Planning for Stream Processing Systems

Anton Riabov and **Zhen Liu**
IBM T.J. Watson Research Center
P.O. Box 704, Yorktown Heights, New York, 10598
riabov@us.ibm.com, zhenl@us.ibm.com

Abstract

With the advent of compositional programming models in computer science, applying planning technologies to automatically build workflows for solving large and complex problems in such a paradigm becomes not only technically appealing but also feasible approach. The application areas that will benefit from automatic composition include, among others, Web services, Grid computing and stream processing systems. Although the classical planning formalism is expressive enough to describe planning problems that arise in a large variety of different applications, it can pose significant limitations on planner performance in compositional applications, in particular, in stream processing systems. In this paper we extend the classical planning formalism by introducing new language constructs that support the structure of stream processing domains. Exposing this structure to the planner can result in dramatic performance improvements: our experiments show exponential planning time reduction in comparison to most recent metric planners.

Introduction

Today, powerful computers and high-bandwidth communication are becoming increasingly affordable, stimulating the growth of high-performance distributed computing. At the same time, recent software development tools and technologies, including open communication protocols and compositional programming models, are paving the way for the arrival of large-scale component-based software architectures, such as web services. In such complex environments it is likely that the same service can be obtained from different components, or from various combinations of the components. The end users or programmers, who would need to assemble the components in order to accomplish their goals, will inevitably be overwhelmed by the possibilities.

There have been increasing research efforts on methods for automatic goal-driven composition of workflows from information processing components. These composition problems are naturally related to planning. Indeed, a sequence of decisions must be made in order to choose and interconnect components such that the resulting processing system satisfies a specified goal.

Planning methods have been successfully applied to Web service composition, Grid computing, and deployment of component-based software (for some examples see (Doshi *et al.* 2004), (Blythe *et al.* 2003), (Kichkaylo, Ivan, & Karamcheti 2003) and references therein). In this work it is typically assumed that two components can be connected once one component can accept as input the type of data produced at the output of the other component, i.e. if there is an exact matching between the input and the output types.

In the classical planning formalism, the preconditions and the effects of the actions are specified in terms of globally defined predicates and fluents. This model is expressive enough to describe planning problems stemming from a wide variety of applications. Motivated by the need of building the workflows automatically in compositional computer systems and applications, we present an extension of this classical planning formalism. We extend this approach by adding planning language constructs that can describe multiple input and output ports of a component, thus partitioning the flat description of action preconditions and effects into data streams flowing into and out of the action. The data streams are described by predicates, and we assume that the actions compute predicates on output streams based on the predicates on input streams. Based on these ideas we designed a planning language referred to as Stream Processing Planning Language (SPPL). While we extend the classical planning formalism to capture the structure of stream processing problems, SPPL formalism is strictly more general than the classical one: if each processing component has exactly one input and one output, the stream planning problem is equivalent to the classical planning problem.

We consider the following variant of the planning problem, which is applicable to stream processing systems, as well as to many other workflow composition applications. In stream processing domains an action represents a stream processing component that produces one or more new data streams from several existing ones. The data flowing through the streams are described with a set of predicates on the streams, and these predicates are used to specify goals, preconditions and effects. The processing components can have multiple inputs and multiple outputs. Formally, we will say that the component exposes multiple input ports, and multiple output ports, and a stream can be connected to each port. Each input port is associated with a

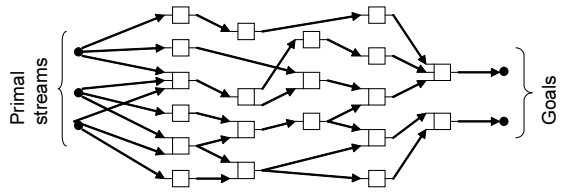

Figure 1: An example of stream processing workflow.

precondition written in terms terms of the predicates bound to the streams connected to it, and each output port has the effect of initializing predicates bound to the output stream. In the initial state only the *primal* streams are available. These are the streams that come into the system from outside. When all input ports of a component are connected to streams, the component produces one output stream for each of the output ports, by filtering, annotating, or otherwise analyzing and transforming the information it receives. Once a stream is produced by one of the components, and the predicates bound to the stream are initialized, the input ports of any number of other components can be connected to that stream. At the same time, none of the components can modify already existing and initialized stream. The goal of the workflow is to produce a number of output streams requested by the user. Each of the goal streams is described with a logical condition on the predicates bound to the stream, similarly to input port preconditions. The solution to the planning problem is a workflow, which we also call a plan graph (Figure 1).

