Problem Solving Using Classical Planners
AAAI Technical Report WS-12-12

Composition of Flow-Based Applications with HTN Planning*

Shirin Sohrabi†
University of Toronto
Toronto, Ontario, Canada

Octavian Udrea, Anand Ranganathan, Anton V. Riabov
IBM T.J. Watson Research Center
Hawthorne, NY, U.S.A.

Abstract

Goal-driven automated composition of software components is an important problem with applications in Web service composition and stream processing systems. The popular approach to address this problem is to build the composition automatically using Artificial Intelligence planning. However, it is shown that some of these popular planning approaches may neither be feasible nor scalable for many real large-scale flow-based applications. Recent advances 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 pattern. In prior work, a flow pattern language called Cascade and its corresponding specialized planner have shown the best performance in these domains. In this paper, we propose to address this problem using Hierarchical Task Network (HTN) planning. To this end, we propose an automated approach of creating an HTN-based problem from the Cascade representation of the flow patterns. The resulting technique not only allows us to use the HTN planning paradigm and its many advantages including added expressivity but also enables optimization and customization of composition with respect to preferences and constraints. Further, we propose and develop a lookahead heuristic and show that it significantly reduces the planning time. We have performed extensive experimentation in the context of the stream processing application and evaluated applicability and performance of our approach.

Introduction

One of the approaches to automated software composition focuses on composition of information flows from reusable software components. This flow-based model of composition is applicable in a number of application areas, including Web service composition and stream processing. There are a number of tools (e.g., Yahoo Pipes and IBM Mashup Center) that support the modeling of the data flow across multiple components. Although these visual tools are fairly popular, the use of these tools 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 Artificial Intelligence (AI) planning is a popular approach to automate the composition of components, Riabov and Liu have shown that Planning Domain Definition Language (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 behind this 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 limitation to the scalability of planning.

In 2009, we proposed a pattern-based composition approach where composition patterns were specified using our proposed language called Cascade and the plans were computed using our specialized planner, MARIO (Ranganathan, Riabov, and Udrea 2009). We made use of the observation that automated composition problem can take advantage of expert knowledge of how different components can be coupled together and this knowledge can be expressed using a composition pattern. 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 paper, we propose a planning approach based on Hierarchical Task Networks (HTNs) to address the problem of automated composition of components. 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 added expressivity, and address optimization and customization of composition with respect to preferences and constraints. We use the preference-based HTN planner **HTNPLAN-P** (Sohrabi, Baier, and McIlraith 2009) for implementation and evaluation of our approach. Moreover, we develop a new lookahead heuristic by drawing inspirations from ideas proposed in (Marthi, Russell, and Wolfe 2007). We also propose an algorithm to derive indexes required by our proposed heuristic.

*This paper also appears in the 6th International Scheduling and Planning Applications woRKshop (SPARK), 2012.

†This work was done at IBM T.J.Watson Research Center.

The contributions of this paper are as follows: (1) we exploit HTN planning with preferences to address modeling, computing, and optimizing the composition of information flows in software components; (2) we develop a method to automatically translate Cascade patterns into HTN domain description and Cascade goals into preferences, and to that end we address several unique challenges that hinder planner performance in flow-based applications; (3) we perform extensive experiments with real-world patterns using IBM InfoSphere Streams applications; and (4) we develop an enhanced lookahead heuristic that improves HTN planning performance by 65% on average in those applications.

Preliminaries

Specifying Patterns in Cascade

The Cascade language has been proposed in (Ranganathan, Riabov, and Udrea 2009) for specifying flow patterns. A Cascade flow pattern describes a set of flows by describing 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.

Figure 1 shows an example of a flow pattern, defining a composite 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 VWAP and the Bargain Index 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 a 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, (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 the paper, 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 1, then any of the 450 flows that it defines can potentially be deployed on the underlying platform.

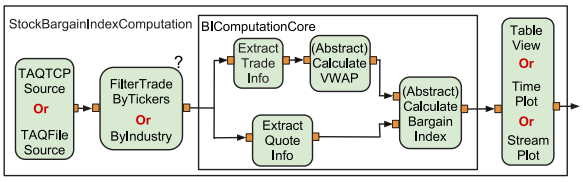

Figure 1: Example of a Cascade flow pattern.

In Cascade, output ports of components (output streams) can be annotated with 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 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 $2 \times 5 \times 5 \times 3$, or 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.

