Scalable Planning for Distributed Stream Processing Systems

Anton Riabov and **Zhen Liu**
IBM T. J. Watson Research Center
P.O. Box 704, Yorktown Heights, N.Y. 10598, U.S.A.
{riabov|zhenl}@us.ibm.com

Abstract

Recently the problem of automatic composition of workflows has been receiving increasing interest. Initial investigation has shown that designing a practical and scalable composition algorithm for this problem is hard. A very general computational model of a workflow (e.g., BPEL) can be Turing-complete, which precludes fully automatic analysis of compositions. However, in many applications, workflow model can be simplified. We consider a model known as the Stream Processing Planning Language (SPPL), applicable in stream processing and other related domains. SPPL replaces the notion of concurrency by timeless functional computation. In addition, SPPL defines workflow metrics of resource consumption and quality of service. Experiments have shown earlier that even a naïve SPPL planning algorithm significantly outperforms existing metric PDDL planners on stream processing workflow composition problems. In this paper we describe an efficient and scalable algorithm for finding high-quality approximate solutions for large instances of SPPL problems. We demonstrate the scalability of the algorithm on synthetic benchmarks that are derived from practical problems. We also give an example of SPPL model for practical problems.

Introduction

In systems based on compositional architectures, such as Web Services and Semantic Grid, applications can be specified by selecting a subset of available components, choosing configuration parameters for the selected components, and defining the interaction between the components. In other words, the applications can be defined by the workflows which are graphs depicting the interactions between the components. Automatic workflow composition relieves users from the burden of developing applications for every new task. Instead, users simply specify a description of the desired result and the system automatically generates a workflow that achieves the result. Needless to say, most compositional systems would benefit from such automatic workflow composition approach: it shortens the workflow development cycle and simplifies user interaction with the system. Various models and methods for workflow composition have been proposed in the literature, including models as diverse and varying in expressiveness and semantics as Petri nets, π-calculus, finite state machines, model checking, etc.

Planning is a natural formalism for describing workflow composition problems, since workflow can often be modeled as a composition of actions, i.e. components, achieving a specified goal, i.e. producing required results. Planning-based composition methods have been proposed for use in many application areas, including Web services (Doshi *et al.* 2004; Koehler & Srivastava 2003; Pistore, Traverso, & Bertoli 2005), Grid computing (Blythe *et al.* 2003; Gil *et al.* 2004), and change management (Brown, Keller, & Hellerstein 2005). We will refer to workflows as plans, and to the family of planning-based composition methods as workflow planning methods.

In this paper we describe a planning-based automatic workflow composition method that we use to implement automatic composition of distributed stream processing applications. Processing of streaming data is an important practical problem that arises in time-sensitive applications where the data must be analyzed as soon as they arrive, or where the large volume of incoming data makes storing all data for future analysis impossible. Stream processing has become hot research topic in several areas, including stream data mining, stream database or continuous queries, and sensor networks.

Our work on automatic planning is carried out in the framework of a new large-scale distributed stream processing system, that we refer to as System S, which allows the automatic deployment of automatically built stream processing plans in the distributed system. In System S, each application is represented by a plan, or a flow graph, where vertices represent processing components and arcs represent data streams. Each component exposes a number of input and output ports. Each data stream, either primal, i.e. arriving from an external source, or derived, i.e. produced by an output port of a component, can be connected to one of the input ports of another component, stored, or sent directly to the end user's console.

In planning for System S we consider the problem of composing the logical workflow separately from the resource allocation problem. The resource constraints supported in our system correspond to the model of execution on a single shared processor. System S includes a scheduler that

achieves the maximal usage of available resources within a distributed system at any time, and therefore the abstraction of a continuous resource pool is appropriate for the system. Note that some of the planning approaches proposed in the literature consider the workflow composition problem jointly with the problem of finding an optimal allocation of components to resources, e.g. machines (Kichkaylo, Ivan, & Karamcheti 2003), or consider only the allocation problem in separation (Gil *et al.* 2004).

For representing the workflow planning problem within System S, we have adopted the formalism of the Stream Processing Planning Language (SPPL), described in (Riabov & Liu 2005). In that paper we have shown that this formalism, derived from PDDL, exhibits significant improvements in planner scalability on stream processing planning domains. In SPPL, the workflow is modeled as a direct acyclic graph (DAG) describing the data flow between the components. The SPPL model assumes that once the input data for a component is available, the component can start producing the output data. In this model there are no control flows, synchronization or other time dependencies: all dependencies between the components are expressed only as data dependencies. These assumptions reflect the structure of stream processing problems, where the workflows are often executed for significant periods of time or being run indefinitely, until the user issues a termination request.

In this paper we first describe how SPPL formalism can be used as a model for workflow planning. We then describe a new scalable planning algorithm for SPPL domains. Using experiments, we show that our proposed approach together with the additional pre- and post-processing steps described in the paper allow significant performance improvements compared to previous work.

SPPL Model

In this section we briefly introduce the model of Stream Processing Planning Language (SPPL). For more details and investigation of associated performance benefits we refer the reader to (Riabov & Liu 2005).

SPPL Workflow Model

A solution to an SPPL planning problem is a workflow. Workflow is a composition of actions. Each action can have multiple input and multiple output ports. In the world model of SPPL, the streams of data from external sources exist before any actions are applied. The streams are described by predicates. An action can be applied only in the state where for each input port there is an existing stream that matches the precondition specified on the port. Once an action is applied, it performs a state transition by creating new streams. The values of predicates on the new stream can depend on input predicate values, and the effects of the action define the formulas for computing the output predicate values. The goal, or multiple goals, are specified similarly to preconditions. If multiple goals are specified, each goal must match one of the output streams of the constructed workflow.

In this model, at each state the workflow consists of the applied actions and the set of streams. The arcs connecting the input and the output ports of the actions correspond to streams. A feasible workflow, or equivalently, a feasible plan, produces a stream or a set of streams that match the goals.

SPPL Representation Language

SPPL extends PDDL by introducing unnamed objects (streams), action effects that create new unnamed objects, precondition and goal expressions that can be bound to the unnamed objects, and default predicate merging rules that are used for computing predicates on the new objects based on the predicates describing the input objects. In SPPL all predicates have an implicit parameter corresponding the the stream on which the predicate is defined. In addition, the predicates may have other parameters. As in PPDL, the parameters can be typed.

