An ASM-based Model for Grid Job Management

Alessandro Bianchi, Luciano Manelli and Sebastiano Pizzutilo
Department of Informatics, University of Bari, via Orabona, 4, 70125 - Bari - Italy
E-mail: alessandro.bianchi@uniba.it, luciano.manelli@gmail.com, sebastiano.pizzutilo@uniba.it

Keywords: grid systems, OGSA services, asynchronous distributed Abstract State Machine.

Received: July 11, 2012

Job Execution Management Services in Grid systems are generally implemented using specific Grid middleware, and they are considered very critical for the success of the entire system. In order to better manage complexity and criticality, literature suggests the use of robust formal models to describe and analyze these services. This paper abstracts strategic services in Grid Systems, proposes an Abstract State Machine-based model to design them, and implements them by the coreASM tool. The obtained results lead to consider the usage of Abstract State Machine models as a concrete control appliance for Grid systems.

Povzetek: Avtorji predlagajo novo orodje coreASM, s katerim upravljajo storitve mreže.

1 Introduction

A Grid system is built to allow researchers in different domains to use multiple heterogeneous resources for problem solving in high performance computational infrastructure. It provides efficient use of multiple machines for executing jobs, in particular for parallel applications, which can distribute the parallel units of their work across the allocated resources through parallel execution. To this end, a Grid system is defined as a set of independent, distributed and cooperating computational units, which are able to achieve a particular computational goal in dynamic multi-institutional Virtual Organizations [1], [2].

Several organizations have proposed standards for requirements, architecture, specification and services for Grid systems. Among them, the Globus Project (now Globus Alliance [3]) proposed the Open Grid Service Architecture (OGSA) in 2006: it provides a set of important requirements and considers several services needed to support Grid systems and applications [4]. In particular, the requirements for *Execution Management Services* (EMSs) deal with managing the execution of jobs during their lifetimes by searching remote resources for scheduling and executing jobs, and by returning output to users.

1.1 Motivation

Grid systems represent fundamental assets in large-scale scientific and engineering research, but several difficulties are encountered in building reliable Grid applications, mainly because of the high complexity of these systems, [24], [25], and because of the lack of proper conceptual framework and supporting tools [26].

These issues are highlighted in [40], where authors state the huge gap between the proposed architectures usually expressed by informal diagrams, and the existing implementations, so concluding that "a good conceptual reference model for grid is missing".

The lack of precise specification heavily impacts EMSs that are generally implemented using specific middleware. Practically, Grid middleware supports the development of service-oriented computing applications and provides services with the aim to lead users to access remote distributed resources and run jobs on them. Different organizations developed different Grid middleware, sometimes implementing the same proposal. For example, two of the most popular middleware are *Globus Toolkit* [3] and *gLite* [5], both obtained by OGSA proposal and both used in several Grid applications. Unfortunately, a uniform coordination of different software is difficult or impossible across two different middleware, so, in the example above jobs originated on Globus Toolkit cannot be forwarded to gLite. According to [39], the lack of this uniform coordination results in different, often incompatibles interface between middleware and services, and drives to non-interoperable "Grid islands" that cannot share resources.

So, a challenge in realizing efficient Grid systems is the development of a specific middleware for the management and the execution of jobs in different environments. The difficulty increases when different Grid architectures are taken into account. According to [20], the need to give clear, formal definition of Grid systems architecture and services is an urgent task that can help in solving these problems.

1.2 Purpose of the work

Our research contributes to execute this "urgent task" by proposing a formal model of Grid system services. Similarly to [40], we do not propose a new architecture, but a formal executable model that precisely describe logical modules in EMSs, with the aim to better understand and improve analysis of job EMSs. Thanks to the abstraction provided by formalism, the system is expressed focusing only the main requirements and not

considering specific implementation issues. In this way the model strictly defines the overall organization and specifies the services provided, without imposing formats for accessing services. Note that the present paper does not face the possible incompatibilities among existing resources, but this is one of the future directions of our research.

The preliminary study [6] examined the lifetime of a job in OGSA Grid architecture and described a simple model for its management and execution. The present paper moves from that lifetime for formally modeling *execution management* components through an approach based upon Abstract State Machine (ASM) [7], [34], and for implementing it using the *coreASM* tool [8], [9].

The *coreASM* provides an executable language and a tool environment for high-level design and experimental validation of ASM models by supporting execution of ASM specifications. It consists in a platform-independent engine implemented as a Java component. Moreover, a graphical interactive environment in form of a plug-in for Eclipse platform provides interactive visualization and control. Finally, the Control State Diagram editor (CSDe, another plug-in for Eclipse [38]) provides an abstract structure for a first simple design of the system in form of diagrams. In this way, our investigation enlarges knowledge and experience in formalizing Grid systems using ASM, and the obtained results represent a first step in the development of a benchmark for formally analyzing alternative implementation of Grid services and architectures.

Next section overviews the literature concerning formal models applied to Grid computing. Section three provides the background on ASM-based approach. Section four designs and models Grid job EMSs. Section five reports on *coreASM* model implementation and on its execution in some typical cases for validating the model. Conclusion and future work are discussed in Section six.

2 Related work

A rich literature deals with the application of formal methods to the design of Grid systems architecture and services, to their verification and validation, and to the analysis of the most critical properties, which can heavily affect the efficiency of provided services. Examples are represented by π-calculus [35], used in [36] for formally specifying dynamic binding and interactive concurrency of Grid services, and by Z notation [28] and Hoare's Communication Sequential Processes - CSP [29], both adopted in [27] for modeling an identity management architecture for accessing Grid systems. Note that these works adopt the formalism for facing just one specific issue, and they do not generalize the approach to the entire system: they are optimal solutions to specific problems, but do not represent a general-purpose model for EMSs. Conversely, our research is aimed at providing a formal framework for specifying Grid services: in fact, in this work we focus on the Job EMS, but the approach we adopt can be easily extended form modelling the entire system.

Petri nets are adopted in researches with wider purposes. For example, composition of applications in a Grid is orchestrated into a unique Petri net-based workflow in [30]. Even though this paper provides high-level access to Grid services, it operates only on one middleware, and interoperability among different middleware is not taken into account.

More general approaches are presented in [31], [32], [10], and [40]. Guan and colleagues ([31]) propose the design and prototype implementation of a scientific Grid infrastructure using a Petri-net-based interface; OGSA resource management and task scheduling is extended by Liu and colleagues ([32]) using Timed Petri nets for allowing services definition; a Colored Petri net-based model is depicted by Zhao and colleagues ([10]) for studying job scheduling problems; Colored Petri nets are also used by van der Aalst and colleagues ([40]) for specifying and validating a reference model for Grid architectures.

All these examples show that Petri nets are effective in designing and analyzing the workflow structure and services provided by the Grid and for studying typical properties of the system, e.g. liveness. Unfortunately, Petri nets present some disadvantages. Storrle and Hausmann [11] summarize Petri nets problems in modeling behavior of exceptions, streaming and traverse-to-completion, and they discuss problems arising when trying to analyses these advanced features. Sarstedt and Guttmann [12] underline that Petri nets are not adequate and intuitive to describe the semantics of many system diagrams, also due to the lack of a unified formalism. Moreover, Eshuis and Wieringa [37] show that Petri nets are suitable for modeling closed and active systems rather than open and reactive/proactive systems, like Grid. Analogously, Atlas and colleagues argue that PN-based approaches are fairly rigid and are not suitable for dynamic environments [41].

