Optimal Workflow Execution in Grid Environments

Walter Binder, Ion Constantinescu, Boi Faltings, and Nadine Heterd

Ecole Polytechnique Fédérale de Lausanne (EPFL)
Artificial Intelligence Laboratory
CH-1015 Lausanne, Switzerland
firstname.lastname@epfl.ch

Abstract. This paper describes an original approach to optimally execute computational tasks in Grid environments. Tasks are represented as workflows that define interactions between different services. We extend the functional description of services with non-functional properties, allowing to specify the resource requirements of services for given inputs. Based on such annotations, we derive a mathematical model to estimate the execution costs of a workflow. Moreover, we present a genetic algorithm that optimally distributes the execution of a workflow in a Grid, supporting the installation of software components on demand, in order to fulfill user requirements, such as a limit on the total workflow execution time. We implemented a testbed to validate our approach on randomly generated workflows in simulated Grids.[1] [2]

Keywords: Grid, Service-Oriented Architectures, Workflow Execution, Service Deployment, Non-Functional Properties, Optimization Algorithms

1 Introduction

Service-oriented computing, a new approach to software development, enables the construction of applications by integrating well-tested services (components) [19]. We assume that services are annotated with semantic descriptions concerning their functionality. Such descriptions may specify the required inputs, generated outputs, preconditions, and effects of a service. Different formalisms have been proposed for the semantic description of services, such as OWL-S [18] or WSMO [21]. In this paper we focus on OWL-S service descriptions.

Interactions between services can be represented as a workflow, which specifies the order in which the individual services have to be invoked and how data

[1] The work presented in this paper was partly carried out in the framework of the EPFL Center for Global Computing and supported by the Swiss National Funding Agency OFES as part of the European projects KnowledgeWeb (FP6-507482) and DIP (FP6-507483).

[2] Authors in alphabetical order.

has to be passed between these services. While in general workflow specifications may include loops, in this paper we address only sequential workflows, because most techniques for automated service composition focus on the generation of sequential workflows [20, 14, 6].

A Grid is a collection of machines providing resources (e.g., CPU cycles, memory, network bandwidth, etc.) and services (i.e., installed software components) to execute computationally expensive tasks [1, 10]. A Grid may span different organizations with heterogeneous infrastructure. In this paper we consider the execution of sequential workflows in a Grid, optimally exploiting the Grid resources (hardware and software).

Our goal is to optimally deploy application components (i.e., individual services of a workflow) on the machines in the Grid. The selection of appropriate machines is based on their configuration (installed software and available resources), as we try to avoid expensive re-configurations of machines. The selection of services favors components that consume less resources for a given input. In order to estimate the resource requirements of a component for a task, we extend semantic service descriptions with non-functional properties (resource requirement formulas). These formulas allow to compute the estimated resource requirements in terms of input properties (such as input size) of a task specification [17].

The selection of machines and components focuses on user requirements, i.e., on objectives related to the performance of task execution [7]. E.g., the user may specify a deadline (an upper bound for the execution time) or require to minimize the execution time. Our task deployment algorithm ensures such user objectives.

While our formalism to describe the resource requirements of components is flexible and extensible to allow custom resource definitions, we focus on the following four resource types in this paper:

- *CPU*: As a Grid is a hetergeneous environment which may include many different types of machines, it is essential to define a platform-independent metric for CPU consumption. The most common metric for CPU consumption is the CPU time measured in seconds. However, this metric is not platform-independent, since the CPU time for the same deterministic program with the same input may vary significantly depending on hardware and operating system. Hence, we assume the ubiquitous presence of a common virtual execution environment, such as the Java Virtual Machine [15], and measure CPU consumption in terms of virtual machine *bytecode instructions*, which is a platform-independent metric [9]. For deterministic programs, measurement results are exactly reproducible, which is particularly useful for benchmarking and profiling. The use of bytecode instruction counting as CPU consumption metric was introduced in [5] and refined in [3, 12]. It has also been used in Grid environments for monitoring [13].
- *Static memory*: This resource relates to the code size of a component, i.e., the storage required for installation. We measure it in MB.

– *Dynamic memory*: This resource corresponds to the memory requirements of a component during execution (loaded code, stack, heap, etc.). We measure it also in MB.
– *Network bandwidth*: The transmission of data between components on different machines that are chained together in a workflow consumes a certain amount of network bandwidth. We measure the size of messages in MB and the bandwidth of network links in MB per second.

The main contributions of this paper are (1) an extension of OWL-S [18] that allows defining non-functional properties of components (in particular resource requirements), (2) a mathematical model to compute the execution costs of a workflow, (3) a genetic algorithm to optimize workflow execution, and (4) a testbed to simulate Grids and to generate random workflows, which we used to evaluate our approach.

This paper is structured as follows: In Section 2 we present our system model and use a concrete example to illustrate how user objectives are expressed and how the Grid manages its resources in order to fulfill the user requirements. Section 3 introduces our formalisms and explains how to express non-functional service properties in OWL-S. Section 4 provides a mathematical model to estimate the costs of workflow execution in terms of execution time. Section 5 presents a genetic algorithm to optimize workflow execution. Section 6 evaluates the performance of the genetic algorithm. Finally, Section 7 concludes this paper.

2 System Model and Example Scenario

Fig. 1 shows the general architecture of our system. The Grid is divided into several domains, each of them administered by a *domain manager*. This allows us modeling disjoint administrative boundaries. Each domain manager is in charge of a number of machines, called *donators*, which offer their (idle) computing resources to the Grid.

