Proceedings of the Twenty-Third International Conference on Automated Planning and Scheduling

HTN Planning for the Composition of Stream Processing Applications

Shirin Sohrabi, Octavian Udrea, Anand Ranganathan, and Anton V. Riabov
IBM T.J. Watson Research Center
PO Box 704, Yorktown Heights, NY 10598, USA

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 AI planning. However, it is shown that some of these planning approaches may neither be feasible nor scalable for many large-scale flow-based applications. Recent advances have proven that the automated composition problem can take advantage of expert knowledge describing the many 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 the use of Hierarchical Task Network (HTN) planning for the composition of stream processing applications. 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 with stream processing applications and evaluated applicability and performance of our approach.

Introduction

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 can be visualized in different ways. This flow-based model of composition is applicable in a number of application areas, including Web Service Composition (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. A stream processing application describes the data flows between the stream processing components.

There are several tools such as IBM Mashup Center, www-01.ibm.com/software/info/mashup-center, and Yahoo Pipes, pipes.yahoo.com, 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 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 large-scale stream processing or flow-based systems (Riabov and Liu 2005; 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, Ranganathan et al. proposed a pattern-based composition approach where composition patterns were specified using their proposed language called Cascade and the plans were computed using their specialized planner, MARIO. They 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, 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 in the context of the stream processing application. 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 domain-independent planning and HTN planning

including added expressivity and modularity, 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 inspiration from ideas proposed by Marthi et al. in (Marthi, Russell, and Wolfe 2007). We also propose an algorithm to derive indexes required by our proposed heuristic. While the focus of this paper is on stream processing applications, our techniques are general enough that they can be used to address the composition of any flow-based application.

The following are the main contributions of this paper: (1) proposed the use of HTN planning with preferences to address modeling, computing, and optimizing composition flows in the stream processing applications, (2) 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, (3) developed an enhanced lookahead heuristic and showed that it improves the HTN planning performance, and (4) performed extensive experimentation with real-world patterns using IBM InfoSphere Streams, www-01.ibm.com/software/data/infosphere/streams.

Preliminaries

As shown by Ranganathan and others (Ranganathan, Riabov, and Udrea 2009), Cascade composition patterns can be translated into a planning domain description in Stream Processing Planning Language, i.e., SPPL (Riabov and Liu 2005). For this, a Cascade to SPPL compiler was implemented in MARIO. The main reason for choosing SPPL over PDDL or other general-purpose planners in MARIO was performance: using several mappings from SPPL to PDDL, Riabov and Liu have shown that the specialized SPPL planner is exponentially faster than PDDL planners.

To counter these prior negative results for the performance of general planners in composition problems, in the next section we will introduce a new mapping from Cascade to HTN domains, which, together with a new search heuristic, allows a general HTN planner to perform comparably with an SPPL planner on Cascade problems.

In the rest of this section, we include a brief overview for readers not previously familiar with HTN or Cascade.

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) (e.g., **SHOP2** (Nau et al. 2003)). 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. More formally, $\pi = o_1 o_2 \dots o_k$ is a plan for HTN planning problem $\mathcal{P} = (s_0, w_0, D)$ if there is a primitive decomposition, w, of w_0 of which π is an instance. A task network

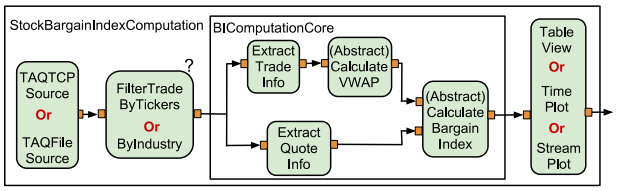

Figure 1: Example of a Cascade flow pattern.

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, described by its name, precondition and effects. 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.

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.

Cascade – Flow Patterns for Stream Processing

The Cascade language has been proposed by Ranganathan and others in 2009 (Ranganathan, Riabov, and Udrea 2009) for describing data flow patterns that can guide automated software composition, and as an alternative to the lower-level planning domain languages like PDDL that are difficult to use as part of software development cycle. Cascade has a programming language syntax that is friendly to software developers, includes integrated development tools, and can be used with different execution environments.

An important example of an execution environment is stream processing middleware (for example, IBM InfoSphere Streams), which facilitates the development of distributed applications that must process high volumes of data in memory. Stream processing applications are constructed as data flow graphs composed of modular software components that communicate via data streams, and described in a programming language, e.g., Streams Processing Language (SPL) (Hirzel et al. 2009). The middleware deploys the components of the application across multiple hardware nodes within a dedicated stream processing cluster, manages them and provides efficient data transport. Cascade flow patterns define the space of valid composed flows, which are then mapped to stream processing data flow graphs in SPL.

