A Novel Graph Based Approach for Automatic Composition of High Quality Grid Workflows

Jun Qin
jerry@dps.uibk.ac.at

Thomas Fahringer
tf@dps.uibk.ac.at

Radu Prodan
radu@dps.uibk.ac.at

Institute of Computer Science, University of Innsbruck
Technikerstr. 21a, 6020 Innsbruck, Austria

ABSTRACT

The workflow paradigm is one of the most important programming models for the Grid. The composition of Grid workflows has been widely studied in the Grid community. However, there is still a lack of a general and efficient approach for automatic composition of Grid workflows. In this paper, we present a STRIPS (Stanford Research Institute Problem Solver) based formal definition of the Grid workflow composition problem, followed by a novel graph based algorithm for automatic composition of high quality (portable, fault tolerant and optimized) Grid workflows. Our algorithm searches for semantic descriptions of workflow activities, i.e., Activity Functions (AFs), defined by ontologies and composes them into Grid workflows using AF Data Dependence (ADD) graphs. The composition process consists of three phases: ADD graph creation, workflow extraction, and workflow optimization. The worst case complexity of our algorithm is quadratic in the number of AFs. An extension of our algorithm to compose Grid workflows with branches and loops is also presented. Experimental results illustrate the effectiveness and efficiency of our approach: (i) the measured worst case execution time of our algorithm further proofs the analyzed time complexity; (ii) the composition of the real world meteorology Grid workflow application MeteoAG with our algorithm takes approximate half a second; and (iii) the execution time of the MeteoAG workflow when running on the Austrian Grid is reduced by up to 25% and the speedup is increased by up to 2.24 by applying our workflow optimization techniques.

Categories and Subject Descriptors

D.1.3 [**Concurrent Programming**]: Distributed programming;
D.1.3 [**Concurrent Programming**]: Parallel programming;
I.2.2 [**Automatic Programming**]: Program synthesis;
I.2.8 [**Problem Solving, Control Methods, and Search**]: Plan execution, formation, and generation

General Terms

Algorithms, Design, Performance

Keywords

Grid Computing, Grid Workflow, Automatic Workflow Composition, Workflow Synthesis

HPDC'09, June 11–13, 2009, Munich, Germany.

1. INTRODUCTION

With the development of Grid technologies, scientists and engineers are building complex applications to execute scientific experiments on distributed Grid resources. Grid workflow, as a programming model, plays a paramount role in this process. A Grid workflow application can be seen as a collection of activities (computational tasks) that are processed in a well-defined order to accomplish a specific goal. Formally, a Grid workflow w is a pair $(A, \overrightarrow{D})$, where A is a finite set of activities and $\overrightarrow{D}$ is a finite set of dependences. Each dependence $\overrightarrow{d_i} \in \overrightarrow{D}$ is either a control flow or a data flow dependence associated with an ordered pair of activities (a_m, a_n), where $a_m, a_n \in A$. The composition of Grid workflows involves the selection of activities, and the specification of control flow and data flow dependences among the selected activities.

Extensive research [13, 2, 8, 17] has proposed abstract descriptions of workflow activities which are independent of the changes of Grid resources. Since an abstract activity represents a group of concrete computational entities (e.g., software components, Web or Grid services) which have the same functionality and the same input and output data structure, the number of abstract activities is usually considerably smaller than the number of concrete computational entities. Consequently, the efforts required for the selection of workflow activities is reduced, and the process of Grid workflow composition is simplified. However, the selection of abstract activities is still a challenging and time consuming process for domain scientists especially when there are hundreds or thousands of such abstract activities that are available for selection.

Ontology [16] technologies have been used by the Grid community [24, 5, 32, 17, 27, 23] for semantic descriptions and discovery of workflow activities. An ontology is a formal representation of a set of concepts within a domain and the relationships among those concepts. *Classes* in an ontology are those concepts that are abstract groups, sets or collections of objects, in contrast to *Individuals* which are instances or objects (the basic or "ground level" objects). A class can subsume or be subsumed by other classes; a class subsumed by another is called a *subclass* of the subsuming class. For example, *Animal* subsumes *Fish*, since (necessarily) anything that is a member of the latter class is a member of the former.

Automatic workflow composition has been widely studied in several areas like Business Process Management, Semantic Web Services and Grid computing. Comparing with other areas, automatic workflow composition in the Grid computing area usually requires that the composed Grid workflows should be (i) *portable* or adaptable to the changes of Grid resources, i.e., the Grid workflows should be abstract and can be concretized when they are actually executed on the Grid; (ii) *fault tolerant*, e.g., Grid workflows should contain alternative execution paths, if available. In case an activity is not available or fails at runtime due to the dynamic nature of the Grid, an alternative execution path should be adopted; and (iii) *op-*

timized: Grid workflows should be optimized for execution time. We call portable, fault tolerant and optimized Grid workflows *high quality Grid workflows*.

To the best of our knowledge, there is still a lack of a general and efficient approach for automatic composition of high quality Grid workflows. With the help of ontology technologies, our previous work [26] presented a semantic based approach for Grid workflow composition by separating concerns between data meaning and data representation, and between Activity Function (semantic description of workflow activities, hereafter abbreviated as AF) and Activity Type (syntactic description of workflow activities, hereafter abbreviated as AT), thus automated data flow composition of Grid workflows. This paper is an extension of our previous work and focuses on automatic control flow composition of Grid workflows. In this paper, based on the STRIPS (Stanford Research Institute Problem Solver) language [12], a base language for expressing automated planning problems in Artificial Intelligence (AI), we present a formal definition of the Grid workflow composition problem, followed by an AI planning based algorithm for automatic Grid workflow composition using an AF Data Dependence (ADD) graph. Our algorithm employs progression to create an ADD graph, and regression to extract workflows, including the alternative ones if available. Our algorithm also optimizes the extracted workflows by analyzing data flow dependences among AFs. In contrast to existing approaches (see Section 7), our algorithm is general, i.e., not limited to any workflow modeling notation such as Petri Nets, and can efficiently and automatically compose high quality Grid workflows. The complexity of our algorithm is a quadratic in the number of AFs. Three series of experimental results illustrate the effectiveness and efficiency of our approach.

Our approach has been implemented as part of the ASKALON Grid application development and computing environment [10], where Grid workflows are described by the Abstract Grid Workflow Language (AGWL) [11]. The remainder of this paper is organized as follows. An overview of AGWL is provided in Section 2. Section 3 presents a formal definition of the Grid workflow composition problem. ADD graph and the related notations are introduced in Section 4. The graph based algorithm for automatic Grid workflow composition is discussed in detail in Section 5 which also includes the complexity analysis of our algorithm and the extension of our algorithm for the composition of Grid workflows with branches and loops. Three series of experimental results are presented in Section 6. Section 7 compares important related work against our approach. The paper ends with a conclusion and an outline of future work.

2. AGWL

AGWL [11] is an XML-based language for describing Grid workflow applications at a high level of abstraction. AGWL has been designed such that users can concentrate on specifying Grid workflows without dealing with the complexity of the Grid.

In AGWL, a concrete computational entity such as an executable, a software component, or a Grid/Web service deployed in the Grid is called an *Activity Deployment (AD)*. An *Activity Type (AT)* is an abstract description of a group of ADs which have the same functionality and the same input and output data structure, but probably different performance behaviors, QoS characteristics and costs. An *Activity Function (AF)* is a more abstract description of a group of ATs which have the same functionality and the same input and output data semantics, but probably different data representations (e.g., where data is stored, what the content type is, etc., see [26]). In other words, ADs, ATs and AFs are concrete, syntactic and semantic descriptions of workflow activities, respectively. Users compose Grid workflows at the semantic level, i.e., using AFs. The mapping from AFs to ATs and further to ADs are done automatically by the ASKALON runtime system. For details, readers may refer to our previous work [26] and [29].

Figure 1: The Activity Function *RAMSHist*

The Abstract Grid Workflow Ontologies (AGWO) [26] are a set of ontologies used to describe AFs. AGWO consists of three main concepts: *Function*, *Data* and *DataRepresentation*. Domain specific functions and data can be defined by subclassing *Function* and *Data*. Two main properties of *Function* are *hasInput* and *hasOutput* whose values are *Data* or its subclasses. With the help of AGWO, an AF is formally defined as follows:

$$AF = \langle F, I, O \rangle$$

where F indicates the function (referring to *Function* or one of its subclasses), I is a set of data classes (referring to *Data* or its subclasses, hereafter abbreviated as DC) indicating the input data, and O is a set of DCs indicating the output data. For convenience, we denote the input and output data classes of an AF by $AF.I$ and $AF.O$, respectively. Fig. 1 shows an example AF *RAMSHist* in the meteorology domain, where $AF.I = \{Month, SeaSurface\}$ and $AF.O = \{RAMSModeledAtmosphere\}$ respectively denotes the input and output data of the function *RAMSHist*.