Each domain manager maintains a number of donators and knows their current configurations, i.e., the locally installed components and the available computing resources (CPU, memory, network bandwidth, etc.). This information is stored in the *donator directory*. Each domain manager also has a *scheduler* to optimally deploy components on the donators.

The *operator* acts as mediator between the user and the Grid environment. Thanks to the *domain manager directory* and to the *implementation profile directory*, the operator is aware of the Grid structure and of available components. Furthermore, it has reasoning capabilities and is in charge of optimally distributing the execution of workflows over the Grid. This is achieved by the *workflow builder* and the *solver*.

We distinguish between 3 types of task/workflow description:

– A high-level *task specification* defining available inputs and required outputs.

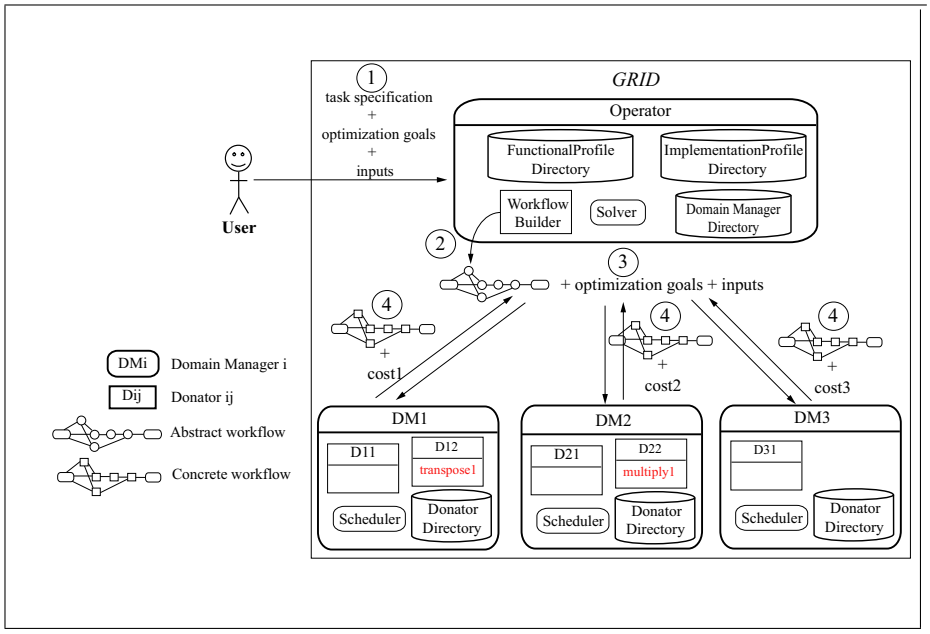

Fig. 1. System overview.

– An *abstract workflow* that refers to a set of services and defines their interactions (dataflow).
– A *concrete workflow* that defines all grounding details (i.e., it specifies which components on which machines to use).

Normally, the user sends a high-level task specification to the operator. As shown in Fig. 1, the operator maintains a *functional profile directory* that contains the semantic descriptions (OWL-S descriptions [18]) of all available service functionalities. With the aid of planning-based service composition techniques, the *workflow builder* is able to automatically generate an abstract workflow from a high-level task specification of available inputs and required outputs. As concrete service composition algorithms are not in the scope of this paper, we refer to [6] for details. Alternatively, the user may directly send an (e.g., manually created) abstract workflow to the operator.

As an example of mapping a scientific computation onto the Grid [8], we consider the following scenario: A user wants to estimate the unknown parameters of a function using the *least squares estimation* method, which can be represented as a matrix resolution problem. As this computation may become computationally hard for large matrices, the user decides to run it on a Grid.

First, the user maps the least squares estimation formula $(X^T X)^{-1} X^T y$ into a task specification. Next, he transmits his request to the Grid operator (Fig. 1 (1)). The request consists of the task specification, some optimization goals, such as a deadline for the results, and the input data. The operator leverages the workflow builder to generate an abstract workflow (Fig. 1 (2)). The

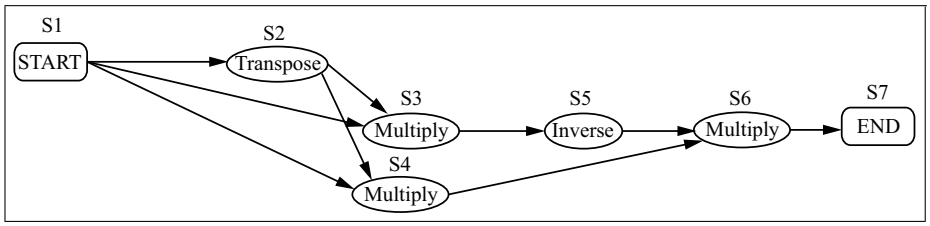

Fig. 2. Abstract workflow representing the least squares computation.

abstract workflow, shown in Fig. 2, uses the following matrix operations: *transpose*, *multiply*, and *inverse*. It takes possible parallelization into account and specifies the order of the matrix operations.

The operator is in charge of deploying the computation so that the execution will be completed before the given deadline. Hence, the operator forwards the abstract workflow to the domain managers (Fig. 1 (3)). Each domain manager tries to find a solution meeting the given deadline by selecting the most efficient components among the advertised ones[3] (e.g., consuming the least CPU and memory resources), and the most powerful machines (in terms of available resources). Each domain manager may install new components only on donators within its domain. I.e., while a concrete workflow may exploit any installed (and advertised) components in the Grid, it may require the installation of new components only within one domain. This gives domains some autonomy, as the installation of new components has to be granted by the domain manager, but cannot be forced by external entities.