Figure 1 graphically represents a Cascade pattern for 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 component is called *StockBargainIndexComputation* and is used to define the root or the top-level component. The flow pattern describes how the data is obtained from sources, processed by components, and visualized in different ways. Source data, Trade and Quote (TAQ), can come either live, or replayed from a file. This data can be filtered according to a set of tickers or according to a specified industry or neither, as the filter components are 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*. The results can be visualized using a table, a time-plot or a stream-plot.

A single flow pattern defines a number of actual flows. That is, 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. 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, (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 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.

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** is an atomic element of the flow graph, and is usually associated with a code fragment, which is used in code generation during flow graph deployment. A **composite component** internally defines a flow of other components. In Figure 1, the *TableView* and *BICompoutationCore* are examples of primitive and composite components respectively. Similarly, an **abstract components** includes the declaration of inputs and outputs, but without code fragment or graph. Instead, separately defined **concrete components** can be declared to implement an abstract component. Note, a concrete component can be primitive or composite. 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** invocation can be used to enumerate several alternatives to be used within the same location in the graph. For example, the pattern in Figure 1 defines a choice between TCP source and file source. The alternatives must have the same number of inputs and the same number of outputs. Any component contained within the **optional** invocation becomes optional. For example, in Figure 1 the choice between trade filters "ByTickers" and "ByIndustry" is made optional, allowing graphs that include no filters at all to be valid instantiations of this pattern.

Cascade		HTN
Flow	→	Plan
Cascade flow pattern	→	HTN planning
Stream	→	Variable
Tag	→	Predicate
Tag hierarchy	→	Axioms
Primitive component	→	Operator
Composite component	→	Method
Choice invocation	→	Task with multiple methods
Optional invocation	→	Task with multiple methods
Abstract component	→	Task
Main composite	→	Initial task network
Cascade goal	→	PDDL simple preferences
Satisfying flow	→	Optimal plan

Figure 2: From Cascade to HTN

In Cascade, output ports of components 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 *ByIndustry*, 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.

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 with a set of 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. Figure 2 shows at high-level how we encode the main elements in Cascade as HTN planning elements. For example, we encode a primitive component as an operator and a composite component as an HTN method. Next, we describe the steps of this transformation while using the example shown in Figure 1 as our running example.

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.

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 composition of the stream processing applications 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; 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 use the *assignment expression*, available in the **SHOP2**'s input language, in the precondition of the operator that creates the new stream. 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. The *sNum* predicate acts as a *counter* to keep track of the *current* value that can be assigned 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)))
Delete 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* (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. 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 *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 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 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 this task, this indicates that there are no concrete components written for this abstract component.

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 invocations, 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 the choice invocation, 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 invocations using methods that simulate the "no-op" task. Abstract components are used in flow patterns to capture high-level components. The abstract 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. For example, the task network of *BIComputationCore* includes the nonprimitive task *CalculateBargainIndex* and different methods written for this task handle the concrete components.

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) preferences. 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.

One reason for why we encode the Cascade goals as preferences is to be able to still find a plan even if the Cascade goals are not achievable or are mutually inconsistent. However, the main reason for why we encode the Cascade goals as preferences is to take advantage of the already existing preference-based heuristics employed by **HTNPLAN-P** as well as our proposed enhanced lookahead heuristic. This allows efficient search pruning and guidance towards the satisfaction of the preferences