A rich set of control flow constructs has been provided in AGWL to simplify the specification of Grid workflow applications such as `sequence`, `parallel`, `if`, `switch`, `while`, `doWhile`, `for`, `forEach`, `parallelFor` and `parallelForEach` with semantics similar to comparable constructs in high-level programming languages. The `dag` construct is also provided for describing a Directed Acyclic Graph (DAG) of activities.

In AGWL, input and output data of activities are described by data ports. The data flow among activities is expressed by data flow links from source data ports to sink data ports. When a source data port is connected to a sink data port with data flow, the data produced at the source data port will arrive at the sink data port at runtime when the data is to be consumed. One source data port may have multiple sink data ports and in that case each sink data port will receive a copy of the data produced at the source data port. AGWL also supports data collections which are ordered lists of data elements provided as initial inputs of a workflow or produced by workflow activities.

Properties and constraints can be defined in AGWL to provide additional information for Grid workflow composition and execution environments for optimization and steering. Particularly, constraints should be complied with by these environments. Users can specify constraints for both activities and data ports. Related to Grid workflow composition, the example constraints are to access the elements of a data collection in parallel (or sequential), etc.

This paper aims to automatic compose Grid workflows at the semantic level. For this purpose, we will only focus on AFs, instead of ATs or ADs. We will also only focus on DCs, instead of data representations or concrete data (e.g., a specific value or file). In the remainder of this paper, the availability of a DC means that of the corresponding concrete data. The terms *data* and *data class (DC)* will be used interchangeably for convenience.

3. DEFINITION OF THE PROBLEM

STRIPS divides its representational scheme into three components, namely, *an initial state*, *a goal state*, and *actions*. A plan for such a planning problem is a sequence of actions that can be executed from the initial state and that leads to the goal state. We use STRIPS as the basis for the definition of the Grid workflow composition problem, where *states* and *actions* are defined as follows.

– **state**: a state is a set of DCs, indicating the availability of data. The *initial state* indicates the user provided data which

can be consumed by workflow activities. The *goal state* indicates the user required data which must be produced by the composed Grid workflow.

– **action**: AFs are the actions. AFs can consume and produce DCs, thereby states can be changed and the goal state can be reached. Note that in the domain of Grid workflow composition, the consumption of data does not make it unavailable because AFs can work on copies of data.

For a given state s_i, an AF can be included (i.e., applied) in a workflow when s_i entails $AF.I$, denoted by $s_i \vDash AF.I$. Given two sets of DCs P and Q, $P \vDash Q$ is defined as follows.

$$P \vDash Q = \begin{cases} true & \forall q \in Q : (q \in P) \vee \\ & (\exists p \in P, p \text{ is a subclass of } q) \\ false & \text{otherwise} \end{cases} \quad (1)$$

$P \vDash Q$ is $true$ if and only if for any $q \in Q$, P subsumes q or P subsumes a subclass of q (see Section 1 for the explanation of *subclass*). For example, if $P = \{D_1, D_{2.1}, D_3\}$, $Q_1 = \{D_1, D_3\}$, $Q_2 = \{D_2, D_3\}$ and $Q_3 = \{D_1, D_4\}$ (D_i or $D_{i.j}$ indicates a data class, and $D_{i.j}$ is a subclass of D_i for any natural number i and j), then both $P \vDash Q_1$ and $P \vDash Q_2$ are *true*, and $P \vDash Q_3$ is *false*, i.e., $P \nvDash Q_3$. For example, the AF illustrated in Fig. 1 can be applied in the state $s_i = \{Month, SeaSurface\}$, where $Month$ and $SeaSurface$ are either the initial input data provided by users or the output data produced by any AF which is already included in the workflow. The new state after applying the AF to s_i is $s_{i+1} = s_i \cup AF.O$, i.e., $\{Month, SeaSurface, RAMSModeledAtmosphere\}$.

We consider a Grid workflow composition problem as an AI planning problem which is defined by the function

$$f : (s_{init}, s_{goal}, \mathcal{AF}) \rightarrow w$$

where each component is described as follows.

– s_{init} is the initial state.

– s_{goal} is the goal state.

– $\mathcal{AF}$ is the set of AFs among which some AFs will be selected for the composition of the Grid workflow w.

– w is a DAG of AFs connected by control flow edges. The AFs in the DAG fulfill the following restrictions:

1) $s_{init} \vDash AF.I$ for any AF which has no incoming control flow edges

2) $s_{init} \cup \left(\bigcup_{AF' \in \mathcal{AF}'} AF'.O\right) \vDash AF.I$ for any AF which has incoming control flow edges. Here $\mathcal{AF}'$ is the set of predecessors of this AF.

3) $s_{init} \cup \left(\bigcup AF.O\right) \vDash s_{goal}$

Automatic composition of Grid workflows with loops and branches is explained in Section 5.5.

4. ADD GRAPH AND NOTATIONS

This section defines the ADD graph and the related notations used for the explanation of our algorithm. The notations are explained through the example ADD graph illustrated in Fig. 2, where rectangles labeled with $D_0, D_1, ...$ are different DCs and round cornered rectangles labeled with $AF_0, AF_1, ...$ are different AFs. The edge connecting from D_j to AF_i indicates that the AF requires D_j as input. The edge connecting from AF_i to D_k indicates that the AF produces D_k as output. The dotted round cornered rectangles are used to group a set of AFs to improve the readability of ADD graphs. The notations at the bottom of Fig. 2 are explained below.

Contributing AF (cAF): For a given state s_i, an AF is said to be a *contributing AF* if and only if $(s_i \vDash AF.I) \wedge (AF.O - s_i \neq \emptyset)$. That is, s_i entails $AF.I$ and $AF.O$ includes some DCs which are not in s_i. In other words, the AF can be applied in state s_i and applying the AF to s_i can produce new DCs. For example, in Fig. 2, AF_0, AF_1 and AF_2 are three cAFs of the superstate S_0 (explained below).

Superstate: Applying all cAFs of a state s_i to this state causes the transition to a new state. We call the new state a *superstate* (analogy to *superset* in *set* theory) because it is a union of all possible states which can be reached from s_i by applying any of these cAFs. Let us denote a superstate by S, the initial superstate by S_0 and let $S_0 = s_{init}$. Let us also denote the set of all cAFs of a superstate S by $cAF(S)$. For example, in Fig. 2, $cAF(S_0) = \{AF_0, AF_1, AF_2\}$, $cAF(S_1) = \{AF_3, AF_4, AF_5\}$, $cAF(S_2) = \{ AF_6, AF_7, AF_8, AF_9 \}$ and $cAF(S_3) = \{ AF_{10} \}$. Applying $cAF(S_i)$ to S_i causes the transition to S_{i+1} which is defined as:

$$S_{i+1} = S_i \cup \left(\bigcup_{AF \in cAF(S_i)} AF.O \right) \quad (2)$$

Contributed DC (cDC): It is obvious that S_{i+1} always contains some newly contributed (i.e., produced) DCs which are not in S_i. Let us denote the set of all cDCs in a superstate S by $cDC(S)$. Then, $cDC(S_{i+1}) = S_{i+1} - S_i$. For example, in Fig. 2, $cDC(S_1) = \{D_2, D_3, D_4\}$, $cDC(S_2) = \{D_5, D_6, D_7\}$, $cDC(S_3) = \{D_8, D_9\}$, and $cDC(S_3) = \{D_{10}\}$, as illustrated by rectangles with blue (or gray) background. We also consider all DCs in S_0 are cDCs, that is, $cDC(S_0) = S_0 = \{D_0, D_1\}$.

ADD Graph: An ADD graph γ is a triple $\langle \mathcal{A}, \mathcal{D}, \overrightarrow{\mathcal{D}} \rangle$, where $\mathcal{D}$ is an ordered list of superstates (namely, $S_0, S_1, ..., S_n$). For each superstate $S_i \in \mathcal{D}$, $cAF(S_i) \in \mathcal{A}$. $\overrightarrow{\mathcal{D}}$ is a set of dependences each connecting either from a DC in S_i to a cAF in $cAF(S_i)$ or from a cAF in $cAF(S_i)$ to a DC in S_{i+1}, where $i \in [0, n)$. The dependences connecting to (or from) a cAF indicate the corresponding input (or output) DCs of the cAF. Note that n will be used in the remainder of this paper to indicate the index of the final superstate in ADD graphs. In the example ADD graph illustrated Fig. 2, $n = 4$, $\mathcal{D} = \{S_0, S_1, ..., S_4\}$, $\mathcal{A} = \{cAF(S_0), cAF(S_1), ..., cAF(S_3)\}$. All dependences in $\overrightarrow{\mathcal{D}}$ are represented as directed edges in Fig. 2. An ADD graph has the following properties:

(a) For any $i, j \in [0, n) \wedge i \neq j$, $cAF(S_i) \cap cAF(S_j) = \emptyset$. That is, all $cAF(S)$ are disjoint. This is obvious because ***(i)*** if an AF is a cAF of S_i, it cannot be a cAF of any later superstate S_j, where $i < j \leq n$, due to the fact that applying the AF can not produce any new DC, and ***(ii)*** the AF also can not be a cAF of any earlier superstate S_k, where $0 \leq k < i$, because it is a cAF of S_i. Therefore, the AF can not be a cAF of any other superstate, that is, all $cAF(S)$ are disjoint.

