HTNPLAN-P** also explores the use of our proposed heuristics and in particular, guides the search by our inadmissible heuristics, designed specifically to guide the search quickly to a good decomposition. The experimental evaluations of our planner shows that our HTN preference-based planner, **HTNPLAN-P**, generates plans that, in all but a few cases, equal or exceed the best preference-based planners in plan quality.

Finally, given our advancements to the state of the art in HTN planning with preferences, we put everything together and go back to address the generation of customized composition through planning. We build our WSC system, **HTNWSC-P**, on top of **HTNPLAN-P**. **HTNWSC-P** is able to handle service selection preferences and enforces hard constraints through pruning. Through experimental evaluation, we show that our approach to WSC is viable and promising. In particular, we show how our techniques can indeed provide a computational basis for the development of effective, state-of-the-art techniques for generating customized compositions of Web services.

Most of the related work has already been discussed in Chapter 2. Here, we review some of the most relevant pieces.

Preference-based planning has been the subject of much interest, spurred on by three IPC-2006 tracks on this subject. A number of planners were developed, all based on the competition's PDDL3 language [Gerevini and Long, 2005]. [Baier et al., 2007], [Hsu et al., 2006], [Edelkamp, 2006] are but a few of the preference-based planning systems based on the competition's PDDL3 language [Gerevini et al., 2009]. A few other planners, that explore the notion of preferences, have emerged more recently (e.g., [Benton et al., 2009, Coles and Coles, 2011, Benton et al., 2012]). The work described in this thesis is distinguished in that it employs Golog/HTN domain control extending PDDL3 with HTN-inspired action-centric constructs.

With respect to advisable HTN planners, Myers was the first to advocate augmenting HTN planning with hard constraints to capture advice on *how to* decompose HTNs, extending the approach to conflicting advice in [Myers, 2000]. Their work is similar in vision and spirit to our work, but different with respect to the realization. In their work, preferences are limited to consistent combinations of HTN advice; they do not include the rich temporally extended state-centric preferences found in PDDL3, nor do they support the weighted combination of preferences into a metric function that defines plan quality. With respect to computing HTN preference-based planning, Myers' algorithm does not exploit lookahead heuristics or sound pruning techniques.

The most notable and closest work to ours that uses both HTNs and preferences developed for IPC-2006 is [Lin et al., 2008]. Unfortunately, the **SCUP** prototype planner is not available for experimental comparison. There are several differences among our works. In particular, they translate user preferences into HTN constraints and preprocess the preferences to check if additional tasks need to be added. They also have an interesting approach to the problem by combining HTN planning with Description Logic (DL), and by using a DL reasoner. However, their preferences are specified in PDDL3, while our preferences can be expressed in the PDDL3 extension that uses HTN-specific preference constructs. Moreover, they do not translate service profiles; hence, they are unable to specify preferences over service selections. Additionally, they do not consider handling regulations, a hallmark of our work. Further, their algorithm cannot handle conflicting user preferences at run-time, and so conflicts need to be detected as a pre-processing step.

Also related is the **ASPEN** planner [Rabideau et al., 2000], which performs a simple form of preference-based planning, focused mainly on preferences over resources. It can plan with HTN-like task decomposition, but its preference language is far less expressive than ours. In contrast to our planners, **ASPEN** performs local search for a local optimum. It does not perform well when preferences are interacting, nested, or not local to a specific activity.

It is important to note that the HTN planners **SHOP2** [Nau et al., 2003] and **ENQUIRER** [Kuter et al., 2004] can be seen to handle some simple user preferences. In particular the order of methods and sorted preconditions in a domain description specifies a user preference over which method is more preferred to decompose a task. Hence users may write different versions of a domain description to specify simple preferences. However, unlike our approach the user constraints are treated as hard constraints and (partial) plans that do not meet these constraints will be pruned from the search space.

Finally, observe that we approached preference-based HTN planning by integrating preference-based planning into HTN planning. An alternative approach would be to integrate HTN into preference-based planning. [Kambhampati et al., 1998] hints at how this might be done by integrating HTN into their plan repair planning paradigm. This might also be done by compiling away the HTN through a reformulation of the planning problem (e.g., [Fritz et al., 2008]). For the integration of HTN into preference-based planning to be effective, heuristics would have to be developed that exploited the special compiled HTN structure. Further, such a compilation would not so easily lend itself to mixed-initiative preference-based planning, a topic for future investigation.

Chapter 6

WSC Problem with Customization: Execution and Optimization

6.1 Introduction

In Chapter 5 we proposed an algorithm (Algorithm 5.6) based on planning with heuristic search that employs a best-first, forward search strategy capable of computing an optimal composition. In this chapter, we consider how to exploit this algorithm in data-intensive settings where the search space can be prohibitively large. Furthermore, in Chapters 3 and 5, we assumed for the most part that relevant information is gathered offline, before composition is constructed. In this chapter we focus on addressing the information-gathering problem with a view to producing high-quality solutions.

Most of the previous work on WSC (and indeed much of the work on the WSC problem with customization) has assumed that all the information required to generate the composition is on hand at the outset, and as such, composition is done offline followed by subsequent execution of the composition, perhaps in association with execution monitoring. However, this is not realistic in many settings. Consider the task of travel planning or any other multi-step purchasing process on the Web. A good part of the composition task for these domains involves data gathering, followed by generation of an optimized composition with respect to that data and other criteria.

Moreover, gathering all the information required for the composition prior to initiating composition generation can result in a lot of unnecessary data access. Further, it results in an enormous search space for a planner. Most state-of-the-art planners require actions to be grounded. However, unlike typical planning applications, many WSC applications are data-

intensive, which results in an enormous number of ground actions and a huge search space. While this space may still be manageable for computing *a* composition, to compute *an optimal* composition, and to guarantee optimality, the entire search space must be searched, at least implicitly. This has the effect that most data-intensive WSC problems that involve optimization of data (like picking preferred flights) will not scale using conventional preference-based planning techniques.

Consider a flexible composition template that describes the Travel domain in terms of the tasks of booking transportation and booking accommodations, with varying options for their realization. We add to this the following preferences: *If destination is more than 500 km away, book a flight, otherwise I prefer to rent a car; I prefer to fly with a Star Alliance carrier; I prefer to book cars with Avis, and if not Budget; I prefer to book a Hilton hotel, and if not a Sheraton.* A naive preference-based planning would access all the flight, car, hotel, etc. information prior to composition and create grounded actions (e.g., *book-car*(*Avis,Pria,Daily,$39,...*)) for each data instance, resulting in a huge set of actions. In order to guarantee optimality of a composition, one needs to guarantee that all compositions were considered, which would (naively) involve considering all combinations of flight-hotel and/or car-hotel. However, there is clearly a smarter way to do this. In particular, either flight information or car rental information (but not both) need to be considered, depending on the distance to destination. Further, the choice of airline is independent of the choice of hotel, so optimality can be guaranteed by optimizing these choices independently. These simple, intuitive observations provide motivation for the work presented in this chapter.

In this chapter, we investigate the class of WSC problems that endeavour to generate high-quality compositions through optimization of service and data selection. We attempt to balance the trade-off between offline composition and online information gathering with a view to producing high-quality compositions. The need to actually execute services to gather data, as well as the potential size and nature of the resultant optimization problem truly distinguishes our WSC problem from previous work on preference-based planning. To this end we exploit a notion of middle-ground execution with a view to producing high-quality compositions that enables information gathering during generation of a WSC. Our objective is to minimize data access and to make optimization as efficient as possible by exploiting the independence of ground actions within the search space. Finally, we wish to ensure that our techniques will maintain the guarantees a more naive approach would afford, including guarantees regarding the soundness of our compositions and their optimality.

Our investigation is performed in the context of our existing preference-based HTN WSC

system, **HTNWSC-P** (see Chapter 5). That is, we assume that the composition template is specified in HTN and the preferences are specified in our modified version of PDDL3. Note, in this chapter we assume that no hard constraints are given, or if given they are specified and enforced as discussed in the previous chapters. Given the specification of preferences and the composition template, in this chapter, we propose a means of analyzing a WSC problem in order to identify places where optimization can be localized while preserving global optimality. Further, building on previous work that addresses the problem of information gathering (e.g., [McIlraith and Son, 2002, Kuter et al., 2005]), we exploit a middle-ground execution system with a view to producing high-quality compositions that executes information-gathering services, as needed, while only *simulating* the execution of world-altering services. In doing so, the HTN WSC system is able to benefit from the further knowledge afforded by information-gathering while still supporting backtrack search, by not actually or not necessarily executing world-altering services.

By exploiting the structure in the preference specification and domain we propose a notion of what we call *localized data optimization* in which the optimization task can be decomposed into smaller, local optimization problems, while preserving global optimality. This notion comes from an observation that in many composition scenarios that involve preferences most of the search time is spent on resolving the optimization that relates to the data (which flight, which car, which hotel). We propose to further improve the search by performing optimization of data choices locally, whenever possible, while still guaranteeing that the choice selected does not eliminate the globally optimal solution. We also identify a condition where performing localized data optimization is sound. We show that our approach to data optimization can greatly improve both the quality of compositions and the speed with which they are generated.

6.1.1 Contributions

The following are the main contributions of this chapter.

- Identified a way to exploit structure in the preference specification and domain in order to generate compositions more efficiently by performing what we call *localized data optimization*
- Identified a condition where performing localized data optimization is sound
- Proposed a notion of middle-ground execution with a view to producing high-quality compositions. To that end we developed an execution system for the WSC problem

with customization that interleaves online information gathering with offline search as deemed necessary

- Modified our **HTNWSC-P** to incorporate the proposed approach
- Identified a case where we could prove the optimality of resulting compositions
- Established the correctness of our approach
- Experimentally evaluated our approach

6.2 Decoupling Data Optimization From Search

Given the HTN domain description of a WSC problem, the initial task network, and specification of preferences expressed in our modified version of PDDL3 (see Chapter 4), we are interested in generating a high-quality (ideally optimal) composition. Unfortunately, unlike the task of generating a composition, its optimization requires considering all alternative compositions, at least implicitly. And even in the case where the composition can be decomposed into independent subproblems, the task of customization over the composition can introduce new inter-dependencies.

In Chapter 5 we proposed an algorithm based on planning with heuristic search that employs a best-first, forward search strategy capable of computing an optimal composition. We elaborate on the algorithm in Section 6.4. In this section, we consider how to exploit this algorithm in data-intensive settings where the search space can be prohibitively large.

Data acquired via information gathering is typically encoded as parameters of the actions that act on that data. E.g., the *book-flight* action would be parameterized by the data associated with a flight, such as airline, origin, destination, fare class, etc. State-of-the-art planning algorithms require actions/operators to be grounded. As such, in data-intensive settings, there can be an enormous number of ground actions and as a consequence an enormous search space to explore. Consider a simplified version of the task of booking a flight, a hotel, a car, and booking a tour for a vacation. Assume that these four tasks can be performed in any order and are completely independent of each other. Given 20 possible flights, 10 hotels, 10 types of car, 5 tours of the city, and 4! ways in which the booking of these items can be performed, there are 20*10*10*5*4! different compositions that need to be explored (at least implicitly) to determine an optimal composition. Using the algorithm proposed in the previous chapter

(Algorithm 5.6), some of these combinations will be eliminated by our exploitation of state-of-the-art heuristic search and sound pruning – a means of pruning partial plans that have no prospect of producing a plan that is superior to the current best plan. Nevertheless, the algorithm is still doing a lot of unnecessary search.

From our experience with WSC applications that involve preferences, we observe that most of the search time is spent on resolving the optimization that relates to the data that we have collected. We henceforth refer to this type of optimization as ***data optimization***. We observe that just as the subtasks afford a degree of independence in many WSC scenarios, so too do the different data choices, and that this independence allows us to perform some optimization *locally*, external to the composition process, or even arbitrarily (if they don't matter) while still guaranteeing that the choice does not eliminate the globally optimal solution. For example, in our simplified scenario we can select the best car, best flight, best hotel, and best tour independently of each other. And in doing so, we can reduce the search space to (20+10+10+5)*4!. More generally, if we are able to identify that subset of the data that is relevant to the optimization of the composition and attempt to localize its optimization then we can significantly streamline our search.

In what follows, we elaborate on the exploitation of three scenarios: (1) a data choice must be done in concert with the composition but choosing optimal data can be localized; (2) a data choice can be optimized in isolation of the composition generation process (i.e., can be done anytime); and (3) a data choice is irrelevant to the optimization of the composition and can be made arbitrarily (i.e., any available data can be selected). We begin by defining the notion of localized data optimization and identifying conditions under which it retains the possibility of finding an optimal solution. Note, we assume that atomic processes are either information gathering or world altering. Examples of information-gathering operators are finding the available hotels or flights. These operators have only outputs. On the other hand *book-flight* or *book-hotel* are world altering (have effects).

Definition 6.1 (Localized Data Optimization with respect to an Operator) *Let $\mathcal{P}'$ be an incomplete HTN planning problem with preferences (defined in Section 6.3). Let N be a search node that represents a partial plan, and let O be the world-altering operator that is to be applied next in our search – the operator that extends the partial plan currently under consideration. Let $N_1 ... N_k$ be different nodes that result from different possible groundings of O from node N. Localized data optimization for O selects node N_i, $1 \leq i \leq k$ if $M(N_i) \leq M(N_j)$, $\forall 1 \leq j \leq k$, where $M(N)$ is the PDDL3 metric value of search node N.*

Note, while in classical planning, the search generates different ground instances of actions by looking at their preconditions, most HTN planning system (**SHOP2** included) assume that operators are already bound (i.e., grounded) before they are applied. Instead methods that embody primitive tasks within their task networks, have parameters that get bound at run time causing different grounding of an operator. Hence, nodes $N_1...N_k$ are nodes that result from grounding the operator at the method level.

According to the above definition, the nodes with the least metric value (i.e., there could be more than one node that have the least metric value) are selected when localized data optimization for an operator is performed. The question is when is such a strategy sound, i.e., when can we do such a local selection without eliminating the overall best solution? For example, assume a best flight among all available flights is selected, but the selected flight arrives at night preventing the planner from booking an activity for that day or the price of the selected flight is too high and as a result there would not be enough money left in the budget to get the hotel that the user really preferred. In such situations, even though the selected flight is the best flight choice among all available flights in isolation (or locally), because of the interactions among operators within and between tasks, this choice is not the best choice for the globally optimal composition.

Definition 6.2 (Sound Localized Data Optimization with respect to an Operator) *Let $\mathcal{P}'$, N, O, $N_1...N_k$ be as in Definition 6.1, and let N_s be a set of nodes that are selected via localized data optimization. Localized data optimization with respect to O is said to be sound if there does not exist a plan extending any node $N_j \notin N_s$, $1 \leq j \leq k$ that would result in a better metric value than any plan extending the node $N_i \in N_s$. Hence, if there exists an optimal plan $\boldsymbol{a}$ from extending the partial plan in node N, $\boldsymbol{a}$ is not achievable from extending any of the nodes N_j.*

This definition has important implications. If localized data optimization is sound, then all nodes N_j can be pruned from the search space because we know an optimal plan cannot be reached by extending any of these nodes.

Now that we know the condition under which localized data optimization is sound, we need to discuss how such an operator can be identified. Doing so involves analyzing the structure of the planning problem to identify operators that are completely independent and have no interactions with the rest of the planning problem including (1) the operators and methods in the domain, (2) the preferences, and (3) the hard constraints, assuming for simplicity that there are no indirect effects that we have to worry about. The following is a syntactic criterion that

can be used to identify operators whose grounding choices will have no impact on the rest of the decisions made during the generation of a composition.

Definition 6.3 (Non-interacting Operator with respect to the Domain)
An operator O is said to be non-interacting with respect to the domain if (1) no predicate in the precondition of O or in the condition of the conditional effect statement of O appears in the effect of any other operator in the domain, and (2) there is no predicate in the effect of O that appears in the precondition (or in the condition of the conditional effect statement) of any other operators or methods[1] *of the planning problem.*

Intuitively this definition says that nothing affects the execution or outcome of this operator. Returning to our example, if the flight booking operator changes anything that is a precondition of another operator, then the flight booking operator interacts with that operator. E.g., if the flight booking operator has the effect of depleting available monetary funds, precluding the booking of a particular hotel, or if it results in arrival at a time that impacts the booking of a tour, then it is considered to interact with other aspects of the planning problem.