An SPPL action can have multiple effects, with one effect corresponding to one output port. The same applies for preconditions and input ports. During planning, the input and output ports of the actions are bound to the unnamed stream objects.

Consider an example of an SPPL action:

```
(:action A
 :parameters (?x - TypeX ?y - TypeY)
 :precondition (and (P1 ?x ?y) (P2 ?x))
 :precondition (and (P2 ?x) (P3))
 :precondition (and (P4) (P1 ?x ?y) (P6))
 :effect (and (P1 ?y ?x) (not (P2 ?x)))
 :effect (and (not (P3)) (P5)) )
```

This action requires 3 input streams, and creates 2 new output streams. Like in STRIPS, the effects are specified using the lists of additions and deletions. However, in SPPL the same predicate, e.g. P3, can be true on one input, and false on another. To determine the value of the predicate to which the effect of the output port will be applied, and hence determine the value of the predicate on each output stream, SPPL planner uses the default predicate propagation rules.

Each predicate belongs to one of the three predicate groups: AND-logic, OR-logic, and non-propagating predicates, or CLEAR-logic. The group to which each predicate belongs is specified when the predicate is declared. When computing value of CLEAR-logic predicate on the new stream, the effect of the corresponding output port is applied to the clear (i.e., false) state of the predicate. In other words, the value of the predicate will be true on the new stream if and only if the predicate is listed in the addition list of the effect corresponding to the output port that created the stream. For the AND-logic predicates the effects are applied to the logical AND of the values of the predicate on all input streams. For the OR-logic predicate the logical OR operation is used to combine the input values before the effects are applied.

Although it may seem that SPPL is limited and can be applied only in a narrow set of domains, SPPL is comparable to STRIPS. It has been shown that any STRIPS problem can be represented in SPPL using the same number of actions and predicates.

Resource and Quality Model of SPPL

In many planning applications, including planning stream processing workflows, the plans have associated cost and quality. The cost can be measured as the number of actions in the plan, for example. In SPPL each action can have a constant cost associated with the action. The total cost of the plan is defined as the sum of costs of individual actions included in the plan. Similarly, quality of the plan is the sum of individual quality values assigned to the actions.

Cost and quality parameters are specified in action definition using `:cost` and `:quality` statements. For example:

```
(:action A
  :cost (1.043)
  :quality (3)
 ... )
```

Depending on user requirements, the planner can be instructed to build a plan with highest quality, lowest cost, or to maximize quality subject to bounded cost.

Using this mechanism SPPL can represent the resource constraints that correspond to resource sharing in a single continuous and additive resource metric. For example, the resource cost of an action can be equal to MIPS requirement of the corresponding stream processing component, and the resource bound can correspond to the total MIPS performed by the processor on which the workflow is executed.

Defining a universal quality metric, which correctly reflects the quality of service provided by the workflow, is a difficult task. The fact that the perceived quality of service is often subjective, and varies from user to user, makes this task even more challenging. The simple additive quality model used in SPPL is designed to represent the tradeoffs between individual actions. It allows to measure the benefit or penalty resulting from replacing one action with another equivalent action by assigning different quality values to the two actions. Therefore, while in some scenarios richer quality models may be required, SPPL model is sufficient for formulating many practical problems.

SPPL Example

We illustrate SPPL model using a simple example of an SPPL planning domain.

In this example the workflow represents an execution plan of a relational database query. The goal corresponds to a query that can be processed using JOIN and SELECT operations. To specify the goal, the user will specify the list of attributes that must appear in resulting relation, and the list of conditions on these attributes. The database schema is represented as the initial state for the planner. The planner will construct a query execution plan for satisfying the query using the minimum number of JOIN and SELECT operations.

To simplify our example, we assume that the database schema is acyclic, i.e. by following any chain of foreign key relationships starting from any relation, the first relation in the chain cannot be reached. All foreign keys are assumed to consist of a single attribute.

We further assume that the SELECT condition is a conjunction of multiple conditions, where each of these conditions is specified on only one attribute (e.g., `age` $\geq$ 21 **and** `residence_state` $=$ `"NY"`). These conditions can then be modeled by partitioning the set of values of the attribute into a number of subsets, and specifying as part of the goal the list of subsets that should be removed by SELECT operations.

SPPL Domain. We will use the predicate `hasAttribute(?a)` to indicate whether an attribute is present in the stream. The pair of predicates `hasSubset(?s)` and `noSubset(?s)` is used to indicate whether a subset has been removed from a relation. One of the subset predicates always equals the negation of the other; both predicates are used in order to express negation in preconditions and goal expressions.

Defining the actions using these predicates is straightforward. We assign cost 1 to each action, since the objective is to minimize the total number of actions.

```
(define (domain RelationalQuery)
  (:types Attribute Subset)
  (:predicates :orlogic
    (hasAttribute ?a - Attribute)
    (noSubset ?s - Subset)   )
  (:predicates :andlogic
    (hasSubset  ?s - Subset)   )
  (:action Join
    :parameters (?a - Attribute)
    :cost (1)
    :precondition (hasAttribute ?a)
    :precondition (hasAttribute ?a)
    :effect () )
  (:action Selection
    :parameters (?s - Subset)
    :cost (1)
    :precondition (hasSubset ?s)
    :effect (and (not (hasSubset ?s))
                 (noSubset ?s)) )  )
```

Above, the merging rules for predicates (`:orlogic` and `:andlogic`) are defined based on JOIN semantics, since JOIN is the only action taking more than one input.

SPPL Problem Example. Each stream in this SPPL model corresponds to a relation. Hence, each of the initial state statements will correspond to a relation available in the database, and the goal will describe the properties of the relation resulting from the query.

