

ODD/SAMURAAIE: Providing HPC Users with a Model-Based on-Demand Deployment Architecture

Boris Daix^{1,2}, Samuel Kortas¹, Christine Morin², and Christian Pérez³

¹ EDF R&D, 1 avenue du Général de Gaulle, 92140 Clamart, France

² INRIA — Centre Rennes - Bretagne Atlantique, France

³ INRIA — Centre Grenoble - Rhône-Alpes, France

{Boris.Daix, Samuel.Kortas}@edf.fr

{Christine.Morin, Christian.Perez}@inria.fr

Abstract. Deploying an application on grids is a complex task, that at the minimum requires resource allocation, installation, and execution of the application. While it turns out to be more or less automatic for simple applications, it still is a challenge for complex applications, such as multiphysic simulations, that are made up of several interconnected parts, using different programming models. This paper presents ODD/SAMURAAIE, a model-based architecture for automatic on-demand deployment, which aims at providing a generic model to fill the gap between the complexity of applications and of grids. This paper demonstrates the benefits of ODD/SAMURAAIE thanks to three widely used HPC deployment cases: multi-threaded parametric applications, message-passing parallel applications, and component-based workflow applications.

1 Introduction

The last decade has seen a major change in the management of computing resources. While these resources used to be well known, homogeneous, and static, they become unknown, heterogeneous, and dynamic. Such a transformation started with grids, that promote a vision where resources are discovered when the application is submitted, and it remains true in clouds or desktop grids. To overcome these difficulties, mechanisms such as JSDL [1] have been developed for grid middleware to describe an application as accurately as possible so as to be able to execute it efficiently and automatically. It is clearly an improvement compared to the beginning where users had to manually retrieve information from Globus Meta-Directory Service (MDS), and write Globus RSL file.

However, we are still facing a similar situation because of the increasing complexity of applications and of programming models. In order to take into account more phenomena so as to obtain more realistic simulations, applications are increasingly complex. Standard HPC applications are based on such libraries as MPI or OpenMP. However, recent applications are made up of several interacting

components that might not even be written with the same programming model. The cooperation between components can be based on workflow or dataflow. For example, numerical simulation platform SALOME [2] provides a component model supporting strong or loose coupling of sequential or parallel codes. It enables its users to do all the steps required to complete their simulations, from defining geometrical structures to visualizing post-processed results, including programming complex computing schemas, etc.

This paper aims at studying a generic model to ease the deployment of any programming model while taking into account the parallelism and distribution, the heterogeneity and the dynamicity of both applications and resources. Section 2 recalls the operations involved while deploying an application and presents related works. ODD/SAMURAAIE, our proposal of model-based on-demand deployment architecture, is presented in Section 3 and it is evaluated for three motivating examples in Section 4. Section 5 concludes the paper.

2 Deploying in HPC environments

2.1 Deployment Process Overview

Deploying an application not only consists of installing its software on some hardware, but rather is a three-step process: *Allocation*, *Installation*, and *Execution*. The *Allocating* step finds available resources fitting given applications requirements. According to the technologies, it may include profiling applications, consulting resource information systems, booking some machines, etc. The *Installation* step sets application artifacts up on the allocated resources. It may include downloading binary code packages, transferring file archives, compiling source code, editing configuration files, setting environment variables, etc. The *Execution* of the application processes is coordinated on the resources where application artifacts are installed. It may include building command lines, writing job submission files, looking after process statuses, sending signals to processes, managing precedences between processes, etc. The deployment process can be straightforward in some contexts however, due to their constraints, it is very complex in the HPC environments.

2.2 Constraints of the HPC environments

Both the resource and application technologies may be parallel and distributed, heterogeneous, and dynamic. *Parallelism and distribution* raise communication topology issues: communicating parts of applications have to run on resources that themselves communicate. *Heterogeneity* raises compatibility issues: applications have to be run on compatible resources only. *Dynamicity* implies a more frequent deployment, which has to be dynamic itself: initially allocated resources may suddenly quit the available pool, or new parts of applications may require additional resources. In a HPC environment, deployment succeeds not only once application runs, but rather when it runs efficiently. For instance, the performance requirements may include processor speed, memory sizes, and storage capacities, but also network, memory, and filesystem latencies and throughputs.