We have made several attempts to formulate stream processing planning problems in PDDL (see (Ghallab *et al.* 1998) for definition of this planning language) and generate plans using the best available metric planners. However the fact that each action makes its output streams available for all subsequent actions makes the problem very difficult for traditional solvers. Specifically, actions need to produce predicates about an object that has not been previously initialized, and this condition turns out to be difficult to express in PDDL. On the other hand, each stream can be described with the same set of predicates, which can greatly simplify planning, provided the planner can take advantage of this property. Our experiments have shown that the extended formalism of SPPL gives significant advantage to the planner, and even simple planning algorithms using this formalism perform much better than the best of traditional planners on common and general examples.

In what follows we describe and analyze the SPPL formalism in detail, and describe experiments that demonstrate the performance advantages of the proposed formalism.

Stream Processing Planning Language

Traditional planning languages assume that actions are applied to the global state of the world, described by predicates. In contrast with this assumption, in distributed stream processing systems each of the processing components works only with the data it receives on incoming streams. Furthermore, descriptions of all streams have the same structure: stream descriptor of a stream can be seen as one object instance of a general stream description class.

Actions in SPPL

Since each processing component can receive more than one input stream, and produce more than one output stream, for each of these streams preconditions and effects must be specified independently. In our planning language we will represent each processing component by an action that has multiple input and output ports, to which the streams can be connected. Correspondingly, the traditional action description is extended as in the following example:

```
(:action A
  :precondition [in1] (and (P1)(P2))
  :precondition [in2] (and (P3)(P2))
  :precondition [in3] (and (P5)(P6)(P7))
  :effect [out1] (and (P4)(P6))
  :effect [out2] (and (P2)) )
```

In this example $in1$, $in2$, and $in3$ are the names of the input ports, and $out1$, $out2$ are the names of the output ports. The predicates $P1, P2, \ldots, P7$ describe properties of the streams. The exact set of the properties is determined by the application. For example, assume that the predicate $P2$ is used to mark video streams. Then, according to the action declaration above, ports $in1$ and $in2$ of the action above must be connected to video streams, and port $out2$ will produce a video stream. Other predicates on the stream may contain additional information about the stream, for example whether this video stream has close captioning, or whether it is encoded with MPEG-2 or MPEG-4.

Semantically this SPPL action is similar to the following PDDL action, assuming that the variables $?in1, ?in2, ?in3$ are bound to already existing streams, and $?out1, ?out2$ are bound to two distinct and uninitialized streams:

```
(:action A
  :parameters (?in1 ?in2 ?in3 ?out1 ?out2)
  :precondition (and (P1 ?in1)(P2 ?in1)
                (P3 ?in2)(P2 ?in2)
                (P5 ?in3)(P6 ?in3)(P7 ?in3))
  :effect (and (P4 ?out1)(P6 ?out1)
                (P2 ?out2)) )
```

Above, for each predicate from the SPPL formulation we have introduced a parameter that corresponds in PDDL to a stream object described by the predicate. The assumption we made above about assigning output port variables to uninitialized streams can be enforced in PDDL by making use of an auxiliary predicate that evaluates to true in the initial state:

```
(:action A
  :parameters (?in1 ?in2 ?in3 ?out1 ?out2)
  :precondition (and
     (notDefined ?out1)
     (notDefined ?out2)
     (P1 ?in1)(P2 ?in1)(P3 ?in2)(P2 ?in2)
     (P5 ?in3)(P6 ?in3)(P7 ?in3))
  :effect (and
     (not (notDefined ?out1))
     (not (notDefined ?out2))
     (P4 ?out1)(P6 ?out1)(P2 ?out2))  )
```

Assigning output to previously unused streams is important for modeling workflow composition problems correctly.