Assume we have 3 tables, `EMPLOYEE(SSN, Name, DeptID)`, `SALARY(SSN, Salary)`, `DEPT(DeptID, Department)`. Consider the following query: `SELECT Department,Name,Salary WHERE Salary>=50,000`. Note that `FROM` clause is not specified, since the planner can identify the relevant tables automatically. The problem of constructing a plan for this inquiry can be represented in SPPL as follows:

```
(define (Problem Query1)
(:domain RelationalQuery)
(:objects SSN
 Name DeptID Salary Department - Attribute
 above50k below50k - Subset)
(:init (and (hasAttribute SSN)
 (hasAttribute Name) (hasAttribute DeptID)
 (hasSubset above50k) (hasSubset below50k)))
(:init (and
```

```
 (hasAttribute SSN) (hasAttribute Salary)
 (hasSubset above50k) (hasSubset below50k)))
(:init (and (hasAttribute DeptID)
 (hasAttribute Department)
 (hasSubset above50k) (hasSubset below50k)))
(:goal (and (hasAttribute Department)
 (hasAttribute Name) (hasAttribute Salary)
 (noSubset below50k)))  )
```

In the problem specification above we define 3 primal streams, one for each table. The goal is described by specifying the required attributes and the subsets that must be removed. An optimal plan created by an SPPL planner for this problem consists of two JOINs on `SSN` and `DeptID` attributes followed by SELECT to remove `below50k` subset.

Planning Algorithm

The algorithm consists of three stages:

1. Parameter substitution;
2. Preprocessing;
3. Backward search.

Stages 2 and 3 receive input from the previous stage, and therefore, over the course of solving the problem, 3 different formulations of the planning problem are created. The search is performed based on the last formulation created, and then the solution constructed for that formulation is refined to create the solution for previous stages, and finally, for the original problem.

Parameter Substitution

At this stage actions are grounded by substituting all possible combinations of objects for action parameters, and a propositional formulation is created. All predicates used in the SPPL file also become ground at this point.

Each of the ground predicates used in problem formulation is added to one of three arrays, as described below, such that each predicate appears with a particular set of actual parameters at most once in one of the arrays. All ground predicates corresponding to the same predicate, but with different parameter sets, must appear within the same array. One array is defined for each type of predicate group (i.e., one of AND, OR, CLEAR logic), and the ground predicates should be added to the arrays corresponding to their group. This procedure allows the algorithm to replace all references to the ground predicates in the actions by the corresponding index in the array. Provided that the array reference, i.e. predicate group type, is known, the index can be traced back to the original ground predicate.

This approach of creating propositional formulation by grounding actions is described in planning literature, and has been used in many automatic planners working with STRIPS or PDDL formalisms. The only modification that we made to this algorithm is the assignment of predicates to one of three groups (AND,OR,CLEAR), which are specific to SPPL and do not exist in PDDL or STRIPS.

During the procedure of grounding actions, the ground predicates that are specified as effects or initial conditions (i.e. in one of the init statements), but never referred to in preconditions or goal statements, can be removed. Similarly, at this step the actions can be removed if they contain in preconditions one or more ground predicates which are not included in any effect of some other operator, or in one of the init statements.

Preprocessing

The following steps of simplification and preliminary analysis are performed at the preprocessing stage:

1. Grouping of actions into super-actions;
2. Indexing of action preconditions and effects;
3. Forward propagation of singleton flags;
4. Efficient representation of the planning problem;
5. Connectivity check;

Below we discuss these steps in more detail.

Action Grouping. The actions that have exactly the same input and output port descriptions are combined to form super-actions. The actions corresponding to one super-action differ only in resource cost and quality values, and have the same preconditions and effects. For faster processing action grouping should be performed after the preconditions and effects have been indexed, since the index significantly increases the speed of finding actions that belong to the same group.

The preprocessing stage creates a new representation of the planning model. Within that new representation a single action can represent a group of actions of the original model, i.e. a super-action. The plans constructed for this modified model will need further refinement to determine exact assignment of actions: each super action included in the plan must be replaced by one of the actions from the group. The choice of the action to use in place of a super-action is made based only on the tradeoff between cost and quality.

The optimization problem of finding the best action assignment to the super-actions in the plan subject to cost bounds and with quality maximization objective is significantly easier to solve than the general planning problem, and well-known methods can be used to find approximate solutions. Our implementation achieves significant speed gain due to grouping of the actions, and subsequent use of approximation algorithm for multiple-choice knapsack. We have implemented a simple approximation algorithm based on dynamic programming, as described, for example, in (Hochbaum & Pathria 1994). More sophisticated approximation algorithms for multiple choice knapsack problems can achieve better performance, see (Kellerer, Pferschy, & Pisinger 2004) for an overview of knapsack algorithms.

Grouping is an optional step, which may also be performed before grounding, at the beginning of Stage 2.

Indexing Preconditions And Effects. To improve search speed, the planning algorithm uses an index of candidate action inputs for each output, and candidate outputs for each input. Since the values of predicates in the CLEAR group are defined independently of preceding actions, these predicates can be used to derive a necessary condition for matching: the add-list in the CLEAR group of the effect defined on the output port must be a subset of the CLEAR group of

the precondition on the matching input port. Another necessary condition is that the delete-list of the output has no intersections with the precondition for the input.

All pairs of inputs and outputs satisfying the necessary conditions are included in the index, so that for each port a list of matching candidates can easily be found. Initial states and goals are also included in the index, as outputs and inputs correspondingly.

While the indexing step is optional, it significantly improves search time by reducing search space, and enables efficient implementation of other preprocessing steps.

Forward Propagation of Singletons. In SPPL, an action can be declared as a singleton, meaning that at most one instantiation of this action can be included in a feasible plan. However, an action is a de-facto singleton if in the state space there can exist only one set of input streams for this action, i.e. for each input port of the action only one stream in the state space matches the corresponding precondition. Multiple instantiations of this action in the plan will not create new vectors, and therefore creating more than one instantiation is unnecessary and wasteful.

The de-facto singletons are detected by using the index of preconditions and effects and tracing back from action inputs to the initial conditions to find whether there exists more than one possible path (i.e. sub-plan) creating the inputs for this action. If at some point during this trace procedure more than one candidate output is found for one of action inputs, the path is not unique, and the action is not a de-facto singleton. However, if the inputs are traced back to initial streams or to other singletons, and no alternative candidates are encountered, the action is a de-facto singleton, and is marked with a singleton flag. During Stage 4 the planner will create at most one instantiation of an action marked with this flag within a plan.