The above condition can be easily checked as a preprocessing step by analyzing the domain definition. However, syntactically identifying how preferences play a role in data interactions is more difficult, particularly when trajectory preferences – preferences expressed in a subset of LTL – are involved. One way to identify interacting operators with respect to the preferences, is to determine whether the operator's add effects – the positive effects of an operator – appear in any preference formulae. More specifically to enforce non-interaction, we need to ensure that the add effects of the operator never appear in the "b part" of preference formulae, where the "b part" is as follows: (*sometime-after* b a) (*always* (*imply* b a)) or (*sometime* (*imply* b a)). This is because the "b part" is the condition that if true requires the preference formula to be true, and in particular, necessitates the "a part" holding. Thus, if the "b part" refers to an add effect of a world-altering operator for which localized data optimization is performed, and the "a part" is hard or impossible to achieve then the choice made in the data optimization interacts with a choice that has to be made later.

To illustrate the problem, consider a user preference that prefers booking a flight with Air Canada *after* a 4-star hotel is booked. So the "b part" is booking a 4-star hotel. Now assume we miss this source of interaction and localized data optimization for the *book-hotel* operator is performed and a 4-star hotel is selected. But then the user did not realize that their origin

[1] Precondition for a method can be specified as a before constraint

or destination is not a Canadian city and Air Canada only flies between two cities if one is in Canada. Hence, because a 4-star hotel is selected we need to satisfy the preference formula which is now unsatisfiable.

Definition 6.4 (Criterion for Sound Localized Data Optimization) *Given that the operator O is non-interacting with respect to the planning domain, the non-interaction with respect to preferences and hard constraints should be defined in such a way that ensures performing sound localized data optimization on this operator as defined in Definition 6.2.*

Determining if an operator is non-interacting with respect to the preferences or hard constraints is beyond the scope of this thesis and is left for future work.

To this point we have defined the notion of localized data optimization and identified some syntactic criteria that will ensure its soundness. Before concluding, we informally discuss two further cases. We observe that in some instances the optimization of data can be completely separated or decoupled from the dynamics of the composition problem and an optimal data choice can be determined as a separate process. For example, if a user's sole preference is to book the cheapest car, then the identification of what car to book can be performed in isolation of the generation of the composition altogether. Further, some data choices have no effect at all on the quality of the composition and as such can be made arbitrarily. For example, if the user does not care what car they rent, then the choice of rental car can be made arbitrarily. In both of these cases, the search space can exclude consideration of the different data values by insertion of a single placeholder value. Execution of the information-gathering service can be delayed until after composition, and the placeholder resolved at that time.

Consider a simple case where the user prefers to book their car from two different rental companies possibly to collect more reward points or keep both their reward programs active. We know that a car can be booked for two purposes: for local transportation and for city-to-city transportation. So if booking a car operator is executed for the second time, it is more preferred that the car be rented from some other rental car agencies. So in this case, data optimization for booking a car operator depends on the current state of the planning problem. However, if every time localized data optimization is performed for an operator, the same node is selected regardless of the planning state (i.e., whether any of the preferences are satisfied or not), then the data optimization for this operator can be completely decoupled from the composition process. In addition, if no preference formula ever mentions the add effects of the operator, and the operator is non-interacting with respect to the domain and preferences, then the selection of the data grounding does not have any effect on the search for an optimal plan. Hence, execution

of the information-gathering services that gathers the relevant information can be delayed to an intermediate step before the execution of the world-altering actions such that the predefined values selected earlier will get their real values by executing the information-gathering services and performing the localized data optimization on them if necessary.

6.3 Middle-Ground Execution

For many WSC problems it is impractical, and often impossible to reduce the WSC problem to a planning problem with complete initial state – i.e., for which all the information necessary to generate a composition (and in our case to optimize it) is known prior to commencement of the search for a composition. In the Travel domain this would necessitate collecting data relating to all the different modes of transportation, means of accommodation, etc. The space of ground actions would be enormous and the planning and optimization task unsolvable. However, one can instead imagine gathering information as it becomes necessary to choice points in the generation and optimization of the composition, and using this to inform the search for different compositions. In this section, we investigate how to perform information gathering in this manner.

The problem of gathering information during composition has been examined in several research papers (e.g., [McIlraith and Son, 2002, Petrick and Bacchus, 2002, Sirin et al., 2005b, Kuter et al., 2004]). McIlraith and Son in [McIlraith and Son, 2002] proposed a so-called middle-ground interpreter that collects relevant information, but only simulates the effects of world-altering actions. Their interpreter works under the **Invocation and Reasonable Persistence (IRP) Assumption** that (1) assumes all information-gathering actions can be executed by the middle-ground interpreter and (2) assumes that the gathered information persists for a reasonable period of time, and none of the actions in the composition cause this assumption to be violated. Even though these assumptions may not hold for time-sensitive data (e.g., stock quotes) or some context-dependent data (e.g., mobile services) these assumptions are true for a large class of information available on the Web, in particular for the type of information that interests us here (e.g., flight schedules, available hotels, tours, etc). Kuter et al. in [Kuter et al., 2004] take a similar approach, but their work focuses on dealing with services that do not return a result (if any) immediately. They provide a Query Manager that allows their planner **ENQUIRER** to continue search without waiting for all of the information-gathering services to return data. They also assume that the information-gathering services are executable (similar to condition 1 of IRP), but they allow the planner itself to change or update

Translate-Atomic-Process-Output(Q)

Input: an OWL-S definition Q of an atomic process A with only outputs.

Output: HTN methods M_1, M_2, and M_3, and HTN operators O_1, O_2, and O_3

Procedure:

(1) let $\boldsymbol{v}$ = the list of input parameters defined for A as in Q

(2) let *Pre* = conjunct of all preconditions of A and $assign(y, callService(A(\boldsymbol{v})))$,
where $callService$ is an external procedure that will handle the call to Web services.

(3) let $\mathit{Eff}_1 = [y \wedge called(A(\boldsymbol{v})) \wedge cached(y, A(\boldsymbol{v}))]$ // cache y for $A(\boldsymbol{v})$

(4) let Eff_3 represent the place holder value for this call.

(5) let $\mathit{Pre}_1 = [\neg called(A(\boldsymbol{v})) \wedge canNotDelay(A(\boldsymbol{v}))]$ // first time calling $A(\boldsymbol{v})$

(6) let $\mathit{Pre}_2 = [called(A(\boldsymbol{v})) \wedge canNotDelay(A(\boldsymbol{v}))]$

(7) let $\mathit{Pre}_3 = \neg canNotDelay(A(\boldsymbol{v}))$

(8) let $O_1 = (A_1(\boldsymbol{v}), \mathit{Pre}, \mathit{Eff}_1)$, and let $M_1 = (A_1(\boldsymbol{v}), A(\boldsymbol{v}), n_1, \mathit{before}(n_1, \mathit{Pre}_1))$,
where A_1 is a new unique name and n_1 is the task node for $A_1(\boldsymbol{v})$ // Case 1

(9) let $O_2 = (A_2(\boldsymbol{v}), \emptyset, \emptyset)$, and let $M_2 = (A_2(\boldsymbol{v}), A(\boldsymbol{v}), n_2, \mathit{before}(n_2, \mathit{Pre}_2))$,
where A_2 is a new unique name and n_2 is the task node for $A_2(\boldsymbol{v})$ // Case 2

(10) let $O_3 = (A_3(\boldsymbol{v}), \emptyset, \mathit{Eff}_3)$, and let $M_3 = (A_3(\boldsymbol{v}), A(\boldsymbol{v}), n_3, \mathit{before}(n_3, \mathit{Pre}_3))$,
where A_3 is a new unique name and n_3 is the task node for $A_3(\boldsymbol{v})$ // Case 3

(11) **Return** O_1, O_2, O_3, and M_1, M_2, M_3

Figure 6.1: Sketch of the translation of the information-gathering atomic process.

the gathered information during planning (a variant of condition 2 of IRP). More recently, Au et al. [Au and Nau, 2007] proposed an approach to relaxing the IRP assumption, however their approach does not seem amenable to generating optimized compositions.

Similar to McIlraith and Son [McIlraith and Son, 2002], we propose a middle-ground execution system, where the relevant information is gathered online via calling information-gathering services (e.g., through a call to external procedures within a HTN planner) while only simulating the effects of world-altering services allowing the planner to backtrack. Our middle-ground execution system is implemented by modifying the translation from OWL-S Web service descriptions [Martin et al., 2007] to HTNs. Our translation builds on the work by Sirin et al. [Sirin et al., 2005b] and our previous work (see Section 3.4.1). We encode each OWL-S atomic process as an HTN operator and each OWL-S composite process as an HTN method. Similarly, we assume that all atomic processes are either information gathering or world altering and distinguish our set of planning operators accordingly. While it may be

hard to separate the two in some cases, this assumption holds true for a large class of atomic processes that interests us.

The fidelity of our translation relies on the **IRP Assumption** [McIlraith and Son, 2002], i.e., none of the actions in the HTN or any exogenous action can violate the assumption. To improve the efficiency of the system by avoiding multiple calls to the same service with the same parameters, we implement a caching system similar to [Sirin et al., 2005b]. However instead of using a monitoring system we modify the translation of information-gathering atomic processes into HTN operators (this operator has preconditions that externally call information-gathering services and add the return response) to explicitly encode the caching for the gathered information. We consider the following 3 cases in our translation:

1. The execution system is calling the information-gathering service for the first time and cannot delay this call.
   $\Longrightarrow$ Call the information-gathering service and cache the gathered information.

2. The execution system have already called the information-gathering service once and cannot delay this call.
   $\Longrightarrow$ Use the cached information. No need to call the information-gathering service again.

3. The execution system can delay the call to the information-gathering service.
   $\Longrightarrow$ Use a placeholder data value.

Our translation relies on the use of a **SHOP2**-based HTN planner; it exploits **SHOP2**'s features to perform run-time binding of variables and to make external procedure calls to invoke services.

In Section 6.2, we discussed circumstances where data optimization can be performed in isolation of the generation of the composition. This can occur when the data is irrelevant to the optimization of the composition. i.e., it is not mentioned in any preferences, or when the data choice does not interact with the dynamics of the composition. For example, consider the book-hotel service and the information-gathering service that gathers information regarding available hotels. If the user has no preference regarding the choice of hotel, then it is efficient to delay the execution of this information-gathering service and the arbitrary selection of a hotel until after the composition is generated. To implement this, we identify these data and associated services a prior and through our modified translation replace occurrences of the data with placeholders.

The information-gathering service is then executed following composition generation (in a postprocessing step) and the placeholder is replaced with an appropriate choice.

Figure 6.1 shows the sketch of the translation of the information-gathering atomic process. Note, we use the **SHOP2** function $assign$ in the precondition term which will bind a variable at run time. Furthermore, we use **SHOP2**'s ability to call external procedures (denoted by $callService$) which invoke a call to a specific Web service. Also note, we modify the domain with three new fluents: $called(A)$, $canNotDelay(A)$, and $cached(A)$, where A is an information-gathering service. The values for $called(A)$ and $cached(A)$ are set to be false at the initial state, while the truth value for the fluent $canNotDelay(A)$ is decided in the preprocessing step by checking if sound localized data optimization for the operator that needs the output of A can be performed in isolation of the generation of the composition.

The translation takes as input the information-gathering atomic process and outputs three methods and three operators representing the three cases we considered. Note, the information-gathering atomic processes call information-gathering services externally to collect information. Case 1 is handled by method M_1 and operator O_1. M_1's precondition (expressed via the *before* constraint) ensures that it is the first time the information-gathering service is called and this call cannot be delayed (i.e., optimization cannot be done in isolation). Operator O_1 will make the call to the information-gathering service and cache the results. Method M_2 will ensure that the call to the information-gathering service has already been made and hence the already cached information can be used. Finally, M_3 is called when we can delay the call to the information-gathering service. Hence, this call is delayed while a place holder value is used instead.

Similar to Kuter et al. [Kuter et al., 2004], let X be a set of information sources. Information source is any external source that can provide information during planning. Then we represent the amount of information that can be obtained from sources as $\delta(X)$. Note that we operate under the IRP assumption, and more specifically, we assume that the results returned from these sources will not change during planning even though we may not know $\delta(X)$ a priori. Also we assume that we can define an information-gathering OWL-S atomic process and in turn an OWL-S service from the specification of an information source.

Definition 6.5 *Complete HTN planning problem with preferences $\mathcal{P}^{\mathcal{C}}$ is identical to an HTN planning with preferences (Definition 3.6). That is $\mathcal{P}^{\mathcal{C}} = (s_0, w_0, D, \preceq)$ where s_0 is the complete initial state, w_0 is the initial task network, D is the HTN planning domain which consists of a set of operators and methods, and $\preceq$ is a preorder between plans. A plan $\boldsymbol{a}$ is a solution to*

$\mathcal{P}^{\mathcal{C}}$ if and only if $\boldsymbol{a}$ is a plan for (s_0, w_0, D) and there does not exists a plan $\boldsymbol{a}'$ for (s_0, w_0, D) such that $\boldsymbol{a}' \prec \boldsymbol{a}$.

Definition 6.6 *An incomplete HTN planning problem with preferences $\mathcal{P}^{\mathcal{I}}$ is a 5-tuple $(s_0^I, w_0, D^I, \preceq, X)$ where s_0^I is what is known of the initial state (i.e., it is incomplete), w_0 and $\preceq$ are defined as in Definition 6.5, D^I is the HTN planning domain which consists of a set of operators and methods some of which can externally call sources, X is a set of available information sources during planning. The total information available about the initial state is defined by $s_0^I \cup \delta(X)$.*

From Definitions 3.5 (Definition of a plan for HTN planning problem) and Definition 6.6, a plan for the incomplete HTN Planning problem with preferences is a primitive decomposition of the task network w_0. To find such a decomposition, some information sources, as dictated by the methods and operators of the domain, have to be called externally to collect the relevant information needed to successfully decompose w_0.

Definition 6.7 (Consistency) *An incomplete HTN planning problem with preferences $\mathcal{P}^{\mathcal{I}} = (s_0^I, w_0, D^I, \preceq, X)$ is consistent with a complete HTN planning problem with preferences $\mathcal{P}^{\mathcal{C}} = (s_0, w_0, D, \preceq)$ if an only if $s_0^I \cup \delta(X) \subseteq s_0$.*

Next, we define an incomplete OWL-S WSC problem with preferences.

Definition 6.8 *Incomplete OWL-S WSC problem with preferences $\mathcal{P}^{\mathcal{IW}}$ is a 5-tuple $(s_0^I, C, K, \phi_{soft}, Y)$, where s_0^I is what is known of the initial state (it may be incomplete), K is a collection of OWL-S process models that includes both atomic processes that produce output (or are information gathering) and atomic processes that produce effects (or are world altering), C is a possibly composite OWL-S process defined in K, and ϕ_{soft} is a set of user preferences [2], and Y is a set of information-gathering services available during planning. A composition π is a solution for $\mathcal{P}^{\mathcal{IW}}$ if and only if it is a solution for (s_0^I, C, K, Y) (i.e., sequence of atomic OWL-S processes such that, when executed at the initial state, achieve C) and there does not exists a plan π' for (s_0^I, C, K, Y) such that π' is more preferred than π.*

Note, the above definition of the OWL-S WSC problem is similar to the one given in Chapter 3. However, in this chapter, we have incomplete initial state.

[2] In this chapter we consider preferences specified in PDDL3.

Definition 6.9 (Equivalency) *Assuming the IRP assumption holds, an incomplete OWL-S WSC problem with preferences* $\mathcal{P}^{\mathcal{IW}} = (s_0^I, C, K, \phi_{soft}, Y)$ *is equivalent to an incomplete HTN planning problem with preferences* $\mathcal{P}^{\mathcal{I}} = (s_0^I, w_0, D^I, \preceq, X)$*, where*

- w_0 *is generated by our modified OWL-S to HTN translation for the OWL-S process* C*,*
- D^I *is the HTN domain description generated by running our modified OWL-S to HTN translation for the collection of OWL-S process models* K*,*
- $\preceq$ *is a preorder between plans as dictated by* ϕ_{soft}*, and*
- X *is the set of information sources that produce the same information as the services* Y*, that is* $\delta(X) = \delta(Y)$*.*

To further elaborate, we represent the body of information that can be obtained from the information-gathering services Y as $\delta(Y)$. More specifically, $\delta(Y)$ represents all possible bindings of the predicates that appear in the `output` or the `postcondition` of the OWL-S descriptions of the atomic processes specified in K. In order for the two problems to be equivalent, we ensure that we have $\delta(X) = \delta(Y)$. That is both X and Y produce exactly the same set of information.