2.3 Existing approaches

To allocate resources, many schedulers are available, mostly in resource technologies but also in some application technologies. Schedulers exist at computer level in operating systems, at cluster levels through batch systems like Maui [3], and at grids level like DAGMan [4]. They are also sometimes provided at application level within workflow engines such as in Pegasus [5].

Applications installation, technologies basically fall in three categories: compiling, configuring, or shipping. The compilation technologies include the GNU autotools and compilers. The configuration technologies include networked machine configuration making, synchronizing, and checking systems such as OSCAR [6], GNU CfEngine [7], or CDDL [8] with HP Smartfrog [9]. The shipping technologies belong to categories as numerous as there are shipping formats: filesystem images, file archives, code packages, and software components. For instance, these technologies include Kadeploy [10], FAI [11], and Debian APT.

To run applications, there are job managers, which include operating systems, single system image operating systems, batch schedulers, grid middleware, and parallel shells such as Kerrighed [12], Vigne [13], XtremOS [14], IBM LoadLeveler [15], OAR [16], JSDL [1] and DRMAA [17] with Torque Resource Manager [18] and Sun Grid Engine [19], Globus Toolkit [20], and DIET [21]. Some application technologies also provide their own launchers like MPICH [22] or engines like in workflow technologies.

Among all deployment-related works, only a few projects attempt to address the whole deployment process: ORYA, Deployware/FDF, and Adage. *ORYA* [23] is a deployment meta-model which defines profiles for application packages, users, and resources. A constraint solver uses this information and applies a strategy to allocate packages on resources. A workflow engine then installs the packages on the resources by using other tools dedicated to ORYA. *Deployware/FDF* [24] is a framework for installing and running distributed n-tier and component-based applications on distributed, heterogeneous, and dynamic resources. Deployware itself is a domain-specific language, and FDF (Fractal Deployment Framework) is its interpreter. *Adage* [25] is a deployment tool that takes resource and application descriptors as an input and that allocates, installs, and runs applications on resources. Resource and application descriptors are XML documents which are technology-specific. Specific application descriptors are translated into GADe (Generic Application Description) so that Adage can deploy parallel and distributed applications made of several technologies.

2.4 Discussion

So far, most existing approaches do not automate the whole deployment process and, whenever they do, they do not yet support all HPC properties. When they exist, the schedulers, launchers, or engines for the applications hopefully lead to better resources allocations than would do schedulers and job managers provided by the resources alone. However, they are not generic enough to let users rely on them for all their applications, in particular for their heterogeneous applications.

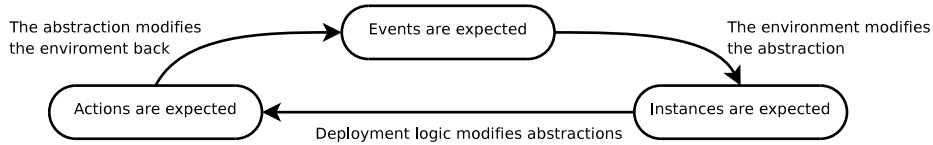


Fig. 1. Permanent cycle of ODD/SAMURAAIE

Moreover, most schedulers do not take into account installation loads which are induced by resources allocation and, to work properly, the launchers and engines have to be configured for each run. To install, the configuration and shipping technologies are easier than compilation ones, however they often are not interoperable. ORYA is an environment distributed across a whole organization but it does not cover HPC constraints. Deployware/FDF has been designed for autonomous applications, so it does not allocate resources. Adage considers the whole deployment process but it does not support dynamicity.

3 ODD/SAMURAAIE

ODD/SAMURAAIE is a model-based on-demand deployment architecture for HPC. It has been designed to manage the three-step process for deploying applications on resources, taking into account distribution, heterogeneity, and dynamicity. As a dynamic architecture, it considers deployment as a permanent cycle (see Figure 1). This cycle has three stages: the environment modifies its abstraction in ODD/SAMURAAIE; the core of ODD/SAMURAAIE reacts to these changes by further modifying the environment abstraction; these latter abstraction changes are committed back into the environment.

In ODD/SAMURAAIE, deployment artifacts of the environment can be resources, services featured by applications running on resources, application files, etc. and even deployment tools. Demands regarding the deployment are expectations for running applications on resources. The model assumes that all those artifacts have ever been deployed, and that those expectations only imply changing their system statuses. ODD/SAMURAAIE supports self-deployment.