Efficient Representation. During preprocessing efficient representation of stream state is used to describe preconditions and effects. The CLEAR group of the add-list of the effect of the action will always be equal to the corresponding group in the state of the stream assigned to action output. Therefore, the state of each stream is represented by a data structure that can be decomposed by groups, and the value of each group can either be specified explicitly, or by reference to another stream state description. This allows the use of pointers instead of copies for constant CLEAR groups when stream state is computed during search.

During search, the state of each stream created in the plan is described by a set of predicates. Since predicates can be enumerated, it is possible to represent stream state as vector, where each element has value of 0 or 1, and corresponds to a predicate. In our implementation we have experimented with both set and vector representation of the stream state. Vector representation allows the planner to achieve better performance if the number of ground predicates used in the propositional formulation is relatively small, i.e. below 200.

Connectivity Check. The index of candidate inputs and outputs also enables the planner to quickly verify whether the plan graph formed by connecting all available actions by directed edges corresponding to candidate connections is such that for each goal there exists at least one directed path from one of the initial streams to the goal. If this condition is violated, the planning problem has no solutions.

Optionally, a shortest path algorithm can also be used here to verify whether the resource bound can be met (for this, resource costs of all actions must be positive). For the purposes of shortest path computation the weight of all input links for each action should be set to the value of resource cost of the action in the selected dimension.

Backward Search

In this section we describe the algorithm that performs plan search. This algorithm is guaranteed to find an optimal solution if one exists.

Search Space. After the preprocessing stage, the resulting instance of the propositional planning problem is described by the following parameters:

- World state $\mathcal{S}$ is a set of streams. Each stream in $\mathcal{S}$ is a vector of predicate values of dimension n: $\mathbf{x} \in \{0,1\}^n$.
- Each property vector $\mathbf{x} \in \{0,1\}^n$ can be represented as a combination of sub-vectors corresponding to three predicate groups: AND, OR, and CLEAR; $\mathbf{x} = (\mathbf{x}_\wedge, \mathbf{x}_\vee, \mathbf{x}_F) \in \{0,1\}^{n_\vee} \times \{0,1\}^{n_\wedge} \times \{0,1\}^{n_F}$, where $n_\vee + n_\wedge + n_F = n$.
- Initial world state $\mathcal{S}^0 = \{\mathbf{x}_i\}_{i=1}^I$ is a set of primal streams received from external sources.
- $\mathcal{A} = \{A_i\}_{i=1}^m$ is the set of all actions. Each action A_i has J_i input ports and K_i output ports, $i = 1..m$. Let $K = \max\{K_i\}_{i=1}^m$ and $J = \max\{J_i\}_{i=1}^m$.
- $c_i \in I\!R_+$ is the resource cost of action A_i; $i{=}1..m$.
- $q_i \in I\!R_+$ is the quality metric of action A_i; $i{=}1..m$.
- $\mathbf{p}^{ij} \in \{0,1\}^n$ is the precondition corresponding to the input port j of action A_i; $i{=}1..m, j{=}1..J_i$.
- $\mathbf{s}^{ik} \in \{0,1\}^n$ is the add list vector corresponding to the output port k of action A_i; $i{=}1..m, k{=}1..K_i$.
- $\mathbf{r}^{ik} \in \{0,1\}^n$ is the delete list vector corresponding to the output port k of action A_i; $i = 1..m, k = 1..K_i$. We assume that $\mathbf{r}^{ik} \wedge \mathbf{s}^{ik} = \mathbf{0}$ for all i and k.
- $\mathcal{G} = \{\mathbf{g}^i\}_{i=1}^G$, $\mathbf{g}^i \in \{0,1\}^n$ – the set of goal vectors.

Action A_i, $i{=}1..m$ is legal in state $\mathcal{S}$ if for each precondition $\mathbf{p}^{ij}$, $j{=}1..J_i$, there exists $\mathbf{x} \in \mathcal{S}$ such that $\mathbf{x} \geq \mathbf{p}^{ij}$. In this case we will say that $\mathbf{x}$ matches $\mathbf{p}^{ij}$.

Let A_i be a legal action in state $\mathcal{S}$, let $\{\mathbf{x}^{ij}\}_{j=1}^{J_i}$ be the streams from $\mathcal{S}$ that match the preconditions $\{\mathbf{p}^{ij}\}_{j=1}^{J_i}$. Action A_i changes the state of the world from $\mathcal{S}$ to $\widehat{\mathcal{S}}$, where $\widehat{\mathcal{S}} = \mathcal{S} \cup \{\widehat{\mathbf{x}}^{ik}\}_{k=1}^{K_i}$. In this new state vectors $\{\widehat{\mathbf{x}}^{ik}\}_{k=1}^{K_i}$ are the stream state vectors of the newly created output streams, which are computed as following:

$$\widehat{\mathbf{x}}^{ik} = \begin{pmatrix} \bigwedge_{j=1}^{J_i} \mathbf{x}_\wedge^{ij} \\ \bigvee_{j=1}^{J_i} \mathbf{x}_\vee^{ij} \\ \mathbf{0} \end{pmatrix} \wedge \neg\mathbf{r} \vee \mathbf{s} \qquad (1)$$

The problem is to construct a plan – a sequence of legal actions that, when applied to the initial state $\mathcal{S}^0$, generates a state $\mathcal{S}^*$, in which for every $\mathbf{g}^i \in \mathcal{G}$ there exists $\mathbf{x}^* \in \mathcal{S}^*$ such that $\mathbf{x}^* \geq \mathbf{g}^i$. The planning problem may also require optimization, such as minimization of total resource cost, maximization of total quality, or maximization of quality subject to resource constraints. Total cost and quality of the plan are equal to the summation of corresponding values of all individual action instances included in the plan.

Backward Search Procedure. The backward search procedure enumerates feasible plans in depth-first-search order, starting from the goal. It follows a branch-and-bound approach of establishing current bounds, and pruning search nodes based on current best solution, however it does not establish bounds by solving linear programs.