Once the operator receives all possible deployment solutions from the domain managers in the form of concrete workflows (Fig. 1 (4)), it chooses the best among them (in terms of shortest execution time), activates it by triggering component installations in the appropriate domain (if necessary), and executes the least squares computation.[4] Finally, the results of the execution are transmitted to the user.

A number of requirements emerge from this scenario: The resource requirements of components, the donators' available resources, and the donators' current configurations have to be represented. Furthermore, the costs of workflow execution must be defined (in terms of execution time), and an algorithm has to be developed to find a deployment that meets the given optimization criteria. These issues are addressed in the following sections.

[3] Advertised components are (1) locally installed components, (2) remote components that are installed on other donators, and (3) components that are not yet installed on any donator but are available for deployment.

[4] The execution of the selected concrete workflow is controlled by task managers within the donators. The execution is based on service invocation triggers, an efficient, lightweight mechanism for distributed workflow execution. The execution of concrete workflows is not in the scope of this paper, we refer to [2].

3 Formalisms

In the following we define the concepts introduced in the previous section more formally.

3.1 General Concepts

Services and Components. Our system comprises:

- A set of *services* F. Each service is specified by a service description, which covers only the service functionality (but not any non-functional aspects, such as resource requirements). F is related to the functional profile directory in Fig. 1.
$$F = \{F_1, F_2, \ldots, F_m\}$$
- Sets of *components*. Each component specifies a service implementation, covering functional and non-functional aspects (i.e., resource requirements). Therefore, for each service F_i, there are possibly j_{F_i} components (implementations of the functionality specified by F_i). These components are represented by the set:
$$C_i = \{C_{i1}, C_{i2}, \ldots, C_{ij_{F_i}}\}$$
The sets of components C_i are related to the implementation profile directory in Fig. 1. The resource requirements of each component C_{ij} are defined as follows:
  - CPU(C_{ij}) represents the CPU requirements of component C_{ij}. It is specified by a function of the input properties of the service and measured in a platform-independent metric, the (approximate) number of required bytecode instructions. As this metric is closely related to algorithm complexity, a general formula of CPU(C_{ij}) can be de derived from a complexity analysis and concrete parameters can be estimated by benchmarking C_{ij} on varying inputs.
  - MemStat(C_{ij}) represents the code size of component C_{ij}. It is specified by a static value and measured in MB.
  - MemDyn(C_{ij}) represents the dynamic memory allocation. It is specified by a function of the input properties of the service and measured in MB.

Domain Managers and Donators. The system has a set of n domain managers:
$$DM = \{DM_1, DM_2, \ldots, DM_n\}$$
Each domain manager maintains a set D_i of j donators
$$D_i = \{D_{i1}, D_{i2}, \ldots, D_{ij}\}$$
and a set L_{DM_i} of components that are locally installed on these donators. The available resources of each donator D_{ik} are defined as follows:[5]:

[5] In our testbed the available resources of donators are specified as static values. In a practical setting, these values change dynamically and are updated by a monitoring infrastructure [13].

- $CPU(D_{ik})$ represents the available CPU resources, measured in bytecode instructions per second.
- $MemDyn(D_{ik})$ represents the available dynamic memory, measured in MB.
- $MemStat(D_{ik})$ represents the available disk storage, measured in MB.
- $Bandwith(D_{ik})$ represents the network bandwidth of donator D_{ik}, measured in MB per second.

Set Relations and Service Grounding. There are several relations between the sets defined above.

In the Grid there are m services, and for each service there is a list of components implementing the service functionality. The set A represents all available components in the Grid and is defined as follows:

$$A = \bigcup_{i=1}^{m} C_i$$

The set I contains the components that are deployed in the Grid, i.e., that are installed on one or more donators. Therefore, $I \subseteq A$. As there are n domain managers, I is defined as follows:

$$I = \bigcup_{i=1}^{n} L_{DM_i}$$

The set $A - L_{DM_i}$ represents all the components that have not yet been deployed in the domain of DM_i, and the set $A - I$ represents the components that have not yet been deployed on any donator.

We define a set of service groundings G as tuples $g_i = \langle c_{ij}, d_{kl} \rangle$, expressing that the component c_{ij} is deployed on donator d_{kl}:

$$G \subseteq C \times D$$

3.2 Workflow Definitions.

Abstract workflow. An abstract workflow $AW = \langle S, R, T \rangle$ is represented as a directed acyclic graph.

- S, the nodes in the graph, represent a set of services that provide all the functionalities used in the abstract workflow.
- $R : S \rightarrow F$ is a function from nodes to services. R gives for each node the corresponding functional service profile.
- T, the edges in the graph, is a set of transmission links, defined as tuples $\langle s_i, s_j \rangle$ of nodes in S ($T \subseteq S \times S$). These links represent the data passed between the services. A service S_i can be invoked only after all incoming links have provided data.

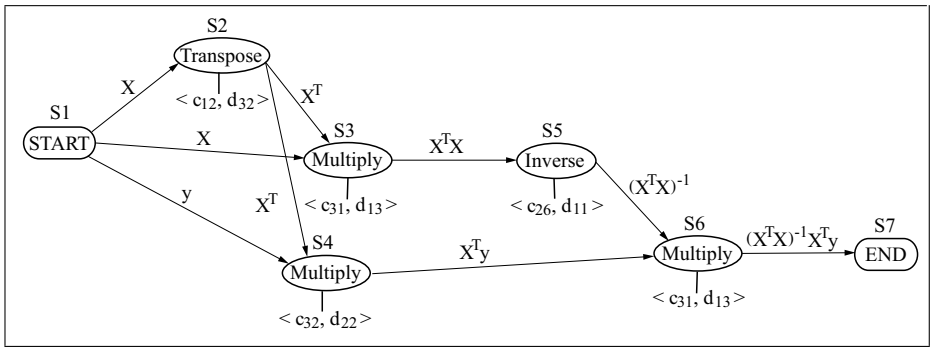

Fig. 3. Concrete workflow representing the least squares computation.