For these reasons, we consider ASM approach [34] more suitable for modeling Grid architecture. This choice is justified by different issues. Several similarities exist between Petri nets and ASM and [7] shows that the run in a Petri net can be expressed by ASM rules, in particular it emphasizes the capability of asynchronous distributed ASMs in modeling the distributed nature of Petri nets. The capability of ASM in describing agent based models allows us taking into account the entire system in both its static and dynamic aspects, through a set of ASMs implementing an asynchronous distributed ASM.

Resuming, we applied ASM approach considering the advantages it provides under three different points of view. When the model expressivity is considered, a rich literature (e.g. [9], [11], [13]) agrees that ASMs show versatility in modeling complex architectures and they have excellent capabilities to capture the behavioral semantics of complex, dynamic, open, and reactive systems, like Grid, where several different processes occur, often with the need to properly model exceptions, streaming and traverse-to-completion. Secondly, considering software engineering development issues, it is worth noting that, starting from the ASM formalism, a

development process has been defined and successfully applied in several complex domains, e.g. telecommunication, programming languages, control systems, and so on [7]. Finally, considering the implementation point of view, the lack of specific environments for translating PN models in executable code is overcome by using an ASM-based approach thanks to tools like AsmL [33] and *coreASM* [8], [9], [22]. Both AsmL and *coreASM* are high-level executable languages based on ASM: the main difference is that the former is integrated with Microsoft .NET platform, the latter with the Eclipse platform.

The existing literature, which applies ASM to Grid systems, encourages our choice. Several papers use ASM to model communication systems behavior and to control their management. A general, abstract communication model for studying message-based communication networks in distributed systems is presented in [14]: the model is implemented and tested using *asmL* environment.

Lemcke and Friesen propose in [15] a composition algorithm of web services defining the execution of business processes and orchestrations by providing ASM representations for these processes and executing them with *coreASM* . Lamch and Wyrzykowski [16] develop a software environment in *asmL* implementing a hybrid approach for integration of a formal model and existing middleware components: the model-based testing approach is useful for investigating properties of specification of Grid middleware using ASM. Both these papers focus on integration issues, but they do not provide a comprehensive view of Grid systems.

In [17], [18], authors propose a semantic model for Grid systems and for traditional distributed systems describing differences between the two systems. They used ASM at a very high level for describing resources and users belong to a Virtual Organization (VO) without developing any tools for simulation. In a different way, [19] analyzes the similarities between distributed and Grid systems, the characteristics and the requirements of Grid systems and programming models. They develop an ASM model and study the validity in Grid environments. Finally, in [20] Zou and others propose a general framework for Grids and model the virtual organization based on ASM, considering quality issues for the user requirements. With respect to these papers, our work models the system at a lower abstraction level and validates its implementation.

In [6], a preliminary, monolithic ASM was built to model the standard mechanisms defined in OGSA for job EMSs. That paper informally details the general description of job EMSs, provides an overview of the ASM-based model, and statically analyzes it. The present work improves that paper, mainly with respect to two issues: firstly, the model is structured in a number of agents, each modeled by an ASM, so obtaining a more realistic Distributed ASM. Secondly, the model is implemented using *coreASM*, and dynamically validated in some typical scenarios.

3 Background on ASM

Abstract State Machine, ASM for short, is a powerful computational model [34], which has been successfully applied in several cases for modeling critical, complex systems, both in industry and in academia: wide discussion about ASM application is in [7]. It simulates every algorithm's behavior through a step-by-step way: each step computes a set of updates with given transition rules. After the completion of a step, all updates are committed simultaneously.

The concept of *abstract state* in ASM extends the usual notion of *state* occurring in Finite State Machines: it is an arbitrarily simple or complex structure, i.e. a collection of domains, with arbitrary functions and relations defined on them. On the other hand, *rules* are basically nested if-then-else clauses with a set of functions in their bodies. In each state, all conditions guarding rules are checked, that is all rules whose conditions are evaluated to true are simultaneously executed, so determining the state transition. All the parameters are stored in a set of so called *locations* and at each time the particular configuration of parameters values determines the current state of the ASM.

The transition from one state to another is described through a set of formulas, in the form:

$$\{\text{if } condition_i \text{ then } updates_i\}_{\ i=1,..,n}$$

where each $condition_i$ (the guard of the i-th rule) is an arbitrary first-order formula, whose interpretation can be true or false, and each $updates_i$ is a finite set of assignments

$$f(t_1,\ldots,t_n) = t$$

whose execution is to be understood as changing the value of the function f at the given parameters t_i, that leads to a new value of the parameter t. The parameters are stored in a set of *locations* and the configuration of parameter values at each step determines the current *state* of the ASM.

For the unambiguous determination of a next state, it is necessary that updates are *consistent*. An update set is consistent if it contains no pair of updates which assign different values to the same location; otherwise, the update set is inconsistent.

The concepts related to modeling monolithic systems through ASMs can be extended to distributed systems. The case of job management in Grid systems is a typical case of *asynchronous distributed systems*. It is *distributed* because its overall behavior is the composition of different, independent, remote elements, each operating on its own. It is *asynchronous* because all involved logical components operate and communicate concurrently, each according to its internal behavior. So, job EMSs in Grid can be modeled by a *Distributed Asynchronous ASM - asyncASM* [7], [21].

Essentially, *asyncASM*s generalize simple ASMs to an arbitrary finite number of independent agents [23], each executing an ASM in its own local state. Formally, an *asyncASM* is given by a family of pairs *(a, ASM(a))* of pairwise different agents, elements of a possibly dynamic finite set *Agent*, each executing its ASM, *ASM(a)*. In this sense, each agent *a* executes its own program, operating

on its own states, so determining a partial view of the entire system. The relation between global and local states is supported by the reserved keyword *self*, used to denote the specific agent executing a rule, and to store information relevant to itself. A new agent can be introduced into the *asyncASM* at any time by extending the set *Agent*.

An ASM-based process for developing complex systems is presented in [7]: it allows capturing system model requirements and refining them through intermediate models to any desired level of detail in a validated and verifiable code. In the present work, modeling and implementation activities have been carried out with the support of the *coreASM* framework [8], [9], [22]. It follows mathematical definition of ASMs and inherits several typical features of the ASM modeling. Its main purpose is to make ASM-based models executable. To this end, the framework includes some language constructs aimed at making easy the development, as, for instance, *forAll*, which allows executing all rules satisfying a given guard condition; *choose*, aimed at expressing non-determinism in the choice of a rule to execute when a condition is satisfied; *seqblock/endseqblock* are the delimiters of block, whose rules must be executed sequentially; *par/endpar* are the delimiters of block, whose rules must be executed concurrently; *enqueue/dequeue* are the operators for adding / removing elements to a queue.

4 Modeling the Grid job EMSs

In order to model the job EMSs, firstly the Grid capabilities and features are informally described, then they are abstracted and formally defined, finally the *asyncASM* is created in the *coreASM* environment [8].

4.1 Informal description of a Grid system

4.1.1 Services

The OGSA standard describes requirements (interoperability and resource sharing, optimization, quality of service, job execution, data services, security, scalability and extensibility), and considers six important independent services that implement such requirements and are needed to support Grid systems and applications [4]:

- ***Execution Management Services*** manage jobs to completion: they concern job submission, description instantiating, scheduling, and provisioning resources (e.g. RAM, disk, CPU, etc.).
- ***Data Services*** focus on the management, access and update of data resources (e.g. files, streams, DBMS, etc…), and provide remote access facilities, replication services and managing of metadata.
- ***Resource Management Services*** manage physical and logical resources.
- ***Security Services*** provide security-related policy, and manage access for cross-organizational users.
- ***Self-Management Services*** support the reducing of the cost and complexity of operating on an IT infrastructure. They provide self-configuration and self-optimization of system components (e.g. computers, networks and storage devices).
- ***Information Services*** concern the manipulation of information about applications, resources and services, support reliability, security, and performance in a Grid. They provide registry, notification, monitoring, discovering, and logging.