Deployment is seen as a side-effect of better scheduling rather than being an off-line process orthogonal to traditional scheduling. Indeed, ODD/SAMURAAIE allocates resources for applications, but it also gathers service requests to install and run them, allocates services for these requests, and enacts these requests. It is based on a permanent cycle between three kinds of abstractions: events (for environment changes), instances (for resources and applications), and actions (for services and service requests). It also requires a translation and connection mechanism, in order to communicate with the concrete HPC technologies of the environment.

Table 1. Abstraction names according to the abstraction level.

	Instance	Action	Event
Contents	<i>Appl.</i>	<i>Program</i>	<i>Messages</i>
Containers	<i>Res.</i>	<i>Services</i>	<i>Sensors</i>
Mappings	<i>Map</i>	<i>Deployment</i>	<i>Journal</i>

Table 2. Vertex names at instance level for applications (1st row) and resources (2nd row).

Aggregate	Data	Comput.	I-O
<i>Process</i>	<i>Blob</i>	<i>Activity</i>	<i>Channel</i>
<i>Host</i>	<i>Memory</i>	<i>Processor</i>	<i>Route</i>

Table 3. Vertex names at action level.

	Aggregate	For data	For computation	For I-O
Programs	<i>Job</i>	<i>Storage request</i>	<i>Execution request</i>	<i>Routing request</i>
Services	<i>System</i>	<i>Storage service</i>	<i>Execution service</i>	<i>Routing service</i>

3.1 System Abstraction Model

SAMURAAIE stands for *System Abstraction Model for User, Resources, and Applications (Actions on, Instances of, and Events from)*. It is a model that abstracts the deployment of user’s applications on resources. It allows the abstraction not only of the instances of application and resource technologies, but also of both the actions on, and the events from them.

The purpose of SAMURAAIE abstractions is not giving exact descriptions of underlying concrete entities, but rather catching their constraints so that automatic deployment succeeds. To catch the constraints, SAMURAAIE uses the metaphor of containers and contents. Given contents may fit in given containers, according to rules that the characteristics of these containers and contents may satisfy. Table 1 lists abstraction names according to their abstraction level and whether they are contents, containers or mapping (links).

SAMURAAIE abstractions are structured as directed graphs. This structure is well-suited to abstract topologies. Vertexes and arcs have attributes to abstract the other characteristics (see below). There are two kinds of vertexes: basic and aggregate. Aggregate vertexes are used to factorize some attributes and connections between basic ones. Finally, vertexes of content graphs may be connected to vertexes of container graphs with *linkage* arcs.

Instances: mapping applications on resources At the instance level, deployment constraints are architectural. Computing basically involves some computation accessing some data through some inputs-outputs. Computation elements may share input-output elements, and input-output elements may share data elements. Data accesses connect computation elements with input-output elements, and input-output elements with data elements. As data accesses are directed, they are abstracted by *access* arcs. Moreover, some data may be copies of some others. Such copy links are abstracted by *copy* arcs. Table 2 gives vertex names at instance level. For a very simple example, see Figure 2.

Computation, input-output, and data elements have other characteristics. A computation element may have a frequency in Hertz, an instruction set architecture, and a number of bits. An input-output element may have a latency in

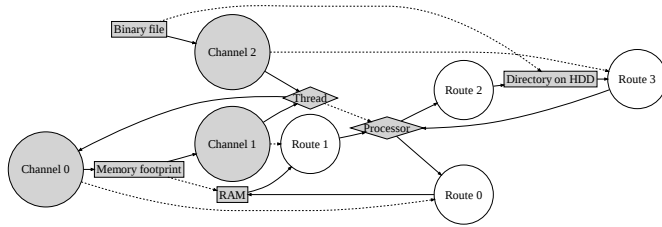


Fig. 2. Example of resource, application, and map abstractions

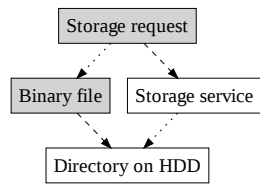


Fig. 3. Example of service, program, and deployment abstractions

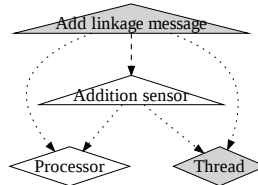


Fig. 4. Example of sensor, message, and journal abstractions

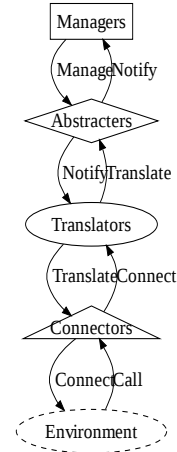


Fig. 5. Organization of actors

second, a throughput in bytes per second, a protocol, and an address or a path. A data element may have a size in bytes, and a medium.