In stream processing, each component creates new output streams, and these streams, once created, are never modified or removed from the workflow by other components. Therefore, SPPL actions should not change properties of any streams that existed before the action was applied, and should create new streams instead. While this rule is known to the SPPL planner natively, domain-independent PDDL planners require additional predicates, as well as preconditions and effects defined in every action definition, to ensure that action effect applies to previously unused streams.

Merging of Input Predicates

In the model that we described above, the predicates that are not listed in the effect statement, will always be false on the stream connected to the output port, independently of corresponding input predicate values. However, in stream processing domains output predicates often depend on the input predicates. For example, a component that processes video may preserve closed captioning, if it was contained in the input stream. Therefore, SPPL must define formulas for computing the output predicate values, that at least allow some predicates to propagate from the input streams to the output streams.

Extending PDDL formalism to single input, multiple output case is straightforward: it is sufficient to specify effects for each of the outgoing streams. Effect statement will specify only the changes that must be applied to predicates on the input stream in order to compute the output stream predicates, and the rest remains unchanged. But what happens if one of the actions has more than one input, and same predicate evaluates to *true* only on a subset of the input streams? In our previous example, action A, if the predicate $P5$ is true for the input streams connected to the ports $in1$ and $in3$, but is false for $in2$, what should the value of $P5$ be on the output stream $out1$? What about $P5$ on $out2$?

In general, the syntax developed for conditional effects in PDDL can be used to specify the formulas for computing output predicates based on the input predicates. However, this approach would require the enumerating all propagated predicates in every effect statement. Since we expect predicate propagation to be very common, it can be unreasonable to assume that in the formulation we can list the names of many predicates in each action definition.

For these reasons, in our current implementation of SPPL we distinguish between two categories of predicates: the predicates that will be propagated from input to output, and the predicates that will not be propagated. The second category of predicates follows the model described in previous subsection.

We further subdivide the predicates that fall within the first category into two groups that we call *AND-logic* and *OR-logic*. The predicates are assigned to groups during problem formulation based on the condition required for the predicates to be propagated from the inputs to the outputs. A predicate from the AND-logic group must be *true* on all input streams of an action in order to be propagated and set to *true* on the output. A predicate belonging to OR-logic group will be propagated to the output if it evaluates to *true* on at least one input stream. Addition and deletion lists in the action effect statement are applied after the propagation rules. Note that once the decision to apply the action is made, the choice of propagation rule for each predicate is independent of the action, and only depends on the predicate itself.

The AND-logic group contains predicates that can be required by a precondition or a goal expression to be *true* on every stream in the workflow that leads to satisfying the constraint. Predicates in the OR-logic group can be used for describing stream properties that can satisfy the precondition as long as they are present on one of the input streams. Non-propagating predicates can be used to implement simple input-to-output matching in cases when the output of the action can be determined independently of the input.

Optimization Metrics in Stream Processing

Many researchers have considered planning domains where a resource metric is used as a minimization objective during construction of the plan. For example, the planning algorithm described in (Kichkaylo, Ivan, & Karamcheti 2003) considers the characteristics of the underlying hardware network, on which the software stream processing components are deployed and make deployment decisions based on resource availability in the network. Due to the anticipated large number of processing components and the need for more expressivity in specifying action effects in our model we decided to decouple the planning and resource allocation problems, and solve them separately.

In the current implementation we assign a scalar resource cost to each action. The optimization objective in this setting is the sum of costs of all actions used in the workflow. This resource model roughly corresponds to scheduling all processing components on the same processor and minimizing utilization of the limited processor resource. Alternatively, it can be seen as minimization of the cost associated with implementing the plan, given the cost of performing each action.

Complexity and Expressivity of SPPL

If each action has one input and one output, the problem of finding a legal plan becomes equivalent to finding a legal plan in STRIPS domain, which is known to be PSPACE (Bylander 1992), and therefore the problems of finding the optimal or feasible plan are at least as hard.

The non-metric version of the problem has a polynomial solution, if the predicates are not propagated from input to output, i.e. AND-logic and OR-logic groups are empty. The metric version, that is NP-hard in general, is polynomially solvable under this conditions, if in addition every action has single input and single output.