Hierarchical Task Network (HTN) Planning

HTN planning is a widely used planning paradigm and many domain-independent HTN planners exist (Ghallab, Nau, and Traverso 2004). The HTN planner is given the HTN planning problem: the initial state s_0, the initial task network w_0, and the planning domain D (a set of operators and methods). HTN planning is performed by repeatedly decomposing tasks, by the application of methods, into smaller and smaller subtasks until a primitive decomposition of the initial task network is found. A task network is a pair (U, C) where U is a set of tasks and C is a set of constraints. A task is *primitive* if its name matches with an operator, otherwise it is *nonprimitive*. An operator is a regular planning action. It can be applied to accomplish a primitive task. A method is described by its name, the task it can be applied to $task(m)$, and its task network $subtasks(m)$. A method m can accomplish a task t if there is a substitution σ such that $\sigma(t) = task(m)$. Several methods can accomplish a particular nonprimitive task, leading to different decompositions of it. Refer to (Ghallab et al. 2004) for more information.

HTNPLAN-P (Sohrabi et al. 2009) is a provably optimal preference-based planner, built on top of a Lisp implementation of **SHOP2** (Nau et al. 2003), a highly-optimized HTN planner. **HTNPLAN-P** takes as input an HTN planning problem, specified in the **SHOP2**'s specification language (not in PDDL). **HTNPLAN-P** performs incremental search and uses variety of different heuristics including the Lookahead Heuristic (LA). We modified **HTNPLAN-P** to implement our proposed heuristic, the Enhanced Lookahead

Heuristic (*ELA*). We also use **HTNPLAN-P** to evaluate our approach.

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 and goals. In particular, we show how to: (1) create an HTN planning domain from the definition of Cascade components (2) represent the Cascade goals as preferences. We refer to the **SHOP2**'s specification language (also **HTNPLAN-P**'s input language) in Lisp. 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.

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 stream processing 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 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; 2006) show poor performance of planners (on an attempt to formulate the problem in PDDL). We believe 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. This can result in a large number of symmetric objects, significantly slowing down the planner.

To address the creation of new uninitialized streams we propose to use the *assignment expression*, available in **SHOP2**'s input language, in the precondition of the operator that creates the new stream (will discuss how to model Cascade components 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 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 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 (must be grounded before any evaluation of state with respect to this predicate), and *Stream* is the variable representing a stream. 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 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 *diff* to ensure the negated tag "AllCompanies" does not propagate downstream.

```
(forall (?tag)(and (tags ?tag) (?tag ?QuoteInfo)
                   (diff ?tag AllCompanies))
             ((?tag ?filteredTradeQuote)))
```

Tag Hierarchy Tags used in Cascade belong to tag hierarchy (or tag taxonomies). This notion is useful in inferring additional tags. In the example in Figure 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*). *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.

Components can inherit from other 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 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 use of high-level components. Structural variations create patterns that capture multiple flows. Enumerations are specified by listing the different possible components. To capture this we use multiple methods applicable to the same task. A component can be specified as optional, meaning that it 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. In HTN, this is already captured by the use of nonprimitive tasks for abstract components and methods for each concrete component.

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 Planning Domain Definition Language (PDDL3) (Gerevini et al. 2009) simple preferences. **Simple preferences** are atemporal formulae that express a preference for certain conditions to hold in the final state of the plan. In PDDL3 the quality of the plan is defined using a metric function. The PDDL3 function `is-violated` is used to assign appropriate weights to different preference formula. Note, inconsistent preferences are automatically handled by the metric function.

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, the 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 some example. If the Cascade goals 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)))
```

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. 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 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 with F shown in Figure 1 and G be 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.

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 given a specification of preference-based HTN planning in hand we select **HTNPLAN-P** 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, and describe how the required indexes for this heuristic can be generated in the preprocessing step.

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 *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. 2009) if used within the **HTNPLAN-P** search algorithm and provably optimal plans can get generated. 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 2 *The ELA function provides sound pruning if the preferences are all PDDL3 simple preferences and the metric function is non-decreasing in the number of violated preferences and in plan length.*

Our notion of reachable tags is similar to the notion of "complete reachability set" in 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 compute a heuristic, which we implement in **HTNPLAN-P**. They use this notion for pruning plans and not necessarily in guiding the search towards a preferred plan.

Generation from HTN

In this section, we briefly discuss how to generate the reachable tags from the corresponding HTN planning problem. Algorithm 1 shows pseudocode of our offline procedure that creates a set of reachable tags for each task. It takes as input

Algorithm 1: The GetRTags (D, w, C) algorithm.