A high-level pseudocode description of the search procedure is presented on Figure 1. In this algorithm, $\tilde{\mathcal{P}}$ is a structure describing the current partial solution. This structure contains the following fields:

- $\tilde{\mathcal{P}}.\mathcal{G}$ – a current set of sub-goals that are not yet satisfied;
- $\tilde{\mathcal{P}}.\mathcal{S}^0$ – a current set of produced streams;
- $\tilde{\mathcal{P}}.\mathbf{g}$ – a reference to the currently selected sub-goal;
- $\tilde{\mathcal{P}}.\mathcal{I}$ – a current set of candidate inputs for satisfying the current sub-goal $\tilde{\mathcal{P}}.\mathbf{g}$;
- $\tilde{\mathcal{P}}.\mathcal{V}$ – action instances in the partial plan that have all inputs connected to the primal streams or other actions in $\tilde{\mathcal{P}}.\mathcal{V}$, and therefore have completely determined vectors describing the output streams;
- $\tilde{\mathcal{P}}.\mathcal{V}^*$ – all other action instances in the partial plan;
- $\tilde{\mathcal{P}}.\mathcal{E}$ – edges in the plan graph corresponding to the streams connecting the input and output ports of action instances to each other, as well as connecting those ports to the goals or to the streams in the initial state.

The last four fields are not used in the algorithm on Figure 1, but they are updated by functions GET_CANDIDATES() and CONNECT_SUBGOAL().

Stack $\mathcal{L}$ is used to store previous partial solutions. We access the stack using subroutines PUSH() and POP() that implement the corresponding stack operations. Finally, set $\mathcal{B}$ is used to store the plans found by the algorithm.

Solution update procedure in lines 14–16 is described generally, ensuring that $\mathcal{B}$ contains all non-dominating solutions during multi-objective optimization. However, if the maximization objective is a one-dimensional plan quality metric, then $\mathcal{B} \leftarrow \mathcal{B} \cup \{\mathcal{P}\}$ should be interpreted as replacing the current best solution with a higher quality plan.

The function IS_VALID($\mathcal{P}$) returns TRUE if the resource cost of the partial plan ($\mathcal{P}$) does not violate the resource bound, and the plan is not labeled as invalid by the sub-goal satisfaction procedures that we discuss below.

GET_CANDIDATES() generates the list of possible candidates for satisfying a sub-goal and CONNECT_SUBGOAL() connects one of the candidates to the sub-goal. The candidates for satisfying a sub-goal are streams that can be divided into the following three categories: an existing stream

```
1.  P̃.G = G; P̃.I = ∅; P̃.S^0 = S^0;
    P̃.V = ∅; P̃.V* = ∅; P̃.E = ∅;
2.  L = {P̃}; B = ∅;
3.  While L ≠ ∅ do
4.    P̃ = POP(L);
5.    If IS_VALID(P̃) Then
6.      If P̃.I = ∅ Then
7.        If P̃.G ≠ ∅ Then
8.          Choose P̃.g ∈ P̃.G;
9.          P̃.G = P̃.G \ {P̃.g};
10.         P̃.I = GET_CANDIDATES(P̃, P̃.g);
11.         If P̃.I ≠ ∅ or P̃.G ≠ ∅ Then
12.           PUSH(L, P̃);
13.         End If
14.       Else If ∀P ∈ B : P̃ ⋠ P Then
15.         B ← B ∪ {P};
16.       End If
17.     Else
18.       Choose c ∈ P̃.I;
19.       P̃.I = P̃.I \ {c};
20.       If P̃.I ≠ ∅ or P̃.G ≠ ∅ Then
21.         PUSH(L, P̃);
22.       End If
23.       P = CONNECT_SUBGOAL(P̃, P̃.g, c);
24.       PUSH(L, P);
25.     End If
26.   End If
27. End While
28. Return(B);
```

Figure 1: Backward Search Algorithm.

in $\tilde{\mathcal{P}}.\mathcal{S}^0$; an effect of a new action instance added to the plan; and, an existing partially specialized stream, produced by one of the effects of action instances in $\tilde{\mathcal{P}}.\mathcal{V}^*$. The implementation of the subroutines GET_CANDIDATES() and CONNECT_SUBGOAL() depends on the category of the candidates, and we will describe the details of the implementation separately for different categories below. After that, we will discuss the methods for selecting sub-goals and candidates during search.

Connecting a sub-goal to an existing stream. If the selected sub-goal matches one or more of the streams existing in the current partial solution, all of these streams become candidates for satisfying this sub-goal. Formally, the following set is added to the set of candidates returned by GET_CANDIDATES(): $\{\mathbf{x} \in \tilde{\mathcal{P}}.\mathcal{S}^0 \mid \mathbf{x} \geq \tilde{\mathcal{P}}.\mathbf{g}\}$.

Connecting the sub-goal to such a candidate requires adding the corresponding link to the set of links in the new partial plan $\mathcal{P}.\mathcal{E}$. Once this link is added, it is possible that for some action $a \in \mathcal{P}.\mathcal{V}^*$, such that $\tilde{\mathcal{P}}.\mathbf{g}$ is a precondition

of a, all preconditions of a become connected to $\mathcal{P}.\mathcal{S}^0$. In that case, the action a must be moved from $\mathcal{P}.\mathcal{V}^*$ to $\mathcal{P}.\mathcal{V}$, and its outputs must be updated using formula (1) and added to $\mathcal{P}.\mathcal{S}^0$. Since this operation adds new streams to $\mathcal{P}.\mathcal{S}^0$, as a result other actions in $\mathcal{P}.\mathcal{V}^*$ may now have all inputs in $\mathcal{P}.\mathcal{S}^0$, and therefore must be moved to $\mathcal{P}.\mathcal{V}$. If that is the case, this procedure is repeated until no other actions can be moved to $\mathcal{P}.\mathcal{V}$. Since OR effects are not propagated to sub-goals, during this operation the OR preconditions of all affected actions must be verified, and $\mathcal{P}$ should be marked invalid if verification fails.

Connecting a sub-goal to a new action instance. Any action effect that may satisfy the sub-goal $\tilde{\mathcal{P}}.\mathbf{g}$ can be used for satisfying the goal, and will be returned as a candidate by `GET_CANDIDATES()`. Connecting the sub-goal to this candidate requires adding new action instance to $\mathcal{P}.\mathcal{V}^*$, and the link to the sub-goal to $\mathcal{P}.\mathcal{E}$. The preconditions of the action instance are updated using the inverse of formula (1): all members of AND group that are not added by the action itself must be added to every precondition. The preconditions are then added to the list of sub-goals $\mathcal{P}.\mathcal{G}$. Note that adding an action instance changes cost and quality of the plan.