The following definition represents an abstract workflow for the least squares computation (see Fig. 2). The special services START (resp. END) refers to the inputs (resp. outputs):

- $S = \{s_1, s_2, s_3, s_4, s_5, s_6, s_7\}$
- $R = \{\ \langle s_1, START\rangle, \langle s_2, Transpose\rangle, \langle s_3, Multiply\rangle, \langle s_4, Multiply\rangle,$
  $\langle s_5, Inverse\rangle, \langle s_6, Multiply\rangle, \langle s_7, END\rangle\ \}$
- $T = \{\ \langle s_1, s_2\rangle, \langle s_1, s_3\rangle, \langle s_1, s_4\rangle, \langle s_2, s_4\rangle, \langle s_2, s_3\rangle, \langle s_3, s_5\rangle, \langle s_5, s_6\rangle, \langle s_4, s_6\rangle,$
  $\langle s_6, s_7\rangle\ \}$

Concrete Workflow. A concrete workflow $CW = \langle S, R, T, G, B\rangle$ is an extension of an abstract workflow, since it specifies additionally a set of bindings B defined as tuples $\langle s_i, g_i\rangle$ assigning a grounding $g_i \in G$ to each node $s_i \in S$, i.e., $B \subseteq S \times G$. The groundings for the START and END nodes define the source of the initial inputs and the destination of the final outputs. The same grounding may appear in different bindings.

Fig. 3 represents a concrete workflow for the least squares computation (here we do not show the groundings for the START and END nodes):

- $B = \{\ (s_2, \langle c_{12}, d_{32}\rangle), (s_3, \langle c_{31}, d_{13}\rangle), (s_4, \langle c_{32}, d_{22}\rangle),$
  $(s_5, \langle c_{26}, d_{11}\rangle), (s_6, \langle c_{31}, d_{13}\rangle)\ \}$

3.3 Extending the OWL-S Ontology

As the current version of the OWL-S [18] ontology does not include any properties to define the resource requirements of a service, we extended it to support such non-functional properties.

Adding Implementation Profiles. Since a distinction must be made between the functionality of a service and the implementation of this functionality (resource requirements are considered implementation aspects), we added two classes to the ontology:

– The *FunctionalProfile* class specifies the service functionality. This class corresponds to the OWL-S *Profile* [18].
– The *ImplementationProfile* class specifies the resource requirements of a component that implements the service. Resource requirements are non-functional properties that are described by the following attributes:
  • *hasCpuResource* represents the CPU requirements of the component, defined as a function of the input properties.
  • *hasDynMemResource* represents the dynamic memory requirements of the component, defined as a function of the input properties.
  • *hasStatMemResource* corresponds to the code size, which is a constant.

  In the *ImplementationProfile* each output property is defined as a function of the input properties. Output properties are important when components are chained (i.e., when the outputs of a component are passed as inputs to another component), in order to know the properties of intermediary results.

Defining Resource Requirements. Resource requirements are defined by functions that may depend on the properties of the service inputs. The ontology had to be extended in order to accept these resource requirement definitions, as shown in Fig. 4.

ResourceMetrics are defined by an expression (*hasMetricsExpression* property) representing the resource usage function and by a metrics type (*hasMetricsType* property), which specifies name and measurement unit of the metrics. For instance, the unit of the CPU resource is bytecode instructions and the unit of the memory resources is MB.

MetricsExpression is either *MetricsLiteral* representing a real value, *MetricsVariable* describing a variable, or *MetricsOperation* specifying an arithmetic operation. *MetricsOperation* has a set of *hasOperand* properties, describing a metrics expression and therefore allowing the definition of nested functions.

3.4 Example

As an example, we show the specification of a component implementing the transposition operation *myTransposeMatrix*. Below is the declaration of the inputs, outputs, and resource requirements:

```
<MatrixTransposeProfile rdf:ID="myTransposeMatrix">
  <hasInput rdf:resource="#input"/>
  <hasOutput rdf:resource="#output"/>
  <hasDynMemResource rdf:resource="#dynMemResource"/>
  <hasCpuResource rdf:resource="#CPUResource"/>
  <hasStatMemResource rdf:resource="#statMemResource"/>
  <hasPrecondition/>
  <hasEffect/>
</MatrixTransposeProfile>

<Matrix rdf:ID="input"> <hasRows rdf:resource="#inputRows"/>
                        <hasColumns rdf:resource="#inputCols"/>
</Matrix>
<Matrix rdf:ID="output"> <hasRows rdf:resource="#inputCols"/>
                         <hasColumns rdf:resource="#inputRows"/>
</Matrix>
<MetricsVariable rdf:ID = "inputRows"/>
<MetricsVariable rdf:ID = "inputCols"/>
```