All elements have names and system statuses. At a given time, a status is one of the following: **REGISTERED** (the element is registered but its status is not further specified), **EXPECTED** (the element is expected to exist or to become up and running), **CORRUPTED** (something wrong happened to the element), **OK** (the element exists or is up and running), **DEPRECATED** (the element is expected to become nonexistent or down and no longer running), **SUSPENDED** (the element exists but it is not yet running or no longer running), **HALTED** (the element has been halted while running, before it completes), **COMPLETED** (the element has completed and is down).

System statuses according to time may be specified by time intervals or by time durations. Intervals are specified by couples of fully qualified dates, durations by amounts of seconds. Intervals are ordered by definition. Durations may be ordered too, with a second kind of arcs after *access: precedence*.

Actions: mapping programs on services At the action level, deployment constraints are logistical. Computing basically involves executing some computation, storing some data, and managing some inputs-outputs. Executing some computation aims at both *processors* and *activities*, storing some data at both *memories* and *blobs*, and managing some inputs-outputs at both *routes* and *channels*. This "aim at" relationship from actions to instances is abstracted by *scope arcs*. Moreover, some *processes* may provide some *services*. This "provide"

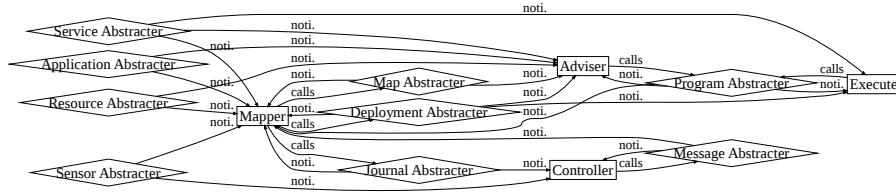


Fig. 6. Interactions between managers and abstracters are notifications or calls.

relationship from instances to services is abstracted by *feature* arcs. Table 3 gives vertex names at action level. Figure 3 shows a very simple example.

All actions have names, system statuses according to time, and they also have one of the following uses: **CREATE** (create content on container), **COPY** (copy content from one container to another), **TEST** (test existence of content on container), **SUSPEND** (suspend content on container), **RESUME** (resume suspended content on container), **REMOVE** (remove content from container).

Events: mapping messages on sensors At the event level, deployment constraints are changes happening in the environment that modify abstractions. Events aim at either instances or actions. This relationship between events and either instances or actions is abstracted by *scope* arcs. Moreover, events may be ordered and this is abstracted by *precedence* arcs. Figure 4 shows a very simple example.

Events have names, system statuses according to time, and they have one of the following uses: **ADD** (add vertex or arc to abstraction), **EDIT** (edit vertex in abstraction), **REMOVE** (remove vertex or arc from abstraction).

3.2 On-Demand Deployment Model

ODD stands for *On-Demand Deployment*. It is a management model designed to automate the deployment of applications on resources. ODD relies on SAMURAAIE to abstract the system. There are four categories of actors in ODD, as shown in Figure 5: *Managers*, *Abstracters*, *Translators* and *Connectors*. *Managers* and *abstracters* are at the core of ODD/SAMURAAIE. *Managers* are responsible for managing the whole deployment process (see the Figure 6). There are four managers: mapper, adviser, executer, and controller. *Mapper* maps contents on containers without breaking mapping rules. *Adviser* gathers actions from instances. *Executer* enacts actions. Finally, *controller* handles events. *Abstracters* are responsible for storing and maintaining abstractions. They are nine, one for each abstraction (see Table 1). *Abstracters* offer transactional interfaces to modify abstractions, and they notify managers or translators as soon as abstractions have been modified. *Translators* and *connectors* are responsible for integrating *abstracters* into the environment. *Translators* translate concrete technology information into abstractions, and the inverse. *Connectors* connect translator technologies with the ones of the environment.