The actions labeled as singletons are allowed only one instance within the plan, and therefore the second instantiation should be prohibited. For that purpose, a Boolean vector with an element for each of the actions is maintained to track the actions that were used in the current partial solution. The corresponding entry is set to true when an action is added to the plan. This enables quick rejection for singleton actions that are already instantiated.

As another search optimization measure, an index of all goals that were analyzed in constructing current partial solution is maintained. This index helps avoid the symmetry when the same goal appears multiple times within one plan. For example, assume there are two equal goals, and both goals can be satisfied by either action A or action B. This situation leads to multiple re-evaluation of the same set of plans. The algorithm can avoid this situation by assigning unique ID numbers to actions, and ensuring that actions are assigned to the goals in non-decreasing order of IDs. Therefore, if B has higher ID number than A, in the previous example the combinations AA, AB, and BB for satisfying the two goals will be possible, but BA will not be considered, because it is symmetric with AB. Combination AB here means that A is used to satisfy the first of the equivalent goals, and B is used to satisfy the second.

Connecting a sub-goal to a partially specified stream. The effects of actions in $\tilde{\mathcal{P}}.\mathcal{V}^*$ constitute the last set of candidates for satisfying the sub-goal. Since these effects are known only partially, only the CLEAR group predicates are considered in matching the effects of actions in $\tilde{\mathcal{P}}.\mathcal{V}^*$ to the sub-goal.

Since the effects are only partially specified, upon connecting the effect to the sub-goal the preconditions of one or more actions in $\tilde{\mathcal{P}}.\mathcal{V}^*$ may be affected. In particular, if any AND group predicates are required in the sub-goal, they must be propagated back, to the preconditions of the action supplying the candidate effect, as well as to the preconditions of the actions that have effects connected to preconditions of this action. If any conflicts are encountered during this propagation, the new plan $\mathcal{P}$ is marked infeasible. As before, the link between the effect and the sub-goal must be added to $\mathcal{P}.\mathcal{E}$.

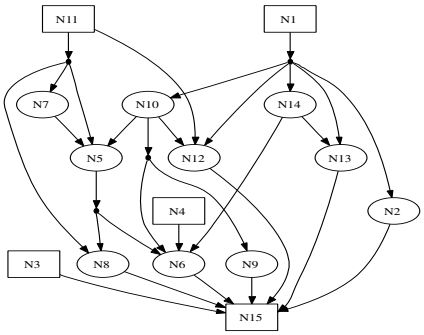

Figure 2: An example of a randomly generated plan.

Selecting sub-goals and candidates during search. The sub-goals are selected in last-in first-out order, and therefore the most recently added sub-goal is explored first. The candidates for satisfying a sub-goal are sorted by the number of predicates in the goal that they satisfy. While all candidates must satisfy all predicates in the CLEAR group, in AND and OR groups the sub-goal may be propagated back to sub-goals corresponding to the inputs of the action. Here we rely on the heuristic observation that in many cases the more predicates are satisfied, the more it is likely that the decision of adding this action will result in a feasible plan. Within the same number of common predicates, the actions are sorted by cost, such that the cheapest actions are considered first.

Scalability Experiments

We compare the performance of the algorithm described in this paper to the performance of 3 algorithms evaluated in (Riabov & Liu 2005) on synthetic benchmarks. We have extended the benchmark framework of that paper by generating plan graphs as general DAGs instead of considering only binary tree plan structures. In each benchmark, we first randomly generate one or more candidate solutions (Figure 2). Next, we define an action for each of the nodes, and define a unique non-propagating predicate without parameters for each output port of the action. Actions that receive input from that port in the generated graph will list that predicate in the precondition. Given this domain definition, and the goal equal to graph output predicate, the planner must reconstruct the graph. This benchmark simulates the practical problem of composing a stream processing workflow based on compatibility between inputs and outputs of components.

Arrows on the figure show the direction of the stream. The splitting arrows are used to show multiple input ports connected to one stream. The rectangular nodes that have no inputs in the graph on Figure 2 accept input from a single primal stream, not shown on the figure.

The DAGs were generated by distributing the nodes ran-

domly inside a unit square, and creating an arc from each node to any other node that has strictly higher coordinates in both dimensions with probability 0.4. The link is established from an existing output port (if one exists) with probability 0.5, otherwise a new port is created. The resulting connected components are then connected to a single output node.

To make the benchmark closer to practical problems, we include in the model one predicate that follows AND-logic propagation. This predicate is true in the initial state, and is required in the goal.

Further, we include an optimization objective, that requires minimization of resources subject to minimum quality constraint. Each action is assigned a positive randomly generated value of quality and resource cost. The quality of a plan is equal to the sum of individual quality values assigned to actions. The resource cost of a plan is the sum of the costs of actions included in the plan. Both resource and quality numbers for actions follow Gaussian distribution with mean 100 and standard deviation 20. For actions that accept the input from the primal stream we use different parameters, the mean quality is 1000 with deviation 200, and resource utilization is 0. In practical problems these actions correspond to the data sources, and the quality of the workflow often significantly depends on the choice of sources.

Testing PDDL Planners on SPPL Domains

Following (Riabov & Liu 2005), we also compare the performance of our algorithm to the performance of Metric-FF (Hoffmann 2003) (recompiled for 200 actions and predicates) and LPG-*td* (Gerevini, Saetti, & Serina 2004), and we use the same approach to represent SPPL problems in PDDL. The results presented in this paper are new, since we generate general random DAGs instead of binary trees.

We note that it is expected that the general-purpose PDDL planners should perform worse compared to the algorithms that are specialized for SPPL domains. We have considered several methods of translating SPPL problems into PDDL in order to achieve the best possible performance of PDDL planners on these problems.