For the transposition operation, both the input and the output are matrices. In this example, the relevant aspects of the input are the matrix dimensions, i.e., the number of rows and columns. These dimensions are represented by two variables *inputRows* and *inputCols*. The relevant aspect of the output are also the matrix dimensions. These properties are defined as a function of the input properties. For the transposition operation, the number of output rows equals the number of input columns and the number of output columns equals the number of input rows.

```
<ResourceMetrics rdf:ID = "CPUResource">
   <hasMetricsExpression>
      <Add>
         <hasOperand> <MetricsLiteral>
                        <hasLiteralValue>a</hasLiteralValue>
                      </MetricsLiteral>
         </hasOperand>
         <hasOperand>
            <Mul> <hasOperand> <MetricsLiteral>
                                 <hasLiteralValue>b</hasLiteralValue>
                               </MetricsLiteral>
                  </hasOperand>
                  <hasOperand rdf:resource="#inputRows"/>
                  <hasOperand rdf:resource="#inputCols"/>
            </Mul>
         </hasOperand>
      </Add>
   </hasMetricsExpression>
   <hasMetricsType> <MetricsType>
                      <hasName>CPU</hasName>
                      <hasUnit>Bytecode Instruction</hasUnit>
                    </MetricsType>
   </hasMetricsType>
</ResourceMetrics>
```

CPU and dynamic memory requirements are defined as functions of the input properties. The example above shows the code for defining the CPU usage of the *myTransposeMatrix* component. In this case, the CPU usage is proportional to the product of the input matrix dimensions: $CPU(myTransposeMatrix) = a + (b * inputRows * inputCols)$. a and b are constants which may be determined by the component developer by benchmarking *myTransposeMatrix* with two different inputs and solving 2 simple equations with two variables. See [4] for details on benchmarking using bytecode instruction counting as metric. Static memory requirements are independent of the input properties and their metrics expressions are simply defined as a *MetricsLiteral* representing a real value.

4 Mathematical Model

This section presents a mathematical model to compute a cost function for a given concrete workflow CW. Costs may be measured in terms of CPU spent, total execution time, bandwidth consumption, memory consumption, etc. While we implemented a large number of different cost functions, we address only the total execution time in this paper because of space limitations.

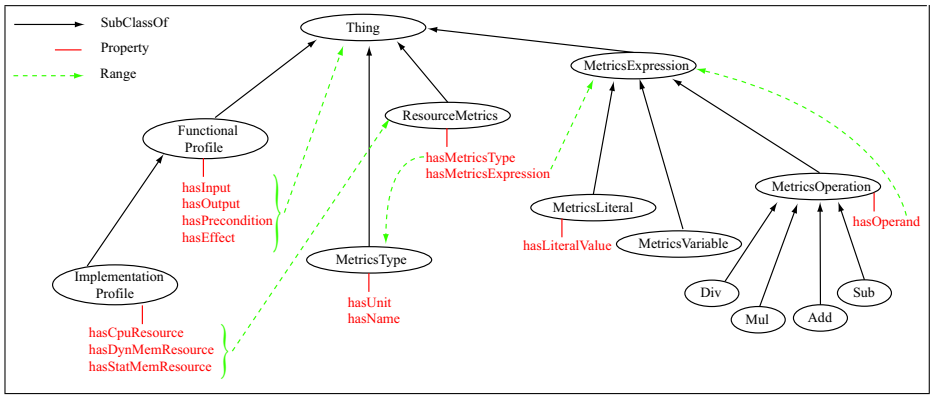

Fig. 4. Extending the OWL-S ontology.

4.1 Cost Measures for Workflow Execution

Considering a concrete workflow, each grounding g_i relates to three kinds of costs (representing the three main stages in the life-cycle of a component):

Activation Cost. $installCost(g_i)$. Any component in a concrete workflow must be activated once, before its first invocation. This is independent of how many times a component is used in the concrete workflow. In fact, this could correspond to the installation of local components on a given donator, or to the installation of local stubs when remotely accessing a component. The $installCost(g_i)$ function defines the cost of activating the component specified by the grounding g_i as a real value.

Data Transfer Cost. $dataTransCost(t, g_i, g_j)$. In order to be able to execute a component, its input parameters must be transferred from the donator corresponding to the previous component in the concrete workflow. The $dataTransCost(t, g_i, g_j)$ function defines the cost of transferring the parameters specified in the data-dependency t between the donators specified in the groundings g_i and g_j corresponding to the two components involved in the data-dependency.

Execution Cost. $invoSerCost(g_i)$. After the component has been activated and its input parameters have been transferred to the current donator, the component can be executed. The $invoSerCost(g_i)$ function defines the execution cost of the component specified by the grounding g_i.

4.2 Computing Costs in Terms of Execution Time

Now we define the cost of a concrete workflow in terms of execution time, taking parallelization (i.e., several components may be executed concurrently) into account. This requires:

- Computing the *activation cost*, which is equivalent to the installation cost.
- Computing the *usage cost*, which represents the execution costs of all the components and the data transfer costs among all the components in the concrete workflow.

Computing Activation Costs. Let G_{sub_j} be a subset of components from G that are installed on a donator D_{ij} managed by the domain manager DM_i. Since in the domain of DM_i we have j donators, we conclude that we also have j subsets of G. In this case, the installation cost is:

$$installCost(CW) = \max\left(\sum_{\forall g_i \in G_{sub_1}} installCost(g_i),\ \ldots,\ \sum_{\forall g_i \in G_{sub_j}} installCost(g_i)\right)$$

where

$$installCost(g_i) = \begin{cases} 0 & \text{if } c_{ij} \in L_{DM_i} \text{ } (c_{ij} \text{ is installed locally}) \\ 0 \textit{ or constant} & \text{if } c_{ij} \in I - L_{DM_i} \text{ } (c_{ij} \text{ is accessed remotely}) \\ \frac{MemStat(c_{ij})}{Bandwith(d_{ik})} & \text{if } c_{ij} \in A - I \text{ } (c_{ij} \text{ is not installed}) \end{cases}$$

Computing Usage Costs with Abstract Interpretation. Abstract interpretation is a theory of sound approximation of the semantics of computer programs. It can be viewed as a partial execution of a computer program to gain information about its semantics (e.g., control structure, flow of information, etc.) without performing all the calculations.