The following theorem establishes correctness of our approach. Note, we assume that the postprocessing step is performed on the generated plan. That is the call to those information-gathering services that have been delayed are made and the placeholder is replaced with an appropriate choice.

Theorem 6.1 *Assuming the IRP assumption holds, let* $\mathcal{P}^{\mathcal{IW}} = (s_0^I, C, K, \phi_{soft}, Y)$ *be an incomplete OWL-S WSC problem with preferences, and let* $\mathcal{P}^{\mathcal{I}} = (s_0^I, w_0, D^I, \preceq, X)$ *be an incomplete HTN planning problem with preferences that is equivalent to* $\mathcal{P}^{\mathcal{IW}}$*. A plan* $\boldsymbol{a} = o_1 ... o_k$ *is a solution to* $\mathcal{P}^{\mathcal{I}}$ *if and only if* $\pi = p_1 ... p_k$ *is a solution to (a composition for)* $\mathcal{P}^{\mathcal{IW}}$ *such that* o_i*,* $1 \leq i \leq k$ *are the primitive operators that correspond to the atomic process* p_i*.*

Proof: The proof is in Appendix A. ■

Note that the theorem establishes a relationship or a one-to-one mapping between the primitive operators that are part of a plan, some of which could be information gathering, and atomic processes that are part of the composition, some of which could be information gathering. Given this theorem, we can generate a solution for $\mathcal{P}^{\mathcal{IW}}$ through HTN planning with preferences as we will see in the next section.

1: **function HTNWSCLOCAL**(s_0^I, w_0, D^I, METRICFN,HEURISTICFN)
2: $frontier \leftarrow \langle s_0', w_0', \emptyset \rangle$ ▷ initialize frontier
3: $bestMetric \leftarrow$ worst case upper bound
4: **while** $frontier$ is not empty **do**
5: $current \leftarrow$ Extract best element from $frontier$
6: $\langle s, w, partialP \rangle \leftarrow current$
7: $lbound \leftarrow$ METRICBOUNDFN(s)
8: **if** $lbound < bestMetric$ **then** ▷ pruning by bounding
9: **if** $w = \emptyset$ and $current$'s metric $< bestMetric$ **then**
10: Output plan $partialP$
11: $bestMetric \leftarrow$ METRICFN(s)
12: $succ \leftarrow$ successors of $current$
13: **if** possible to perform sound localized data optimization **then**
14: $succ \leftarrow$ the best nodes among successors of $current$ ▷ pruning other nodes
15: $frontier \leftarrow$ merge $succ$ into $frontier$

Figure 6.2: A sketch of our HTN WSC algorithm with sound localized optimization.

6.4 Computing a Preferred Composition

We update the algorithm discussed in Section 5.4 in order to deal with incomplete information and perform middle-ground execution and sound localized data optimization on some already identified non-interacting operators. The updated algorithm takes as input an incomplete HTN planning problem with preferences $\mathcal{P}^{\mathcal{I}} = (s_0^I, w_0, D^I, \preceq, X)$ as defined in Definition 6.6. Our algorithm just as before performs best-first, incremental search and uses state-of-the-art heuristics proposed in the previous chapter [Sohrabi et al., 2009]. The updated algorithm is outlined in Figure 6.2.

If computing a successor to $current$ generates many different groundings of a primitive task, causing different groundings of an operator whose localized data optimization is known to be sound, then then data optimization on this node will select the best successors according to METRICFN and replace $succ$ with the selected nodes (line 13 and 14). The resulting $succ$ is then merged into the $frontier$. Note that $succ$ will have only the nodes chosen based on the localized data optimization, that is all other nodes will get pruned from the search space. The search terminates when $frontier$ is empty.

6.4.1 Properties of the Algorithm

The search space for computing the preferred composition is significantly reduced by the HTN representation of the composition template, by pruning performed from incremental search,

and by the localized data optimization (line 13 and 14). So, under sound pruning we can guarantee that by exhausting the search space, an optimal plan can be found. We use the OM function to estimate the lower bound. Baier et al. [Baier et al., 2009] show that the OM function provides sound pruning under certain conditions (see Proposition 5.1).

Theorem 6.2 *If the OM function used to calculate the lower bound provides sound pruning, and any localized data optimization performed is sound, then the last plan returned, if any from the algorithm, is optimal.*

Proof: The proof is similar to the proof for **HTNPLAN-P** Theorem 5.4 using Definition 6.3 and Definition 6.4. We know each planning episode returns a better plan. We also know that the algorithm stops only when the final planning episode has rejected all possible plans. Moreover, because of sound pruning, the algorithm never prunes states incorrectly from the search space. Therefore, no better plan than the last returned plan exists. ■

Corollary 6.1 *Assuming the IRP assumption holds, let $\mathcal{P}^{\mathcal{IW}} = (s_0^I, C, K, \phi_{soft}, Y)$ be an incomplete OWL-S WSC problem with preferences, and let $\mathcal{P}^{\mathcal{I}} = (s_0^I, w_0, D^I, \preceq, X)$ be an incomplete HTN planning problem with preferences that is equivalent to $\mathcal{P}^{\mathcal{IW}}$. If $\boldsymbol{a} = o_1...o_k$ is the last returned plan by the algorithm, then $\pi = p_1...p_k$ is a solution to $\mathcal{P}^{\mathcal{IW}}$ such that o_i, $1 \leq i \leq k$ are the primitive operators that correspond to the atomic process p_i.*

Proof: We know the input to the algorithm is the incomplete HTN planning problem with preferences $\mathcal{P}^{\mathcal{I}}$. Since $\boldsymbol{a}$ is the last plan, by Theorem 6.2 $\boldsymbol{a}$ is optimal, hence $\boldsymbol{a} = o_1...o_k$ is a solution to $\mathcal{P}^{\mathcal{I}}$. Then by Theorem 6.1 $\pi = p_1...p_k$ is a solution to (a composition for) $\mathcal{P}^{\mathcal{IW}}$ such that o_i are the primitive operators that correspond to the atomic process p_i. ■

6.5 Implementation and Evaluation

We implemented our WSC system with two modules: a preprocessor and a preference-based HTN planner. The preprocessor reads PDDL3 problems and generates an HTN planning problem. Additionally, it finds non-interacting operators, making it possible to perform sound localized data optimization on this selection. Our implementation builds on **HTNWSC-P** (discussed in Chapter 5), itself a modification of the LISP version of **SHOP2** [Nau et al., 2003], that implements the algorithm and heuristics described above. We have three main objectives

in our experimental evaluation: (1) to measure the search time gain as well as the quality improvement by performing localized data optimization, (2) to see if performing localized data optimization helps in finding an optimal plan, (3) to investigate if the improvement (both time and quality) depends on other dimensions of search such as the heuristics used or the difficulty of the domain.

We use the Travel domain as our benchmark. Note that there are no standard benchmark that we were aware of that could exercise the kinds of behaviour that we were looking for in order to evaluate our approach (i.e., they are either not data-intensive or assume they have complete information about the initial state). Therefore, we created our own 8 problem sets each with 6 different instances (we have 48 instances in total). In half of the problem sets we allowed interleaving of tasks and in the other half we did not. An example of interleaving is one that allows booking an accommodation when a transportation is booked, but not paid for (i.e., the transportation task is not done yet). Furthermore, the problem sets within the allowed (or not allowed) interleaving group differ in the difficulty of their top-level task. In the easiest case, the order of the execution of all tasks in *arrange-travel* (e.g., *arrange-trans*, *arrange-acc*, and *arrange-activity*) was known, and in the hardest case, these tasks could be carried out in any order. As explained earlier, if there are n tasks and they can be carried out in any order, then in the worst case there are $n!$ different combinations to evaluate in order to find an optimal composition. Finally in each problem set we know the number of non-interacting operators, but intentionally select the percentage of the one identified from this range [0, 20, 40, 60, 80, 100]. So in the 0% case none of the non-interacting operators are identified, hence, no localized data optimization can be performed, on the other hand in the 100% case all of the non-interacting operators are known, and localized data optimization is performed whenever possible. We used a 60 minute time out and a limit of 1 GB per process.

We ran all of the instances in two modes, one that makes use of the LA heuristic and one that does not. To compare the relative performance between the two modes, we averaged the percent metric difference of the final plan (relative to the worst plan) for all our 48 instances. This difference is 43% indicating that not surprisingly, using the LA heuristic greatly improves the quality of search. In particular, without the use of the LA heuristic, an optimal plan was not found in any of the instances. However, when the LA heuristic was used, many instances found an optimal plan. In particular, in four of the problem sets (we named them cases 1-4 in Figure 7.3), an optimal plan was found even without any localized data optimization The result shows (see Figure 7.3) that as the percentage of identified non-interacting operators increases (i.e., more localized data optimization is done), the time it took to find an optimal plan decreases.

% of Identified Non-Interactions	Case 1 Time(sec)	Case 2 Time(sec)	Case 3 Time(sec)	Case 4 Time(sec)	Average STI	Average PMI
0%	128	131	136	277	1.00	50.89%
20%	80	80	88	221	1.51	50.89%
40%	41	39	50	178	2.69	50.89%
60%	29	29	40	119	3.66	50.89%
80%	23	23	33	89	4.62	50.89%
100%	17	18	30	30	7.14	44.91%

Figure 6.3: Time comparison between the four cases that found an optimal plan even without localized data optimization. STI is the search time improvement between each case and the no data optimization case (i.e., 0% case). PMI is the percent metric improvement. i.e., the percent difference between the metric of the first and the last plan returned relative to the first plan.

We averaged this improvement and show it in the STI column (search time improvement with respect to the 0% case). This column shows that optimal plans are found for example, 2.69 times faster than the 0% case in the 40% case, and 7.14 times faster in the 100% case. Also recall that our algorithm is incremental, performing search in a series, each one returning a better-quality plan than the last. To see how effective this approach is, we calculated the *percent metric improvement* (PMI), i.e., the percent difference between the metric of the first and the last plan returned relative to the first plan. The result shows that the incremental approach improves the quality of the plan almost by 50%.

Finally, we looked at the other four cases where without localized data optimization an optimal plan was not found. Out of these, in two, an optimal plan was found in the 100% case and this was found 3.5 times faster than the time it took to find a non-optimal plan in the 0% case. This suggests that doing localized data optimization for these harder problem sets is helpful. In the remaining two cases, an optimal plan was not found even with optimization. This is not surprising, since the search space in these sets is very large, and pruning even though helpful, is not able to exhaust the search space; in these cases interleaving was allowed and the top level tasks were unordered. However, we observed that with optimization, the quality of the final plan was improved by 10%, and the time spent on finding this better quality plan was 5 times faster.

6.6 Summary and Discussion

A significant number of WSC problems involve both optimization of the composition and the collection of information. Work on WSC problem with customization has begun to address this

problem but much of the work has ignored the important information-gathering component, assuming that all information is given a priori. In this thesis, we are motivated by the observation that even though some classes of WSC problems can be addressed without the need for any execution during the composition phase, without explicit consideration of the data, and without consideration of preferences that distinguish high-quality solutions, many interesting and useful compositions must be done hand in hand with the data collection and optimization. Specifically this is done following execution of some information-gathering services.

The main contributions of this chapter include: identification of a way to exploit structure in the preference specification and domain in order to generate compositions more efficiently by performing what we call *localized data optimization*, identification of a condition where performing localized data optimization is sound, development of an execution system for the WSC problem with customization that interleaves online information gathering with offline search as deemed necessary, and identification of a case where we could prove the optimality of resulting compositions. To assess the effectiveness of our approach to the WSC problem, we performed experiments to evaluate the performance of our system. We showed that our approach to data optimization has the potential to greatly improve the quality of compositions and the speed with which they are generated. While the focus of this chapter was reasonably narrow, the problem it presents and the advances it makes are important first steps in addressing a broad and important problem.

While most of the related work has already been discussed, the works by Thakkar et.al. and Knoblock [Thakkar et al., 2005, Knoblock, 1995] are of notable mention here with respect to information gathering. In contrast, their focus is not on quality but rather on improving the efficiency of the execution by reducing the number of information gathering calls. In particular, they plan and optimize for information gathering by generating optimized query plans and use a data integration technique called the tuple-level filtering to insert sensing operations in the integration plan to optimize the execution of the compose Web services.

Also notable is the body of research on quality-driven WSC that was overviewed in Chapter 2 (e.g., [Lécué, 2009, Zeng et al., 2003, Alrifai and Risse, 2009]). Our work differs in many ways. In particular, in our framework we are able to find a composition that is optimal with respect to the user's preferences some of which are over the entire composition, and we can do so while interleaving execution and search. Further, we are concerned with optimizing the selection of data within the services in addition to the selection of services themselves based on their quality.

Chapter 7

Beyond Web Services

7.1 Introduction

In previous chapters, we discussed how to address the WSC problem with customization by exploiting and advancing state-of-the-art techniques for planning with preferences. To that end, we defined a correspondence between generating a customized composition of Web services and non-classical planning. We then proposed algorithms and systems that computed high-quality compositions. While the techniques we developed are motivated by the WSC problem, in this chapter we explore the possibility of applying our techniques to analogous problems. Hence, in this chapter, we discuss how we can use and adapt our framework to address two applications: requirements engineering and stream processing.

We first discuss the requirements engineering application briefly. This is the result of our collaboration with the software engineering group at the University of Toronto. Requirements engineering (RE) is a field in software engineering that includes the investigation of the goal modeling frameworks. While many of the stakeholders' goals are mandatory, traditional frameworks cannot address optional requirements, goals that are not mandatory but desirable. In [Liaskos et al., 2010, Liaskos et al., 2011], we extend these traditional frameworks to support the representation of optional goals (aka preferences). In addition, we also translated the goal models and the preference specifications into an HTN planning problem with preferences, where preferences were specified in PDDL3. This enabled the use of our preference-based HTN planner, **HTNPLAN-P** (see Chapter 5) to search for alternative plans that best meet the given preferences. The result showed that **HTNPLAN-P** can greatly benefit the proposed goal modeling framework where there is a need for better understanding the impact of stakeholders optional (or soft) goals as well as their hard goals.

From this point on, in this chapter, we will focus on the research that was conducted in collaboration with IBM T.J. Watson Research Center. In this work, we address the problem of automated composition of flow-based components in the context of the stream processing application.

A class of problems in automated software composition focuses on composition of information flows from reusable software components. An information flow is obtained from sources, processed by software components in order to transform the raw data into useful information, and is finally visualized in different ways. This flow-based model of composition is applicable in a number of application areas, including WSC and stream processing. In a stream processing application, large volume of input data, from telecommunications, finance, health care, and other industries, are integrated, aggregated, processed, and analyzed on the fly, or immediately as relevant information arrives from sources. There are a number of tools such as Yahoo Pipes[1] and IBM Mashup Center[2] that support the modeling of the data flow across multiple components. Although these visual tools are fairly popular, their use becomes increasingly difficult as the number of available components increases, even more so, when there are complex dependencies between components, or other kinds of constraints in the composition.

While automated AI planning is a popular approach to automate the composition of components, Riabov and Liu have shown that the PDDL-based planning approach may neither be feasible nor scalable when it comes to addressing real large-scale stream processing systems or other flow-based applications (e.g., [Riabov and Liu, 2006]). The primary reason is that while the problem of composing flow-based applications can be expressed in PDDL, in practice the PDDL-based encoding of certain features poses significant limitations to the scalability of planning [Riabov and Liu, 2005, Riabov and Liu, 2006].

Recent advances, including the work and techniques discussed in this thesis, have proven that the automated composition problem can take advantage of expert knowledge restricting the ways in which different reusable components can be composed. This knowledge can be represented using an extensible composition template or a composition pattern and can be further customized to meet the soft and hard constraints.

As discussed in this thesis, there are a number of different ways a composition template can be specified (e.g., HTNs or Golog (see Chapter 2)). Another way to represent a composition template is to use a language called Cascade [Ranganathan et al., 2009]. HTNs and Cascade resemble each other in many ways, and we study the relationship between the two in this chap-

[1] pipes.yahoo.com
[2] www-01.ibm.com/software/info/mashup-center

ter. The Cascade language is used in a specialized planner MARIO [Ranganathan et al., 2009], to allow domain experts to explore the space of possible flows and help them construct and deploy applications. For software engineers, who are usually responsible for encoding composition patterns, doing so in Cascade is easier and more intuitive than in PDDL or in other planning specification languages. The MARIO planner achieves fast composition times due to optimizations specific to Cascade, taking advantage of the structure of flow-based composition problems, while limiting expressivity of domain descriptions.