Basic PDDL Encoding. In the basic method of translating SPPL models to PDDL, the unnamed stream objects of SPPL are modeled by named PDDL objects. In SPPL the streams are created by actions. We model this in PDDL by creating enough stream objects, and labeling the object "initialized" when it is used for a newly created stream. We use `free(?s)` predicate to indicate that stream `?s` is not initialized; this predicate is set to true in the initial state for all stream objects. Then, SPPL action

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

is represented in PDDL as

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

The AND-logic predicate propagation rule is modeled in PDDL using conditional effects:

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

If the conditional effects are not supported by the planner, it is possible to construct an equivalent model by creating several copies of the action for all combinations of preconditions and effects.

Improved PDDL Encoding. In the improved method, to avoid the symmetry in the choice of objects, we create 2 copies of each action, where each copy of the action is bound to a particular set of output objects by a predicate. For example, for action `A` we create two copies `A1` and `A2`. Action `A1` has precondition `freeA1out(?s)`, which it sets to false. Similarly, action `A2` has precondition `freeA2out(?s)`. The predicate `freeA1out` (correspondingly, `freeA2out`) is initialized in the initial state on only one stream object.

We refer to (Riabov & Liu 2005) for a more detailed discussion of modeling SPPL problems in PDDL.

Experiment Results

The planners were run on 3.0 Ghz Pentium 4 computer with 500 megabytes of memory. In each table of results discussed below the "Plan Size" column shows the number of actions in the plan. "Actions" column shows the number of SPPL actions included in the planning domain. We measured the average running time of each planner on 10 randomly generated SPPL planning tasks for each problem size. The tables show the average, maximum and minimum running time in seconds over 10 runs. The experiments were terminated if the running time exceeded 10 minutes, and "TIME" symbol indicates that in the table. Also, in many cases PDDL planners terminated abnormally due to memory allocation errors, and these cases are shown using "MEM" symbol.

For constructing our benchmarks we used a set of simple problems that are likely to appear as sub-problems within practical planning problems. The planners that show poor scalability in these tests are therefore more likely to perform worse in practice.

Single Plan. Tables 1 and 2 show the experiment results in the case where the only actions defined in the SPPL domain are the actions that are used in the plan. The difference between the two tables is only in the PDDL encoding used, and both SPPL planners show similar performance across the tables. The improved PDDL encoding increases maximum solvable problem size from 7 to 11 nodes, but in larger problems the PDDL planners are not able to find the solution because of memory allocation problems. The fact that in our experiment the PDDL planners could not solve stream processing problems with 13 actions or more stresses the need for models and algorithms that are specialized for these problems, and can scale to practical problem sizes. Both SPPL algorithm implementations perform much better.

Plan size	*Actions*	*Metric-FF*			*LPG-td*			*Naive SPPL*			*Improved SPPL*		
		min	avg	max	min	avg	max	min	avg	max	min	avg	max
5	5	0.09	1.74	6.83	0.48	TIME	TIME	0.08	0.09	0.10	0.07	0.08	0.09
7	7	0.65	MEM	MEM	63.30	MEM	MEM	0.08	0.13	0.39	0.08	0.09	0.12
9	9	TIME	MEM	MEM	MEM	MEM	MEM	0.07	0.09	0.10	0.08	0.09	0.11
11	11	MEM	MEM	MEM	TIME	MEM	MEM	0.08	0.09	0.11	0.08	0.10	0.14
15	15	MEM	MEM	MEM	MEM	MEM	MEM	0.08	0.09	0.11	0.08	0.10	0.11
25	25	MEM	MEM	MEM	MEM	MEM	MEM	0.09	0.10	0.11	0.11	0.12	0.14
50	50	MEM	MEM	MEM	MEM	MEM	MEM	0.14	0.18	0.25	0.17	0.21	0.24
100	100	MEM	MEM	MEM	MEM	MEM	MEM	1.06	2.68	4.41	0.49	0.55	0.64
500	500	MEM	MEM	MEM	MEM	MEM	MEM	TIME	TIME	TIME	9.77	10.12	10.37

Table 1: A single plan, basic PDDL encoding.

Plan size	*Actions*	*Metric-FF*			*LPG-td*			*Naive SPPL*			*Improved SPPL*		
		min	avg	max	min	avg	max	min	avg	max	min	avg	max
5	5	0.04	0.09	0.14	0.35	0.41	0.55	0.07	0.09	0.11	0.07	0.08	0.11
11	11	5.36	MEM	MEM	183.68	MEM	MEM	0.08	0.09	0.10	0.08	0.09	0.10
13	13	MEM	MEM	MEM	MEM	MEM	MEM	0.08	0.09	0.11	0.08	0.09	0.11
15	15	MEM	MEM	MEM	MEM	MEM	MEM	0.08	0.09	0.10	0.08	0.10	0.12
25	25	MEM	MEM	MEM	MEM	MEM	MEM	0.09	0.10	0.12	0.10	0.11	0.13
50	50	MEM	MEM	MEM	MEM	MEM	MEM	0.16	0.18	0.20	0.18	0.21	0.25
100	100	MEM	MEM	MEM	MEM	MEM	MEM	0.64	1.46	5.23	0.47	0.52	0.58
500	500	MEM	MEM	MEM	MEM	MEM	MEM	TIME	TIME	TIME	9.83	10.16	10.63

Table 2: A single plan, improved PDDL encoding.

Plan size	*Actions*	*Metric-FF*			*LPG-td*			*Naive SPPL*			*Improved SPPL*		
		min	avg	max	min	avg	max	min	avg	max	min	avg	max
20	66	MEM	MEM	MEM	MEM	MEM	MEM	0.06	0.68	1.16	0.12	0.14	0.15
20	75	MEM	MEM	MEM	MEM	MEM	MEM	0.09	0.22	0.77	0.14	0.19	0.40
20	90	MEM	MEM	MEM	MEM	MEM	MEM	0.11	0.15	0.34	0.20	0.21	0.25
20	99	MEM	MEM	MEM	MEM	MEM	MEM	0.12	0.13	0.15	0.20	0.22	0.26
20	105	MEM	MEM	MEM	MEM	MEM	MEM	0.13	0.20	0.70	0.20	0.23	0.25
20	111	MEM	MEM	MEM	MEM	MEM	MEM	0.13	0.15	0.16	0.21	0.24	0.26
20	120	MEM	MEM	MEM	MEM	MEM	MEM	0.14	0.18	0.37	0.24	0.26	0.28
20	210	MEM	MEM	MEM	MEM	MEM	MEM	0.29	0.34	0.67	0.48	0.51	0.54
20	360	MEM	MEM	MEM	MEM	MEM	MEM	0.54	0.57	0.67	1.02	1.08	1.11
20	1560	MEM	MEM	MEM	MEM	MEM	MEM	2.54	2.76	3.17	14.92	15.52	15.91

Table 3: 3 candidate plans of 20 actions each, and a large number of unrelated actions.