(b) For any $i, j \in [0, n] \wedge i < j$, $S_i \subsetneq S_j$. That is, each superstate contains at least one more DC than any previous superstate. This is illustrated by Eq. (2)

(c) For any AF in $cAF(S_i)$, $AF.I \cap cDC(S_i) \neq \emptyset$ and $AF.O \cap cDC(S_{i+1}) \neq \emptyset$. That is, any cAF in $cAF(S_i)$ must consume (or produce) at least one cDC in S_i (or S_{i+1}). This is obvious based on the definition of cAF.

Dependence: In an ADD graph, a node N can be either a DC or a cAF. Node N_j depends on node N_i, denoted by $N_i \ \delta \ N_j$, if there exists a path from N_i to N_j in the ADD graph. For example, in Fig. 2, $D_1 \ \delta \ D_3$, $AF_0 \ \delta \ D_7$, $AF_5 \ \delta \ AF_7$, and $D_3 \ \delta \ AF_{10}$. Obviously, D_8 does not depend on D_4. In addition, we also say that a node N depends on itself, that is, $N \ \delta \ N$. It is obvious that the dependence relationship is transitive. For reasons of simplicity, $N_i \ \delta \ s_{goal}$ will be used in the reminder of this paper to indicate that $\exists D \in s_{goal}, N_i \ \delta \ D$, i.e., s_{goal} depends on N_i.

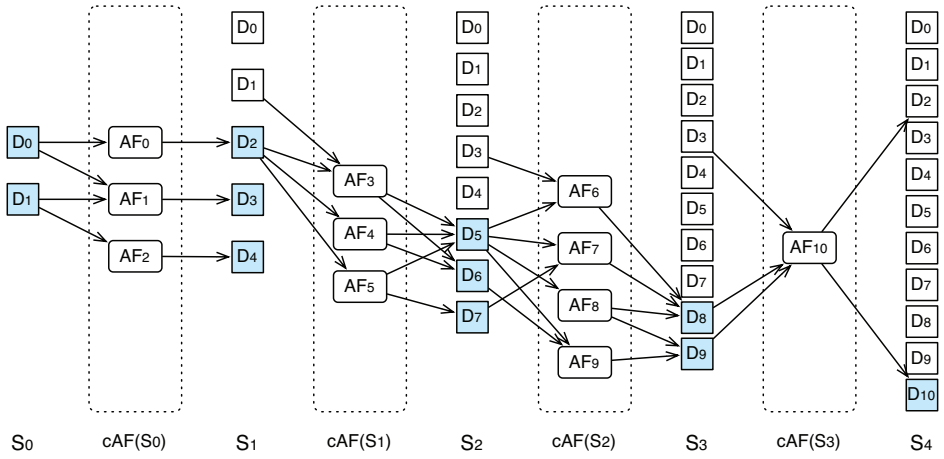


Figure 2: An AF Data Dependence (ADD) Graph

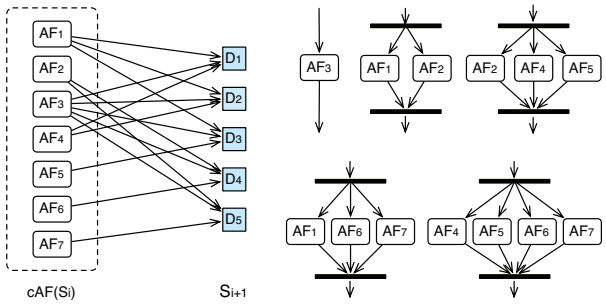


Figure 3: Alternative AF Combinations

Necessary cAF (ncAF): A cAF AF_i in an ADD graph is said to be necessary if and only if s_{goal} depends on this cAF, i.e., $AF_i \ \delta \ s_{goal}$. Let us denote all ncAFs in $cAF(S)$ by $ncAF(S)$. Then, $ncAF(S) = \{AF | AF \in cAF(S) \wedge AF \ \delta \ s_{goal}\}$. For example, in Fig. 2, $ncAF(S_2) = \{AF_6, AF_7, AF_8, AF_9\}$. Because AF_2 is not necessary, $ncAF(S_0) = \{AF_0, AF_1\}$.

Necessary cDC (ncDC): Similarly, a cDC D_i in an ADD graph is said to be necessary if and only if $D_i \ \delta \ s_{goal}$. The set of all ncDCs in $cDC(S)$ is $ncDC(S) = \{D | D \in cDC(S) \wedge D \ \delta \ s_{goal}\}$. For example, $ncDC(S_3) = \{D_8, D_9\}$. $ncDC(S_0) = \{D_2, D_3\}$ because D_4 is not necessary.

Alternative AF Combination: According to the definition of ncAF and the definition of ncDC, it is all ncAFs in $ncAF(S_i)$ that produce all ncDCs in $ncDC(S_{i+1})$ when the transition changes from the superstate S_i to S_{i+1}. In some cases, instead of all ncAFs in $ncAF(S_i)$, a combination of those ncAFs are enough to produce those ncDCs. And there may also exist multiple alternatives of such AF combinations. Let us denote the set of all alternative AF combinations in $ncAF(S)$ by $altAF(S)$. Although the order of the AF combinations is not important in our algorithm, we denote the k-th AF combination by $altAF(S)_k$ for convenience. For example, in Fig. 2, $altAF(S_3)_1 = \{AF_6, AF_9\}$, $altAF(S_3)_2 = \{AF_7, AF_9\}$, or $altAF(S_3)_3 = \{AF_8\}$ can produce $\{D_8, D_9\}$. Although other AF combinations such as $\{AF_8, AF_9\}$ can also produce $\{D_8, D_9\}$, we consider this AF combination redundant and will ignore it in our algorithm because all ncDCs which can be produced by AF_9 can also be produced by AF_8. Similarly, $\{AF_3, AF_5\}$ and $\{AF_4, AF_5\}$ are two alternative AF combinations in $cAF(S_2)$ each of which can produce $\{D_5, D_6, D_7\}$.

The calculation of alternative AF combinations is a nontrivial task which is exemplified by part of an ADD graph as shown in Fig. 3(a). In this case, there are in total five alternative AF combinations which can produce $\{D_1, D_2, D_3, D_4, D_5\}$, as illustrated in Fig. 3(b), where ForkNode and JoinNode (each represented as a thick line segment) are UML Activity Diagram notations and used to indicate parallel execution.

Simple ADD Graph: A simple ADD graph is an ADD graph where all AFs are ncAFs, all DCs are ncDCs, and each $cAF(S_i)$ contains only a single AF combination, i.e., $|altAF(S_i)| = 1$ for all $i \in [0, n)$. The ADD graph (Fig. 6) discussed in Section 5.5 is an example simple ADD graph. Note that the DCs in some superstates (the ones without blue or gray background) are actually cDCs in previous superstates and they are necessary. Comparing Fig. 6 with Fig. 2, AF_2 and D_4 are eliminated because they are not necessary. Only AF_3 is kept in $cAF(S_1)$ because $\{AF_3\}$ is one of the alternative AF combinations in $cAF(S_1)$. Similar is done for AF_8. D_7 is eliminated because AF_5 is not included. As we will see later, a simple ADD graph contains only a single workflow.

5. DESCRIPTION OF THE ALGORITHM

In order to solve a STRIPS-like problem, AI planners typically use techniques such as progression, regression, and partial-ordering. Our algorithm for automatic Grid workflow composition employs both progression to create an ADD graph, and regression to extract workflows, including alternative workflows if available. Our algorithm also optimizes extracted workflows in order to deal with the heterogeneous nature of the Grid and workflow activities. This constitutes three phases of our algorithm: ADD graph creation, workflow extraction and workflow optimization. The algorithm of each phase is explained in the following subsections. In the remainder of this paper, unless explicitly stated, *our algorithm* refers to all three phases as mentioned above.

5.1 ADD Graph Creation

The algorithm of ADD graph creation is illustrated by Algorithm 1. The ADD graph is built starting from the initial superstate S_0 (line 1). Before expanding the ADD graph to the next superstate S_{i+1}, the algorithm checks whether the current superstate S_i entails s_{goal} (line 3). If so, the creation of the ADD graph is finished and the algorithm returns with the created ADD graph (line 12). Otherwise, the algorithm searches AFs in $\mathcal{AF}$ which are not cAF of all previous superstates and calculates $cAF(S_i)$ (line 4). If $cAF(S_i) == \emptyset$, the algorithm returns *null* (line 6), i.e.,*no solution found*. If $cAF(S_i) \neq \emptyset$, the algorithm expands the ADD graph to the next superstate S_{i+1} (line 8) and evaluates the superstate S_{i+1} again as illustrated above.