The specific implementation of these services is delegated to Grid middleware. Practically, when a job is submitted, the middleware creates a job manager process for that job. The job manager manages the single job's lifetime, matches job requirements with the needed resources, and controls the relative allocation in different way for different Grid middleware. The allocation consists in assigning and queuing the job to local manager in resources. Note that in the following we only focus Execution Management Services.

4.1.2 Architecture

We assume a Grid is over three levels, operating on distributed heterogeneous resources, namely *Grid Application* level, *Middleware* level, *Resources Pool* level. Note that, this layered view is an abstraction of OGSA proposal, often used in literature, for instance in [40], and it fits our purposes.

The *Grid Application* is a higher-level structure built on top of the architecture that for our purposes is only aimed at partitioning each user job in *jobs* to submit to the lower level. In fact, since it is usually considered that a *job* is the smallest unit managed by a Grid environment, we assume a user submits one or more *User Jobs* (say UJ_1, UJ_2, … ,UJ_n), each composed by one or more *jobs* (say job_{11}, job_{12},…, job_{1k} for UJ_1; job_{21}, job_{22},…, job_{2h} for UJ_2;… job_{n1}, job_{n2}, …, job_{nm} for UJ_n). Note that possible interactions among jobs are managed by Grid Application level, but this issue is outside the purposes of present work.

The *Middleware level* implements fundamental functionality needed by Grid applications and required by OGSA. It interacts with the *Application level*, managing *jobs* until their completions and returning results to users. In this context the EMSs concern searching candidate resource and executing and managing *jobs* until end, so users can transparently execute their *UJ*-s on distributed resources. These tasks are critical because their incorrect execution can heavily affect the quality of provided services [24], so adoption of formal method is useful and sometimes mandatory.

The *Resources Pool level* is characterized by distributed and heterogeneous resources of a Grid. For the purposes of our work *a resource* is a logical entity with specific features needed to job execution, with its local workload and managed by a local resource manager.

For sake of simplicity, the architecture is summarized in figures 1 to 3: Figure 1 is an overview of the three levels: the box labeled "*Grid Application Level*" is outside the purposes of present work, so it is not further detailed. The boxes "*Grid Middleware Level*" and

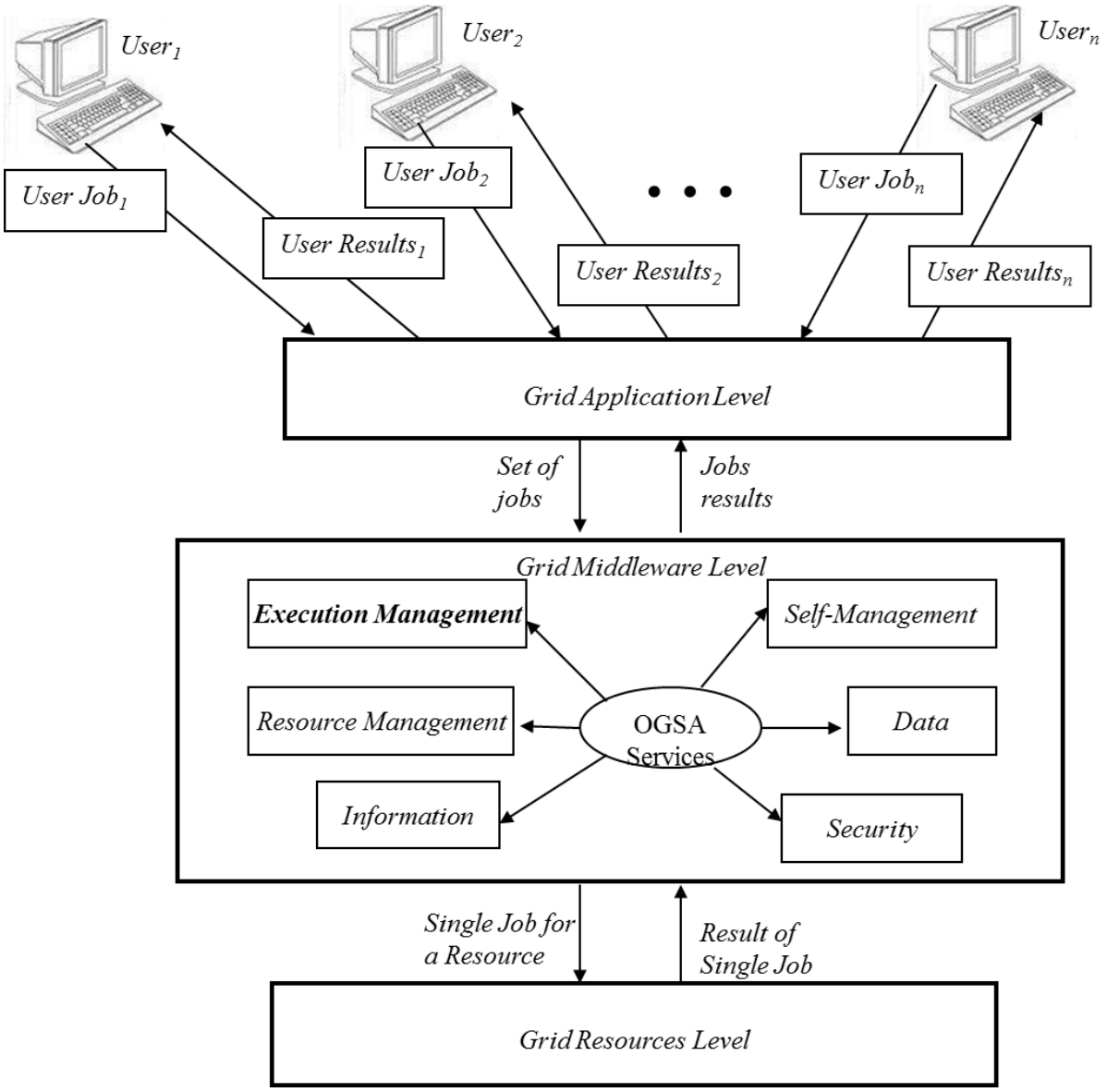

Figure 1: Architecture Overview.

"*Grid Resources Level*" are detailed in Figure 2 and Figure 3, respectively.

The *UJ*-s submitted by users are processed by the *Application level* and each of them is partitioned into composing *jobs*. Next, each *job* is sent to the *Grid Middleware level*, through a queue called *waitingJobs*, according to a First-In-First-Out (FIFO) policy. Moreover, the *Application level* receives the results of *jobs* executions by lower level and it re-composes them so that to obtain the results of UJs to send back to users.

A *job* must be executed on a resource that satisfies all its computational constraints. To this end, the EMSs chooses the proper resource in a shared pool, according to *job* performance requests, allocates the *job* on the selected resource for execution, and controls the *job* lifetime until completion.

In order to achieve these goals, we assume that Execution Management module is composed by two logical components: a *Dispatcher*, and a set of *Job Managers* (Figure 2), each accessed in mutual exclusion. The former schedules execution of each *job* to a *Job Manager*. Therefore, the role of the *Dispatcher* consists in the following activities: extracting the *job* at the top of the queue, searching for a free instance of a *Job Manager*, and activating the *Job Manager* instance on the submitted *job*.