With abstract interpretation we can take parallelization and resource constraints into account. In our case, the concrete workflow is interpreted over an abstract domain: Components are not really invoked and data is not really transferred, but a simulation of the relevant aspects is done. These aspects are the starting and ending execution time of components and the resource allocation of donators.

For each donator in the concrete workflow, an interval list of its activation is kept (this list is initially empty). For the concrete workflow, the availability time of each data transfer message is stored.

In each step, a component c_{ij} is selected which has all its input messages $M = \{m_1, \ldots, m_n\}$ available. The component c_{ij} has as

- starting time, the maximum of its messages' arrival times.

$$starting\ time = \max(arrivalTime(m_1), \ldots, arrivalTime(m_n))$$

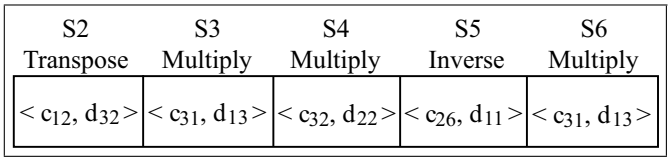

Fig. 5. Chromosome describing the concrete workflow of Fig. 3.

– ending time, the sum of its starting time and its invocation cost in terms of execution time.

$$ending\ time = starting\ time + \frac{CPU(c_{ij})}{CPU(d_{ik})}$$

After calculating the starting and ending time of a component, the activation list of the corresponding donator is examined. We search for the next free interval (where the donator has not yet been assigned to a task) and register the computation of the component there. If necessary, the computation is split into several pieces.

Having updated the activation list, we know when the output messages of the component are available. For each output message, we use the (estimated) message size (remember that our extension to OWL-S allows defining output properties as functions of input properties) and the bandwidth of communication links (data transfer costs) to compute the arrival times at the message destinations.

5 Optimization Using Genetic Algorithms

In Section 4 we defined an optimization function in terms of execution time. Now we must choose an optimization method well adapted to this problem. As we are considering a large-scale search space, we decided to use a non-deterministic optimization method. We chose a *genetic algorithm* [16], since it was straightforward to implement; other optimization techniques would be applicable as well. In order to formalize the concrete workflow cost optimization problem as a genetic algorithm (GA), we have to define the population, the individual representation, and the operators of the GA.

Population and Individual Representation. In our case, the *population* is a set of concrete workflows. Each of these concrete workflows represents an individual and is described as a chromosome having the nodes of the concrete workflow as *genes* (the set S representing functional profiles).

Each of these genes has a specific alphabet or *alleles*. This alphabet is described by a set of groundings (the set G). Each component of the groundings must be included in the set of components of the function specified by the given node. An example is shown in Fig. 5.

Fitness Function and Termination Condition. The fitness of an individual is the execution time of the concrete workflow it represents. The smaller the execution time, the better the individual fits in the environment. The genetic algorithm terminates when a given deadline is met or when a convergence value is detected, i.e., when the improvement after the last n steps is smaller than a given delta δ.

Selection Operator. There are several different types of selection mechanisms [22]. We use the k-way tournament selection method. The idea of the k-way tournament selection is to hold a tournament among k randomly picked individuals. The winner of the tournament is the individual with the highest fitness, which is inserted into the mating pool. We repeat this process until we have n (the size of the population) winners.

Recombination Operator. The recombination operator is performed on a couple of individuals: The parents' chromosomes are cut at the same random position and their second halves are swapped between them, thus yielding two new individuals, each containing genes from both parents.

Mutation Operator. The mutation operator is performed on an individual and not on a couple as the recombination operator. Each gene of each individual has a very small probability of modifying the allele representing this gene (i.e., the grounding). The modification is done by choosing randomly another grounding in the alphabet corresponding to the given gene.

Preventing Premature Convergence. Premature convergence is a serious concern in genetic algorithms [11]. It means that the population converges towards a local optimum instead of the global optimum. This occurs when the crossover operator has a high probability; i.e., when diversity is lost. In short, the 'wrong' part of the search space is explored. Therefore, mutation is needed to increase diversity in the population and to explore other parts of the search space. Two mechanisms were implemented in order to prevent premature convergence:

The first mechanism is to start the execution of the genetic algorithm with a very high mutation rate and a low recombination rate in order to explore the search space. Then, after a certain number of generations, we slowly decrease the mutation rate and increase the recombination rate in order to direct the search towards the 'good' part.

The second implemented technique is to initialize a proportion of the population randomly. Every ten generations, 10% of the individuals of the new generation are created randomly.