The expressivity of our model, as defined in (Nebel 2000), is at least that of STRIPS and less than that of planning models with general conditional effects, since for any STRIPS problem a corresponding formulation for stream planning can be constructed, and since the values of the output predicates are computed using restricted merging formulas.

The advantage of using stream planning formulation directly, instead of constructing a PDDL formulation first, solving it, and later converting the solution back to streams

and stream processing components, lies in the added efficiency that the search algorithm can gain from the additional structure explicitly specified in the SPPL formulation. The approach of translating SPPL to PDDL using variables to identify streams, as described in Section m will likely cause a metric PDDL planner to generate at least $O(N!)$ candidate plans for each unique candidate workflow graph of length N due to the symmetry in assigning stream objects to action parameters. Although methods for detecting different types of symmetries in general planning domains are known (Fox & Long 2002), SPPL makes the symmetry explicit and simplifies the planner's task by ensuring that the planner has the most complete knowledge of problem structure.

Planning Algorithm

We have implemented a general branch-and-bound procedure for solving the problem, which allows us to experiment with different search methods for finding optimal plans. Branch and bound is a standard approach to solving combinatorial problems, and it has been shown to be a successful method for solving planning problems.

Currently, the backward search (from the goal) is implemented: at each branching node a goal is chosen from the set of available nodes, and is connected to an existing primal or derived stream or to a newly instantiated action. If an action is instantiated, the input ports of the action are registered as new goals to be satisfied at the next step. The preconditions of the action in combination with the constraints on the output of the action are used to specify the new goal constraint for each of the inputs. The search tree is pruned if the cost of partial workflow exceeds the bound on cost, defined by the current solution or by problem constraints.

The algorithm gains additional efficiency from precomputing pairs of commutative actions and considering only one of the two possible orderings in the pair, therefore achieving the same effect as GraphPlan (Blum & Furst 1995). This is a stream equivalent of applying the commutative actions in parallel, extending this approach to the more general stream planning scenario. Potential conflicts (mutexes) are precomputed as well, and for each input port a lists of compatible actions is constructed. These lists are further filtered during branching according to the revised goals.

Experiments

We performed numerical experiments to study the performance benefits of our formalism. In these experiments we compare the performance of our planner to the performance of Metric-FF (Hoffmann 2003) [1] and LPG-*td* (Gerevini, Saetti, & Serina 2004). We have chosen these planners because they have demonstrated top performance among metric solvers in the International Planning Competition (in 2002 and 2004 resp.), and were available for evaluation.

In our experiments we generated a simple instance of the stream processing planning problem, and formulated it in both PDDL and the extended planning language. We then varied the size of the example and measured the time it took each of the planners to find the solution. We have constructed the examples that represent elementary planning problems that are likely to occur (in combinations) in practical stream processing planning scenarios. Therefore, the performance demonstrated by Metric-FF and LPG-*td* on these examples is indicative of the performance on significantly more complex problems that arise in practice.

All planners were run in sequence on the same 3.0 Ghz Pentium 4 computer with 500 megabytes of memory. For the same problem size, performance of the same planner can vary due to the randomness in problem instance generation and the random decisions taken by the planner (for example, LPG-*td* employs random restarts). Therefore we measured the average planning time on 15 randomly generated instances for each problem size. The experiments were terminated if the running time exceeded 10 minutes.

Generating Stream Planning Problems

The problem instances are constructed by first generating a flow graph: a random binary in-tree rooted at the goal stream, in which nodes have 2 incoming links with probability 0.3. For each arc in this tree we assign a unique predicate that is created as an effect of the action corresponding to the tail node, and require the same predicate in the precondition of the action associated with the head node. The predicate produced by the root is listed in the goal requirements. This assignment of predicates mimics the stream data type compatibility constraints. It ensures that the generated flow graph can be reconstructed during planning.

The leaf nodes in this tree are actions corresponding to the data sources. The data sources are the generators of the primal streams. They do not require any specific inputs and can be inserted in the plan at any time. However, while regular actions can appear in the plan multiple times, at most one instance of each data source can be included in the plan. This condition may be enforced directly by the planner, as in our implementation, or via a global predicate.