After activation, the *Job Manager* becomes unavailable for other *jobs*. Then, it searches a suitable *Resource* in the *Resource Pool level*, matching the required *job* performance. If a resource is found, the Job Manager assigns it the *job*, waiting for control return, and it manages the lifetime and the state of the *job* until completion. In case of failure, an error is reported. Each *Job Manager* becomes again available for other *jobs* only after *job* completion.

A *Resource* in the *Resources pool* receives the *job* by the *Job Manager*, enqueues the *job* in the *local queue* and processes it, according to the position in the queue.

The set of resources is assumed to be fixed (from *1* to *R*), and during execution resources cannot be added or

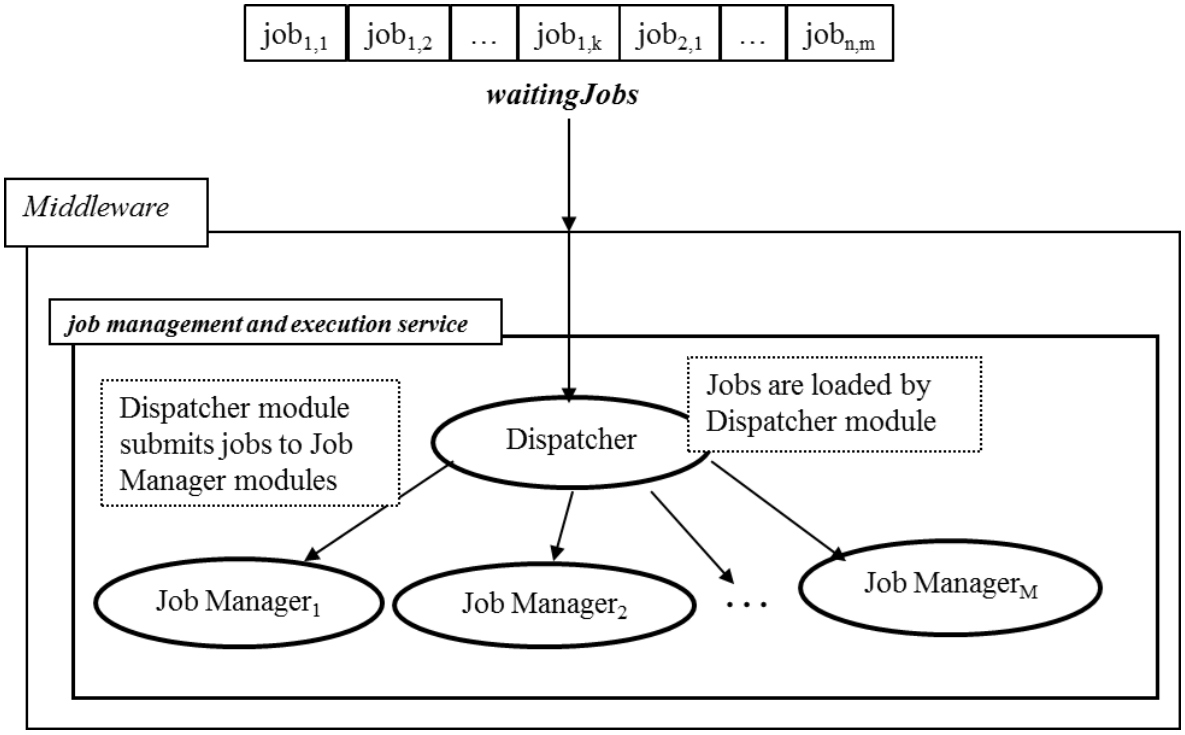

Figure 2: Overview of Grid Middleware Level.

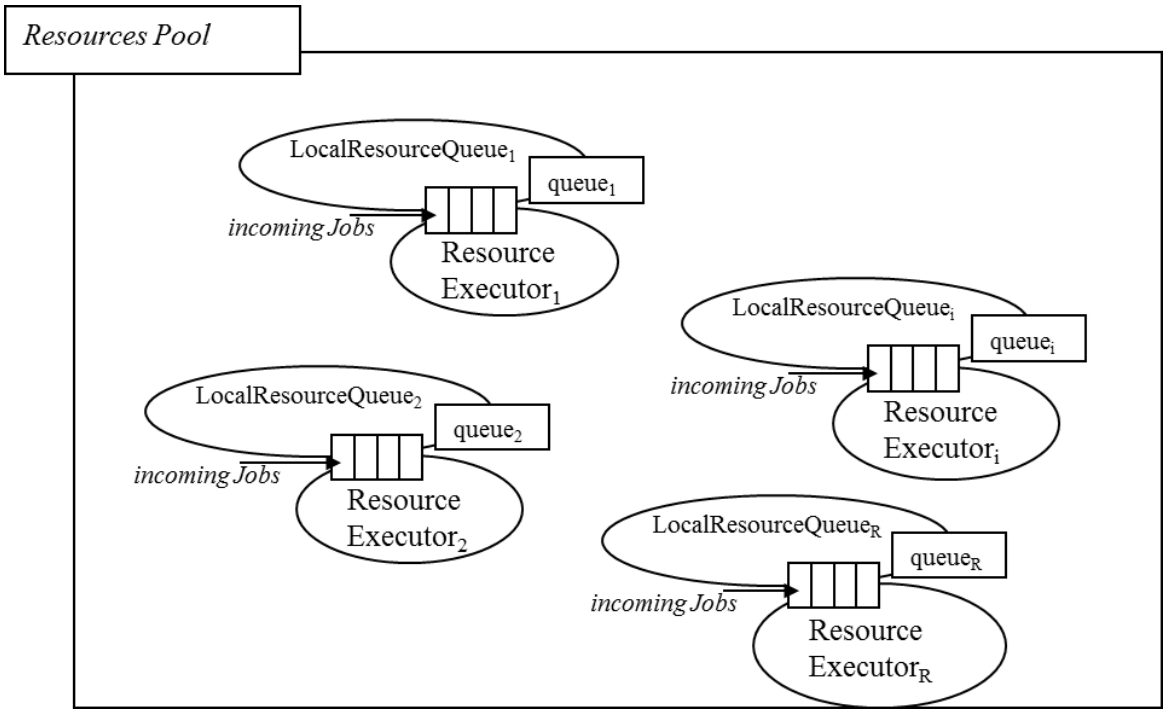

Figure 3: Representation of the Resource pool.

removed. For abstraction purposes, we suppose that a *Resource* computes *jobs* with a FIFO policy. If no resource in the pool is able to satisfy *job* requirements, the *job* fails due to a lack of available resources. Moreover, we assume that a *Resource* is composed by two logical components that work on the same *job*: a *ResourceLocalQueue* module, devoted to enqueuing the incoming *jobs*, and a *ResourceExecutor* module, for *jobs* execution.

If problems occur in resource during *job* execution, the *Job Manager* catches errors and stops the computation. When the execution completes, either successfully or unsuccessfully, middleware sends a message to the user application. In Figure 3 a representation of Resources pool is presented.

As a final remark, it is worth noting that users can cancel their *user jobs* (*UJ*-s) before completion; if so, all corresponding *jobs* are removed by the Grid system.

4.1.3 Abstraction

Informal description above can be abstracted in the following set of requirements:

Req.1 A Grid receives requests for *UJ-s* from clients; each *UJ* is decomposed in a set of atomic *jobs*, and each *job* is queued waiting for service.