```
1  initialize global Map R;  T ← ∅;
2  if w is a task network then
3      if w = ∅ then return C;
4      else if w = (:ordered t_1 ... t_n) then
5          for i=n to 1 do C ← GetRTags(D, t_i, C);
6      else if w = (:unordered t_1 ... t_n) then
7          for i=1 to n do
8              T_{t_i} ← GetRTags(D, t_i, ∅); T ← T_{t_i} ∪ T;
9          for i=1 to n do
10             C_{t_i} ← ⋃_{j=1,j≠i}^{n} T_j ∪ C;
               GetRTags(D, t_i, C_{t_i});
11 else if w is a task then
12     if R[w] is not defined then R[w] ← ∅;
13     else if t is primitive then T ← add-list of an operator that
       matches;
14     else if t is nonprimitive then
15         M' ← {m_1, ..., m_k} such that task(m_i) match with
           t;
16         U' ← {U_1, ..., U_k} such that U_i = subtask(m_i);
17         foreach U_i ∈ U' do T ← GetRTags(D, U_i, C) ∪ T;
18     R[w] ← R[w] ∪ T ∪ C;
19 return T ∪ C
```

the planning domain D, a set of tasks (or a single task) w, and a set of tags to carry over C. The algorithm is called initially with the initial task network w_0, and $C = \emptyset$. To track the produced tags for each task we use a map R. If w is a task network then we consider three cases: 1) task network is empty, we then return C, 2) w is an ordered task network, then for each task t_i we call the algorithm starting with the right most task t_n updating the carry C, 3) w is unordered, then we call GetRTags twice, first to find out what each task produces (line 8), and then again with the updated set of carry tags (line 10). This ensures that we overestimate the reachable tags regardless of the execution order.

If w is a task then we update its returned value $R[w]$. If w is primitive, we find a set of tags it produces by looking at its add-list. If w is nonprimitive then we first find all the methods that can be applied to decompose it and their associated task networks. We then take a union of all tags produced by a call to GetRTags for each of these task networks.

Our algorithm can be updated to deal with recursive tasks by first identifying when loops occur and then by modifying the algorithm to return special tags in place of a recursive task's returned value. We then use a fixed-point algorithm to remove these special tags and update the values for all tasks.

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 use of our 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 **HTNPLAN-P**, on the created problems. To address our second objective, we implemented the preprocessing algorithm discussed earlier and modified **HTNPLAN-P** to incorporate the enhanced lookahead heuristic within its search strategy and then examined its performance. A search strategy is a prioritized sequence of heuristics that determines if a node is better than another.

We had 7 domains and more than 50 HTN planning problems in our experiments. The created HTN problems come 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 2. 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 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 2 indicates the applicability and feasibility of our approach as we increase the difficulty of the problem. All problems were solved within 35 seconds by at least one of the two modes used. The result also indicates that not surprisingly, the *LA* heuristic performs better at least in the harder cases (indicated in bold). This is 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 (3rd domain Problems 3 and 4), an optimal plan was only found when the *LA* heuristic was used.

We had two sub-objectives in evaluating our proposed heuristic, the Enhanced Lookahead Heuristic (*ELA*): (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. To address our objectives, we identified cases where **HTNPLAN-P** has difficulty finding the optimal solution. In particular we chose the third and fourth domain and tested with goal tags that appear deep in the right branch of the HTN search tree. These problems are difficult because achieving the goal tags are harder and the *LA* heuristic fails in providing sufficient guidance.

Figure 3 shows a subset of our results. *LA* then *ELA* (resp. *ELA* then *LA*) column indicates that we use a strategy in

Dom	Prob	Plan Length	# of Plans	*No-LA* Time (s)	*LA* Time (s)
1	1	11	81	0.04	0.05
	2	11	162	0.10	**0.01**
	3	11	81	0.18	**0.04**
2	1	11	162	0.04	0.05
	2	11	162	0.13	**0.01**
	3	11	81	0.25	**0.04**
3	1	38	2^{26}	0.08	0.08
	2	38	2^{13}	276.11	**0.09**
	3	20	2^{13}	OM	**0.14**
	4	38	2^{26}	OM	**0.14**