To specify optimization criteria, we have generated two random numbers for each node of the flow graph: the resource utilization and the quality contribution.

```
(:action N1i0   ...
 :effect (and (increase (cost) 111)
    (increase (quality) 81) ... ) )
```

For stream processing elements, both numbers follow Gaussian distribution with mean 100 and standard deviation 20. For sources, the mean quality is 1000 with deviation 200, and resource utilization constant 0. We have specified the sum of resource utilization numbers as the metric for minimization, and a minimum value for the sum of quality values as a goal constraint:

```
(:metric MINIMIZE (cost) )
(:goal (and  (>= (quality) 5110) ... ))
```

The total plan quality is computed as the sum of quality values assigned to planned actions. The quality metric roughly corresponds to the profit from investment in resources.

Additional effort is needed to prevent traditional planners from constructing plans that generate unused streams. Since the plans corresponding to the generated instances have in-tree structure, we can require that each generated stream is

[1] In our version of Metric-FF we have increased the maximum number of predicates and actions to 200 from the default value 50.

connected to the goal or to one of the input ports. To enforce this we used a predicate defined on stream objects indicating whether the stream is in a valid state (i.e. both ends of the stream are connected or the stream objects is not used). At the goal all stream objects must be in a valid state.

Important feature of stream processing is the functional dependency of output predicate sets on input predicates of the actions. To illustrate this, we have included one predicate that follows AND-logic during propagation. Conditional effects make it easy to write the AND formula that computes the value of output stream predicate based on the values corresponding to the incoming streams:

```
(:action N3i0
 :parameters ( ?in1 ?in2 ?out - stream )
 ...
 :effect (and ...
    (when (and (catA ?in1) (catA ?in2) )
          (and (catA ?out)) ))  )
```

Of the two planners that we evaluated, only Metric-FF allows conditional effects. For LPG-*td* we translated this action into two actions. One action required the predicate on both input streams in the precondition and created the predicate on the output stream, and another did not require or produce the predicate. The stream planner can recognize and support AND-logic natively, based on the declaration of the predicate as an AND-logic predicate.

First Scalability Experiment

There are multiple ways to encode stream planning problems in PDDL. We begin with evaluating the simplest encoding that maps one stream processing action to a single PDDL action, provided that conditional effects are supported.

In our encoding we use action parameters to refer to stream objects connected to inputs and outputs of the action. Since each stream processing action produces new stream objects without changing existing ones, each output stream object must be instantiated for the first time. In problem definition we declare sufficient number of stream objects (twice the number of actions), and initialize `(notDefined ?s)` predicate for each object. This predicate is removed once the stream object is used in effect of an action.

The following example illustrates this encoding. Consider the stream processing action:

```
(:action N1
 :precondition [in1] (and (T1) )
 :precondition [in2] (and (T2) )
 :effect [out] (and (T3) )  )
```

The corresponding PDDL action will be:

```
(:action N1
 :parameters ( ?in1 ?in2 ?out - stream )
 :precondition (and
    (notDefined ?out) (T1 ?in1) (T2 ?in2))
 :effect (and
    (not (notDefined ?out)) (T3 ?out)) )
```

The following table shows the results of the experiments. The first column contains the number of actions in the plan and the other columns contain the minimum, maximum and average running time in seconds for all planners that we tested. Column titled *Stream* corresponds to our SPPL planner implementation. In the table, "*" indicates that the solution was not found after 10 minutes, and "#" indicates that the solver terminated abnormally due to insufficient memory or other reasons. In this and other experiments the timeouts were often caused by thrashing that occurs when planners allocate and use more memory than is physically available.