The following are example preferences that encode Cascade goals *ByTickers* and *TableView*. These PDDL3 simple preferences are over the predicate "(*Tag Stream*)". Note that we need to define a **metric function** for the generated preferences. Recall, 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 and we assume that the metric is always being minimized. 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 (ByTickers ?finalStream)))
(preference g2 (at end (TableView ?finalStream)))
```

Flow-Based HTN Planning Problem with Preferences

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 composite M; D is the HTN planning domain, consisting of a set of operators and methods derived from the components $v \in \mathcal{V}$; and $\preceq$ is a reflexive and transitive relation defined through PDDL3 metric function over the set of Cascade goals G; $\alpha \preceq \alpha'$ states that α is *at least as preferred as* α'. A plan α is a solution to (i.e., an optimal plan for) P if and only if: α is a plan and there does not exists another plan α' such that it is more preferred (i.e., $\alpha' \preceq \alpha$ and $\alpha \not\preceq \alpha'$).

Proposition 1 *Let $P^F = (F, G)$ be a Cascade flow pattern problem. Let $P = (s_0, w_0, D, \preceq)$ be the corresponding HTN planning problem with preferences. If G is realizable, then there exists an optimal plan for P. Furthermore, if G is realizable and α is an optimal plan for P, then we can construct a flow (based on α) that is a satisfying flow for P^F.*

Consider the Cascade flow pattern problem $P^F = (F, G)$ with F shown in Figure 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 also 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 with a specification of preference-based HTN planning in hand we select **HTNPLAN-P** to compute these optimal plans that later translate 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 given the set of known tags. In the next section, we will evaluate the performance of **HTNPLAN-P** with the proposed heuristic incorporated within its search strategy. As we will discuss, our proposed heuristic helps improve the HTN planning performance, especially in the harder problem sets; a problem can be harder if the goal tags appear in the harder to reach branches of the search space. In addition, the proposed heuristic improves the HTN performance making it even comparable with an SPPL planner on Cascade problems. On the other hand, the notion behind our proposed heuristic and how we generated the required indexes is general enough to be used within other HTN planners.

Enhanced Lookahead Heuristic (*ELA*)

The enhanced lookahead function estimates the PDDL3 metric value achievable from a search node N based on the reachable tags from this node. 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 at least one 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. 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. 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, when computing the *ELA* function, 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 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 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 (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. Thus, the *ELA* function provides sound pruning. ■

Our notion of reachable tags is similar to the notion of "complete reachability set" from Marthi et al. in (Marthi, Russell, and Wolfe 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, Russell, and Wolfe 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, which is non-trivial.

Generation of the Heuristic Indexes

In this section, we briefly discuss how to generate the reachable tags from the corresponding HTN planning problem. We can also generate the set of reachable tags from the description of the Cascade flow patterns; however, that description is omitted from this paper.

Algorithm 1 shows pseudocode of our offline procedure that creates a set of reachable tags for each task. It takes as input the planning domain D, a set of tasks (or a single task) w, and a set of tags to carry over C. The algorithm should be called initially with the initial task network w_0, and $C = \emptyset$. 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 call to GetRTags 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. 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

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

initialize global Map R; $T \leftarrow \emptyset$;
if w *is a task network* **then**
  if $w = \emptyset$ **then return** C;
  else if $w = (:ordered\ t_1 \ldots t_n)$ **then**
    for *i=n* **to** *1* **do** $C \leftarrow \mathsf{GetRTags}(D, t_i, C)$;
  else if $w = (:unordered\ t_1 \ldots t_n)$ **then**
    for *i=1* **to** *n* **do**
      $T_{t_i} \leftarrow \mathsf{GetRTags}(D, t_i, \emptyset)$; $T \leftarrow T_{t_i} \cup T$;
    for *i=1* **to** *n* **do**
      $C_{t_i} \leftarrow \bigcup_{j=1, j \neq i}^{n} T_j \cup C$; $\mathsf{GetRTags}(D, t_i, C_{t_i})$;
else if w *is a task* **then**
  if $R[w]$ *is not defined* **then** $R[w] \leftarrow \emptyset$;
  else if t *is primitive* **then**
    $T \leftarrow$ add-list of an operator that matches;
  else if t *is nonprimitive* **then**
    $M' \leftarrow \{m_1, ..., m_k\}$ such that $task(m_i)$ match with t;
    $U' \leftarrow \{U_1, ..., U_k\}$ such that $U_i = subtask(m_i)$;
    foreach $U_i \in U'$ **do** $T \leftarrow \mathsf{GetRTags}(D, U_i, C) \cup T$;
  $R[w] \leftarrow R[w] \cup T \cup C$;
return $T \cup C$

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. Note that this 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 can 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 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. These patterns vary from having 30 to 170 components and 10 to 200 tags. 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 new heuristic within its search strategy and then examined its performance.

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 or choice invocations, 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".

Domain	Problem	Plan Length	# of Plans	*No-LA* Time (sec)	*LA* Time (sec)
1	1	11	162	0.01	0.07
	2	11	81	0.04	0.05
	3	11	162	**0.10**	**0.01**
	4	11	81	**0.18**	**0.04**
2	1	11	81	0.04	0.04
	2	11	162	0.04	0.05
	3	11	162	**0.13**	**0.01**
	4	11	81	**0.25**	**0.04**
3	1	38	2^{13}	**276.11**	**0.09**
	2	38	2^{26}	**OM**	**0.13**
	3	38	2^{26}	**OM**	**0.14**
	4	20	2^{13}	**OM**	**0.14**
4	1	22	4608	0.01	0.01
	2	44	4608^2	0.09	0.11