In this chapter, we propose an HTN Planning approach to address the problem of automated composition of flow-based applications. To this end, we propose a novel technique for creating an HTN-based planning problem with preferences from the Cascade representation of the patterns, together with a set of user-specified Cascade goals. The resulting technique enables us to explore the advantages of using domain-independent planning and HTN planning including robustness and expressivity, and address optimization and customization of composition with respect to constraints. We use the preference-based HTN system **HTNPLAN-P** [Sohrabi et al., 2009] (see Chapter 5) for implementation and evaluation of our approach. Moreover, we develop a new lookahead heuristic by drawing inspiration from ideas proposed by Marthi et al. in [Marthi et al., 2007]. We also propose an algorithm to derive indexes required by our proposed heuristic. Our proposed heuristic helps the modified version of our HTN planner achieve fast planning response comparable to that of the specialized planner MARIO. We have performed extensive experimentation in the context of the stream processing application. Our evaluation showed the applicability of the proposed approach and great promise for the proposed approach.

7.1.1 Contributions

The following are the main contributions of this chapter.

- Proposed the use of HTN planning with preferences to address modeling, computing, and optimizing composition flows in the stream processing applications
- Developed a method to automatically translate Cascade flow patterns into HTN domain description and Cascade goals into preferences, and to that end we addressed several unique challenges that hinder planner performance in flow-based applications
- Developed an enhanced lookahead heuristic and showing that it improves the performance of our HTN planner by 65% on average for the problems we used. We also

developed an algorithm to derive indexes required by the proposed heuristic from HTN planning domains, as well as Cascade problems

- Performed extensive experimentation with real-world patterns using IBM InfoSphere Streams `www-01.ibm.com/software/data/infosphere/streams`.

7.2 Preliminaries

7.2.1 Specifying Patterns in Cascade

The Cascade language has been proposed by Ranganathan et al in 2009 [Ranganathan et al., 2009] for describing data flow patterns. A Cascade flow is as a directed acyclic graph that describes the flow of information between components. That is, it describes how the data is obtained from one or more sources, processed by one or more components, and finally visualized in different ways. A Cascade flow pattern describes a set of flows by specifying different possible structures of flow graphs, and possible components that can be part of the graph. Components in Cascade can have zero or more input ports and one or more output ports. A component can be either primitive or composite. A primitive component embeds a code fragment from a flow-based language (e.g., SPADE [Gedik et al., 2008]). These code fragments are used to convert a flow into a program/script that can be deployed on a flow-based information processing platform. A composite component internally defines a flow of other components. Cascade flows are defined by specifying connections between components, which must connect each input port of each component to an output port of another, or to an external input of the composite.

Figure 7.1 shows an example of a flow pattern of a stream processing application from financial domain. This application helps financial experts decide whether a current price of a stock is a bargain. The main composite is called *StockBargainIndexComputation*. Source data can be obtained from either *TAQTCP* or *TAQFile*. This data can be filtered by either a set of tickers, by an industry, or neither as the filter components is optional (indicated by the "?"). The Volume-Weighted Average Price (VWAP) and the Bargain Index (BI) calculations can be performed by a variety of concrete components (which inherit from abstract components *CalculateVWAP* and *CalculateBargainIndex* respectively). The final results can be visualized using a table, a time- or a stream-plot. Note, the composite includes a sub-composite *BIComputationCore*.

A single flow pattern defines a number of actual flows. As an example, let us assume there are 5 different descendants for each of the abstract components. Then, the number of possible flows defined by *StockBargainIndexComputation* is $2 \times 3 \times 5 \times 5 \times 3$, or 450 flows.

A flow pattern in Cascade is a tuple $F = (\mathcal{G}(\mathcal{V}, \mathcal{E}), M)$, where $\mathcal{G}$ is a directed acyclic graph and M is called the main composite. Each vertex, $v \in \mathcal{V}$, can be the invocation of one or more of the following: (1) a primitive component, (2) a composite component, (3) a choice of components, (4) an abstract component with descendants, or (5) a component, optionally. Each directed edge, $e \in \mathcal{E}$ in the graph represents the transfer of data from an output port of one component to the input port of another component. Throughout this chapter, we refer to edges as **streams**, outgoing edges as **output streams**, and ingoing edges as **input streams**. The main composite, M, defines the set of allowable flows. For example, if *StockBargainIndexComputation* is the main composite in Figure 7.1, then any of the 450 flows that it defines can potentially be deployed on the underlying platform.

A **concrete component** is an atomic element of the pattern graph, and is usually associated with a code fragment, which is used in code generation during flow graph deployment. The declaration of a concrete component includes zero or more named input ports, and zero or more named output ports. Port names can be referenced in the code fragment to assist the code generator to establish connections between fragments. In Figure 7.1, the *ExtractTradeInfo* component is an example of a concrete component.

Abstract components are defined similarly to concrete components, including declaration of inputs and outputs, but without code fragment. Unlike with concrete components, the abstract component declaration does not include a code fragment. Instead, separately defined concrete components or composites can be declared to implement an abstract component. For example, in Figure 7.1 the *CalculateBargainIndex* component is an abstract component. Including an abstract component within a graph pattern (i.e., a composite) defines a point of variability of the graph, allowing any implementation of the abstract to be used in place of the abstract.

Cascade includes two more constructs for describing graph variability. The **choice** component can be used to enumerate several alternatives to be used within the same location in the graph. For example, the pattern in Figure 7.1 defines a choice between "TAQ TCP Source" and "TAQ File Source". The alternatives must have the same number of inputs and the same number of outputs. Any component contained within the **optional** component becomes optional. This requires the contained component to have the same number of inputs and outputs. For example, in Figure 7.1 the choice between filter trade "ByTickers" and "ByIndustry" is made

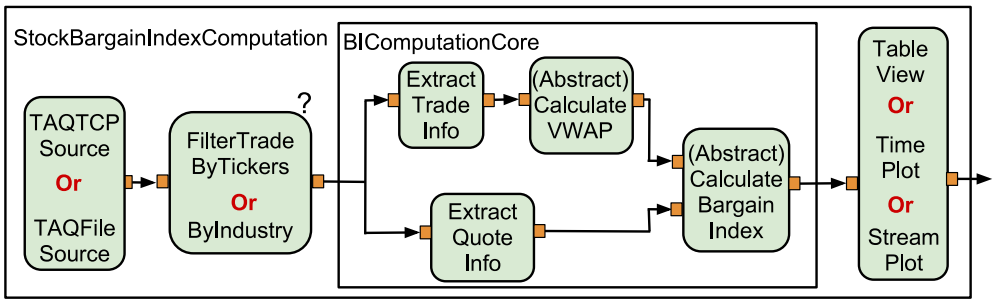

Figure 7.1: Example of a Cascade flow pattern.

optional, allowing graphs that include no filters at all to be valid instantiations of this pattern.

In Cascade, output ports of components (output streams) can be annotated with user-defined tags to describe the properties of the produced data. Tags can be any keywords related to terms of the business domain. Tags are used by the end-user to specify the composition goals; we refer to as the **Cascade goals**. For each graph composed according to the pattern, tags associated with output streams are propagated downstream, recursively associating the union of all input tags with outputs for each component. Cascade goals specified by end users are then matched to the description of graph output. Graphs that include all goal tags become candidate flows (or **satisfying flows**) for the goal. For example, if we annotate the output port of the *FilterTradeByIndustry* component with the tag *ByIndusty*, there would be $2 \times 5 \times 5 \times 3 = 150$ satisfying flows for the Cascade goal *ByIndustry*. Planning is used to find "best" satisfying flows efficiently from the millions of possible flows, present in a typical domain.

As shown by Ranganathan et al. [Ranganathan et al., 2009], Cascade planning problems can be encoded in the SPPL planning domain description language [Riabov and Liu, 2005]. For this, a Cascade to SPPL compiler was implemented as part of MARIO. The specialized SPPL planner has been shown to exhibit exponentially better performance on flow composition domains compared to PDDL planners. The main difficulty in representing these problems in PDDL comes from the fact that the same component can be used in different places within the same graph, and therefore unique names must be given to new output streams produced when the component is included in the graph. In this chapter, we will show that HTN allows an elegant solution to this problem.

7.2.2 Specifying Preferences

To specify preferences for the automated composition of flow-based applications, we use the PDDL3 [Gerevini et al., 2009] preference specification language (see Section 4.5.1). Currently we exploit the use of simple preferences. Recall that **simple preferences**, or final-state preferences are atemporal formulae that express a preference for certain conditions to hold in the final state of the plan. For example, preferring that a particular tag appears in the final stream is a simple preference. This is the preference form we exploit in the following section to encode the user-specified Cascade goals.

Since PDDL3 allows specification of temporally extended preference, we could also consider specification of temporally extended preference. However, as shown by Baier et at. [Baier et al., 2007], temporal extended preferences can be compiled away so that the planning problem is left with only simple preferences.

Recall that in PDDL3 the quality of the plan is defined using a metric function. The PDDL3 function `(is-violated name)` is used to assign appropriate weights to different preference formula. Note, inconsistent preferences are automatically handled by the metric function.

7.3 From Cascade Patterns to HTN Planning

In this section, we describe an approach to create an HTN planning problem with preferences from any Cascade flow pattern together with a set of given Cascade goals. In particular, we show how to: (1) create an HTN planning domain (specified in **SHOP2**, the base planner for **HTNPLAN-P**) from the definition of Cascade components, and (2) represent the Cascade goals as preferences (specified in PDDL3 simple preferences). We employ **SHOP2**'s specification language written in Lisp when describing the planning elements or when giving examples. We consider ordered and unordered task networks specified by keywords ":ordered" and ":unordered", distinguish operators by the symbol "!" before their names, and variables by the symbol "?" before their names.

7.3.1 Creating the HTN Planning Domain

In this section, we describe an approach to translate the different elements and unique features of Cascade flow patterns to operators or methods, in an HTN planning domain.

Creating New Streams

One of the features of the flow-based composition domains is that components produce one or more new data streams from several existing ones. Further, the precondition of each input port is only evaluated based on the properties of connected streams; hence, instead of a single global state, the state of the world is partitioned into several mutually independent ones. Although it is possible to encode parts of these features in PDDL, the experimental results in [Riabov and Liu, 2005, Riabov and Liu, 2006] show poor performance of planners they ran (in an attempt to formulate the problem in PDDL). They conjectured that the main difficulty in the PDDL representation is the ability to address creating new objects that have not been previously initialized to represent the generation of new streams. In PDDL, this can result in a symmetry in the choice for the object that represents the new uninitialized stream, significantly slowing down the planner.

To address the creation of new uninitialized streams we propose to use the *assignment expression*, available in the **SHOP2** input language, in the precondition of the operator that creates the new stream. We will discuss how to model Cascade components as operators and methods next. We use numbers to represent the stream variables using a special predicate called *sNum*. We then increase this number by manipulating the add and delete effects of the operators that are creating new streams. This *sNum* predicate acts as a *counter* to keep track of the *current* value that we can assign for the new output streams.

The assignment expression takes the form "(assign v t)" where v is a variable, and t is a term. Here is an example of how we implement this approach for the "bargainIndex" stream, the outgoing edge of the abstract component *CalculateBargainIndex* in Figure 7.1. The following precondition, add and delete list belong to the corresponding operators of any concrete component of this abstract component.

```
Pre: ((sNum ?current)
     (assign ?bargainIndex ?current)
     (assign ?newNum (call  + 1 ?current)))
Delele List: ((sNum ?current))
Add List:    ((sNum ?newNum))
```

Now for any invocation of the abstract component *CalculateBargainIndex*, new numbers, hence, new streams are used to represent the "bargainIndex" stream.

Tagging Model for Components

In Cascade output ports of components are annotated with tags to describe the properties of the produced data. Some tags are called *sticky* tags, meaning that these properties propagate to all downstream components unless they are *negated* or removed explicitly. The set of tags on each stream depends on all components that appear before them or on all *upstream* output ports.

To represent the association of a tag to a stream, we use a predicate "(*Tag Stream*)", where *Tag* is a variable or a string representing a tag, for example, bargainIndex, and *Stream* is the variable representing a stream. Note that *Tag* should be grounded before any evaluation of state with respect to this predicate. To address propagation of tags, we use a *forall expression*, ensuring that all tags that appear in the input streams propagate to the output streams unless they are negated by the component.

A forall expression in **SHOP2** is of the form "(forall X Y Z)", where X is a list of variables in Y, Y is a logical expression, Z is a list of logical atoms. Here is an example going back to Figure 7.1. *?tradeQuote* and *?filteredTradeQuote* are the input and output stream variables respectively for the *FilterTradeQuoteByIndustry* component. Note, we know all tags ahead of time and they are represented by the predicate "(tags ?tag)". Also we use a special predicate *different* to ensure the negated tag *AllCompanies* does not propagate downstream.

```
(forall (?tag)(and (tags ?tag) (?tag ?QuoteInfo)
                   (different ?tag AllCompanies))
              ((?tag ?filteredTradeQuote)))
```

Tag Hierarchy

Tags used in Cascade belong to tag hierarchies (or tag taxonomies). This notion is useful in inferring additional tags. In the example in Figure 7.1, we know that the *TableView* tag is a sub-tag of the tag *Visualizable*, meaning that any stream annotated with the tag *TableView* is also implicitly annotated by the tag *Visualizable*. To address the tag hierarchy we use **SHOP2** axioms. **SHOP2** axioms are generalized versions of Horn clauses, written in this form :- *head tail*. The *tail* can be anything that appears in the precondition of an operator or a method. The following are axioms that express the hierarchy of views.

```
:- (Visualizable ?stream)((TableView ?stream))
:- (Visualizable ?stream)((StreamPlot ?stream))
```

Component Definition in the Flow Pattern

Next, we put together the different pieces described so far in order to create the HTN planning domain. In particular, we represent the abstract components by nonprimitive tasks, enabling the use of methods to represent concrete components. For each concrete component, we create new methods that can decompose this nonprimitive task (i.e., the abstract component). If no method is written for handling a task, this is an indication that the abstract component had no children or descendants.

Components can inherit from other components (usually only from abstract components). The net (or expanded) description of an inherited component includes not only the tags that annotate its output ports, but also the tags defined by its parent. We represent this inheritance model directly on each method that represents the inherited component using helper operators that add to the output stream, the tags that belong to the parent component.

We encode each primitive component as an HTN operator. The parameters of the HTN operator correspond to the input and output stream variables of the primitive component. The preconditions of the operator include the "assign expressions" as mentioned earlier to create new output streams. The add list also includes the tags of the output streams if any. The following is an HTN operator that corresponds to the *TableView* primitive component.

```
Operator: (!TableView ?bargainIndex  ?output)
Pre: ((sNum ?current) (assign ?output ?current)
      (assign ?newNum (call + 1 ?current)))
Delete List: ((sNum ?current))
Add List:((sNum ?newNum)(TableView ?bargainIndex)
          (forall (?tag) (and (tags ?tag)
           (?tag ?bargainIndex))((?tag ?output))
```

We encode each composite component as HTN methods with task networks that are either ordered or unordered. Each composite component specifies a *graph clause* within its body. The corresponding method addresses the graph clause using task networks that comply with the ordering of the components. For example, the graph clause within the *BIComputationCore* composite component in Figure 7.1 can be encoded as the following task. Note, the parameters are omitted. Note also, we used ordered task networks for representing the sequence of components, and an unordered task network for representing the split in the data flow.

```
(:ordered (:unordered (!ExtractQuoteInfo)
                      (:ordered (!ExtractTradeInfo) (CalculateVWAP)))
          (CalculateBargainIndex))
```

Structural Variations of Flows

There are three types of structural variation in Cascade: enumeration, optional components, and the use of high-level components. Structural variations create patterns that capture multiple flows. Enumerations (choices) are specified by listing the different possible components. To capture this we use multiple methods applicable to the same task. For example, in order to address choices of *source*, we use two methods, one for *TAQTCP* and one for *TAQFile*. A component can be specified as optional, meaning that it may or may not appear as part of the flow. We capture optional components using methods that simulate the "no-op" task. Abstract components are used in flow patterns to capture high-level components. These components can be replaced by their concrete components (children). In HTN, this is already captured by the use of nonprimitive tasks for abstract components and methods for each concrete component. For example, the task network of *BIComputationCore* includes the nonprimitive task *CalculateBargainIndex* and different methods written for this task handle the concrete components.