Plan	*Stream*			*Metric-FF*			*LPG-td*		
size	min	avg	max	min	avg	max	min	avg	max
5	0.1	0.5	2.6	0.1	0.3	1.1	0.5	1.8	4.2
7	0.1	0.1	0.3	0.1	*	*	1.5	75.2	267.0
9	0.1	0.1	0.2	0.2	*	*	36.1	*	*
11	0.0	0.1	0.1	0.5	*	*	345.4	*	*
15	0.1	0.1	0.1	*	*	*	#	#	#
25	0.1	4.1	10.0	*	*	*	#	#	#
50	4.3	7.5	9.8	#	#	#	#	#	#
500	8.5	10.4	15.0	#	#	#	#	#	#

In this experiment we generated trivial stream planning problems, which have a single feasible plan. Moreover, finding this plan is trivial due to the unique matching of input and output predicates on each link. However, starting from just 11 actions, the planning problem becomes hard to solve for Metric-FF and LPG-*td*. We believe that the reason for such poor performance is excessive enumeration of different assignments of stream objects to stream variables, caused by the PDDL encoding. In the next experiment we modify the encoding to help planners avoid this enumeration.

Improved Scalability Experiment

The previous encoding allowed the planners to substitute different stream objects in action parameters, generating exponentially many alternative variable assignments. To help planners avoid this inefficiency we can assign fixed stream objects to the output stream variables. This can be achieved by defining unique predicate on the stream object and including that predicate in the precondition of the action:

```
(:action N1
 :parameters ( ?in1 ?in2 ?out - stream )
 :precondition (and
    (notDefN1 ?out) (T1 ?in1) (T2 ?in2))
 :effect (and
    (not (notDefN1 ?out)) (T3 ?out)) )
```

Then, at the initialization, the predicate `notDefN1` is defined only for one stream name: `(notDefN1 S1)`. However, this approach allows to use action `N1` at most once. To allow multiple instances of the same action within one plan, we generate multiple copies of each action with different stream objects connected to output streams.

Two copies of each action are generated in this experiment. The table below summarizes experiment results.

Plan	*Stream*			*Metric-FF*			*LPG-td*		
size	min	avg	max	min	avg	max	min	avg	max
5	0.0	0.1	0.2	0.0	0.1	0.3	0.3	0.4	0.7
11	0.0	0.1	0.1	0.2	10.7	56.6	0.5	1.0	2.3
15	0.1	1.1	2.8	46.0	*	*	0.9	12.7	19.8
19	0.1	1.8	4.9	*	*	*	1.5	19.8	25.7
30	0.3	1.1	2.5	*	*	*	12.8	29.8	57.5
35	0.3	10.2	29.9	*	*	*	329.0	*	*
50	0.1	6.7	13.5	#	#	#	#	#	#
500	8.4	9.9	12.5	#	#	#	#	#	#

Experiment shows that this encoding can indeed improve performance of the traditional planners, and the effect is especially noticeable in LPG-*td*. However, the problems of size above 50 can only be solved by stream processing planner. We employ this encoding in our following experiments.

Growing Action Set Experiment

In this experiment we considered a scenario in which there exist exactly 2 candidate plans, each composed of 6 actions. The planner must choose the best plan based on resource and quality characteristics. To scale the problem we generated actions that do not participate in candidate plans.

The first column in the table below shows the number of stream processing actions generated. The number of PDDL actions is higher due to the expansion during translation.

Stream	*Stream*			*Metric-FF*			*LPG-td*		
actions	min	avg	max	min	avg	max	min	avg	max
18	0.1	0.1	0.3	0.2	*	*	0.8	5.4	31.8
27	0.1	0.1	0.1	0.4	*	*	1.2	6.0	25.4
39	0.1	3.2	8.1	2.3	*	*	4.6	47.2	147.8
45	0.3	6.8	11.0	3.7	*	*	27.0	*	*
51	5.4	7.5	11.5	5.4	*	*	56.4	#	#
57	5.5	7.8	10.0	57.1	*	*	#	#	#
75	5.5	7.1	10.9	#	#	#	#	#	#
312	6.1	7.3	8.9	#	#	#	#	#	#
1512	34.8	36.7	42.9	#	#	#	#	#	#

This experiment shows again that the stream planning formalism allows our planner to scale much better than the traditional planners: it can solve problems of 1500 actions in under 40 seconds, while it takes the other planners more than 10 minutes to solve problems of just 57 actions.