4 Examples

ANGE, which stands for *Adage New Generation*, is the in-progress implementation of ODD/SAMURAAIE. Written in Python 2.5, its core has been used as a simulator to generate SAMURAAIE abstractions for the three widely used cases that follow. The experiments simulate a deployment on behalf of a user who is logged in a desktop workstation. ANGE is assumed to be deployed on this workstation, and to have proper translators and connectors to communicate with the concrete resource and application technologies involved in the environment.

4.1 Deploying multi-threaded parametric applications on computers

In the first example, the user wants to deploy a multi-threaded parametric application on a remote computer. This computer has a multi-processor and some free space on a local hard disk drive. The application consists of a launcher and a parametric code. The launcher is a shell script that takes as an input both the path to the parametric code, and the study parameters. The parametric code is multi-threaded and it takes as an input the case-by-case parameters. These parameters are the paths to input and output files. The launcher, the parametric code, and the input files are located in the hard disk drive of the user's workstation.

The user asks ANGE to deploy the application. The stage of the permanent cycle in which the environment modifies its abstraction starts: The user provides the connectors with the authentication parameters for the resources, and the study parameters for the application. With this information, the resource connector probes the remote computer, and the application connector both probes local application files, and gathers the application from the study parameters. The connectors then provide their corresponding translators with the concrete technology information they have just obtained, such as the processor frequencies, the memory latencies, the network addresses, etc., and such as the input file sizes, the number of parametric code instances, etc. The translators call the abstracters to modify the resource, service, application, and map abstractions.

Here is a word about the abstractions that have been modified. The user's workstation, the remote computer, and their network interconnection are abstracted in the resource abstraction. The services that these computers provide for execution and storage are abstracted in the service abstraction. The application files, the expected output files, and the expected processes are abstracted in the application abstraction. Finally, the location of the application files in the user's workstation is abstracted in the map abstraction.

The next stage of the permanent cycle is the reaction of the managers to the abstraction modifications when the resource, service, application, and map abstracters notify them. Mapper maps, on the remote computer, as many expected processes as this computer may host at the same time, respecting the mapping rules at the instance level. Because the application files are already mapped in the user's workstation, it adds copies to the application abstraction, and then it maps these copies in the remote computer. Adviser notices that some expected

files and processes are mapped, so it gathers, orders, and adds copy and submission requests to the program abstraction. Then mapper maps these expected requests respectively on storage and execution services. Executer turns OK the expected requests, respecting the precedence in which they have been connected (copy before submit).

Finally, the changes made by the managers are committed back into the environment. The application translator gathers the file paths from the abstractions and hence builds the command lines. Then, thanks to its resource connector, the deployment translator submits them to the concrete execution service of the remote computer: the application runs.

When it notices that some processes have completed, the application connector notifies it to the application translator. In turn, the application translator calls the application abstracter to turn COMPLETED the corresponding processes. The mapper then notices that there is some room back for other processes to be hosted, and so on and so forth until every process has completed.

4.2 Deploying message-passing parallel applications on clusters

In the second example, the user wants to deploy a message-passing parallel application on a remote cluster of computers. The remote cluster is made up of a frontal node, which is the computer described in the first example, and of several compute nodes which have the same hardware characteristics. In addition, the frontal node exports some more free space that every compute node mounts locally, at a different mounting point than the one of the frontal node. The frontal and compute nodes are interconnected by a commodity switch. The compute nodes are not reachable from the user's workstation: process and file manipulations on them happens through the frontal node on which a batch scheduler is running.

The frontal node is mandatory for the user's workstation to reach the compute nodes. The resource and service abstractions abstract both the networked cluster of computers, the shared filesystem, and the cluster-wide services of storage and execution. The frontal node has two different network addresses: the one of the route from the user's workstation (the same as in the first example), and the one of the routes from compute nodes. All node processors have accesses to the shared filesystem of the frontal node and, like for network addresses, the mounting points are abstracted on the routes of these accesses. Finally, the frontal node services have all compute node processors and hard disk drives into their scope.

The message-passing parallel application is made up of a launcher and several intercommunicating processes. The launcher takes as an input the number of processes, the path to the binary file, and the arguments for this binary file. The binary file is linked to a shared library. In that example, the processes have read-only accesses to a shared input file, whose path is given as the binary file argument.