7.3.2 Specifying Cascade Goals as Preferences

While Cascade flow patterns specify a set of flows, users can be interested in only a subset of these. Thus, users are able to specify the Cascade goals by providing a set of tags that they would like to appear in the final stream. We propose to specify the user-specified Cascade goals as PDDL3 [Gerevini et al., 2009] simple preferences. The advantage of encoding the Cascade goals as preferences is that the users can specify them outside the domain description as an additional input to the problem. Also, by encoding the Cascade goals as preferences, if the goals are not achievable, a solution can still be found, but with an associated quality measure. In addition, our preference-based planner, **HTNPLAN-P**, can potentially guide the planner towards achieving these preferences; can do branch and bound with sound pruning using admissible heuristics, whenever possible, to guide the search toward a high-quality plan.

The following are example preferences that encode Cascade goals *ByIndustry*, *TableView*, and *LinearIndex*. These PDDL3 simple preferences are over the predicate "(*Tag Stream*)". Note that we need to define a metric function for the generated preferences. If the Cascade goals, now encoded as preferences are mutually inconsistent, we can assign a higher weight to the "preferred" goal. Otherwise, we can use uniform weights when defining a metric function.

```
(preference g1 (at end (ByIndustry ?finalStream)))
(preference g2 (at end (TableView ?finalStream)))
(preference g3 (at end (LinearIndex ?finalStream)))
```

7.3.3 Flow-based HTN Planning Problem with Preferences

In this section, we characterize a flow-based HTN planning problem with preferences and discuss the relationship between satisfying flows and optimal plans.

A Cascade flow pattern problem is a 2-tuple $P^F = (F, G)$, where $F = (\mathcal{G}(\mathcal{V}, \mathcal{E}), M)$ is a Cascade flow pattern (where $\mathcal{G}$ is a directed acyclic graph, and M is the main composite), and G is the set of Cascade goals. α is a satisfying flow for P^F if and only if α is a flow that meets the main composite M. A set of Cascade goals, G, is realizable if and only if there exists at least one satisfying flow for it.

Given the Cascade flow pattern problem P^F, we define the corresponding flow-based HTN planning problem with preferences as a 4-tuple $P = (s_0, w_0, D, \preceq)$, where: s_0 is the initial state consisting of a list of all tags and our special predicates; w_0 is the initial task network encoding of the main component M; D is the HTN planning domain, consisting of a set of operators and methods derived from the Cascade components $v \in \mathcal{V}$; and $\preceq$ is a preorder between plans dictated by the set of Cascade goals G.

Proposition 7.1 *Let $P^F = (F, G)$ be a Cascade flow pattern problem where G is realizable. Let $P = (s_0, w_0, D, \preceq)$ be the corresponding flow-based HTN planning problem with preferences. If α is an optimal plan for P, then we can construct a flow (based on α) that is a satisfying flow for the problem P^F.*

Consider the Cascade flow pattern problem $P^F = (F, G)$ with F shown in Figure 7.1 and G the *TableView* tag. Let P be the corresponding flow-based HTN problem with preferences. Then consider the following optimal plan for P: [TAQFileSource(1), ExtradeTradeInfo(1,2), VWAPByTime(2,3), ExtractQuoteInfo(1,4), BISimple(3,4,5), TableView(5,6)]. We can construct a flow in which the components mentioned in the plan are the vertices and the edges are determined by the numbered parameters corresponding to the generated output streams. The resulting graph is not only a flow, but a satisfying flow for the problem P^F.

7.4 Computation

In the previous section, we described a method that translates Cascade flow patterns and Cascade goals into an HTN planning problem with preferences. We also showed the relationship between optimal plans and satisfying flows. Now with a specification of preference-based HTN planning in hand we select **HTNPLAN-P**, our preference-based HTN planning system

(see Chapter 5), to compute these optimal plans that later get translated to satisfying flows for the original Cascade flow patterns. In this section, we focus on our proposed heuristic for this task, and describe how the required indexes for this heuristic can be generated in the preprocessing step.

7.4.1 Enhanced Lookahead Heuristic (*ELA*)

The enhanced lookahead function estimates the metric value achievable from a search node N. To estimate this metric value, we compute a set of reachable tags for each task within the initial task network. A set of tags are reachable by a task if they are reachable by any **plan** that extends from decomposing this task. Note, we assume that every nonprimitive task can eventually have a primitive decomposition.

The *ELA* function is an underestimate of the actual metric value because we ignore deleted tags, preconditions that may prevent achieving a certain tag, and we compute the set of all reachable tags, which in many cases is an overestimate. Nevertheless, this does not necessarily mean that the *ELA* function is a lower bound on the metric value of any plan extending node N. However, if it is a lower bound, then it will provide sound pruning (following Baier et al. [Baier et al., 2009]) if used within the **HTNPLAN-P** search algorithm and provably optimal plans can get generated (see Proposition 5.4). A pruning strategy is sound if no state is incorrectly pruned from the search space. That is whenever a node is pruned from the search space, we can prove that the metric value of any plan extending this node will exceed the current bound best metric. To ensure that the *ELA* is monotone, for each node we take the intersection of the reachable tags computed for this node's task and the set of reachable tags for its immediate predecessor.

Proposition 7.2 *The ELA function provides sound pruning if the preferences are all PDDL3 simple preferences over a set of predefined tags and the metric function is non-decreasing in the number of violated preferences and in the plan length.*

Proof: The *ELA* function is calculated by looking at a reachable set of tags for each task. Hence, it will regard as violated, preferences that have tags that do not appear in the set of reachable tags. This means that these tags are not reachable from node N. Given that we used the above trick to ensure the *ELA* function does not decrease and all our preferences are PDDL3 simple preferences over a set of predefined tags, the is-violated function for the hypothetical node N_E, that *ELA* is evaluating the metric for, is less than or equal to any node

N' reachable from node N (for each preference formula). Moreover, since we assume that the metric function is non-decreasing in the number of violated preferences and in plan length (NDVPL) [Baier et al., 2009], the metric function of the hypothetical node N_E will be less than or equal to the metric function of every successor node N' reachable from node N. This shows that the *ELA* evaluated at node N returns a lower bound on the metric value of any plan extending N. Therefore, the *ELA* function provides sound pruning. ■

Our notion of reachable tags is similar to the notion of "complete reachability set" in Marthi et al. ([Marthi et al., 2007]). While they find a superset of all reachable states by a "high-level" action a, we find a superset of all reachable tags by a task t; this can be helpful in proving a certain task cannot reach a goal. However, they assume that for each task a sound and complete description of it is given in advance, whereas we do not assume that. In addition, we are using this notion of reachability to estimate a heuristic which we implement in **HTNPLAN-P**. They use this notion for pruning plans and not necessarily in guiding the search toward a preferred plan. Marthi et al. in their follow up paper [Marthi et al., 2008] address the problem of finding an optimal plan with respect to action costs. This paper uses a notion of optimistic and pessimistic description, a generalization of their previous terms. This paper uses some notion of heuristic search in addition to limited hierarchical lookahead by exploiting an abstract lookahead tree. However, they again made an assumption that both the optimistic and pessimistic descriptions are given for each task in advance. Finding these descriptions, especially when the task network has loops both direct or indirect, is not trivial.

Next, we briefly overview our proposed solution of generating the set of all achievable tags. We will describe how we can generate this set from the description of the Cascade flow patterns as well as their corresponding HTN planning problem. Note, we assume that the set of tags that belong to the given Cascade pattern is known.

7.4.2 Generation from Cascade

To compute the *ELA* function from Cascade, we must compute an estimate of the tags "reachable" after any concrete or composite component in the pattern. We do this incrementally, by associating each component with an empty set of tags initially and then adding tags to these sets as we perform a traversal of the Cascade pattern.

We traverse the Cascade pattern in topological sort order – since Cascade patterns are acyclic graphs, we are guaranteed to find a topological sort not only for the entire pattern,

but also for any subgraph contained within a composite component. When we encounter a composite component during our traversal (e.g., *BIComputationCore* in Figure 7.1), we expand the composite and traverse its subgraph in topological sort order. For every component reached we propagate the complete set of predecessors, both concrete and composite, on the streams of the flow pattern. As an example, when reaching *ExtractTradeInfo* in Figure 7.1, the set of predecessors includes *FilterTradeByTickers*, *FilterTradeByIndustry*, *TAQTCPSource* and *TAQFileSource*. We add *ExtractTradeInfo* to this set and propagate it on the outgoing stream towards *CalculateVWAP*.

For every component we reach, we take the set of "positive" (i.e., non-negated) tags on all output ports and add it to the *ELA* entry for every predecessor, as well as for the current component itself. In essence, this "informs" predecessors that tags asserted on current output ports are reachable after them in the flow. Since in order to calculate the *ELA* function, we overestimate of the set of reachable tags, it is safe to ignore tags negated on the output. For example, if *ExtracTradeInfo* is tagged with *Trade*, this tag is be added to the *ELA* entry for all its four predecessors. Two adjustments to this general approach are needed to ensure correct computation of the *ELA* function.

First, when we compute the set of output tags for a component *C*, we must also include all tags for components that inherit from *C*, since a component can be replaced (or *will* be replaced if *C* is abstract) by one of its inheritance descendants in some possible plans. We can precompute sets of output tags for components that take inheritance into account before we start the traversal in time proportional to the number of components in the pattern.

Second, we must account for parallel branches of the flow pattern. For example, in Figure 7.1 the branches containing *ExtractTradeInfo* and *ExtractQuoteInfo* denote parallel computation, which means that a tag appearing in one branch is "reachable" after components on the other branch. Therefore, whenever we have multiple streams "joining" (e.g., multiple input ports into *CalculateVWAP*), we make additions to the entries of the *ELA* map to account for parallelism. For any two distinct branches B_1 and B_2, it is enough to take the current entry in the *ELA* map for the first component of B_1 and add it to the *ELA* entries for *all* components in B_2. In Figure 7.1, if *ExtractTradeInfo* is tagged with *Trade*, *ExtractQuoteInfo* with *Quote* and *CalculateVWAP* with *VWAP* (ignoring inheritance), the adjustment is made when we reach *CalculateBargainIndex*. The *ELA* for *ExtractTradeInfo* contains *Trade* and *VWAP*; we add these tags to the *ELA* entry for *ExtractQuoteInfo*. We also add *Quote* to the *ELA* entry for both *ExtractTradeInfo* and *CalculateVWAP*. This makes the total computational complexity at most cubic in the size of the pattern (the size of the planning domain).

7.4.3 Generation from HTN

Algorithm 7.2 shows pseudocode of our offline procedure that creates a set of tags for each task. It takes as input the planning domain D, a set of tasks (or a single task) w, a set of tags to carry over C, and a stack of parent tasks S. The algorithm should be called initially with the initial task network w_0, and $C = S = \emptyset$. The reason for why we keep track of the sets of tags to carry over is because we want to make sure we calculate not only a set of tags produced by a decomposition of a task network (or a task), but also we want to find a set of reachable tags for all possible plan extensions from this point on. The reason for why we keep a stack of parent tags D is to be able to detect a possible loop.

The call to GetReachableTags will produce a set of tags reachable by the set of tags w (produced by w and C). To track the produced tags for each task we use a map R (line 2). In order to keep track of the produced tags for each task we use a map R (line 2). If w is a task network then we consider three cases. If the task network is empty, we just return C. If we have an ordered task network then for each task (or a task network) within it, we call the algorithm starting with the right most task t_n, and updating the tag to carry c to take into account the tags produced later on. If the task network is unordered, then we call GetReachableTags twice, first to find out what each task produces, and then again with the updated set of carry tags. This ensures that we overestimate the reachable tags regardless of the execution order. Note that the first call to the algorithm (line 11) is with an empty C, because we have to call GetReachableTags again (line 15), this time with the updated set of tags to carry.

If w is a task then we are able to update its returned value $R[w]$. Hence we start off with initializing this value to an empty set (line 17). We then check to see if this task was ever called before to find out if a possible loop has occurred (line 18). If so we do not record anything for this task (because we are sure we will get to this task sometime later). We then return the carry together with a special tag x. This tag is special because it is not among one of our known tags, $AllTags$, and that it can take a parameter. If the task is primitive, we will find a set of tags it produces by looking at its add-list. If it is nonprimitive then we have to first find all methods that can be applied to decompose this task and their associated task networks. We then take a union of all tags produced by a call to GetReachableTags for each of these task networks.

In the presence of loops, there may be extra tags $x(w)$, where w is recursive, for stored tags of some tasks. In order to update the values for each task t that has the tag $x(w)$ (there could be multiple ones), we go over each stored tag for t (i.e., $R[t]$) and replace each $x(w)$ with the tags that are stored for w (i.e., $R[w]$). We may need to do this multiple times to reach a fix point,

1: **function GETREACHABLETAGS**(D, w, C, S)
2: **initialize** global Map R
3: $T \leftarrow \emptyset$
4: $x \leftarrow$ choose a name not in $AllTags$
5: **if** w is a task network **then**
6: **if** $w = \emptyset$ **then return** C
7: **else if** $w =$ (:ordered t_1 ... t_n) **then**
8: **for** i=n **to** 1 **do** $C \leftarrow \mathsf{GetReachableTags}(D, t_i, C, S)$
9: **else if** $w =$ (:unordered t_1 ... t_n) **then**
10: **for** i=1 **to** n **do**
11: $T_{t_i} \leftarrow \mathsf{GetReachableTags}(D, t_i, \emptyset, S)$
12: $T \leftarrow T_{t_i} \cup T$
13: **for** i=1 **to** n **do**
14: $C_{t_i} \leftarrow \bigcup_{j=1, j \neq i}^{n} T_j \cup C$
15: $\mathsf{GetReachableTags}(D, t_i, C_{t_i}, S)$
16: **else if** w is a task **then**
17: **if** $R[w]$ is not defined **then**$R[w] \leftarrow \emptyset$
18: **if** $t \in S$ **then return** $C \cup \{x(w)\}$ ▷ loop detected
19: **else if** t is a primitive task **then**
20: $a \leftarrow$ operator whose name match with t
21: $T \leftarrow$ add-list of $a \cap AllTags$
22: **else if** t is a nonprimitive task **then**
23: $M' \leftarrow \{m_1, ..., m_k\}$ such that $task(m_i)$ match with t
24: $U' \leftarrow \{U_1, ..., U_k\}$ such that $U_i = subtask(m_i)$
25: **for all** $U_i \in U'$ **do**
26: $T \leftarrow \mathsf{GetReachableTags}(D, U_i, C, S \cup \{t\}) \cup T$
27: $R[w] \leftarrow R[w] \cup T \cup C$
28: **return** $T \cup C$

Figure 7.2: A sketch of the GetReachableTags (D, w, C, S) algorithm.

that is adding $R[w]$ does not change the set of tags for t. Once a fixed point is reached for each task, we remove occurrences of $x(w)$ for all w.

7.5 Experimental Evaluation

We had two main objectives in our experimental analysis: (1) evaluate the applicability of our approach when dealing with large real-world applications or composition patterns, (2) evaluate the computational time gain that may result from the use of the proposed heuristic. To address our first objective, we took a suite of diverse Cascade flow pattern problems from patterns described by customers for IBM InfoSphere Streams and applied our techniques to create the

corresponding HTN planning problems with preferences. We then examined the performance of our preference-based HTN planner **HTNPLAN-P** (see Chapter 5) on the created planning problems. To address our second objective, we implemented the preprocessing algorithm discussed earlier and modified **HTNPLAN-P** to incorporate the new heuristic within its search strategy and then examined its performance. All our domains and problems are created from real patterns described for IBM InfoSphere Streams.

We had 7 domains and more than 50 HTN planning problems in our experiments. The HTN problems were constructed from patterns of varying sizes and therefore vary in hardness. For example, a problem can be harder if the pattern had many optional components or many choices, hence influencing the branching factor. Also a problem can be harder if the tags that are part of the Cascade goal appear in the harder to reach branches depending on the planner's search strategy. For **HTNPLAN-P**, it is harder if the goal tags appear in the very right side of the search space since it explores the search space from left to right if the heuristic is not informing enough. All problems were run for 10 minutes, and with a limit of 1GB per process. "OM" stands for "out of memory", and "OT" stands for "out of time".