Req.2 A *Dispatcher module* in middleware loads a *job*, and sends it to a specific *Job Manager module*.

Req.3 For each *job*, *Job Manager* finds the most adequate resource satisfying *job* requirements and runs it on that resource.

Req.4 That *Resource* enqueues and processes the *job*.

Req.5 If there are no failures, the *job* is completed with success.

Req.6 If errors occur the *job* is aborted.

Req.7 A user can cancel or remove a User Job, so cancelling/removing the corresponding *jobs*.

Req.8 At the end of the computation *Resource* is released.

Req.9 At the end of the computation the result is communicated to the end user.

Moreover, the model of system operations must be able to execute the following actions:

Act.1 The Dispatcher takes the first *job* in the *waitingJob* queue and sends it to the first available instance of Job Manager, if the system is ready.

Act.2 The system needs to find available resources necessary for the computation: if so, resources are reserved, else the *job* is immediately rejected.

Act.3 The *job* sent to a Resource is enqueued in its local queue and when possible the computation starts;

Act.4 The *job* can complete the computation with success and the system traces the result of the computation; if problems arise, the execution fails and the systems returns to the idle state.

Act.5 At the end of the computation, the system resets and returns in a state of inactivity and user is noticed about the result.

Act.6 The user can cancel UJ before execution or remove it before completion.

4.2 The ASM models

Modeling job management services in a Grid system is so reduced to modeling one *Dispatcher Agent*, a set of *JobManager Agents* and a pool of *ResourceExecutor Agents*, each with a *Resource Local Queue Agent*.

Note that the ASMs of the agents will be described separately, but all of them are a unique Distributed Asynchronous ASM - *asyncASM*.

In the resulting *asyncASM* there is one instance of the *Dispatcher ASM*, up to M instances of the *Job Manager ASM*, and R instances for both *Resource Local Queue* and *Resource Executor ASMs*. In real word, M is established by the capability of the middleware, and R depends on the actually available resources.

Figures 4 to 7 show the graphical view of the ASMs modeling Dispatcher, Job Manager, Resource Local Queue and Resource Executor agents, respectively. They are screenshots of the graphical representation of the ASMs, obtained by the ASM Control State Diagram editor (CSDe [38]). According to the usual notation of ASMs, circles represent *states*, diamonds represent *conditions* and boxes represent *rules*. Moreover, we added asterisks "*", "**", "***" to the screenshots for indicating the logical link between Dispatcher and Job Manager, between Job Manager and Resource Local Queue, and between Resource Local Queue and Resource Executor, respectively. For space reasons, in the following each ASM will be described in general and only some parts will be detailed.

After activation, Dispatcher ASM (Figure 4) goes in "IDLEDISP" state. When one or more *jobs* are in *waitingJobs* queue, the condition labeled "OneJob" evaluates to true, so the "LOAD" rule is activated. As a result, the *job* at the top of *waitingJobs* queue is loaded in Dispatcher. Then the ASM waits for the availability of at least one JobManager. The ASM stops waiting when the value of the location parameter *inactive* for a JobManager JM_i evaluates to true, for some *i*, i.e., JM_i is not currently processing, so it is available for executing that *job*.

Then the rule "ACTIVATION" is fired: this results in activating JobManager JM_i, so changing the value of the location parameter *inactive* to false, the state of the Dispatcher ASM evolves to "JOBSUBMITTED", and the Job Manager ASM (Figure 5) can start its execution. The computations of both Dispatcher ASM and Job Manager ASM then continue asynchronously: Dispatcher ASM checks if more *jobs* are waiting; meanwhile, Job Manager ASM processes the *job*.

After activation, if user does not cancel the *job*, the Job Manager ASM checks whether a proper resource, which satisfies *job* requirements exists in resource pool. To this end the condition labeled "exists resource in Resource" is evaluated. If the condition is not satisfied, then the *job* is rejected and the run stops, otherwise, "ACCEPT" rule is executed. The result of rule is assigning *job* execution to one of the available proper resources, selected in a non-deterministic way using the "choose" construct of *coreASM* language, and the location parameter *submitJobToResource* is set to the couple <identifier of chosen resource; identifier of job>.

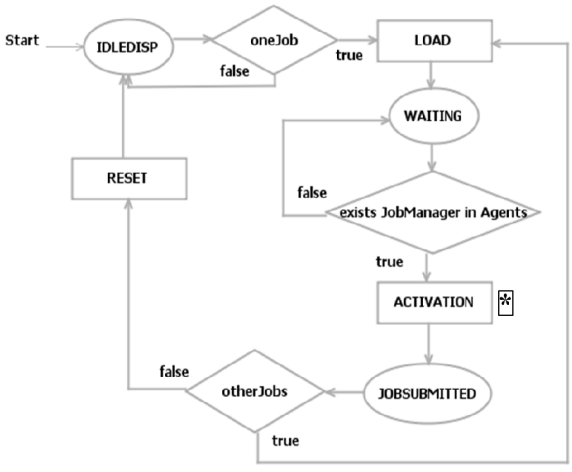

Figure 4: Representation of the Dispatcher ASM.