4	1	44	4608^{2}	0.09	0.11
	2	92	4608^{4}	0.64	**0.61**
	3	184	4608^{8}	4.80	**4.50**
	4	368	4608^{16}	43.00	**35.00**

Figure 2: Evaluating the applicability of our approach by running **HTNPLAN-P** (two modes) as we increase problem hardness.

Dom	Prob	*LA* then *ELA* Time (s)	*ELA* then *LA* Time (s)	Just *ELA* Time (s)	Just *LA* Time (s)	*No-LA* Time (s)
3	5	1.70	1.70	**0.07**	0.13	OM
	6	1.70	1.70	**0.07**	1.50	OM
	7	1.80	1.80	**0.07**	1.60	OM
	8	1.70	1.70	**0.07**	OM	OM
	9	1.40	1.40	**0.07**	OM	OM
	10	1.40	1.30	**0.07**	OM	OM
4	5	0.58	0.45	**0.02**	0.56	0.12
	6	2.28	2.24	**0.07**	3.01	0.38
	7	14.40	14.28	**0.44**	19.71	1.44
	8	104.70	102.83	**3.15**	147.00	8.00
	9	349.80	341.20	**10.61**	486.53	18.95
	10	OT	OT	**24.45**	OT	40.20

Figure 3: Evaluation of the *ELA* heuristic.

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. Our results show that the ordering of the heuristics does not seem to make any significant change in the time it takes to find an optimal plan. The results also show that using the *ELA* heuristic improves the search time compared to other search strategies. In particular, there are cases in which the planner fails to find the optimal plan when using *LA* or *No-LA* but the 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 *ELA* for the problems we used. This shows that our enhanced lookahead heuristic seems to significantly improve the performance.

Summary and Related Work

There is a large body of work that explores the use of AI planning for the task of automated Web service composition

(e.g., (Pistore et al. 2005)). Additionally some explore the use of some form of expert knowledge (e.g., (McIlraith and Son 2002)). While similarly, many explore the use of HTN planning, they rely on the translation of OWL-S (Martin et al. 2007) service descriptions of services to HTN planning (e.g., (Sirin et al. 2005)). Hence, the HTN planning problems driven from OWL-S generally ignore the data flow aspect of services, a major focus of Cascade flow patterns.

In this paper, we examined the correspondence between HTN planning and automated composition of flow-based applications. We proposed use of HTN planning and to that end proposed a technique for creating an HTN planning problem with user preferences from Cascade 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 developed a lookahead heuristic and showed that it improves the performance of **HTNPLAN-P** for the domains we used. The proposed heuristic is general enough to be used within other HTN planners. We have performed extensive experimentation that showed applicability and promise of the proposed approach.

References

Baier, J. A.; Bacchus, F.; and McIlraith, S. A. 2009. A heuristic search approach to planning with temporally extended preferences. *Artificial Intelligence* 173(5-6):593–618.

Gedik, B.; Andrade, H.; lung Wu, K.; Yu, P. S.; and Doo, M. 2008. SPADE: the System S declarative stream processing engine. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 1123–1134.

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.

Ghallab, M.; Nau, D.; and Traverso, P. 2004. *Hierarchical Task Network Planning. Automated Planning: Theory and Practice*. Morgan Kaufmann.

IBM. IBM InfoSphere Streams. http://www.ibm.com/software/data/infosphere/streams/. [online; accessed 14-05-2012].

IBM. IBM Mashup Center. http://www-01.ibm.com/software/info/mashup-center/. [online; accessed 14-05-2012].

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)*, 232–239.

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.

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)*, 482–493.

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.

Pistore, M.; Marconi, A.; Bertoli, P.; and Traverso, P. 2005. Automated composition of Web services by planning at the knowledge level. In *Proceedings of the 19th International Joint Conference on Artificial Intelligence (IJCAI)*, 1252–1259.

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)*, 711–720.

Riabov, A., and Liu, Z. 2005. Planning for stream processing systems. In *Proceedings of the 20th National Conference on Artificial Intelligence (AAAI)*, 1205–1210.

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)*, 31–41.

Sirin, E.; Parsia, B.; Wu, D.; Hendler, J.; and Nau, D. 2005. HTN planning for Web service composition using SHOP2. *Journal of Web Semantics* 1(4):377–396.

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)*, 1790–1797.

Yahoo. Yahoo pipes. http://pipes.yahoo.com. [online; accessed 14-05-2012].