The application abstraction abstracts the intercommunicating processes as the resource abstraction does for the networked cluster of computers: every pro-

cess activity (thread) has write-only accesses to all other process memory footprints. Compared with a multi-threaded process like in the first example, there are more channels between intercommunicating processes: one process writes to the memory footprint of the others, the latter read their memory footprints, and symmetrically.

This time, the mapper must take the resource and application topologies into account in order to correctly map the application on the resources. Depending on the throughput required by the processes to access the read-only input file, the mapper can decide adding as many copies as there are compute nodes, hence mapping these copies on their local disk drives. On the contrary, it can prefer adding only one copy, and mapping it on the filesystem shared by all compute nodes. In both cases, the adviser gathers as many copy requests as needed. During the cycle stage in which changes are committed back into the environment, the application translator builds the parameters required by the concrete launcher, and the resource translator builds the corresponding batch scheduler jobs.

Whenever a compute node fails, the permanent cycle goes for another round: the remote cluster connector notices it and transmits that information to its translator, which turns `CORRUPTED` the system status of that node in the resource abstraction. The resource abstracter then notifies the adviser, which analyzes the situation. Whether a process was running on that node when it failed, the adviser may add requests to kill the whole application.

4.3 Deploying component-based workflow applications on grids

In the third and last example, the user wants to deploy a component-based workflow application on a remote grid of computers. The remote grid is made up of a first cluster, which is the cluster described in the second example, and of a new cluster. The latter is made up of another frontal node, similar to the one of the first cluster, and of several diskless compute nodes which are interconnected in another network by a faster interconnection technology. These compute nodes also have different instruction sets and better memory latencies. Grid information system and job manager are running on the new frontal node.

In addition to different ISA attribute values between the processors of the newer and of the older compute nodes, the heterogeneity of the grid is abstracted by the different latency values of the routes. When the memories and the processors are aggregated into the same hosts, the latencies of the routes between them are memory latencies; and when they are not aggregated into the same hosts, the route latencies are the network ones. Also, the resource abstraction abstracts the fact that the compute nodes of one cluster are not connected to any compute node of the other, and that every compute node of any cluster is connected to both frontal nodes.

The application is a component-based workflow. One component of this application is a workflow engine, which dynamically loads, connects, disconnects, and unloads other components, according to a workflow description received as an input. Basically, components are loaded in processes, these processes being

component containers. Binary files for components and containers respectively are shared library and executable files. The application also uses a message-passing parallel application, which has been shipped as a component in order to be connected with other components. The application manipulates data stored in a database component, which reads from and writes to a large file given as a component property. This workflow description is diamond-shaped, where the last task is the parallel application (workflow engine and database not being parts of this description). In this example, the common component containers can run only on processors having newer instruction sets.

The application abstraction connects some processes with precedence arcs, according to the flow of components. The heterogeneity between common components and the parallel one is abstracted by different attribute values of both the process technology and ISA. The workflow engine and the database components have to run on nodes that all compute nodes can reach: the frontal nodes. The parallel component requires compute nodes in the same network. For other components, the only constraints are that their implementations have to be compatible with the nodes they are mapped on.

When the user asks for deployment, the application connector calls the application translator and this latter calls application abstracter to abstract only the workflow engine component. When this engine is up and running, it is given a reference to the application connector, and the workflow description. The engine calls the application translator, and so on and so forth. When the engine notices that all but the latest task in the flow have completed, it asks to deploy the parallel component. Things happen as in the first and second examples: parts are still running when other parts are asked to be deployed (first example), and deploying a parallel application is detailed in the second example.

5 Conclusion

This paper has presented ODD/SAMURAAIE, a model-based architecture for automatic on-demand deployment. It targets the establishment of a well defined model to enable advanced programming models, possibly incorporating multi-technology applications, to easily and automatically be deployed on grids. Its originality lies in its support of dynamicity while being generic: it does not enforce a particular programming model. The model has been validated on three kinds of applications with the ANGE prototype. It turns out it was straightforward to write all translators and connectors. The simulator correctly generates the graph and the actors behaves as expected. Hence, it is able to support MPI as well as Salome applications.