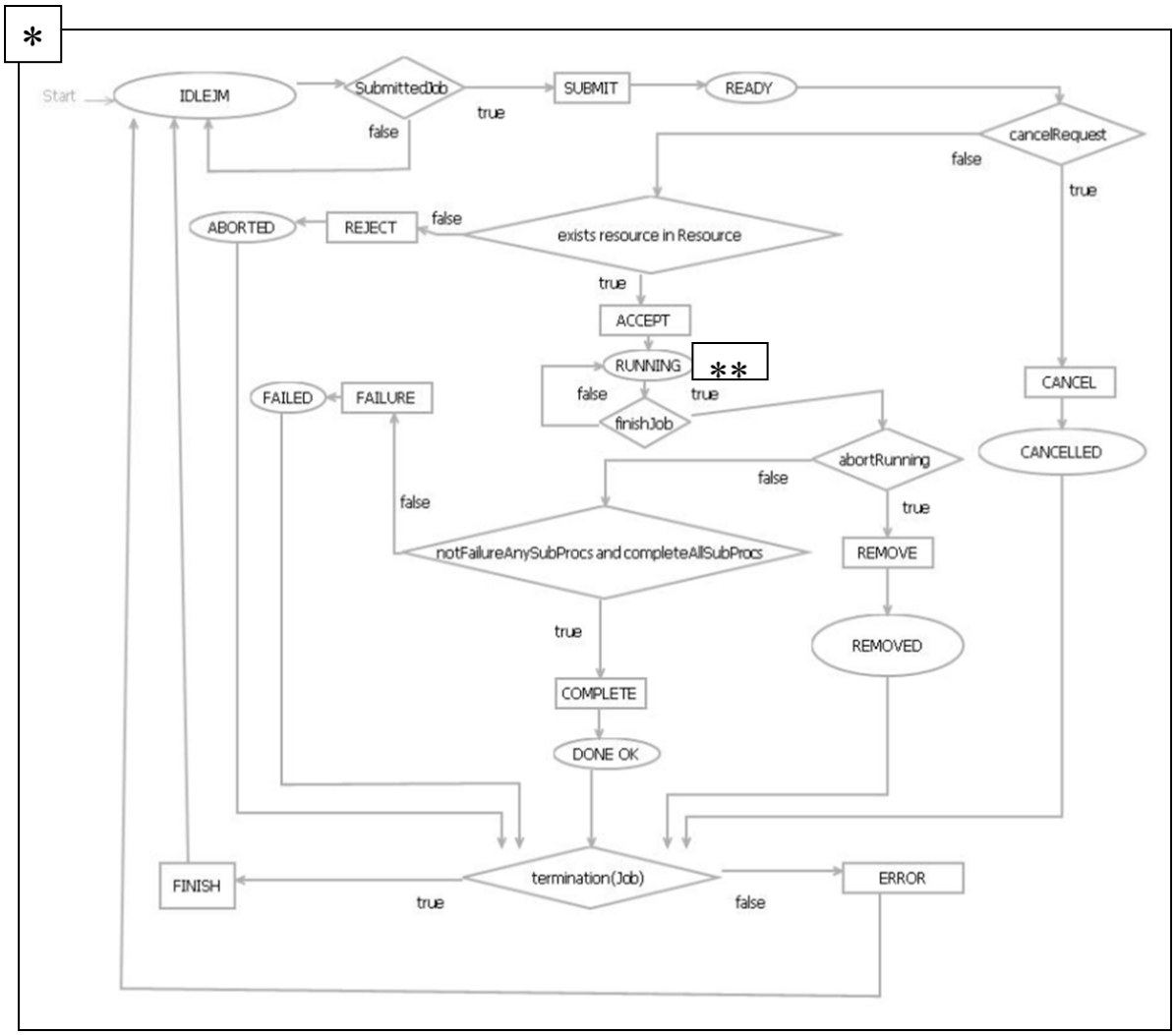

Figure 5: Representation of the Job Manager ASM.

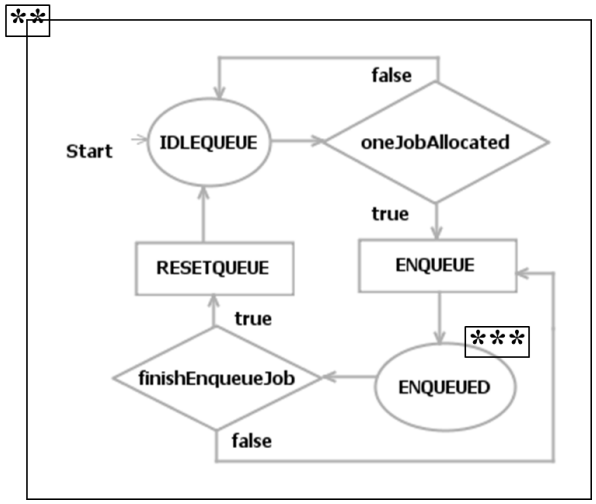

Figure 6: Representation of the Resource Local Queue ASM.

After execution of this rule, the Job Manager ASM evolves in "RUNNING" state, and the Resource Local Queue ASM (Figure 6) can start its execution, asynchronously with respect to Job Manager ASM.

The condition labeled "OneJobAllocated" in Resource Local Queue ASM is satisfied when the location parameter *submitJobToResource* establishes the relation between the resource and a *job*. Therefore, after activation of Resource Local Queue ASM, this condition evaluates to true and the rule "ENQUEUE" fires. This rule sets to true location parameter *jobAllocated*, so the condition labeled "OneJobInQueue" in the Resource Executor ASM (Figure 7) becomes true, and therefore the rule "SCHEDULE" can be executed. Next, Resource Local Queue ASM and Resource Executor ASM perform their computations asynchronously.

5 Model implementation and simulation

According to the ASM-based method for designing critical systems presented in [7], the modeled behavior is validated through simulation. So, after implementation, the ASMs shown above have been executed in some typical scenarios. Note that simulations are aimed at validating the main functionality, without the purpose to verify the correct behavior of the whole system.

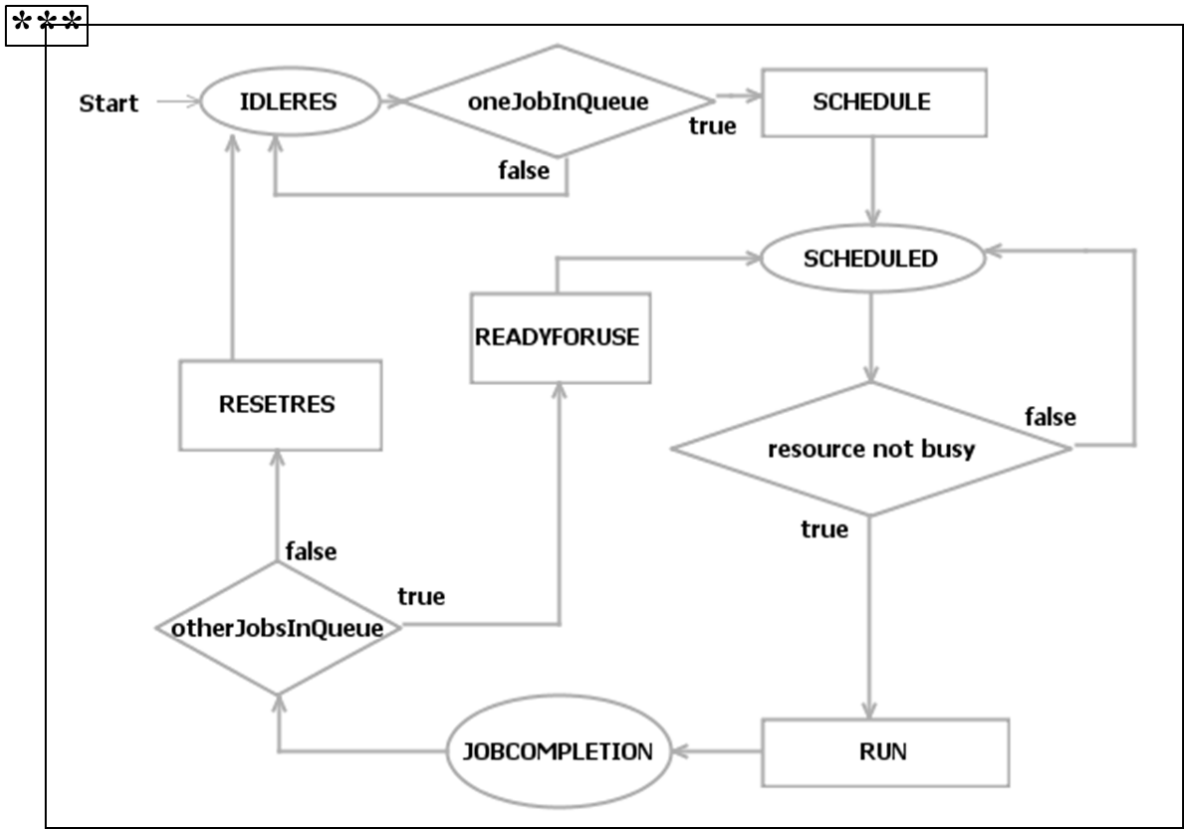

Figure 7: Representation of the Resource Executor ASM

5.1 Implementation

The executable code of the modeled ASMs, have been obtained through a 4-steps process. In the first step each ASM is edited using the Control State Diagram editor (CSDe [38]), an Eclipse plugin for creating and modifying ASMs and translating them into *coreASM* specifications. The second step is the production of the *coreASM* specification for every ASM. It is automatically executed by the CSDe. In the third step the four files obtained so far, one for each ASM, are merged in a unique *coreASM* file, which specifies the unique *asyncASM* modeling the system. Finally, the specific behavior of the obtained *asyncASM* is manually customized by the programmer by adding instructions for rules and conditions.