	3	92	4608^4	**0.64**	**0.61**
	4	184	4608^8	**4.80**	**4.50**
	5	276	4608^{12}	**16.00**	**15.00**
	6	368	4608^{16}	**43.00**	**35.00**

Figure 3: Evaluating the applicability of our approach by running **HTNPLAN-P** (two modes) as we increase problem hardness.

We show a subset of our results in Figure 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 short, the *LA* heuristic is computed by first solving the current node up to a fixed depth, and then computing a single plan for each of the resulting nodes using depth-first search and returning the best metric value among the decomposed nodes. We measure plan length for each solved problem as a way to show the number of generated output streams since each plan consists of a set of operators and operators generate new 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 3 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 2-5), 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 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 4.

The results show that there are some hard problems for the *LA* heuristic. See Problem 8-10 for Domain 3 and Problems 11 and 12 for Domain 4. These problems are difficult because achieving the goal tags are difficult for the planner. There are a number of reasons, some of which are planner specific, for why these problems are hard. For example, in Problem 8, there are many optional invocations, and the planner chooses the "no-op" task whereas the goal tags are not achievable if the "no-op" task is selected. Therefore, since the search space is large, the exploration of the path that leads to achieving the goal tags gets delayed. Some of these problems are easier because the goal is to achieve easier to reach tags. It could also be the case that the *LA* heuristic's sampling technique evaluates the correct branch of the search space; hence, it can provide the right level of guidance to the planner. However, from the result shown in Figure 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 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 4. This shows that the *ELA* heuristic seems to significantly improve the time it takes to find an optimal plan.

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	7	0.58	0.45	**0.02**	0.56	0.12
	8	2.28	2.24	**0.07**	3.01	0.38
	9	14.40	14.28	**0.44**	19.71	1.44
	10	104.70	102.83	**3.15**	147.00	8.00
	11	349.80	341.20	**10.61**	486.53	18.95
	12	OT	OT	**24.45**	OT	40.20

Figure 4: Evaluation of the *ELA* heuristic.

Summary and Related Work

There is a large body of work that explores the use of AI planning for the task of automated WSC (e.g., (Traverso and Pistore 2004)). 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, Riabov, and Udrea 2009; Feblowitz et al. 2012)). Additionally some explore the use of some form of expert knowledge or composition template (e.g., (McIlraith and Son 2002)) that help guide the composition. 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)). There are several key differences between OWL-S and Cascade flow patterns. In particular, Cascade is specially designed to address the data flows and propagations of properties throughout the pattern, while OWL-S, aims to support Web service discovery and composition, not focused on modeling the data flow interactions of services. Representing the expert knowledge in Cascade can be used in other applications such as WSC, especially if data flow in the composition is of interest, in addition to control flow.

In this paper, we examined the correspondence between HTN planning and automated composition of stream processing applications. We proposed the use of HTN planning and to that end proposed a technique for creating an HTN planning problem with 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 developed a lookahead heuristic and an algorithm to derive indexes required by the proposed heuristic. It has been shown that 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), Fast Downward (Helmert 2006), LAMA (Richter, Helmert, and Westphal 2008). Our challenge here was to develop a suitable heuristic for HTN planning that gives guidance towards optimal solutions without exhaustively searching the search space.

Our experimental evaluation showed the applicability and promise of the proposed approach for the problem of automated composition of stream processing applications. In

particular, we showed that our proposed heuristic improves the performance of **HTNPLAN-P** for the domains we used. In addition, its performance with the proposed heuristic is comparable with an SPPL planner on Cascade problems; similar size plans were created in similar plan time based on looking at the results table from (Riabov and Liu 2006). Note, the proposed heuristic is general enough to be used within other HTN planners. As part of the future work we would like to evaluate the performance of our proposed heuristic for the general HTN planning problems. While the focus of this paper is on stream processing applications, our techniques are general enough that they can be used to address the composition of any flow-based application.

Acknowledgements

The authors thank Jorge Baier, Sheila McIlraith, and Nagui Halim for their valuable feedback.

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.

Bonet, B., and Geffner, H. 2001. Planning as heuristic search. *Artificial Intelligence* 129(1-2):5–33.

Feblowitz, M. D.; Ranganathan, A.; Riabov, A. V.; and Udrea, O. 2012. Planning-based composition of stream processing applications. In *ICAPS-12 System Demonstrations and Exhibits Track*.

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.

Helmert, M. 2006. The Fast Downward planning system. *Journal of Artificial Intelligence Research* 26:191–246.

Hirzel, M.; Andrade, H.; Gedik, B.; Kumar, V.; Losa, G.; M. Mendell, H. N.; Soule, R.; ; and Wu., K.-L. 2009. SPL stream processing language specification. Technical Report RC24897, IBM Research.

Hoffmann, J., and Nebel, B. 2001. The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research* 14:253–302.

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.

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)*, 222–231.

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.

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.

Richter, S.; Helmert, M.; and Westphal, M. 2008. Landmarks revisited. In *Proceedings of the 23rd National Conference on Artificial Intelligence (AAAI)*, 975–982.

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.

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)*.