Algorithm 1: ADD Graph Creation

Input : initial state s_{init}; goal state s_{goal}; set of AFs $\mathcal{AF}$
Output: An ADD graph if a solution exists, or null otherwise

1 $S_0 := s_{init}$ *// initialize superstate S_0*
2 $i := 0$; *// index of current superstate*
3 **while** $S_i \not\models s_{goal}$ **do**
4 $cAF(S_i) := \{AF | AF \in (\mathcal{AF} - \bigcup_{j\in[0,i)} cAF(S_j)) \wedge (S_i \vDash AF.I) \wedge (AF.O - S_i) \neq \emptyset\}$ *// calculate $cAF(S_i)$*
5 **if** $cAF(S_i) == \emptyset$ **then**
6 **return** null *// no solution found*
7 **else**
8 calculate S_{i+1} based on Eq. (2) *// next superstate*
9 $i := i + 1$ *// index of next superstate*
10 **end**
11 **end**
12 **return** the ADD graph based on the created superstates, the calculated cAF(S) and the dependences among them.

When the ADD graph is expanded to S_1, all AFs in $\mathcal{AF}$ have been accessed once, thereby $\bigcup_{AF\in\mathcal{AF}} AF.O \vDash s_{goal}$ is determined. If it is evaluated to be *false* (i.e., some DCs in s_{goal} cannot be produced by any AF in $\mathcal{AF}$), our algorithm returns *null* immediately. For reasons of simplicity, this part is not shown in Algorithm 1.

Fig. 2 is an ADD graph where the initial state $s_{init} = \{D_0, D_1\}$, the goal state $s_{goal} = \{D_9, D_{10}\}$, and the set of all AFs $\mathcal{AF} = \{AF_0, AF_1, ..., AF_{10}, ...\}$. The AFs which are not cAFs of any superstate are not included in the ADD graph and thereby not shown. The final superstate $S_4 = \{D_0, D_1, ..., D_{10}\}$ satisfies $S_4 \vDash s_{goal}$.

5.2 Workflow Extraction

The algorithm of workflow extraction is illustrated by Algorithm 2. First, it calculates $ncDC(S_i)$, $ncAF(S_i)$ and $altAF(S_i)$ from S_n to S_1 (line 3–9) by analyzing AF data dependences. The principle is that if a cDC $D_j \in cDC(S_i)$ is a ncDC, then the cAF $AF_k \in cAF(S_{i-1})$ producing D_j is a ncAF, and the DC $D_{j'} \in cDC(S_{i-1})$ inputting to AF_k is a ncDC, and so on. Through this regression, all $ncDC(S)$ and $ncAF(S)$ can be determined. $ncDC(S_0)$ is calculated based on $ncAF(S_0)$ at the outside of the loop (line 9). For example, in Fig. 2, $D_{10}\ \delta\ s_{goal}$, therefore, $AF_{10}\ \delta\ s_{goal}$ and $D_8\ \delta\ s_{goal}$. Since $D_2\ \delta\ s_{goal}$ and $D_3\ \delta\ s_{goal}$ (note that there exists a path D_3-AF_{10}-D_{10}), $ncAF(S_0) = \{AF_0, AF_1\}$. AF_2 is redundant and will be eliminated. $altAF(S_i)$ is calculated when $ncAF(S_i)$ and $ncDC(S_{i+1})$ are determined. This is done by enumerating all possible AF combinations in $ncAF(S_i)$, such that each of the AF combinations is sufficient to produce all DCs in $ncDC(S_{i+1})$, and eliminating the redundant AF combinations (the corresponding algorithm is omitted here due to space limitation).

Once all $ncDC(S)$, $ncAF(S)$, and $altAF(S)$ are determined, the algorithm eliminates non-necessary cDCs and cAFs, and checks whether the size of each $altAF(S_i)$ is 1. If so, the ADD graph is a simple ADD graph, i.e., it contains a single workflow. The workflow can then be optimized in the next phase (line 12). Otherwise, depending on the selection of alternative AF combinations in each $altAF(S)$, multiple ADD graphs can be extracted (line 14). Each extracted ADD graph is then recursively extracted again (line 16).

In the ADD graph shown in Fig. 2, $ncAF(S_1)$ has two alternative AF combinations $\{AF_3, AF_5\}$ and $\{AF_4, AF_5\}$, $ncAF(S_2)$ has three $\{AF_8\}$, $\{AF_6, AF_9\}$ and $\{AF_7, AF_9\}$, $ncAF(S_0)$ and $ncAF(S_3)$ each have one. Six ADD graphs are extracted in the first invocation of Algorithm 2. When $\{AF_3, AF_5\}$ and $\{AF_8\}$ are selected, there exist again two alternative AF combinations in $ncAF(S_1)$, i.e., $\{AF_3\}$ and $\{AF_5\}$. It is similar when $\{AF_4, AF_5\}$

Algorithm 2: Workflow Extraction: extractWorkflows()

Input : ADD graph γ; goal state s_{goal};
Output: set of extracted simple ADD graphs Γ

1 $\Gamma := \emptyset$ *// initialize Γ*
2 $R := s_{goal}$ *// set of DCs to be produced by previous superstates*
3 **for** $S_i := S_n$ **to** S_1 **do** *// regression*
4 $ncDC(S_i) := R \cap cDC(S_i)$
5 $ncAF(S_{i-1}) := \{AF \mid AF \in cAF(S_{i-1}) \wedge AF.O \cap ncDC(S_i) \neq \emptyset\}$
6 calculate $altAF(S_{i-1})$ *// alternative AF Combinations*
7 $R := (R \cup (\bigcup_{AF\in ncAF(S_{i-1})} AF.I)) - ncDC(S_i)$
8 **end**
9 $ncDC(S_0) := \bigcup_{AF\in ncAF(S_0)} AF.I$
10 eliminate non-necessary cAFs and cDCs from γ
11 **if** $\forall i \in [0, n) : |altAF(S_i)| == 1$ **then**
12 $\Gamma := \Gamma \cup \gamma$ *// is simple ADD graph*
13 **else**
14 $E := \prod_{i\in[0,n)} altAF(S_i)$ *// cartesian product for extraction*
15 **forall** $e \in E$ **do** *// for each extracted graph*
16 $\Gamma := \Gamma \cup$ extractWorkflows(e, s_{goal}) *// recursion*
17 **end**
18 **end**
19 **return** Γ

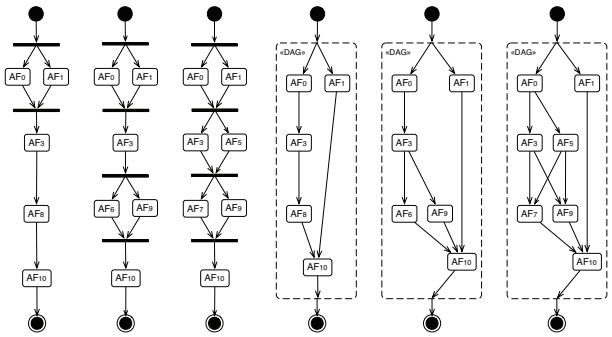


Figure 4: Automatically Composed Grid Workflows

and $\{AF_8\}$ are selected. Because two of the extracted ADD graphs are identical, in total, we obtain seven alternative workflows when Algorithm 2 returns. For all of these workflows, the AFs in the same $ncAF(S_i)$ can be executed in parallel (via the AGWL construct `parallel`), and all AFs in the $ncAF(S_i)$ are executed before the AFs in $ncAF(S_{i+1})$ (via the AGWL construct `sequence`). Fig. 4(a), (b) and (c) show three generated workflows, represented with UML Activity Diagram [30]. We can obtain the other four workflows by substituting AF_3 with AF_4 in these three workflows and AF_3 with AF_5 in the first workflow (Fig. 4(a)).

The extracted workflows provide alternative execution paths in case of failure at runtime. For example, if AF_8 in Fig. 4(a) cannot be mapped to any AD or the execution of AF_8 fails, instead of returning an error to users, the workflow execution can continue with AF_6 and AF_9 (Fig. 4(b)). If AF_6 fails again, AF_7 can be used as a replacement (Fig. 4(c)).

The selection of these alternative workflows for execution is based on two factors: ***(i)*** user preferences: users may specify which AFs they want (or do not want) to be included in the extracted workflows (e.g., based on their experiences with previous workflow executions), and ***(ii)*** scheduling decisions: the workflow scheduler may try to schedule (without actual execution) all of these alternative workflows and select the best one (e.g., with minimum execution time or minimum cost) for the actual execution.