We show a subset of our results in Figure 7.3. Columns 5 and 6 show the time in seconds to find an optimal plan. We ran **HTNPLAN-P** in its existing two modes: *LA* and *No-LA*. *LA* means that the search makes use of the *LA* (lookahead) heuristic (*No-LA* means it does not). Note, **HTNPLAN-P**'s other heuristics are used to break ties in both modes.We measure plan length for each solved problem as a way to show the number of generated output streams. We show the number of possible optimal plans (# of Plans) for each problem as an indication of the size of the search space. This number is a lower bound in many cases on the actual size of the search space. Note, we only find one optimal plan for each problem through the incremental search performed by **HTNPLAN-P**.

The results in Figure 7.3 illustrate the applicability and feasibility of our approach as we increase the difficulty of the problem. All problems were solved within 36 seconds by at least one of the two modes used. The results also indicate that, not surprisingly, the *LA* heuristic performs better at least in the harder cases (indicated in bold) partly because the *LA* heuristic forms a sampling of the search space. In some cases, due to the possible overhead in calculation of the *LA* heuristic, we did not see an improvement. Note that in some problems (e.g., 3rd domain Problems 3-5), an optimal plan was only found when the *LA* heuristic was used.

So far, the experiments we ran showed that an optimal solution was found within a reasonable time using the *LA* mode of the planner. Next, we identify cases where the planner will have difficulty finding an optimal solution. To show this we chose the third and fourth domain

Domain	Problem	Plan Length	# of Plans	*No-LA* Time (sec)	*LA* Time (sec)
1	1	10	243	0.00	0.00
	2	11	162	0.01	0.07
	3	11	81	0.04	0.05
	4	11	162	**0.10**	**0.01**
	5	11	81	**0.18**	**0.04**
2	1	10	243	0.00	0.00
	2	11	81	0.04	0.04
	3	11	162	0.04	0.05
	4	11	162	**0.13**	**0.01**
	5	11	81	**0.25**	**0.04**
3	1	38	2^{26}	0.08	0.08
	2	38	2^{13}	**276.11**	**0.09**
	3	38	2^{26}	**OM**	**0.13**
	4	38	2^{26}	**OM**	**0.14**
	5	20	2^{13}	**OM**	**0.14**
4	1	22	4608	0.01	0.01
	2	23	4608	0.02	0.02
	3	44	4608^2	0.08	0.08
	4	44	4608^2	0.09	0.11
	5	88	4608^4	0.59	0.59
	6	92	4608^4	**0.64**	**0.61**
	7	176	4608^8	4.50	4.50
	8	184	4608^8	**4.80**	**4.50**
	9	264	4608^{12}	14.88	14.88
	10	276	4608^{12}	**16.00**	**15.00**
	11	352	4608^{16}	35.65	35.65
	12	368	4608^{16}	**43.00**	**35.00**

Figure 7.3: Evaluating the applicability of our approach by running **HTNPLAN-P** (two modes) as we increase problem hardness.

and we tested with goals that appear deep in the right branch of the HTN search tree (or the search space). The result is shown in the right two most columns of Figure 7.4.

The results show that there are some hard problems for the *LA* heuristic. See Problem 10-12 for Domain 3 and Problems 17 and 18 for Domain 4. These problems are difficult because achieving the goal tags are difficult for the planner. There are a number of reasons (most are planner specific) for why these problems are hard. For example, in Problem 10, the way the domain is written indicates to the planner that the optional components need to be explored first, but the goal is only achievable with the non-optional choices; hence, exploring the particular branch that needs to get explored gets delayed. Some of these problems are easier because

the goal is to achieve easier to reach tags. It is also the case that the *LA* heuristic's sampling technique evaluates the right branch of the search space first; hence, it can provide the right level of guidance to the planner. However, from the result shown in Figure 7.4 we can conclude that while the *LA* heuristic greatly improves the time to compute an optimal plan, it may have difficulty when dealing with the hard to reach goal tags.

We had two sub-objectives in evaluating our proposed heuristic (the *ELA* heuristic): (1) to find out if it improves the time to find an optimal plan (2) to see if it can be combined with the planner's previous heuristics, namely the *LA* heuristic. *LA* then *ELA* (resp. *ELA* then *LA*) column indicates that we use a strategy in which we compare two nodes first based on their *LA* (resp. *ELA*) values, then break ties using their *ELA* (resp. *ELA*) values. In the Just *ELA* and Just *LA* columns we used either just *LA* or *ELA*. Finally in the *No-LA* column we did not use either heuristics.

The results (subset shown), also in Figure 7.4, show that the ordering of the heuristics does not seem to make any significant change in the time it took to find an optimal plan. That is, using the *ELA* heuristic combined with the *LA* heuristic at least for the problems considered does not seem to improve the performance of the planner compared to just using the *ELA* heuristic. The results also show that using the *ELA* heuristic alone performs best compared to the other search strategies. In particular, there are cases in which the planner fails to find an optimal plan when using *LA* or *No-LA*, but an optimal plan is found within the tenth of a second when using the *ELA* heuristic. To measure the gain in computation time from the *ELA* heuristic technique, we computed the percentage difference between the *LA* heuristic and the *ELA* heuristic times, relative to the worst time. We assigned a time of 600 to those that exceeded the time or memory limit. The results show that on average we gained 65% improvement when using the *ELA* heuristic for all the problems we used; we gained 90% improvement on the problems shown in Figure 7.4. This shows that the *ELA* heuristic seems to significantly improve the time it takes to find an optimal plan.

7.6 Summary and Discussion

There is a large body of work that explores the use of AI planning for the task of automated WSC as overviewed in Chapter 2. Following this line of research, there is also a line of work that explores the use of AI planning for the task of composing information flows, an analogous problem to WSC (e.g., [Riabov and Liu, 2006, Ranganathan et al., 2009]). In this chapter, we explored the possibility of applying our techniques to stream processing applications, applica-

Domain	Problem	*LA* then *ELA* Time (s)	*ELA* then *LA* Time (s)	Just *ELA* Time (s)	Just *LA* Time (s)	*No-LA* Time (s)
3	6	0.16	0.16	**0.06**	0.13	OM
	7	1.70	0.17	**0.07**	0.13	OM
	8	1.70	1.70	**0.07**	1.50	OM
	9	1.80	1.80	**0.07**	1.60	OM
	10	1.70	1.70	**0.07**	OM	OM
	11	1.40	1.40	**0.07**	OM	OM
	12	1.40	1.30	**0.07**	OM	OM
4	13	0.58	0.45	**0.02**	0.56	0.12
	14	2.28	2.24	**0.07**	3.01	0.38
	15	14.40	14.28	**0.44**	19.71	1.44
	16	104.70	102.83	**3.15**	147	8.00
	17	349.80	341.20	**10.61**	486.53	18.95
	18	OT	OT	**24.45**	OT	40.20

Figure 7.4: Evaluation of the *ELA* heuristic.

tions that involve composing information flows. In particular, we examined the correspondence between HTN planning and automated composition of flow-based applications. We proposed the use of HTN planning and to that end proposed a technique for creating an HTN planning problem with user preferences from Cascade representation of flow patterns and user-specified Cascade goals. This opens the door to increased expressive power in flow pattern languages such as Cascade, for instance the use of recursive structures (e.g., loops), user preferences, and additional composition constraints.

We also exploited the use of a lookahead heuristic and showed that it improves the performance of **HTNPLAN-P** for the domains we used, making it comparable with domain-specific planners (e.g., MARIO). The proposed heuristic is general enough that it can be used within other HTN planners. We have performed extensive experimentation that showed the applicability and promise of the proposed approach for the problem of automated composition of flow-based applications.

Chapter 8

Conclusion and Future Work

8.1 Conclusion

In this thesis, we established the correspondence between generating a customized composition of Web services and non-classical AI planning where the objective of the planning problem is specified as a form of control knowledge, such as a workflow or template, together with a set of constraints to be optimized or enforced. This enabled us to bring to bear many of the theoretical and computational advances in reasoning about actions to the task of WSC. In particular, we exploited and advanced techniques in preference-based planning to generate customized compositions of Web services. Moreover, we established that techniques in (preference-based) planning can indeed provide a computational basis for the development of effective, state-of-the-art techniques for generating customized compositions of Web services. While our research has been motivated by Web services, the theory and techniques we have developed are amenable to many analogous problems such as business process modeling, component software composition, requirements engineering, Big Data and stream processing, some of which were discussed in Chapter 7.

To evaluate our thesis, in Chapter 3 we first characterized the WSC problem with customization, establishing that there is a correspondence between generating a customized composition of Web services and a non-classical planning problem where the objective of the planning problem can be specified as a composition template together with a set of constraints. While a composition template can be represented in a variety of different ways, in this thesis, we consider two representations: Golog and HTNs. In Chapter 4, we developed specification languages that met our set of desirable criteria for soft and hard constraints specification. Given the characterization of the WSC problem with customization, the specification of composition

templates, and the specification of constraints, we developed algorithms and techniques, based on heuristic search, to compute customized compositions. Furthermore, in Chapter 5 we discussed our four systems, each of which differ with respect to the specification of the composition template, Golog or HTNs, the specification of preferences, $\mathcal{LPP}$, $\mathcal{LPH}$, or PDDL3, the specification of hard constraints, and the nature of the heuristics they use. In Chapter 6, we addressed the information-gathering problem when generating customized compositions and discussed how localized optimization can increase the performance of the search. Finally, in Chapter 7 we discussed some of the applications of our proposed languages and techniques.

In the rest of this chapter, we repeat the problems we addressed and our major contributions. Finally, we discuss potential future work.

8.2 Problems and Contributions

The general problem we face in this thesis is to investigate principled techniques for composing Web services, that support user customization. This manifests itself in a number of specific research challenges that we identify and address in this thesis.

Characterize the WSC Problem with Customization We characterized the WSC problem with customization, where the objective of the planning problem is represented in some form of composition template (either in Golog or HTNs) together with a set of constraints that need to be optimized or enforced. This characterization enables us to generate customized compositions of Web services through non-classical planning.

Specify the Soft and Hard Constraints We designed a set of desirable criteria, evaluated the existing specification languages with respect to this set, and extended the existing languages to meet our set of desirable criteria. In particular, we proposed a language called $\mathcal{LPH}$ and extended PDDL3 with HTN-specific preference constructs. Moreover, we proposed a means of specifying and optimizing preferences over service and data selection using our extension. We provide a semantics for our preference languages through the situation calculus [Reiter, 2001].

Compute Optimized Compositions We proposed algorithms that integrate preference-based reasoning, advanced state-of-the-art planning with preferences, proved properties of these algorithms, implemented and evaluated systems to show the applicability of our proposed approach. In particular, we have developed four systems: **GOLOGPREF**, **HTNPLAN**,

HTNPLAN-P, and **HTNWSC-P**. Each of these systems uses heuristic search but differ with respect to the forms of the composition templates, Golog or HTNs, the specification of soft constraints, $\mathcal{LPP}$, $\mathcal{LPH}$, or PDDL3, and the specification of the hard constraints, and the nature of the heuristics they use.

Execute and Optimize We proposed a notion of middle-ground execution system for the WSC problem with customization that interleaves online information gathering with offline search as deemed necessary. We proposed to further improve the search by performing optimization of data choices locally, whenever possible, while still guaranteeing that the choice selected does not eliminate the globally optimal solution. We showed that our approach to data optimization can greatly improve both the quality of compositions and the speed with which they are generated.

Explore Applicability Beyond WSC We investigated how to use and adapt our framework to address two applications: requirements engineering and stream processing. We proposed different uses of HTN planning with preferences and adapted our techniques accordingly. For example, we proposed a heuristic tailored to stream processing applications in order to improve the performance of search in generating an optimal solution.

8.3 Future Work

There are a number of future directions for the work presented in this thesis. Next, we list a subset of these.

Incorporate Diagnosis and WSC In our recent work [Sohrabi et al., 2011], we addressed the problem of generating explanations for observed behaviour with respect to a model of the behaviour of a dynamical system with a focus of generating *preferred* explanations. In particular, we addressed the problem of how to specify the preference criteria, and how to compute preferred explanations using planning technology. This problem arises in a diversity of applications including diagnosis of dynamical systems. In particular, our work in [Sohrabi et al., 2010] focuses on establishing the relationship between diagnosis and planning, and generating (high-quality) diagnoses using planning technology. It remains to show how this line of research can help with diagnosing faulty services and how it can fit within our framework of the WSC problem with customization.

Identify Non-Interacting Operators In Chapter 6 we gave a condition under which performing localized data optimization is sound and discussed how this relates to identifying non-interacting operators. We also gave some syntactic criteria in order to identify non-interacting operators with respect to the domain. It remains to determine conditions for when an operator is non-interacting with respect to the preferences or hard constraints. Further, given these conditions, identifying non-interacting operators automatically is also left for future work.

Mixed Initiate Planning Another interesting future direction would be to address dynamic and changing preferences and constraints. An ideal solution to address this problem needs to be mixed-initiate. To that end, we need to design a user-friendly interface that possibly not only takes the user's preferences, objectives, policies, but also interacts with the user in a mixed-initiative manner during the composition construction time. Hence, instead of soliciting preferences from the user a priori, the system solicits user preferences as they become relevant during the planning process.

Explore Other Applications We claim that the techniques and languages developed in this thesis are general enough that they can be applied to other applications including multi-agent systems, business process modeling, and social and computational behaviour modeling and verification. We partially supported this claim in Chapter 7 by applying our techniques in two applications, requirements engineering and stream processing. It remains to show how our techniques are applicable to other applications such as business process modeling.

Appendix A

Proof of Theorem 6.1

The proof[1] of Theorem 6.1 relies on the following lemma.

Definition A.1 (Search trace) *Search trace is a sequence of methods and operator instances that have been applied to decompose a task network.*

Recall D^I includes HTN operators and methods that externally call the set of information sources X during planning. In the following Lemma we assume these operators and methods can be identified. Also, in order to address the case where we may delay a call to an information source, we assume s_0 (the complete initial state) includes the information about our placeholder values.

Lemma 1 *Assuming the IRP assumption holds, let $\mathcal{P}^{\mathcal{I}} = (s_0^I, w_0, D^I, \preceq, X)$ be an incomplete HTN planning problem with preferences, and let $\mathcal{P}^{\mathcal{C}} = (s_0, w_0, D, \preceq)$ be a complete HTN planning problem with preferences that is consistent with $\mathcal{P}^{\mathcal{I}}$. Let st be a search trace of the search tree for $\mathcal{P}^{\mathcal{I}}$, and st' be a search trace obtained from st by removing the operators and methods that externally call the set of information sources X during planning. Then st is a search trace of the search tree for $\mathcal{P}^{\mathcal{C}}$.*

Proof: The proof is by induction of the length d of a search trace, assuming the IRP assumption holds.

Let $d = 0$, then since both problems $\mathcal{P}^{\mathcal{I}}$ and $\mathcal{P}^{\mathcal{C}}$ have the same initial task network w_0, the root of their search tree is identical. This proves the base case.

[1]The steps of this proof is similar to the correctness proof of **ENQUIRER** [Kuter et al., 2005]

Now assume that every search trace of the search tree with length d for $\mathcal{P}^{\mathcal{I}}$ once stripped from the operators and methods that correspond to X is also a search trace of the search tree for $\mathcal{P}^{\mathcal{C}}$. Let $n = \langle s, w, partialP \rangle$ be the leaf node of the search trace with length d, where s is a plan state, w is a task network, and $partialP$ is a partial plan. If $w = \emptyset$ then, n is a leaf node in the search tree for both problems and by the inductive hypothesis the proof holds. If w is not $\emptyset$, let t be the task selected next from the task network w. Then, we consider the following two cases: (1) t correspond to gathering information, hence the search tree for $\mathcal{P}^{\mathcal{I}}$ once stripped from the operators and methods that correspond to X (descenders of t) is also a search trace of the search tree for $\mathcal{P}^{\mathcal{C}}$ (2) t does not correspond to gathering information. Then we consider the following two sub cases (1) t is a primitive task. Then we know that the operator that can be applied is a world-altering operator and is in D and D^I. Then both problems result in the same search node that is obtained by removing t from w and updating the partial plan $partialP$, (2) t is nonprimitive. Then we know that the methods application to t are all in D (as they do not correspond to X). Also from the definition of consistency Definition 6.7 we know $s_0^I \cup \delta(X) \subseteq s_0$, hence the set of methods applicable to t (the children of node n) for problem $\mathcal{P}^{\mathcal{I}}$ is a subset of the children of nodes n for the problem $\mathcal{P}^{\mathcal{I}}$ for depth $d + 1$. Note again that we assumed that the place holder values are in s_0. Hence, the set of search traces going out of the leaf node n for problem $\mathcal{P}^{\mathcal{I}}$ is a subset of those going out of the leaf node n for the problem $\mathcal{P}^{\mathcal{C}}$. ■

Note, the above Lemma is in one direction because $s_0^I \cup \delta(X) \subseteq s_0$. That is we showed that the search trace for an incomplete problem corresponds to a search tree for a complete version of the problem if we ignore the operators and methods that correspond to gathering information from sources X.