Plan size	*Actions*	*Metric-FF*			*LPG-td*			*Naive SPPL*			*Improved SPPL*		
		min	avg	max	min	avg	max	min	avg	max	min	avg	max
5	10	0.09	0.13	0.17	0.48	0.79	1.48	0.08	0.10	0.15	0.07	0.08	0.10
9	18	MEM	MEM	MEM	MEM	MEM	MEM	0.09	0.11	0.18	0.07	0.12	0.23
11	22	MEM	MEM	MEM	MEM	MEM	MEM	0.08	0.11	0.17	0.09	0.10	0.12
15	30	MEM	MEM	MEM	MEM	MEM	MEM	0.21	0.53	1.12	0.09	0.14	0.25
19	38	MEM	MEM	MEM	MEM	MEM	MEM	0.61	6.90	19.13	0.10	0.20	0.82
21	42	MEM	MEM	MEM	MEM	MEM	MEM	0.98	79.90	363.03	0.10	0.20	0.47
23	46	TIME	MEM	MEM	MEM	MEM	MEM	5.76	TIME	TIME	0.11	1.74	15.33
50	100	MEM	MEM	MEM	MEM	MEM	MEM	599.78	TIME	TIME	0.21	7.87	15.75
200	400	MEM	MEM	MEM	MEM	MEM	MEM	TIME	TIME	TIME	15.48	16.90	18.12
500	1000	MEM	MEM	MEM	MEM	MEM	MEM	TIME	TIME	TIME	21.81	23.75	25.42

Table 4: Resource/quality tradeoff: 2 alternatives for each action.

The naive SPPL implementation shows more variability in running time compared to the improved SPPL planning algorithm described in this paper. On large problems over 100 actions the improved SPPL algorithm finishes more than 10 times faster than the naive one. This difference is mostly due to propagation of singletons, which is very effective in this type of problems.

Unrelated Actions. In practical problems of stream processing workflow composition the number of available actions is likely to be large, but the size of the plan will be limited. We model this scenario by generating 3 candidate plan DAGs of 20 actions each, plus a varying number of actions that are not included in the plan. Table 3 shows the scalability of the planners with respect to the number of actions. From previous experiment we know that PDDL planners face memory allocation problems starting from 11 actions, and in this experiment, where the smallest plan has 20 actions, these planners cannot find the solution. Notice that in this experiment the improved SPPL planner performs worse then the naive algorithm on very large problems due to extra time spent in pre-processing. However, the performance penalty is not significant: the problem with 1,560 actions is solved in 15 seconds. To avoid the penalty completely, a hybrid solution can be used, i.e. preprocessing should be performed only after it is determined that blind search takes significant time.

Resource/Quality Tradeoff. Table 4 shows the results of the experiment in which we investigate the effect of alternative actions. As in the first experiment, we generate a single DAG of fixed size, and study planner scalability with respect to plan size. However, in this experiment we generate two alternative actions for each DAG node. Both action alternatives have the same preconditions and effects, and differ only in the quality and resource cost values, which are independent and randomly generated as we described earlier. As expected, the time spent by search-based algorithm increases exponentially with the number of actions, and the naive SPPL planner requires more than 10 minutes to find the optimal plan of 23 nodes. The improved algorithm takes advantage of action grouping, and in 25 seconds finds a near-optimal solution even for 500 nodes. In this experiment the solution found by the improved SPPL algorithm is within 0.1% of optimal.

The experiments show that our approach is particularly suitable for large scale systems, where the new technique exhibits order of magnitude improvement.

Conclusion

In this paper we described the application of planning for composition of stream processing workflows. This planning method was developed for the automatic building and deployment of applications in System S, a large scale distributed stream processing system. Stream Processing Planning Language (SPPL) formalism is a good match for describing the planning problems arising in this context. We have proposed a scalable planning algorithm for SPPL problems and we have studied the performance improvement of our approach using synthetic benchmarks that were derived from practical scenarios.

An important aspect of workflow planning is modeling the semantics of a data stream and the semantics of stream processing components. Many approaches developed in Semantic Web area, such as the OWL-S standard, can be used in stream processing environment. Semantic annotations on streams may allow richer goal expressions that refer to the context in which the data has been derived. We leave the issues related to semantics for future work.

Another future work topic is the investigation of richer models for modeling quality of service associated with the workflow. While SPPL quality model allows scalable planning and can express many practical aspects of quality, in some scenarios it can unnecessarily limit the capabilities of the automatic composition module. Alternatively, the quality of service is a measure of the difference between the submitted request and the produced result. Recently proposed rule-based approaches for specifying user preferences suggest another direction in which the quality model can be improved further.

References

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. of ICAPS-03*.

Brown, A.; Keller, A.; and Hellerstein, J. 2005. A model of configuration complexity and its application to a change management system. In *Proc. of IM-05*.

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

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

Gil, Y.; Deelman, E.; Blythe, J.; Kesselman, C.; and Tangmurarunkit, H. 2004. Artificial intelligence and grids: Workflow planning and beyond. *IEEE Intelligent Systems*.

Hochbaum, D. S., and Pathria, A. 1994. Node-optimal connected k-subgraphs. *Manuscript*.

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

Kellerer, H.; Pferschy, U.; and Pisinger, D. 2004. *Knapsack Problems*. Springer, ISBN 3-540-40286-1.

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

Koehler, J., and Srivastava, B. 2003. Web service composition: Current solutions and open problems. In *Proc. of ICAPS-03, Workshop on Planning for Web Services*.

Pistore, M.; Traverso, P.; and Bertoli, P. 2005. Automated composition of web services by planning in asynchronous domains. In *Proc. of ICAPS-05*.

Riabov, A., and Liu, Z. 2005. Planning for stream processing systems. In *Proc. of AAAI-05*.