Future works include the connection of ANGE to concrete resource management services and to integrate scheduling results. While ODD/SAMURAAIE proposes a generic model to deal with complex programming models and resources, it requires smart mapper and adviser to behave efficiently. Moreover, as the model tolerates faults, it would be interesting to also link ANGE to an existing fault tolerant system.

References

1. Anjomshoaa, A., Brisard, F., Drescher, M., Fellows, D., Ly, A., McGough, S., Pulsipher, D., Savva, A.: GFD.136: Job submission description language (JSDL) specification, version 1.0. Open Grid Forum (sep 2008) REC.
2. Ribes, A., Caremoli, C.: Salome platform component model for numerical simulation. *Computer Software and Applications Conf., Annual Intl.* **2** (2007) 553–564
3. Cluster Ressources: Maui cluster scheduler. <http://www.clusterresources.com>
4. The Condor Team: Directed acyclic graph manager (web site) <http://www.cs.wisc.edu/condor/dagman/>.
5. Gil, Y., Ratnakar, V., Deelman, E., Mehta, G., Kim, J.: Wings for pegasus: Creating large-scale scientific applications using semantic representations of computational workflows. (2007) 1767–1774
6. The OSCAR team: Oscar <http://svn.oscar.openclustergroup.org>.
7. Burgess, M., Ralston, R.: Distributed resource administration using cfengine. *Softw., Pract. Exper.* **27**(9) (1997) 1083–1101
8. Toft, P., Loughran, S.: Configuration description, deployment and lifecycle management working group (cddlm-wg) final report. Technical report, HP (2008)
9. Goldsack, P., Guijarro, J., Lain, A., Mecheneau, G., Murray, P., Toft, P.: SmartFrog: Configuration and automatic ignition of distributed applications. In: HP OpenView University Association conf. (HP OVUA), Genève, Suisse (July 2003)
10. Leduc, J., Peaquin, G.: Kadeploy <http://kadeploy.imag.fr>.
11. Lange, T.: Fully automated install <http://www.informatik.uni-koeln.de/fai/>.
12. Morin, C., Lottiaux, R., Vallée, G., Gallard, P., Utard, G., Badrinath, R., Rilling, L.: Kerrighed: A single system image cluster operating system for high performance computing. In: Euro-Par. (2003) 1291–1294
13. Jeanvoine, E., Rilling, L., Morin, C., Leprince, D.: Using overlay networks to build operating system services for large scale grids. In: ISPDC. (2006) 191–198
14. Laforenza, D.: Xtreemos: Towards a grid-enabled linux-based operating system. In: BNCOD. (2008) 241–243
15. Kannan, S., Roberts, M., Mayes, P., Brelsford, D., Skovira, J.F.: Workload Management with LoadLeveler. IBM redbook.
16. Capit, N., Costa, G.D., Georgiou, Y., Huard, G., Martin, C., Mounié, G., Neyron, P., Richard, O.: A batch scheduler with high level components. (2005) 776–783
17. participants, D.W.G.: DRMAA Interface Specification
18. Cluster Ressources: Torque ressource manager. <http://www.clusterresources.com>
19. SUN: Sun grid engine <http://gridengine.sunsource.net/>.
20. Globus Alliance: Globus toolkit. <http://www.globus.org/>
21. Caron, E., Desprez, F.: DIET: A scalable toolbox to build network enabled servers on the grid. *Intl. Jour. of High Perf. Computing Applications* **20**(3) (2006) 335–352
22. Argonne National Laboratory: Mpich2 <http://www-unix.mcs.anl.gov/mpi/mpich2>.
23. Cunin, P.Y., Lestideau, V., Merle, N.: Orya: A strategy oriented deployment framework. In Dearle, A., Eisenbach, S., eds.: *Component Deployment*. Volume 3798 of *Lecture Notes in Computer Science.*, Springer (2005) 177–180
24. Fliissi, A., Dubus, J., Dolet, N., Merle, P.: Deploying on the Grid with DeployWare. In: *Proceedings of the 8th International Symposium on Cluster Computing and the Grid (CCGRID'08)*, Lyon, France, IEEE (may 2008) 177–184 Rank (CORE) : A.
25. Lacour, S., Pérez, C., Priol, T.: Generic application description model: Toward automatic deployment of applications on computational grids. In: *6th IEEE/ACM International Workshop on Grid Computing (Grid2005)*, Seattle, WA, USA, Springer-Verlag (nov 2005)