The entire process was executed in about four days by one person, without any previous experience in *coreASM* language. The step that consumed more time was the last, which required about three days. Note that the few time spent for designing the ASMs during the first step is due to the previous study of the model, which required much more effort.

The result of the process is one file, about 650 lines long: about 300 were automatically generated and about 350 manually produced.

5.2 Simulation setting

After implementation, the model has been simulated in five typical scenarios.

Scenario 1 is the standard ideal scenario, in which the Grid system is able to process all *jobs* submitted, all resources are available for all submitted *jobs*, and no user stops the submitted UJ. It is expected that each user receives the result concerning the computation of the submitted *job*.

In **Scenario 2** the system is able to process all *jobs* submitted, but constraints for some of them are not satisfied by any resource. Moreover, in this scenario the users do not stop computation before end. It is expected an error message to users that submitted unsatisfied *jobs*.

Scenario 3 is analogous to the first one, i.e. the system is able to process all submitted *jobs*, and all resources are available for all submitted *jobs*, but one of them (say, *stopped_job*) is stopped by an explicit user request, so a message confirming *job* deletion due to user action is expected.

Scenario 4 simulates the behavior of the Grid when the number of submitted *jobs* is greater than the total number of available Job Managers. Moreover, in this scenario the users do not stop the computations. It is expected that some *jobs* are not dispatched to a *Job Manager Agent* as well as they are submitted, because all *Job Manager Agents* are busy, so their execution must be delayed; nevertheless they should be correctly processed later.

Scenario 5 is aimed at validating the behavior of the Grid when only one resource satisfies the constraints for the submitted *jobs*. So, it is expected that if the users do not stop computations, all the submitted *jobs* are queued and they are processed according to arrival order.

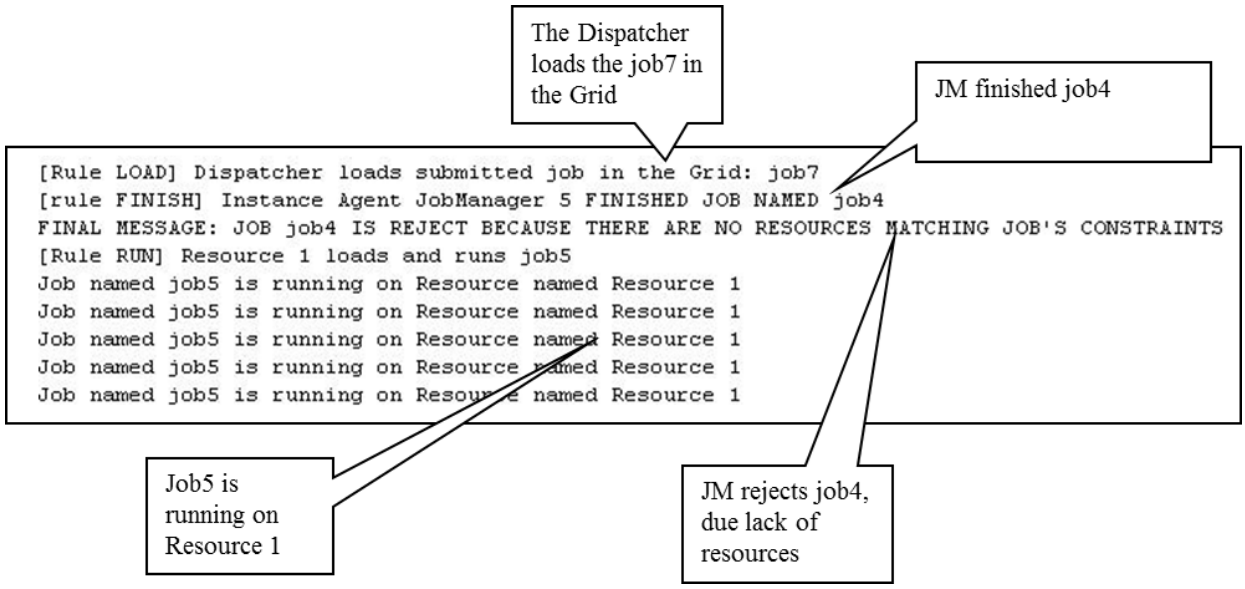

Figure 8: Console Output for Scenario 2.

For each scenario, ten simulations have been executed, and in all cases the Grid system is composed of one instance of Dispatcher ASM, 10 instances of Job Manager ASM, and 10 instances of both Resource Local Queue ASM and Resource Executor ASM. In scenarios 1, 2, 3, and 5 the number of *jobs* queued in the *waitingJob* is variable, but always lower than the total number of Job Managers. Instead, in scenario 4 the number of *jobs* is always greater.

5.3 Simulation execution

For each scenario, after setting the initial condition, the computation executed by the involved ASMs have been observed, looking at both the locations at each step, and the output shown in *coreASM* console. An example of the *coreASM* console output produced during execution of scenario 2 is in Figure 8.

For all scenarios, the *Dispatcher Agent* activates a *JobManager* instance for each submitted *job*, then the *JobManager Agent* searches for an adequate resource matching job constraints. During simulation of scenarios 1 and 3, all needed resources are found and reserved for *job* execution. Then, in scenario 1, each resource processes its *job* until completion and finally the expected message is sent back to each user. Instead, in scenario 3 the *JobManager* associated to the *stopped_job* ends its own execution when the guard condition *abortRunning* is encountered. In this case, *stopped_job* is removed by the *JobManager Agent*, and a message confirming deletion is correctly sent to the user.

Execution of scenario 2 shows that due to the lack of proper resources for some *job*s, they cannot be satisfied. In other words, some *JobManager Agents* fail in finding adequate resources, and their *job*s are rejected. The final messages for these *job*s correctly show the failure.

In scenario 4 some *job*s correctly wait for *Job* Manger availability; when a *Job Manager Agent* becomes ready, it accepts the *job* at the top of the queue. The final message sent to users correctly shows the result of *job* execution.

In scenario 5 all submitted *job*s are correctly enqueued, each waiting for the availability of the resource, and they are all correctly executed, according to their arrival order.

Therefore, in all cases we executed, the model correctly evolves according to the expected behavior, and, after completion all resource are correctly released, so becoming available for processing new *job*s, and the agents return to the idle state.

6 Conclusion and future work

Grid technology makes available a great extent of computational power for solving many application problems with acceptable resource consumption. In this context, the specific middleware adopted for executing Users Jobs is a very critical requirement, which can affect success of the system. Its high complexity requires the use of formal methods for guaranteeing correct behavior within required quality of service. This paper is part of our research aimed at building a formal framework for studying Grid systems. Since ASMs have proven their practical benefits for the specification and analysis of several complex systems, we apply this formalism in Grid systems domain.

Here, we provide a formal description of the job Execution Management Services in terms of *asyncASM*s, and its implementation into *coreASM* tool. Job EMS is expressed as a composition of interoperable, always refineable, building blocks, and the resulting *asyncASM* model is an effective choice for defining a precise semantic foundation of Grid system. This solution allows coordination among different logical components in the Grid.

The simulation-based validation of the model provides an informal evidence of requirement satisfaction, and it makes possible a preliminarily analysis of some system properties. For example we can

see that each state can be reached starting from the initial idle state, and that all rules can always be fired, so the modeled system is deadlock free. Moreover, we can observe that it is always possible returning to the initial state, so that it is always possible implementing a proper recovery procedure in case of failures. Finally, it is worth noting that the Dispatcher Agent can be a bottleneck for the system, because it has to manage a lot of *job*s.