6 Evaluation

In order to evaluate our approach, we implemented a testbed to simulate different Grid environments. The testbed allows varying many parameters, such as the

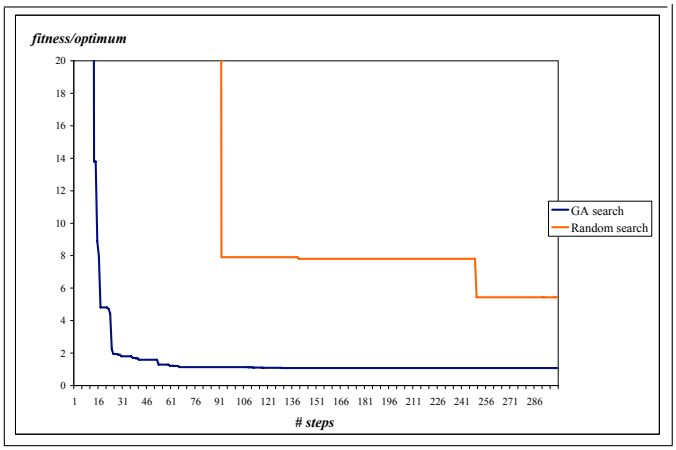

Fig. 6. Evaluation results: Installation of components on donators is NOT possible.

topology of the grid (the number of domain managers), the number of donators, the donators' network bandwidth, the donators' provided resources, as well as the available components and their respective resource requirements. Moreover, our testbed allows generating random abstract workflows. While we validated our approach in a large number of different settings, we selected 3 representative experiments for the discussion below.

We launched the GA on large-scale systems with different configurations (in terms of the number of donators, the number of domain managers, etc.) and compared the results with a random search strategy, which explores the search space randomly, always preserving the best solution found so far. We used the following methodology for our evaluation:

- First, the GA was launched until convergence was reached. The result (i.e., the shortest execution time) is called C and represents the optimum fitness. In smaller settings, we used exhaustive search to verify that C was the global optimum. However, in large-scale settings an exhaustive search is not viable because of the large search space.
- Then the GA and the random search algorithm were launched over 300 steps.
  - $F_{ga(n)}$ represents the best fitness achieved after n steps using the GA.
  - $F_{rs(n)}$ represents the best fitness achieved after n steps of random search.

For both search algorithms, we always generated 30 individuals in each step (generation). For all experiments, we used a configuration with two domain managers, each controlling 50 donators. For each experiment, we randomly configured the properties of each donator (available resources, connectivity and bandwidth, installed components, etc.).

In order to present the results of our evaluation, we wanted to show both the quality of the results (fitness) and the speed of convergence (over the number of steps) in a single figure. Therefore, we represent the results in a graph, were

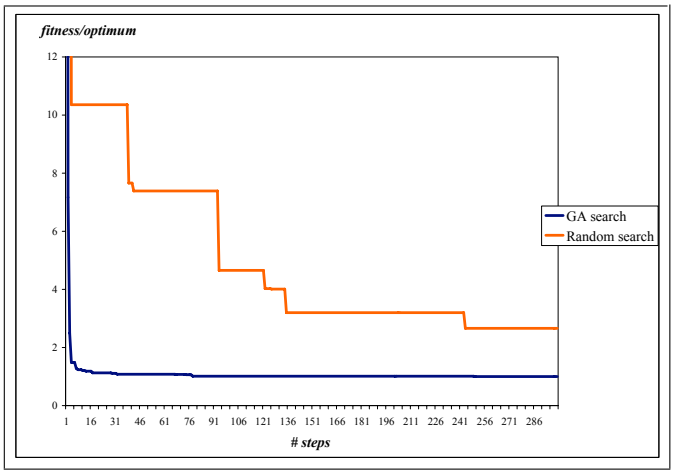

Fig. 7. Evaluation results: Installation of components on donators is possible.

the X-axis corresponds to the number of steps n and the Y-axis shows the best relative fitness obtained until step n (a value of 1 means that the optimum C has been reached). We plot two curves over the number of algorithm steps: One for the GA and one for the random search. I.e., the curves represent the function $\frac{F_{ga(n)}}{C}$ for the GA, resp. the function $\frac{F_{rs(n)}}{C}$ for the random search.

In Fig. 6 and Fig. 7 the least squares computation is considered. Fig. 6 shows the results obtained with a configuration where the installation of components on donators is impossible (i.e., donators do not have enough static memory resources to allow installation). Thus, the solution can only use locally installed components or remotely accessible components. Fig. 7 shows the results when the installation of components on donators is possible (i.e., donators have enough resources in terms of static memory). In Fig. 7 the GA converges rather quickly compared to Fig. 6, where it takes about 60 steps to find a solution close to the optimum.

Fig. 8 shows the result obtained with a randomly generated abstract workflow of 10 nodes (i.e., the connections between the 10 nodes are set randomly). In this kind of experiment, we are interested to see the behavior of the GA and of the random search for abstract workflows with different complexity. The installation of components on donators is allowed.

To sum up, in all settings the genetic algorithm converges quickly towards the optimal solution, whereas the random search algorithm converges much slower and never reaches the optimum within 300 steps.

7 Conclusion

In this paper we presented a novel, original approach to optimally distribute and execute workflows in a Grid. Our system takes as inputs a high-level task specification (from which we derive a workflow defining the functionalities and

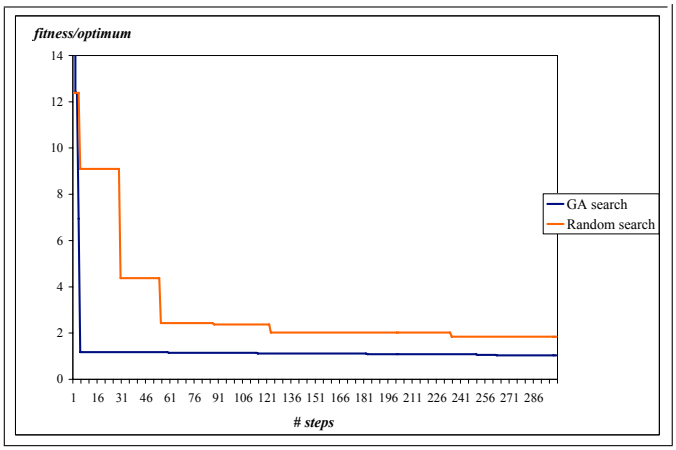

Fig. 8. Evaluation result with a random abstract workflow composed of 10 nodes.

dependencies of required services), optimization goals (e.g., a deadline for the completion of the workflow execution), and input data. Based on these inputs, the system selects the best suited components that implement the desired functionalities and deploys them optimally across the machines of the Grid, favoring already installed components and taking CPU consumption, static and dynamic memory requirements, and network bandwidth into account.