Now back to the proof of Theorem 6.1. Note again that we assume that the postprocessing step is performed on the generated plan. That is the call to those information-gathering services that have been delayed are made and the placeholder is replaced with an appropriate choice. Here we repeat the statement of the Theorem 6.1.

Theorem A.1 *Assuming the IRP assumption holds, let $\mathcal{P}^{\mathcal{IW}} = (s_0^I, C, K, \phi_{soft}, Y)$ be an incomplete OWL-S WSC problem with preferences, and let $\mathcal{P}^{\mathcal{I}} = (s_0^I, w_0, D^I, \preceq, X)$ be an incomplete HTN planning problem with preferences that is equivalent to $\mathcal{P}^{\mathcal{IW}}$. A plan $\boldsymbol{a} = o_1 ... o_k$ is a solution to $\mathcal{P}^{\mathcal{I}}$ if and only if $\pi = p_1 ... p_k$ is a solution to (a composition for) $\mathcal{P}^{\mathcal{IW}}$ such that o_i, $1 \leq i \leq k$ are the primitive operators that correspond to the atomic process p_i.*

Assuming the IRP assumption holds, let $\mathcal{P}^{\mathcal{IW}} = (s_0^I, C, K, \phi_{soft}, Y)$ be an incomplete OWL-S WSC problem with preferences, and let $\mathcal{P}^{\mathcal{I}} = (s_0^I, w_0, D^I, \preceq, X)$ be an incomplete HTN planning problem with preferences that is equivalent to $\mathcal{P}^{\mathcal{IW}}$. By the definition of equivalency Definition 6.9 we know that w_0 is generated by our modified OWL-S to HTN translation for the OWL-S process C, D^I is the HTN domain description generated by running our modified OWL-S to HTN translation for the collection of OWL-S process models K, $\preceq$ is a preorder between plans as dictated by ϕ_{soft}, and X is the set of information sources that produce the same information as the services Y, that is $\delta(X) = \delta(Y)$.

The proof is done by showing that there is a bijection between the set of plans for $\mathcal{P}^{\mathcal{I}}$ and the set of compositions for $\mathcal{P}^{\mathcal{IW}}$. The proof relies on looking at the complete version of the two problems.

Let $\mathcal{P}^{\mathcal{CW}} = (s_0, C, K^c, \phi_{soft})$ be a complete version of the OWL-S WSC problem with preferences $\mathcal{P}^{\mathcal{IW}}$ such that $s_0 = s_0^I \cup \delta(Y)$ and K^c includes only atomic processes with effects (they are world altering). Since both $\mathcal{P}^{\mathcal{CW}}$ and $\mathcal{P}^{\mathcal{IW}}$ have the same goal process C and the same set of information is available for both problems (one has all the information in the initial state, the other gathers information during planning), then the set of compositions for $\mathcal{P}^{\mathcal{IW}}$ once the atomic process with outputs are removed from them is the same as the set of compositions for $\mathcal{P}^{\mathcal{CW}}$.

Let $\mathcal{P}^{\mathcal{C}} = (s_0, w_0, D, \preceq)$ be a complete HTN planning problem with preferences that corresponds to (simpler version of equivalence where we have complete information) $\mathcal{P}^{\mathcal{CW}}$. Given the correctness of the translation [Sirin et al., 2005b] and given that our modification to the translator did not change the correctness of the translation, we know from [Sirin et al., 2005b] that there is a bijection between the set of plans for $\mathcal{P}^{\mathcal{C}}$ and the set of compositions $\mathcal{P}^{\mathcal{CW}}$.

Then by Lemma 1 and knowing that (1) $s_0 = s_0^I \cup \delta(X)$ since $s = s_0^I \cup \delta(Y)$ and (2) postprocessing step replaced the values of the place holders with their real values, we know that the set of plans for $\mathcal{P}^{\mathcal{C}}$ is the same as the set of plans for $\mathcal{P}^{\mathcal{I}}$ once the atomic process with outputs are removed from them.

Hence, it follows that there is a bijection between between the set of plans for $\mathcal{P}^{\mathcal{I}}$ and the set of compositions for $\mathcal{P}^{\mathcal{IW}}$. ■

Bibliography

[Alrifai and Risse, 2009] Alrifai, M. and Risse, T. (2009). Combining global optimization with local selection for efficient QoS-aware service composition. In *Proceedings of the 18th International World Wide Web Conference (WWW)*, pages 881–890.

[Andrews and et al, 2002] Andrews, T. and et al (2002). 1.1 online: http://www.ibm.com/developerworks/library/specification/ws-bpel/.

[Au and Nau, 2007] Au, T.-C. and Nau, D. S. (2007). Reactive query policies: A formalism for planning with volatile external information. In *Proceedings of the IEEE Symposium on Computational Intelligence and Data Mining (CIDM)*, pages 243–250.

[Bacchus and Kabanza, 1998] Bacchus, F. and Kabanza, F. (1998). Planning for temporally extended goals. *Annals of Mathematics and Artificial Intelligence*, 22(1-2):5–27.

[Bacchus and Kabanza, 2000] Bacchus, F. and Kabanza, F. (2000). Using temporal logics to express search control knowledge for planning. *AI Magazine*, 16:123–191.

[Baier et al., 2009] Baier, J., Bacchus, F., and McIlraith, S. (2009). A heuristic search approach to planning with temporally extended preferences. *Artificial Intelligence*, 173(5-6):593–618.

[Baier et al., 2007] Baier, J. A., Bacchus, F., and McIlraith, S. A. (2007). A heuristic search approach to planning with temporally extended preferences. In *Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1808–1815.

[Battle et al., 2005] Battle, S., Bernstein, A., Boley, H., Grosof, B., Gruninger, M., Hull, R., Kifer, M., Martin, D., McIlraith, S., McGuinness, D., Su, J., and Tabet, S. (2005). Semantic Web service ontology (SWSO) first-order logic ontology for Web services (FLOWS). www.daml.org/services/swsl/report/.

[Benton et al., 2012] Benton, J., Coles, A. J., and Coles, A. (2012). Temporal planning with preferences and time-dependent continuous costs. In *Proceedings of the 22nd International Conference on Automated Planning and Scheduling (ICAPS)*.

[Benton et al., 2009] Benton, J., Do, M. B., and Kambhampati, S. (2009). Anytime heuristic search for partial satisfaction planning. *Artificial Intelligence*, 173(5-6):562–592.

[Berardi et al., 2005] Berardi, D., Calvanese, D., Giacomo, G. D., Lenzerini, M., and Mecella, M. (2005). Automatic service composition based on behavioral descriptions. *International Journal of Cooperative Information Systems*, 14(4):333–376.

[Bertoli et al., 2010] Bertoli, P., Pistore, M., and Traverso, P. (2010). Automated composition of web services via planning in asynchronous domains. *Artificial Intelligence*, 174(3-4):316–361.

[Bienvenu et al., 2006] Bienvenu, M., Fritz, C., and McIlraith, S. (2006). Planning with qualitative temporal preferences. In *Proceedings of the 10th International Conference on Knowledge Representation and Reasoning (KR)*, pages 134–144.

[Bienvenu et al., 2011] Bienvenu, M., Fritz, C., and McIlraith, S. A. (2011). Specifying and computing preferred plans. *Artificial Intelligence*, 175(7–8):1308–1345.

[Bonet and Geffner, 2001] Bonet, B. and Geffner, H. (2001). Planning as heuristic search. *Artificial Intelligence*, 129(1-2):5–33.

[Box and et al, 2003] Box, D. and et al (2003). 1.1. Online:www.w3.org/TR/SOAP/.

[Bruijn et al., 2006] Bruijn, J. D., Lausen, H., Polleres, A., and Fensel, D. (2006). The Web service modeling language WSML: An overview. Technical report, DERI.

[Burch et al., 1992] Burch, J. R., Clarke, E. M., McMillan, K. L., Dill, D. L., and Hwang, L. J. (1992). Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170.

[Calvanese et al., 2008] Calvanese, D., Giacomo, G. D., Lenzerini, M., Mecella, M., and Patrizi, F. (2008). Automatic service composition and synthesis: the Roman Model. *IEEE Data Engineering Bulletin*, 31(3):18–22.

[Casati et al., 2000] Casati, F., Ilnicki, S., Jin, L.-j., Krishnamoorthy, V., and Shan, M.-C. (2000). Adaptive and dynamic service composition in eFlow. In *Proceedings of the 12th International Conference on Advanced Information Systems Engineering (CAiSE)*, pages 13–31.

[Cheung and Gil, 2007] Cheung, W. K.-W. and Gil, Y. (2007). Privacy enforcement through workflow systems in e-science and beyond. In *Proceedings of the ISWC'07 Workshop on Privacy Enforcement and Accountability with Semantics (PEAS)*.

[Chinnici and et al, 2001] Chinnici, R. and et al (2001). 1.2. Online: www.w3.org/TR/wsdl/.

[Chun et al., 2004] Chun, S. A., Atluri, V., and Adam, N. R. (2004). Policy-based Web service composition. In *Proceedings of the 14th International Workshop on Research Issues on Data Engineering: Web Services for E-Commerce and E-Government Applications (RIDE)*, pages 85–92. IEEE Computer Society.

[Chun et al., 2005] Chun, S. A., Atluri, V., and Adam, N. R. (2005). Using semantics for policy-based Web service composition. *Distributed and Parallel Databases*, 18(1):37–64.

[Cimatti et al., 1997] Cimatti, A., Giunchiglia, F., Giunchiglia, E., and Traverso, P. (1997). Planning via model checking: A decision procedure for *AR*. In *Proceedings of the 4th European Conference on Planning (ECP)*, pages 130–142.

[Coles and Coles, 2011] Coles, A. J. and Coles, A. (2011). Lprpg-p: Relaxed plan heuristics for planning with preferences. In *Proceedings of the 21st International Conference on Automated Planning and Scheduling (ICAPS)*.

[Currie and Tate, 1991] Currie, K. and Tate, A. (1991). O-plan: The open planning architecture. *Artificial Intelligence*, 52(1):49–86.

[De Giacomo et al., 2000] De Giacomo, G., Lespérance, Y., and Levesque, H. (2000). ConGolog, a concurrent programming language based on the situation calculus. *Artificial Intelligence*, 121(1-2):109–169.

[Edelkamp, 2006] Edelkamp, S. (2006). Optimal symbolic PDDL3 planning with MIPS-BDD. In *5th International Planning Competition Booklet (IPC-2006)*, pages 31–33, Lake District, England.

[Emerson, 1990] Emerson, E. A. (1990). Temporal and modal logic. *Handbook of theoretical computer science: formal models and semantics*, B:995–1072.

[Erol et al., 1994] Erol, K., Hendler, J. A., and Nau, D. S. (1994). UMCP: A sound and complete procedure for hierarchical task-network planning. In *Artificial Intelligence Planning Systems*, pages 249–254.

[Fox and Long, 2002] Fox, M. and Long, D. (2002). Domains of the 3rd international planning competition. In *Proceedings of the 6th International Conference on Artificial Intelligence Planning and Scheduling (AIPS)*.

[Fritz et al., 2008] Fritz, C., Baier, J. A., and McIlraith, S. A. (2008). Congolog, sin trans: Compiling congolog into basic action theories for planning and beyond. In *Proceedings of the 11th International Conference on Knowledge Representation and Reasoning (KR)*, pages 600–610.

[Gabaldon, 2002] Gabaldon, A. (2002). Programming hierarchical task networks in the situation calculus. In *AIPS'02 Workshop on On-line Planning and Scheduling*.

[Gabaldon, 2004] Gabaldon, A. (2004). Precondition control and the progression algorithm. In *Proceedings of the 9th International Conference on Knowledge Representation and Reasoning (KR)*, pages 634–643. AAAI Press.

[Gedik et al., 2008] Gedik, B., Andrade, H., lung Wu, K., Yu, P. S., and Doo, M. (2008). SPADE: the system s declarative stream processing engine. In *ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 1123–1134.

[Gerevini et al., 2009] Gerevini, A., Haslum, P., Long, D., Saetti, A., and Dimopoulos, Y. (2009). Deterministic planning in the fifth international planning competition: PDDL3 and experimental evaluation of the planners. *Artificial Intelligence*, 173(5-6):619–668.

[Gerevini and Long, 2005] Gerevini, A. and Long, D. (2005). Plan constraints and preferences for PDDL3. Technical Report 2005-08-07, Department of Electronics for Automation, University of Brescia, Brescia, Italy.

[Gerevini et al., 2003] Gerevini, A., Saetti, A., and Serina, I. (2003). Planning through stochastic local search and temporal action graphs in lpg. *Journal of Artificial Intelligence Research*, 20:239–290.

[Gerth et al., 1995] Gerth, R., Peled, D., Vardi, M. Y., and Wolper, P. (1995). Simple on-the-fly automatic verification of linear temporal logic. In *Proceedings of the 15th International Symposium on Protocol Specification, Testing and Verification (PSTV)*, pages 3–18.

[Ghallab et al., 2004] Ghallab, M., Nau, D., and Traverso, P. (2004). *Hierarchical Task Network Planning. Automated Planning: Theory and Practice*. Morgan Kaufmann.

[Gil et al., 2004] Gil, Y., Deelman, E., Blythe, J., Kesselman, C., and Tangmunarunkit, H. (2004). Artificial intelligence and grids: Workflow planning and beyond. *IEEE Intelligent Systems*, 19(1):26–33.

[Green, 1969] Green, C. (1969). Application of theorem proving to problem solving. In *Proceedings of the 1st International Joint Conference on Artificial Intelligence (IJCAI)*, pages 219–240.

[Hamadi and Benatallah, 2003] Hamadi, R. and Benatallah, B. (2003). A Petri net-based model for Web service composition. In *Proceedings of the 14th Australasian database conference (ADC)*, pages 191–200. Australian Computer Society, Inc.

[Helmert, 2006] Helmert, M. (2006). The Fast Downward planning system. *Journal of Artificial Intelligence Research*, 26:191–246.

[Hoffmann, 2001] Hoffmann, J. (2001). FF: The fast-forward planning system. *AI Magazine*, 22(3):57–62.

[Hoffmann et al., 2009a] Hoffmann, J., Bertoli, P., Helmert, M., and Pistore, M. (2009a). Message-based Web service composition, integrity constraints, and planning under uncertainty: A new connection. *Journal of Artificial Intelligence Research*, 35:49–117.

[Hoffmann et al., 2007] Hoffmann, J., Bertoli, P., and Pistore, M. (2007). Web service composition as planning, revisited: In between background theories and initial state uncertainty. In *Proceedings of the 22nd National Conference on Artificial Intelligence (AAAI)*, pages 1013–1018.

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

[Hoffmann et al., 2009b] Hoffmann, J., Weber, I., and Governatori, G. (2009b). On compliance checking for clausal constraints in annotated process models. *Journal Information Systems Frontiers*.

[Hoffmann et al., 2008] Hoffmann, J., Weber, I., Scicluna, J., Kaczmarek, T., and Ankolekar, A. (2008). Combining scalability and expressivity in the automatic composition of semantic Web services. In *Proceedings of the 8th International Conference on Web Engineering (ICWE)*, pages 98–107.

[Horrocks et al., 2003] Horrocks, I., Patel-Schneider, P., and van Harmelen, F. (2003). From $\mathcal{SHIQ}$ and RDF to OWL: The making of a Web ontology language. *Journal of Web Semantics*, 1(1):7–26.

[Hsu et al., 2006] Hsu, C.-W., Wah, B., Huang, R., and Chen, Y. (2006). Handling soft constraints and goals preferences in SGPlan. In *5th International Planning Competition Booklet (IPC-2006)*, pages 39–41, Lake District, England.

[Hsu et al., 2007] Hsu, C.-W., Wah, B., Huang, R., and Chen, Y. (2007). Constraint partitioning for solving planning problems with trajectory constraints and goal preferences. In *Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1924–1929, Hyderabad, India.