In practice the problems are likely to exhibit this structure, having small optimal plan size together with a large number of actions that are not used in the solution. The assumption of only 2 candidate plans is too strong to hold in practice. We will address this issue in our next experiment.

Resource/Quality Tradeoff Experiment

In our last experiment we study how well our planner can scale with respect to the size of the problem, if the choice of action is not trivial, and the number of plan candidates is large. We generate a single plan graph for each problem instance. For each non-leaf node of the graph we now generate two alternative actions that have the same input and output requirements, but different resource and quality values. Constructing the optimal plan now requires choosing one of the alternatives in each of the nodes.

Plan	*Stream*			*Metric-FF*			*LPG-td*		
size	min	avg	max	min	avg	max	min	avg	max
5	0.1	0.2	0.3	0.1	0.1	0.2	0.4	0.5	0.8
9	0.1	0.3	0.5	0.2	*	*	0.9	1.2	1.9
11	0.3	0.7	1.1	0.5	*	*	1.5	3.1	15.8
15	2.3	5.1	9.3	*	*	*	5.1	11.1	35.5
19	13.7	46.6	71.4	426.6	*	*	49.9	*	*

This combinatorial search problem quickly becomes hard for our simple implementation, but it still has advantage over Metric-FF and LPG. We conclude that even in complex practical problems the structure expressed in SPPL is likely to lead to performance improvements, since simple binary tree graphs commonly appear as sub-graphs in the solution.

Conclusion

In this paper we investigated the application of planning for automatic construction of stream processing workflows, the new and important area of planning applications. We found that an extension of the classical planning formalism targeted towards stream processing planning problems can contribute to significant improvement in planner performance. This new formalism is flexible, backward-compatible with the classic formalism, and can be used in many automatic workflow composition applications.

We see two general directions in which our work can be further extended. First, more sophisticated and efficient planning algorithms can be developed for this formalism. We have implemented a simple backward search algorithm, and nevertheless were able to demonstrate advantages of the approach. However, there is a need in more efficient algorithms due to typically large sizes of these planning problems. Because SPPL extends classical planning, most of the heuristics and bounds developed for classical planning can be extended or applied directly in this framework. Second, the stream planning formalism we proposed can be enriched to include such PDDL extensions as conditional effects and functions computed on streams. The language we use for describing stream processing domains can be easily extended to incorporate corresponding statements.

Acknowledgments

We thank our colleague Genady Ya. Grabarnik for the assistance and valuable insights that he provided to us in preparation of this work. We also thank the anonymous reviewers for their very helpful comments and suggestions.

References

Blum, A. L., and Furst, M. 1995. Fast planning through planning graph analysis. In *Proc. IJCAI-95*, 1636–1642.

Blythe, J.; Deelman, E.; Gil, Y.; Kesselman, K.; Agarwal, A.; Mehta, G.; and Vahi, K. 2003. The role of planning in grid computing. In *Proc. ICAPS-03*.

Bylander, T. 1992. Complexity results for serial decomposability. In *Proc. AAAI-92*.

Doshi, P.; Goodwin, R.; Akkiraju, R.; and Verma, K. 2004. Dynamic workflow composition using Markov decision processes. In *Proc. ICWS-04*.

Fox, M., and Long, D. 2002. Extending the exploitation of symmetries in planning. In *Proc. AIPS-02*, 83–91.

Gerevini, A.; Saetti, A.; and Serina, I. 2004. Planning in PDDL2.2 domains with LPG-TD. In *International Planning Competition, ICAPS-04*.

Ghallab, M.; Howe, A.; Knoblock, C.; and McDermott, D. 1998. PDDL. The planning domain definition language. Technical Report DCS TR-1165, Yale Center for Computational Vision and Control.

Hoffmann, J. 2003. The Metric-FF planning system: Translating "ignoring delete lists" to numeric state variables. *Journal of AI Research* 20:291–341.

Kichkaylo, T.; Ivan, A.; and Karamcheti, V. 2003. Constrained component deployment in wide-area networks using AI planning techniques. In *Proc. IPDPS-03*.

Nebel, B. 2000. On the compilability and expressive power of propositional planning formalisms. *Journal of AI Research* 12:271–315.