In order to model this optimization problem, we extended the OWL-S ontology with non-functional properties to describe the resource requirements (in terms of input properties) of different components. With the aid of these descriptions, we derived a mathematical model to estimate the overall execution time of a workflow in a particular setting. The model, which may become rather complex because it also has to take resource conflicts into account, is evaluated using abstract interpretation. Finally, we leverage genetic algorithms to search for an optimal workflow execution that satisfies the given requirements. An evaluation of our system using randomly generated workflows in simulated Grids revealed that our optimization algorithm converges quickly towards an optimal solution.

References

1. F. Berman, G. Fox, and T. Hey. The grid: Past, present, future. In F. Berman, G. Fox, and T. Hey, editors, *Grid Computing: Making the Global Infrastructure a Reality*, pages 9–50. John Wiley & Sons Inc., 2003.
2. W. Binder, I. Constantinescu, and B. Faltings. Efficiently distributing interactions between composed information agents. In *Second European Workshop on Multi-Agent Systems (EUMAS-2004)*, Barcelona, Spain, December 2004.
3. W. Binder and J. Hulaas. A portable CPU-management framework for Java. *IEEE Internet Computing*, 8(5):74–83, Sep./Oct. 2004.
4. W. Binder and J. Hulaas. Using bytecode instruction counting as portable CPU consumption metric. In *QAPL'05 (3rd Workshop on Quantitative Aspects of Pro-*

gramming Languages), ENTCS (Electronic Notes in Theoretical Computer Science), Edinburgh, Scotland, April 2005.
5. W. Binder, J. G. Hulaas, and A. Villazón. Portable resource control in Java. *ACM SIGPLAN Notices*, 36(11):139–155, Nov. 2001. Proceedings of the 2001 ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA'01).
6. I. Constantinescu, B. Faltings, and W. Binder. Large scale, type-compatible service composition. In *IEEE International Conference on Web Services (ICWS-2004)*, San Diego, CA, USA, July 2004.
7. A. Dan, C. Dumitrescu, and M. Ripeanu. Connecting client objectives with resource capabilities: An essential component for grid service management infrastructures. In *Proceedings of the 2nd International Conference on Service Oriented Computing*, Nov. 2004.
8. E. Deelman, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, S. Patil, M.-H. Su, K. Vahi, and M. Livny. Pegasus: Mapping scientific workflows onto the grid. In *European Across Grids Conference*, pages 11–20, 2004.
9. B. Dufour, K. Driesen, L. Hendren, and C. Verbrugge. Dynamic metrics for Java. *ACM SIGPLAN Notices*, 38(11):149–168, Nov. 2003.
10. L. Ferreira, V. Berstis, J. Armstrong, M. Kendzierski, A. Neukoetter, M. Takagi, R. Bing-Wo, A. Amir, R. Murakawa, O. Hernandez, J. Magowan, and N. Bieberstein. *Introduction to Grid Computing with Globus*. IBM Redbook, second edition, September 2003.
11. D. B. Fogel. *Evolutionary Computation: Toward a New Philosophy of Machine Intelligence*. Wiley-IEEE Press, second edition, 18 August 1999.
12. J. Hulaas and W. Binder. Program transformations for portable CPU accounting and control in Java. In *Proceedings of PEPM'04 (2004 ACM SIGPLAN Symposium on Partial Evaluation & Program Manipulation)*, pages 169–177, Verona, Italy, August 24–25 2004.
13. J. Hulaas, W. Binder, and G. D. M. Serugendo. Enhancing Java grid computing security with resource control. In *International Conference on Grid Services Engineering and Management (GSEM 2004)*, Erfurt, Germany, Sept. 2004.
14. O. Lassila and S. Dixit. Interleaving discovery and composition for simple workflows. In *Semantic Web Services, 2004 AAAI Spring Symposium Series*, 2004.
15. T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, Reading, MA, USA, second edition, 1999.
16. D. Mange and M. Tomassini. *Bio-Inspired Computing Machines Toward Novel Computational Architectures*. Presses polytechniques et universitaires romandes, Lausanne, Switzerland, 1998.
17. M. Meyerhofer and K. Meyer-Wegener. Estimating non-functional properties of component-based software based on resource consumption. In *Software Composition Workshop (SC 2004) affiliated with ETAPS 2004*, April 2004.
18. OWL-S. DAML Services, http://www.daml.org/services/owl-s/.
19. M. P. Papazoglou and D. Georgakopoulos. Introduction: Service-oriented computing. *Communications of the ACM*, 46(10):24–28, Oct. 2003.
20. S. Thakkar, C. A. Knoblock, J. L. Ambite, and C. Shahabi. Dynamically composing web services from on-line sources. In *Proceeding of the AAAI-2002 Workshop on Intelligent Service Integration*, pages 1–7, Edmonton, Alberta, Canada, July 2002.
21. WSMO. Web Service Modeling Ontology, http://www.wsmo.org/.
22. B.-T. Zhang and J.-J. Kim. Comparison of selection methods for evolutionary optimization. *Evolutionary Optimization*, 2:55–70, 2000.