[Hull, 2005] Hull, R. (2005). Web services composition: A story of models, automata, and logics. In *Proceedings of the IEEE International Conference on Web Services (ICWS-05)*.

[Hull and Su, 2005] Hull, R. and Su, J. (2005). Tools for composite web services: a short overview. *SIGMOD Record*, 34:86–95.

[Kambhampati, 2007] Kambhampati, S. (2007). Model-lite planning for the web age masses: The challenges of planning with incomplete and evolving domain models. In *Proceedings of the 22nd National Conference on Artificial Intelligence (AAAI)*, pages 1601–1604.

[Kambhampati et al., 1998] Kambhampati, S., Mali, A. D., and Srivastava, B. (1998). Hybrid planning for partially hierarchical domains. In *Proceedings of the 15th National Conference on Artificial Intelligence (AAAI)*, pages 882–888.

[Klusch et al., 2005] Klusch, M., Gerber, A., and Schmidt, M. (2005). Semantic Web service composition planning with OWLS-Xplan. In *AAAI-05 Fall Symposium*.

[Knoblock, 1995] Knoblock, C. A. (1995). Planning executing sensing and replanning for information gathering. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1686–1693.

[Kolovski et al., 2005] Kolovski, V., Parsia, B., Katz, Y., and Hendler, J. A. (2005). Representing Web service policies in OWL-DL. In *Proceedings of the 4th International Semantic Web Conference (ISWC)*, pages 461–475.

[Kuter and Nau, 2004] Kuter, U. and Nau, D. S. (2004). Forward-chaining planning in nondeterministic domains. In *Proceedings of the 19th National Conference on Artificial Intelligence (AAAI)*, pages 513–518.

[Kuter et al., 2009] Kuter, U., Nau, D. S., Pistore, M., and Traverso, P. (2009). Task decomposition on abstract states, for planning under nondeterminism. *Artificial Intelligence*, 173(5-6):669–695.

[Kuter et al., 2004] Kuter, U., Sirin, E., Nau, D. S., Parsia, B., and Hendler, J. A. (2004). Information gathering during planning for Web service composition. In *Proceedings of the 3rd International Semantic Web Conference (ISWC)*, pages 335–349.

[Kuter et al., 2005] Kuter, U., Sirin, E., Parsia, B., Nau, D. S., and Hendler, J. A. (2005). Information gathering during planning for Web service composition. *Journal of Web Semantics*, 3(2-3):183–205.

[Kvarnström and Doherty, 2000] Kvarnström, J. and Doherty, P. (2000). Talplanner: A temporal logic based forward chaining planner. *Annals of Mathematics Artificial Intelligence*, 30(1-4):119–169.

[Lago et al., 2002] Lago, U. D., Pistore, M., and Traverso, P. (2002). Planning with a language for extended goals. In *Proceedings of the 18th National Conference on Artificial Intelligence (AAAI)*, pages 447–454.

[Lécué, 2009] Lécué, F. (2009). Optimizing QoS-aware semantic Web service composition. In *Proceedings of the 8th International Semantic Web Conference (ISWC)*, pages 375–391.

[Lécué et al., 2008] Lécué, F., Léger, A., and Delteil, A. (2008). DL reasoning and AI planning for Web service composition. In *Web Intelligence*, pages 445–453.

[Liaskos et al., 2010] Liaskos, S., McIlraith, S. A., Sohrabi, S., and Mylopoulos, J. (2010). Integrating preferences into goal models for requirements engineering. In *Proceedings of the 10th International Requirements Engineering Conference (RE)*, pages 135–144.

[Liaskos et al., 2011] Liaskos, S., McIlraith, S. A., Sohrabi, S., and Mylopoulos, J. (2011). Representing and reasoning about preferences in requirements engineering. *Requirements Engineering*, 16:227–249. 10.1007/s00766-011-0129-9.

[Lin et al., 2008] Lin, N., Kuter, U., and Sirin, E. (2008). Web service composition with user preferences. In *Proceedings of the 5th European Semantic Web Conference (ESWC-08)*, pages 629–643.

[Manna and Waldinger, 1980] Manna, Z. and Waldinger, R. J. (1980). A deductive approach to program synthesis. *ACM Transactions on Programming Languages and Systems*, 2(1):90–121.

[Marthi et al., 2007] Marthi, B., Russell, S. J., and Wolfe, J. (2007). Angelic semantics for high-level actions. In *Proceedings of the 17th International Conference on Automated Planning and Scheduling (ICAPS)*, pages 232–239.

[Marthi et al., 2008] Marthi, B., Russell, S. J., and Wolfe, J. (2008). Angelic hierarchical planning: Optimal and online algorithms. In *Proceedings of the 18th International Conference on Automated Planning and Scheduling (ICAPS)*, pages 222–231.

[Martin et al., 2007] Martin, D., Burstein, M., McDermott, D., McIlraith, S., Paolucci, M., Sycara, K., McGuinness, D., Sirin, E., and Srinivasan, N. (2007). Bringing semantics to Web services with OWL-S. *World Wide Web Journal*, 10(3):243–277.

[McDermott, 1998] McDermott, D. V. (1998). PDDL — The Planning Domain Definition Language. Technical Report TR-98-003/DCS TR-1165, Yale Center for Computational Vision and Control.

[McDermott, 2002] McDermott, D. V. (2002). Estimated-regression planning for interactions with Web services. In *Proceedings of the 6th International Conference on Artificial Intelligence Planning and Scheduling (AIPS)*, pages 204–211.

[McIlraith and Fadel, 2002] McIlraith, S. and Fadel, R. (2002). Planning with complex actions. In *Proceedings of the 9th International Workshop on Non-Monotonic Reasoning (NMR-02)*, pages 356–364.

[McIlraith and Son, 2002] McIlraith, S. and Son, T. (2002). Adapting Golog for composition of semantic Web services. In *Proceedings of the 8th International Conference on Knowledge Representation and Reasoning (KR)*, pages 482–493.

[McIlraith et al., 2001] McIlraith, S., Son, T., and Zeng, H. (2001). Semantic Web services. *IEEE Intelligent Systems. Special Issue on the Semantic Web*, 16(2):46–53.

[Murata, 1989] Murata, T. (1989). Properties, analysis and applications. *IEEE*, 77(4):541–580.

[Myers, 2000] Myers, K. L. (2000). Planning with conflicting advice. In *Proceedings of the 5th International Conference on Artificial Intelligence Planning and Scheduling (AIPS)*, pages 355–362.

[Narayanan and McIlraith, 2002] Narayanan, S. and McIlraith, S. (2002). Simulation, verification and automated composition of Web services. In *Proceedings of the 11th International World Wide Web Conference (WWW)*.

[Nau et al., 2001] Nau, D., Muñoz-Avila, H., Cao, Y., Lotem, A., and Mitchell, S. (2001). Total-order planning with partially ordered subtasks. In *Proceedings of the 17th International Joint Conference on Artificial Intelligence (IJCAI)*, pages 425–430.

[Nau et al., 2003] Nau, D. S., Au, T.-C., Ilghami, O., Kuter, U., Murdock, J. W., Wu, D., and Yaman, F. (2003). SHOP2: An HTN planning system. *Journal of Artificial Intelligence Research*, 20:379–404.

[Palacios and Geffner, 2007] Palacios, H. and Geffner, H. (2007). From conformant into classical planning: Efficient translations that may be complete too. In *Proceedings of the 17th International Conference on Automated Planning and Scheduling (ICAPS)*, pages 264–271.

[Peltz, 2003] Peltz, C. (2003). Web services orchestration and choreography. *Computer*, 36(10):46–52.

[Petrick and Bacchus, 2002] Petrick, R. P. A. and Bacchus, F. (2002). A knowledge-based approach to planning with incomplete information and sensing. In *Proceedings of the 6th International Conference on Artificial Intelligence Planning and Scheduling (AIPS)*, pages 212–222.

[Petrick and Bacchus, 2004] Petrick, R. P. A. and Bacchus, F. (2004). Extending the knowledge-based approach to planning with incomplete information and sensing. In *Proceedings of the 14th International Conference on Automated Planning and Scheduling (ICAPS)*, pages 2–11.

[Pistore et al., 2004] Pistore, M., Barbon, F., Bertoli, P., Shaparau, D., and Traverso, P. (2004). Planning and monitoring Web service composition. In *Proceedings of the 11th International Conference on Artificial Intelligence: Methodology, Systems, Applications(AIMSA)*, pages 106–115.

[Pistore and Traverso, 2001] Pistore, M. and Traverso, P. (2001). Planning as model checking for extended goals in non-deterministic domains. In *Proceedings of the 17th International Joint Conference on Artificial Intelligence (IJCAI)*, pages 479–484.

[Pistore et al., 2005] Pistore, M., Traverso, P., and Bertoli, P. (2005). Automated composition of Web services by planning in asynchronous domains. In *Proceedings of the 15th International Conference on Automated Planning and Scheduling (ICAPS)*, pages 2–11.

[Ponnekanti and Fox, 2002] Ponnekanti, S. R. and Fox, A. (2002). Sword: A developer toolkit for Web service composition. In *Proceedings of the 11th International World Wide Web Conference (WWW)*.

[Rabideau et al., 2000] Rabideau, G., Engelhardt, B., and Chien, S. A. (2000). Using generic preferences to incrementally improve plan quality. In *Proceedings of the 5th International Conference on Artificial Intelligence Planning and Scheduling (AIPS)*, pages 236–245.

[Ranganathan et al., 2009] Ranganathan, A., Riabov, A., and Udrea, O. (2009). Mashup-based information retrieval for domain experts. In *Proceedings of the 18th ACM Conference on Information and Knowledge Management (CIKM)*, pages 711–720.

[Rao et al., 2004] Rao, J., Küngas, P., and Matskin, M. (2004). Logic-based Web services composition: From service description to process model. In *Proceedings of the IEEE International Conference on Web Services (ICWS)*, page 446. IEEE Computer Society.

[Reiter, 2001] Reiter, R. (2001). *Knowledge in Action: Logical Foundations for Specifying and Implementing Dynamical Systems*. MIT Press.

[Riabov and Liu, 2005] Riabov, A. and Liu, Z. (2005). Planning for stream processing systems. In *Proceedings of the 20th National Conference on Artificial Intelligence (AAAI)*, pages 1205–1210.

[Riabov and Liu, 2006] Riabov, A. and Liu, Z. (2006). Scalable planning for distributed stream processing systems. In *Proceedings of the 16th International Conference on Automated Planning and Scheduling (ICAPS)*, pages 31–41.

[Richter et al., 2008] Richter, S., Helmert, M., and Westphal, M. (2008). Landmarks revisited. In *Proceedings of the 23rd National Conference on Artificial Intelligence (AAAI)*, pages 975–982.

[RuleML, 2008] RuleML (2008). Rule markup language (RuleML). ruleml.org/.

[Schuster et al., 2000] Schuster, H., Georgakopoulos, D., Cichocki, A., and Baker, D. (2000). Modeling and composing service-based and reference process-based multi-enterprise processes. In *Proceedings of the 12th International Conference on Advanced Information Systems Engineering (CAiSE)*, pages 247–263.

[Shaparau et al., 2006] Shaparau, D., Pistore, M., and Traverso, P. (2006). Contingent planning with goal preferences. In *Proceedings of the 21st National Conference on Artificial Intelligence (AAAI)*.

[Sirin and Parsia, 2004] Sirin, E. and Parsia, B. (2004). Planning for semantic Web services. In *Semantic Web Services Workshop at the 3rd International Semantic Web Conference*.

[Sirin et al., 2005a] Sirin, E., Parsia, B., and Hendler, J. (2005a). Template-based composition of semantic Web services. In *AAAI-05 Fall Symposium on Agents and the Semantic Web*.

[Sirin et al., 2005b] Sirin, E., Parsia, B., Wu, D., Hendler, J., and Nau, D. (2005b). HTN planning for Web service composition using SHOP2. *Journal of Web Semantics*, 1(4):377–396.

[Sohrabi et al., 2010] Sohrabi, S., Baier, J., and McIlraith, S. (2010). Diagnosis as planning revisited. In *Proceedings of the 12th International Conference on the Principles of Knowledge Representation and Reasoning (KR)*, pages 26–36.

[Sohrabi et al., 2009] Sohrabi, S., Baier, J. A., and McIlraith, S. A. (2009). HTN planning with preferences. In *Proceedings of the 21st International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1790–1797.

[Sohrabi et al., 2011] Sohrabi, S., Baier, J. A., and McIlraith, S. A. (2011). Preferred explanations: Theory and generation via planning. In *Proceedings of the 25th National Conference on Artificial Intelligence (AAAI)*, pages 261–267. Accepted as both oral and poster presentation.

[Sohrabi and McIlraith, 2008] Sohrabi, S. and McIlraith, S. A. (2008). On planning with preferences in HTN. In *Proceedings of the 12th International Workshop on Non-Monotonic Reasoning (NMR)*, pages 241–248.

[Sohrabi and McIlraith, 2009] Sohrabi, S. and McIlraith, S. A. (2009). Optimizing Web service composition while enforcing regulations. In *Proceedings of the 8th International Semantic Web Conference (ISWC)*, pages 601–617.

[Sohrabi and McIlraith, 2010] Sohrabi, S. and McIlraith, S. A. (2010). Preference-based Web service composition: A middle ground between execution and search. In *Proceedings of the 9th International Semantic Web Conference (ISWC)*, pages 713, 729.

[Sohrabi et al., 2006] Sohrabi, S., Prokoshyna, N., and McIlraith, S. A. (2006). Web service composition via generic procedures and customizing user preferences. In *Proceedings of the 5th International Semantic Web Conference (ISWC)*, pages 597–611.

[Sohrabi et al., 2012] Sohrabi, S., Udrea, O., Ranganathan, A., and Riabov, A. (2012). Composition of flow-based applications with HTN planning. In *International Scheduling and Planning Applications woRKshop (SPARK)*, pages 1–7. This paper also appears in the AAAI-12 Workshop on Problem Solving using Classical Planners (CP4PS).

[Son and Pontelli, 2006] Son, T. C. and Pontelli, E. (2006). Planning with preferences using logic programming. *Theory and Practice of Logic Programming*, 6(5):559–607.

[Srivastava and Koehler, 2003] Srivastava, B. and Koehler, J. (2003). Web service composition - current solutions and open problems. In *ICAPS 2003 Workshop on Planning for Web Services*, pages 28–35.

[Tate, 1977] Tate, A. (1977). Generating project networks. In *Proceedings of the 5th International Joint Conference on Artificial Intelligence (IJCAI)*, pages 888–893.

[Thakkar et al., 2005] Thakkar, S., Ambite, J. L., and Knoblock, C. A. (2005). Composing, optimizing, and executing plans for bioinformatics Web services. *VLDB Journal*, 14(3):330–353.

[Tonti et al., 2003] Tonti, G., Bradshaw, J. M., Jeffers, R., Montanari, R., Suri, N., and Uszok, A. (2003). Semantic Web languages for policy representation and reasoning: A comparison of KAoS, Rei, and Ponder. In *Proceedings of the 2rd International Semantic Web Conference (ISWC)*, pages 419–437.

[Traverso and Pistore, 2004] Traverso, P. and Pistore, M. (2004). Automatic composition of semantic Web services into executable processes. In *Proceedings of the 3rd International Semantic Web Conference (ISWC)*.

[Valero et al., 2009] Valero, V., Cambronero, M. E., Díaz, G., and Macià, H. (2009). A Petri net approach for the design and analysis of web services choreographies. *Journal of Logic and Algebraic Programming*, 78(5):359 – 380.

[Waldinger, 2001] Waldinger, R. J. (2001). Web agents cooperating deductively. In *Proceedings of the 1st International Workshop on Formal Approaches to Agent-Based Systems-Revised Papers (FAABS)*, pages 250–262.

[Wilkins, 1988] Wilkins, D. E. (1988). *Practical planning: extending the classical AI planning paradigm*. Morgan Kaufmann, San Francisco, CA.

[WS-Policy, 2006] WS-Policy (2006). Web service policy framework (WS-policy). www.w3.org/Submission/WS-Policy/.

[Zeng et al., 2003] Zeng, L., Benatallah, B., Dumas, M., Kalagnanam, J., and Sheng, Q. Z. (2003). Quality driven web services composition. In *Proceedings of the 12th International World Wide Web Conference (WWW)*, pages 411–421.

[Zhovtobryukh, 2007] Zhovtobryukh, D. (2007). A Petri net-based approach for automated goal-driven web service composition. *Simulation*, 83(1):33–63.