Algorithm 3: Workflow Optimization

Input : simple ADD graph
Output: DAG of AFs

```
1  mark all AFs in ncAF(S_0) as the root nodes of the DAG
2  for ncAF(S_i) := ncAF(S_1) to ncAF(S_{n-1}) do
3      forall AF' ∈ ncAF(S_i) do
4          U := AF'.I                          // unsolved inputs
5          k := i                              // index of current superstate
6          L := ∅                              // located predecessors
7          repeat
8              T := ∅                          // temporarily unsolved inputs
9              forall D ∈ U do
10                 if D ∈ ncDC(S_k) then
11                     L := L ∪ {AF | AF ∈ ncAF(S_{k-1}) ∧
                       D ∈ AF.O}               // mark as predecessors
12                 else
13                     put D into T            // mark as unsolved
14                 end
15             end
16             forall AF'' ∈ L do
17                 T := T − AF''.I             // ignore inputs of
                   predecessors
18             end
19             k := k − 1                      // move to previous ncDC(S)
20             U := T                          // update unsolved inputs
21         until U == ∅                        // until all inputs are solved
22         mark all AFs in L as the predecessors of AF'
23     end
24 end
25 return the DAG
```

5.3 Workflow Optimization

The workflows extracted in the previous phase may still be optimized in terms of control flow. For example, in Fig. 2, AF_3 can be executed as long as AF_0 is finished, even if AF_1 is not finished at that time, because AF_3 does not depend on AF_1. Using only `parallel` and `sequence` constructs prevents such execution from happening, thereby may increase the workflow execution time due to unnecessary waiting.

In order to eliminate such delays in execution time, we introduce a workflow optimization phase, which, instead of using `sequence` or `parallel` constructs, composes AFs into DAGs, as illustrated by Algorithm 3. In this algorithm, two nested loops (line 2 and 3) are used to locate predecessors of each AF in each $ncAF(S_i)$. This is done by solving (i.e., finding the producers of) input DCs of each AF. For each AF, Algorithm 3 initializes unsolved input DCs U with $AF.I$ (line 4), then tries to solve them by checking against previous $ncDC(S_k)$ (line 9–15). If an unsolved input DC is contained in $ncDC(S_k)$, we mark all the AFs in $ncAF(S_{k-1})$ which produce this DC as predecessors of the AF (line 11). Otherwise, the input DC is marked as unsolved temporarily (line 13). When all unsolved input DCs in U are checked against $ncDC(S_k)$, U is updated through T by eliminating input DCs of the located predecessors (line 16 –18, 20). This is because these input DCs have been solved in previous iterations and they are input to the predecessors of the AF. This process continues until all input DCs are solved (line 21).

Fig. 4(d), (e) and (f) show the optimized workflows of Fig. 4(a), (b) and (c), respectively. The optimization can be significant when the execution time of AF_1 is much longer than that of AF_0. Let us denote the execution time of AF_i by $t(AF_i)$. Fig. 5 compares the execution of the workflow shown in Fig. 4(a) and that of its optimized version in Fig. 4(d), where $t(AF_1) > t(AF_0) + t(AF_3) + t(AF_8)$. The processors P_1 and P_2 may be located on two different Grid resources. For reasons of simplicity, the communication

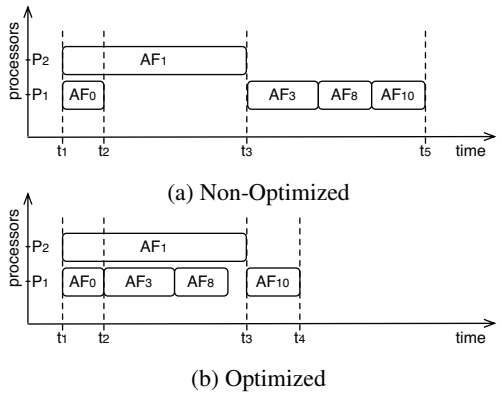


Figure 5: Improvement of Workflow Optimization

time between two processors and the overhead of the Grid middleware are ignored. In Fig. 5, compared with the non-optimized execution, AF_3 and AF_8 are executed in parallel with AF_1 in the optimized execution. This results in a total execution time of $t_4 - t_1 = t(AF_1) + t(AF_{10})$, which is less than the execution time of the non-optimized workflow $t_5 - t_1 = t(AF_1) + t(AF_3) + t(AF_8) + t(AF_{10})$. The actual performance improvement achieved by employing workflow optimization on a real world Grid workflow application is discussed in Section 6.

The Grid workflows composed by our algorithm through the three phases achieve high quality because of ***(i)*** Portability: the workflows are abstract and represented at the semantic level; ***(ii)*** Fault tolerance: the workflows may include alternative workflows thereby provide alternative execution paths in case some activities fail; and ***(iii)*** Performance: the workflows are optimized for execution time.

5.4 Algorithm Analysis

The execution time of our algorithm is caused mostly by the phase of ADD graph creation because this phase involves matching inputs of AFs with superstates through reasoning. This fact can also be proved by the experimental results shown in Section 6. Therefore, the complexity analysis of our algorithm focuses only on the phase of ADD graph creation.

For convenience, let us denote the set of all possible workflows by W and denote the number of AFs by x. As we demonstrate in the following propositions, the worst case execution time taken by our algorithm is a quadratic in x.

Proposition 1. *Given a set of x AFs, an initial state s_{init} and a goal state s_{goal}, the time taken by our algorithm to find an element of W is a quadratic in x if $W \neq \emptyset$. If such an element is not found by our algorithm, then necessarily $W = \emptyset$.*

PROOF. According to Section 5.1, the *basic operation* in the phase of ADD graph creation is to check whether an AF is a cAF for a given superstate. The number of the basic operation is determined by the number of AFs and the number of the superstates that are in the ADD graph. Note that the set of cAFs of each superstate are disjoint. Given x AFs, the maximum number of superstates in the ADD graph is $x + 1$, in which case, all sets of cAFs of each superstate have the minimum number of cAFs, i.e., 1. In other words, $\forall i \in [0, n) : |cAF(S_i)| = 1$. Here we assume that there are enough DCs defined in AGWO and associated with AFs such that the number of newly produced DCs in each superstate is greater than 0, i.e., $cDC(S_i) \neq \emptyset$. Note that if there are fewer DCs, the maximum number of superstates is less than $x+1$; if there are more DCs, the maximum number of superstates is not affected. Therefore, given x AFs, the maximum number of superstates is $x + 1$. To expand the ADD graph to the superstate S_1, x basic operations

ncAF(S)	Input	Output
$ncAF(S_0)$	$\{D_0, D_1\}$	$\{D_2, D_3\}$
$ncAF(S_1)$	$\{D_1, D_2\}$	$\{D_5, D_6\}$
$ncAF(S_2)$	$\{D_5\}$	$\{D_8, D_9\}$
$ncAF(S_3)$	$\{D_3, D_8, D_9\}$	$\{D_2, D_{10}\}$

Table 1: ncAF(S) and Their Input and Output DCs

are required because there are x AFs to be compared. Because $|cDC(S_i)| = 1$, expanding to S_2 requires $x-1$ basic operations, expanding to S_3 requires $x-2$ basic operations, and so on. The maximum total number of the basic operations is:

$$x + (x-1) + (x-2) + ... + 1 = \frac{x^2 + x}{2}$$

Note that if $\bigcup_{AF \in \mathcal{AF}} AF.O \not\models s_{goal}$ (in which case $W == \emptyset$), our algorithm can return *no solution found* after the first x basic operations. □

Proposition 2. *If an element of W is found by our algorithm, the number of the superstates of the ADD graph is minimum, which also means that the length[1] of the DAG workflow is minimum.*

PROOF. Let us assume the algorithm finds a solution when expanding the ADD graph to S_n. This means that $\forall i \in [0, n), S_i \not\models s_{goal}$. Therefore, n is minimum. □

5.5 Branches and Loops

With the help of AGWL constraints [11], our algorithm can be extended to compose Grid workflows with loops and branches. The example constraints related to workflow composition are to access elements of a data collection in parallel, to produce a DC when another DC is produced, etc.

Branches: Users can specify constrains for the goal state to obtain workflows with branches. For example, the goal state $s_{goal} = \{D_9, D_{10}$(*agwl:precondition="D_6==true"*)$\}$ means that the composed Grid workflow must produce D_9 and D_{10}, and D_{10} must be produced when D_6=*true* (assuming D_6 is a *boolean*). In this case, we first compose a workflow by invoking our algorithm with $s_{goal1} = \{D_6, D_9\}$ as the goal state. The result is an ADD graph with $S_n \models s_{goal1}$. Then we compose another workflow with $s_{init} = S_n$ and $s_{goal2} = \{D_{10}\}$ as the goal state. Then we connect these two workflows through a *DecisionNode* which has an outgoing edge with the guard $[D_6$=*true*$]$ connecting to the second workflow. The $else$ branch is empty in this case. Two branches are merged at the end of the second workflow.

Parallel Loops: In the initial state, users can also specify a data collection and how each data element is allowed to be accessed. For example, $s_{init} = \{ D_1$ (*agwl:cardinality=multiple, agwl:access-order=parallel*$), D_2\}$ means that the user provided data is D_2 and a collection of D_1, and all D_1 can be accessed in parallel. In this case, we can compose a workflow by invoking our algorithm with $s_{init} = \{D_1, D_2\}$ as the initial state. Then we can invoke the composed workflow in the loop body of a `parallelForEach` which iterates over the data collection of D_1. Automatic composition of workflows with `parallelFor` is also possible if users specify the initial state, for example, $s_{init} = \{D_1$(*agwl:value-range=1:10:3, agwl:access-order=parallel*$), D_2\}$, where *1:10:3* indicates the loop counter, i.e., from 1 to 10 step by 3.

Sequential Loops: Users may specify a goal state $\{D_9, D_{10}$ (*agwl:postcondition="$D_{10} < 0.1$"* $\}$ which means when D_{10} is produced, its value must be less than 0.1 (e.g., a threshold value). A workflow with a sequential loop is required in this case. To compose a workflow with sequential loops, our algorithm first composes a workflow with $s_{goal} = \{D_9, D_{10}\}$. Then it checks the possibility of sequential loops. Let us assume the workflow illustrated in Fig. 4(d) is considered for sequential loop generation. The corresponding simple ADD graph is shown in Fig. 6. Based on the ADD graph, we can obtain the input and output DCs of $ncAF(S)$ as Table 1. Our algorithm checks output DCs starting from $ncAF(S_3)$ (because it produces D_{10}) to see which DCs can be input to any previous $ncAF(S_i)(i < 3)$ and thereby produces new D_{10}. It is obvious that D_3 produced by AF_{10} can be input to AF_4 (the red dash line), which consequently updates D_{10} (the blue solid line). Based on this, a workflow with a `doWhile` loop can be generated, and AF_4, AF_8 and AF_{10} are sequentially executed in the loop body. In case of multiple possibilities of sequential loops found, user interactions are required. The detailed algorithm for generation of sequential loops is illustrated by Algorithm 4. Line 3 – 9 find in which superstate the input DC D is first produced. Then, the algorithm tries to find DCs $\in S_k$ which are not ncDCs and can be input to a ncAF of a previous superstate and thereby further update D (line 11 – 17). This process is done for all previous superstates except S_0 (line 10 – 19).

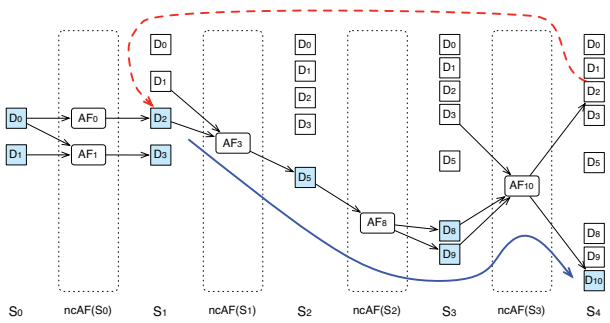


Figure 6: An ADD Graph for Generation of Sequential Loops

6. EXPERIMENTAL RESULTS

Our algorithm is implemented in Java as part of the ASKALON Grid application development and computing environment [10]. The reasoning part is implemented using the Jena APIs [20]. We evaluate our approach through three series of experiments: (i) the composition of a Grid workflow in a simulated domain, to test how our algorithm behaves in the case where thousands of AFs are available; (ii) the composition of a real world Grid workflow application in the meteorology domain to illustrate the execution time of our algorithm for a real world case; and (iii) the comparison of the execution time of optimized and non-optimized real world Grid workflows. The first two series of experiments are conducted on a normal desktop computer with 2 GB memory and one 2.4 GHz Intel Core 2 Duo CPU. The Java runtime environment used is JRE 1.5.0_16. In the third experiment, we execute a real world Grid workflow application on the Austrian Grid [1] through ASKALON [10]. A subset of the computational resources which have been used for the third experiment is summarized in Table 2.

In the first experiment, we developed an ontology which contains thousands of AFs (namely, $AF_0, AF_1, ...$) and thousands of DCs (namely, $D_0, D_1, ...$). In order to measure the worst case execution time of our algorithm, as described in Section 5.4, these AFs are defined in the following way: AF_i accepts D_i and a random number (between 0 and 10) of DCs from $\{D_0, ..., D_{i-1}\}$ as input, and produces D_{i+1} and a random number (between 0 and 10) of DCs in $\{D_0, ..., D_{i-1}\}$ as output. Thus, all AFs can be pipelined. The reason that we use a random number (between 0 and 10) of DCs as input and output of AFs is based on our experiences in multiple scientific domains such as material science, astrophysics and meteorology where the number of the input and output ports of workflow activities varies from 0 to around 10. This observation actually makes the composition of Grid workflows more difficult if a classic state space searching based AI planning algorithm is

[1]The length of a finite DAG is the number of edges of the longest directed path.

Algorithm 4: Sequential Loop Generation

Input : simple ADD graph
data class $D \in s_{goal}$ associated with a postcondition
Output: A workflow with a sequential loop if success, or a message otherwise.

build the $ncAF(S)$ input and output table, as Table 1
$B := \emptyset$ *// set of triples each indicating a possible sequential loop*
$k := 0$ *// index of a superstate where D is first produced*
for $ncDC(S_i) := ncDC(S_n)$ **to** $ncDC(S_0)$ **do**
  if $D \in ncDC(S_i)$ **then**
    $k := i$;
    break;
  end
end
repeat
  forall $D' \in \left(\bigcup_{AF \in ncAF(S_{k-1})} AF.O - ncDC(S_k)\right)$ **do**
    for $ncAF(S_j) := ncAF(S_{k-1})$ **to** $ncAF(S_0)$ **do**
      if $\exists AF' \in ncAF(S_j) \wedge D' \in AF'.I \wedge AF'\ \delta\ D$ **then**
        // D' in S_k can be input to AF' at $ncAF(S_j)$
        put a triple $\langle D', k, j \rangle$ into B *// found a seq. loop*
      end
    end
  end
  $k := k - 1$
until $k == 0$
if $B == \emptyset$ **then**
  return sequential loop is impossible
else if $|B| == 1$ **then**
  return create a workflow with a `doWhile` loop based on B
else if $|B| > 1$ **then**
  return user interaction is required
end

Grid Site	*CPU*	*#*	*GHz*	*LRM*	*Location*
karwendel	Dual Core AMD Opteron	8	2.4	SGE	Innsbruck
altix1	Itanium 2	8	1.4	PBS	Innsbruck
schafberg	Itanium 2	8	1.4	PBS	Salzburg
altix1jku	Itanium 2	8	1.4	PBS	Linz
c703-pc1801	Pentium 4	8	2.8	Torque	Innsbruck
c703-pc2601	Pentium 4	8	2.8	Torque	Innsbruck

Table 2: The Austrian Grid Testbed

used, due to the huge number of states. In this experiment, AFs are defined as below:

$$AF_0 : (D_0) \rightarrow (D_1)$$
$$AF_1 : (D_0, D_1) \rightarrow (D_2)$$
$$AF_2 : (D_2) \rightarrow (D_1, D_3)$$
$$AF_3 : (D_0, D_1, D_3) \rightarrow (D_4)$$
$$\ldots$$

Then we compose Grid workflows using these AFs. Obviously, we obtain a workflow consisting of a sequence of AFs. Although the structure of the composed workflow is simple, the number of the basic operations invoked in our algorithm is maximized. By specifying a suitable initial state, e.g., $s_{init} = \{D_0\}$, and a goal state, e.g., $s_{goal} = \{D_{1000}\}$, we can obtain the worst case execution time of our algorithm.

Fig. 7 illustrates the experimental results of the execution time of our algorithm with two curves: the total execution time, the execution time of the ADD graph creation phase. The x-axis is the number of AFs. We can observe that the execution time of ADD graph creation is very close to the total execution time, i.e., most of the execution time of our algorithm is spent in the phase of ADD graph creation. The trend analysis shows that the worst

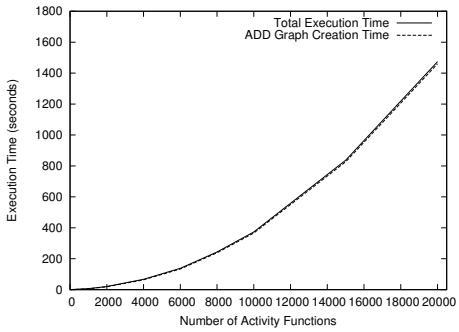


Figure 7: Execution Time of Our Algorithm in the Worst Case

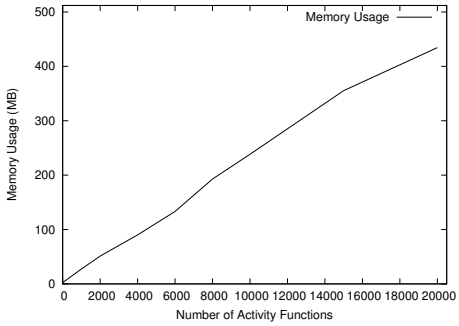
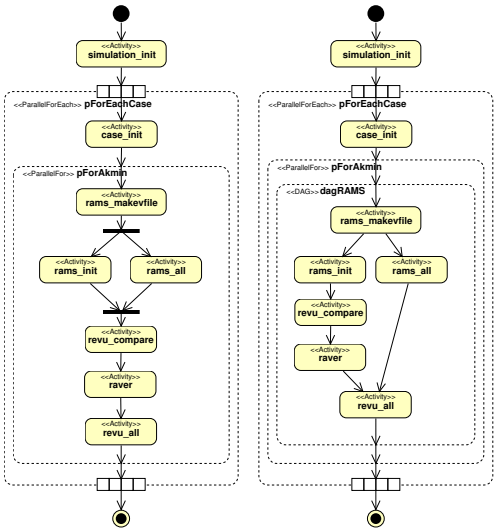


Figure 8: Memory Usage of Our Algorithm in the Worst Case

case total execution time of our algorithm to compose a Grid workflow is a quadratic in the number of AFs. If 1000 AFs are defined in ontologies in a certain domain, our algorithm can compose a Grid workflow (or return *no solution found*) in 7.04 seconds in the worst case. It is 20.70 seconds in case of 2000 AFs. We also measured the *heap* memory usage in Java virtual machine (JVM) when composing these Grid workflows with our algorithm. The results are illustrated in Fig. 8, where the values are calculated based on the measurements of *java.lang.management.MemoryMXBean*, the management interface for the memory system of JVM provided since Java 1.5. Because of the automatic garbage collection mechanism of JVM, we consider the *heap* memory usage illustrated here as an estimation of the amount of the memory that are allocated for the objects of our workflow composition program, including the objects for storing the ontologies and the objects required by our algorithm itself. According to what we observed, roughly 10% of the *heap* memory is used by the latter. We can conclude that our algorithm uses a reasonable amount of memory. Note that the actual memory usage of our workflow composition program as provided by the operating system is usually higher than those measured here.

In the second experiment, we compose the real world Grid workflow application MeteoAG using our algorithm. MeteoAG [28] is a meteorology simulation application based on the numerical model RAMS [7]. The simulations produce precipitation fields of heavy precipitation cases over the western part of Austria at a spatial and temporal grid in order to resolve most alpine watersheds and thunderstorms. The ontology, provided by the Institute of Meteorology and Geophysics, University of Innsbruck, consists of 19 AFs. We composed the MeteoAG workflow with and without the workflow optimization phase enabled respectively and obtained two versions of the workflow as illustrated in Fig. 9. Note that the workflow consists of two nested levels of parallel loops. The loop body of the inner loop (corresponding to a *simulation case*) is a DAG, in case of the optimized version, or a sequence of activities and a parallel section, in case of the non-optimized version. The total execution time for the composition of the two workflows is 0.64 seconds for the optimized version and 0.54 for the non-optimized version. We conclude that our algorithm is fast enough in this case.



(a) Non-Optimized (b) Optimized

Figure 9: Two Versions of the MeteoAG Workflow

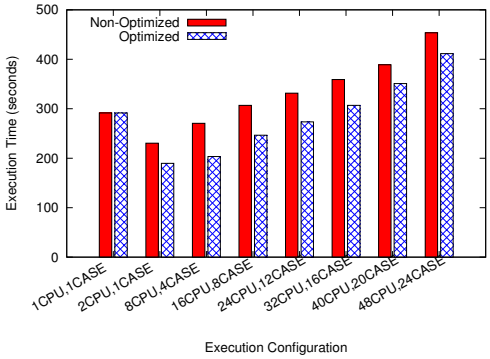


Figure 10: Execution Time of Both MeteoAG Workflows

In the third experiment, we executed the optimized and the non-optimized MeteoAG workflows on the Austrian Grid respectively. First, we executed one simulation case using one CPU (denoted by "1CPU,1CASE", similar for the others). Obviously the execution time of both workflows are similar in this case because no parallelism can be employed. Since the maximum number of parallel jobs in one simulation case is two, we then increased the number of simulation cases and the number of CPUs such that the number of CPUs is twice the number of simulation cases. Note that if the number of CPUs is more than twice the number of simulation cases, the additional CPUs can not be used. If the number of CPUs is less than twice of the number of simulation cases, the execution time of both MeteoAG workflows are similar because the execution of all simulation cases are interleaved and the differences in control flow dependences in single simulation cases become insignificant. Fig. 10 compares the execution time of both MeteoAG workflows in our experiments. We can see that the execution time of the optimized MeteoAG workflow is less than that of the non-optimized one. Specifically, when we execute 4 simulation cases on 8 CPUs on *karwendel*, 25% execution time is reduced (270.64s vs. 203.37s). The percentages of reduced execution time in other cases vary from 9% to 20%. The corresponding speedup is improved by up to 2.24, as illustrated in Fig. 11.

7. RELATED WORK

Lautenbacher et al. [21] presented a survey of available workflow composition approaches in areas like Grid workflows, Semantic Web Services, and Business Process Management. The authors mainly focus on the question what features are supported by the existing approaches, e.g., whether semantic technologies are included, or whether the approach works at an abstraction level. Little information about algorithms of automatic workflow composition is given.

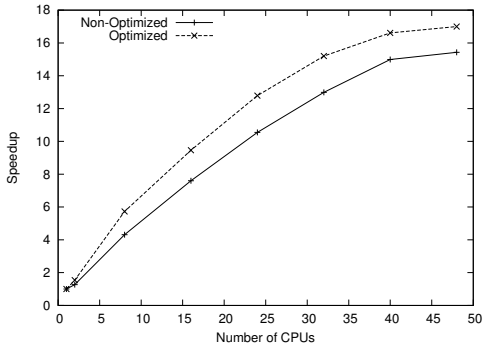


Figure 11: Speedup of Both MeteoAG Workflows

Gil [14] mentioned some ideas to use semantic technologies (e.g., application specific ontologies) and AI planning algorithms to support assisted workflow composition. The author pointed out several cases where automatic completion of workflows is desirable, such as creating an executable workflow from a workflow instance by assigning tasks to resources, completing under-specified workflow templates by mapping abstract tasks to specific components or adding data conversion steps. However, no specific algorithms were discussed. Wings[15] is a workflow creation system that combines semantic representations with planning techniques. While Wings can generate workflows of computations for given data collections, it can only generate workflow instances based on workflow templates. The creation of workflow templates is still done manually. Furthermore, the creation of workflows is limited to DAGs. Gubała et al. [18] presented the Workflow Composition Tool (WCT) for automatic Grid workflow composition. The main idea of their work is to iteratively solve unsatisfied data dependences by contacting service registries until some dependences cannot be satisfied. The algorithm is limited to Petri Nets and cannot handle alternative control flows like the case presented in Fig. 3. In addition, our algorithm works at a higher level of abstraction than the WCT approach. A planning approach presented in [3] automatically constructs data processing workflows where the inputs and outputs of services are relational descriptions. Their planner uses relational subsumption to connect the output of a service with the input of another. Their approach focuses on cases where inputs and outputs of services are relational descriptions.

In the area of Semantic Web Services, some AI planning based approaches for automatic web service composition are proposed. Wu et al. [31] adapted the graphplan algorithm [6] with semantics and focused on addressing both process heterogeneity and data heterogeneity of web services composition problems. Meyer et al. [25] uses an extension of the Enforced Hill-Climbing planning algorithm [19] for automatic service composition. Although the idea of graph based planning is similar to our approach, there are several differences: (i) our approach can generate alternative workflows, and (ii) we consider workflow optimization. In addition, our algorithm focuses on Grid workflow composition and models workflow activities with input and output data classes and the consumption of data classes does not make them unavailable. Therefore, mutual exclusion links, as used in the graphplan algorithm, are not needed in our approach. Ambite et al. [4] models the web service composition problem as a Triple logic program and uses Triple logic engine to generate workflows. Duan et al. [9] sketched an algorithm to synthesis BPWL4WS abstract processes. They assume that tasks

are associated with ranks which, however, is not feasible for Web or Grid services which are developed by different organizations.

In addition, Lelarge et al. [22] presented an AI planning based approach for automatic composition of secure workflows in the domain of stream processing systems. No Semantic Web technologies such as Ontology are involved.

8. CONCLUSIONS AND FUTURE WORK

In this paper, we formalized the Grid workflow composition problem based on the STRIPS language and presented a novel graph based algorithm for automatic composition of high quality (portable, fault tolerant and optimized) Grid workflows. Our algorithm composes semantic described workflow activities, i.e., Activity Functions (AFs), into Grid workflows by employing progression to create an AF Data Dependence (ADD) graph and regression to extract workflows, including the alternative ones if available. Our algorithm also optimizes extracted workflows in order to deal with the heterogeneous nature of the Grid and workflow activities. The time complexity of our algorithm is a quadratic in the number of AFs. With the help of AGWL constraints, we also extended our algorithm for composition of Grid workflows with branches, parallel and sequential loops. The composition of Grid workflows with our algorithm in domains with up to 20000 AFs further proves the analyzed time complexity. The composition of a real world meteorology workflow MeteoAG takes around half a second. By applying our workflow optimization techniques, the execution time of the MeteoAG workflow is reduced by up to 25% and the speedup is increased by up to 2.24. We believe that our graph based approach of automatic Grid workflow composition is the most feasible one among all methods introduced so far. Assuming that ontologies for application domains are given, it demonstrates good potential to be used as part of production workflow environments, which is in contrast to most related work that has been largely evaluated against experimental implementations. Our approach has been integrated in the ASKALON Grid application development and computing environment and is used by numerous application groups for their daily work and research with Grid workflows.

The initial state, the specification of the initial available data classes, is considered fully known in our algorithm. We will investigate possible extensions to our algorithm to deal with cases with partially known initial states for further simplification of the Grid workflow composition process.

Acknowledgment

This work is partially funded by the European Union through the IST-034601 edutain@grid project and the Austrian BMBWK (Federal Ministry for Education, Science and Culture) through the GZ BMWF-10.220/0002-II/10/2007 Austrian Grid project.

9. REFERENCES

[1] Austrian Grid. http://www.austriangrid.at.

[2] I. Altintas, C. Berkley, E. Jaeger, M. Jones, B. Ludäscher, and S. Mock. Kepler: An Extensible System for Design and Execution of Scientific Workflows. In *16th Intl. Conf. on Scientific and Statistical Database Management (SSDBM'04)*, Santorini Island, Greece, June 21-23, 2004.

[3] J. L. Ambite and D. Kapoor. Automatically Composing Data Workflows with Relational Descriptions and Shim Services. In *Proceedings of 6th International Semantic Web Conference and 2nd Asian Semantic Web Conference (ISWC 2007 + ASWC 2007)*, Busan, Korea, November 2007. Springer.

[4] J. L. Ambite and M. Weathers. Automatic Composition of Aggregation Workflows for Transportation Modeling. In *Proceedings of the 2005 National Conference on Digital Government Research*, pages 41–49, 2005.

[5] C. Berkley, S. Bowers, M. Jones, B. Ludäscher, M. Schildhauer, and J. Tao. Incorporating Semantics in Scientific Workflow Authoring. In *SSDBM'2005: Proceedings of the 17th international conference on Scientific and statistical database management*, pages 75–78, Berkeley, CA, US, 2005.

[6] A. L. Blum and M. L. Furst. Fast Planning Through Planning Graph Analysis. *Artificial Intelligence*, 90:281–300, 1997.

[7] W. R. Cotton, R. A. Pielke, R. L. Walko, G. E. Liston, C. J. Tremback, H. Jiang, R. L. McAnelly, J. Y. Harrington, M. E. Nicholls, G. G. Carrio, and J. P. McFadden. RAMS 2001: Current status and future directions. *Meteorology and Atmospheric Physics*, 82:5–29, 2003.

[8] E. Deelman, G. Singh, M.-H. Su, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, K. Vahi, G. B. Berriman, J. Good, A. Laity, J. C. Jacob, and D. Katz. Pegasus: a Framework for Mapping Complex Scientific Workflows onto Distributed Systems. *Scientific Programming Journal*, 13(2), November 2005.

[9] Z. Duan, A. Bernstein, P. Lewis, and S. Lu. Semantics Based Verification and Synthesis of BPEL4WS Abstract Processes. In *Proceedings of the IEEE International Conference on Web Services*, Washington, DC, USA, 2004.

[10] T. Fahringer, R. Prodan, R. Duan, J. Hofer, F. Nadeem, F. Nerieri, S. Podlipnig, J. Qin, M. Siddiqui, H.-L. Truong, A. Villazon, and M. Wieczorek. *Workflows for eScience, Scientific Workflows for Grids*, chapter ASKALON: A Development and Grid Computing Environment for Scientific Workflows. Springer Verlag, 2007.

[11] T. Fahringer, J. Qin, and S. Hainzer. Specification of Grid Workflow Applications with AGWL: An Abstract Grid Workflow Language. In *Proceedings of IEEE International Symposium on Cluster Computing and the Grid 2005 (CCGrid 2005)*, Cardiff, UK, May 9-12, 2005.

[12] R. E. Fikes and N. J. Nilsson. STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving. *Artificial Intelligence*, 2:189–208, 1971.

[13] I. Foster, J. Voeckler, M. Wilde, and Y. Zhao. Chimera: A Virtual Data System for Representing, Querying, and Automating Data Derivation. In *14th International Conference on Scientific and Statistical Database Management (SSDBM'02)*, Edinburgh, Scotland, July 2002.

[14] Y. Gil. *Workflows for e-Science – Scientific Workflows for Grids*, chapter Workflow Composition: Semantic Representations for Flexible Automation. Springer Verlag, 2007.

[15] Y. Gil, V. Ratnakar, E. Deelman, G. Mehta, and J. Kim. Wings for Pegasus: Creating Large-Scale Scientific Applications Using Semantic Representations of Computational Workflows. In *Proceedings of the Nineteenth Conference on Innovative Applications of Artificial Intelligence (IAAI-07)*, Vancouver, British Columbia, Canada, July 2007.

[16] T. R. Gruber. A Translation Approach to Portable Ontology Specifications. *Knowl. Acquis.*, 5(2):199–220, 1993.

[17] T. Gubała, D. Herężlak, M. Bubak, and M. Malawski. Constructing Abstract Workflows of Applications with Workflow Composition Tool. In *Proceedings of Cracow Grid Workshop (CGW'06)*, 2006.

[18] T. Gubała, D. Herężlak, M. Bubak, and M. Malawski. Semantic Composition of Scientific Workflows Based on the Petri Nets Formalism. In *Proc. of the 2nd IEEE International Conference on e-Science and Grid Computing*, Amsterdam, The Netherlands., December 2006.

[19] J. Hoffmann and B. Nebel. The FF Planning System: Fast Plan Generation Through Heuristic Search. *Journal of Artificial Intelligence Research*, 14:253–302, 2001.

[20] Jena Team. Jena Semantic Web Framework API. http://jena.sourceforge.net/.

[21] F. Lautenbacher and B. Bauer. A Survey on Workflow Annotation & Composition Approaches. In *Proceedings of the Workshop on Semantic Business Process and Product Lifecycle Management (SemBPM) in the context of the European Semantic Web Conference (ESWC)*, Innsbruck, Austria, 2007.

[22] M. Lelarge, Z. Liu, and A. V. Riabov. Automatic Composition of Secure Workflows. Technical Report W0607-005, IBM Research Division, July 2006.

[23] M. Lemos, M. A. Casanova, L. F. B. Seibel, J. A. F. de Macedo, and A. B. de Miranda. Ontology-Driven Workflow Management for Biosequence Processing Systems. In *Proceedings of 15th International Conference Database and Expert Systems Applications (DEXA 2004)*, Zaragoza, Spain, 2004.

[24] P. Lord, P. Alper, C. Wroe, and C. Goble. *The Semantic Web: Research and Applications*, chapter Feta: A Light-Weight Architecture for User Oriented Semantic Service Discovery, pages 17–31. Springer, 2005.

[25] H. Meyer and M. Weske. Automated Service Composition using Heuristic Search. In *Proceedings of the Fourth International Conference on Business Process Management (BPM 2006)*, Vienna, Austria, 2006.

[26] J. Qin and T. Fahringer. A Novel Domain Oriented Approach for Scientific Grid Workflow Composition. In *SC'08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, Austin, Texas, USA, 2008.

[27] L. Salayandia, P. P. da Silva, A. Q. Gates, and A. Rebellon. A Model-Based Workflow Approach for Scientific Applications. In *Proceedings of the 6th OOPSLA Workshop on Domain-Specific Modeling*, 2006.

[28] F. Schüller, J. Qin, F. Nadeem, R. Prodan, T. Fahringer, and G. Mayr. Performance, Scalability and Quality of the Meteorological Grid Workflow MeteoAG. In *Proceedings of 2nd Austrian Grid Symposium*, Innsbruck, Austria, September 21-23, 2006. OCG Verlag.

[29] M. Siddiqui, A. Villazon, J. Hofer, and T. Fahringer. GLARE: A Grid Activity Registration, Deployment and Provisioning Framework. In *SC'05: Proceedings of the ACM/IEEE conference on Supercomputing*, Seattle, WA, USA, 2005.

[30] The Object Management Group (OMG). UML 2 Activity Diagram. http://www.omg.org/spec/UML/2.1.2/Superstructure/PDF/.

[31] Z. Wu, A. Ranabahu, K. Gomadam, A. P. Sheth, and J. A. Miller. Automatic Composition of Semantic Web Services using Process and Data Mediation. Technical report, LSDIS lab, University of Georgia, February 2007.

[32] J. Zhang. Ontology-Driven Composition and Validation of Scientific Grid Workflows in Kepler: a Case Study of Hyperspectral Image Processing. In *Proceedings of 5th International Conference on Grid and Cooperative Computing Workshops*, 2006.