The ASM approach can be seen as a reference model for Grids studies: it can help both researchers and practitioners to better understand Grid behavior, to clarify concepts at the abstract desiderate formal level, to improve the efficiency and reduction of development costs, and to compare different solutions. In fact, thanks to the abstraction process, and to tools like *coreASM*, it is quite easy building an implementation of the model, spending few efforts, and different Grid strategies, depending on different middleware, can be derived as different refinements of the same abstraction. In this way, researchers can verify and validate the behavior of solutions they propose, and practitioners can easily compare the implementation of different proposals.

Future development of research is aimed at a twofold goal: on one hand, the model will be completed for encompassing proper management of some side aspects, for example resource allocation policy, and management of a pool of Dispatchers. On the other hand, since ASM-based approach enables the analysis of the model for evaluating computationally interesting properties, the obtained models will be analyzed for identifying possible weaknesses. In this sense, it can be challenging for researchers and practitioners investigating how the ASM models can help the interoperability and the standardization of Grid systems achieving optimal performance and reduction of costs.

Acknowledgement

This work has been partially funded by the Italian Ministry of Education, University and Research (MIUR), within the Piano Operativo Nazionale - PON02_00563_3489339.

The authors are very grateful to the anonymous reviewers for their constructive remarks and comments.

References

[1] I. Foster, C. Kesselman, and S. Tuecke, “The anatomy of the Grid: Enabling scalable virtual organizations”, *International Journal of High Performance Computing Application*, vol. 15, no.3, pp. 200-222, 2001.

[2] I. Foster, “What is the Grid? A Three Point Checklist”, *Global Grid Forum*, Available: http://www.globus.org/alliance/publications/papers.php, 2002.

[3] Globus Alliance – Globus Toolkit – www.globus.org

[4] I. Foster, I., Kishimoto, H., Savva, A., Berry, D., Djaoui, A., Grimshaw, A., Horn, B., Maciel, F., Siebenlist, F., Subramaniam, R., Treadwell, J., Reich, J.V. Reich, “The Open Grid Services Architecture, Version 1.5” , GFD-I.080, *Open Grid Forum*, Available: www.ogf.org/documents/GFD.80.pdf, 2006.

[5] http://glite.cern.ch/

[6] A. Bianchi, L. Manelli and S. Pizzutilo, “A Distributed Abstract State Machine for Grid Systems: A Preliminary Study”, in P. Iványi and B.H.V. Topping (Eds.) *Proceedings of the Second International Conference on Parallel, Distributed, Grid And Cloud Computing For Engineering, Civil-Comp Press*, Ajaccio, France, Paper 84, April 2011

[7] E. Börger, R. Stärk, *Abstract State Machine*, Springer, 2003.

[8] www.coreasm.org

[9] R. Farahbod, V. Gervasi, and U. Glaesser, "CoreASM: An extensible ASM execution engine", *Fundamenta Informaticae*, vol. 77, no.1-2, pp. 71-103, 2007.

[10] X. Zhao, B. Wang, L. Xu, “Grid Application Scheduling Model Based on Petri Net with Changeable Structure”, *Proceedings of the 6th International Conference on Grid and Cooperative Computing*, Los Alamitos, CA, pp.733-736, 2007.

[11] H. Storrle, J. Hausmann, “Towards a formal semantics of UML 2.0 activities”, in *Software Engineering*, Lecture Notes in Informatics vol. P-64, P. Liggesmeyer, K. Pohl, M. Goedicke, Eds., pp. 117-128, 2005.

[12] S. Sarstedt, and W. Guttmann, "An ASM Semantics of Token Flow in UML 2 Activity Diagrams", *Proceedings of the 6th International Andrei Ershov memorial conference on Perspectives of systems informatics*, I. Virbitskaite and A. Voronkov Eds., LNCS 4378, pp. 349-362, 2007.

[13] W. Reisig, "The Expressive Power of Abstract State Machines", *Computing and Informatics*, vol. 22, no.3-4, pp. 1-10, 2003.

[14] U. Glässer, Y. Gurevich, M. Veanes, “Abstract Communication Model for Distributed Systems”, *IEEE Transactions on Software Engineering*, vol.30, no.7, pp. 458-472, 2004.

[15] J. Lemcke, and A. Friesen, “Composing Web-service-like abstract state machines (ASMs)”, *Proceedings of the IEEE Congress on Services*, Salt Lake City, Utah, pp. 262–269, July 2007

[16] D. Lamch, R. Wyrzykowski, “Specification, analysis and testing of Grid environments using Abstract State Machines”, *Proceedings of the International Conference on Parallel Computing in Electrical Engineering*, Bialystok, Poland, pp. 116-120, September 2006.

[17] Z.Nemeth, V. Sunderam, “Characterizing Grids: Attributes, Definitions and Formalism”, *Journal of Grid Computing*, Vol. 1, no.1, pp. 9-23, 2003.

[18] Z.Nemeth, V. Sunderam, “A Formal Framework for Defining Grid Systems”, *Proceedings of the 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid*, Berlin, pp. 202-211, May 2002.

[19] M.Parashar, J.C. Browne, "Conceptual and implementation models for the Grid", *Proceedings of the IEEE*, vol.93, no 3, pp.653-668, 2005.

[20] D. Zou, W. Qiang, Z. Shi, "A Formal General Framework and Service Access Model for Service Grid", *Proceedings of the 10th IEEE International Conference on Engineering of Complex Computer Systems*, Shanghai, China, pp. 349-356, June 2005.

[21] A. Blass, Y. Gurevich, "Abstract State Machines Capture Parallel Algorithms", *ACM Transactions on Computational Logic*, Vol. 4 no. 4, pp.578-651, 2003.

[22] R. Farahbod, and U. Glasser, "The CoreASM modeling framework", *Software – Practice and Experience*, 41, no.2, pp. 167–178, 2011.

[23] G.M.P. O'Hare, N.R. Jennings, *Foundations of Distributed Artificial Intelligence*, John Wiley & Sons, 1996.

[24] J. Yu, R. Buyya, "A taxonomy of scientific workflow management systems for grid computing", *ACM SIGMOD Record*, Vol. 34, no.3, pp. 44-49, 2005.

[25] J. Montes, A. Sánchez, J.J. Valdés, M.S. Pérez, P. Herrero, "Finding order in Chaos: A behavior model of the whole grid", *Concurrency and Computation: Practice & Experience*, Vol.22 no. 11, pp.1386-1415, 2010.

[26] D. Gannon, R. Bramley, G. Fox, S. Smallen, A. Rossi, R. Ananthakrishnan, F. Bertrand, K. Chiu, M. Farrellee, M. Govindaraju, S. Krishnan, L. Ramakrishnan, Y. Simmhan, A. Slominski, Y. Ma, C. Olariu, and N. Rey-Cenvaz, "Programming the Grid: Distributed Software Components, P2P and Grid Web Services for Scientific Applications", *Cluster Computing*, Vol.5, no3, pp.325-336, 2002.

[27] A.N. Haidar, P. V. Coveney, A.E. Abdallah, P. Y. A. Ryan, B. Beckles, J. M. Brooke and M.A.S. Jones "Formal Modelling of a Usable Identity Management Solution for Virtual Organisations", *Proceedings of the 2nd Workshop on Formal Aspects of Virtual Organisations*, Electronic Proceedings in Theoretical Computer Science - EPTCS 16, pp. 41-50, 2010.

[28] J. Woodcock, and J. Davies, *Using Z Specification, Refinement, and Proof*. C.A.R Hoare series editor, Prentice Hall International, 1996.

[29] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[30] F. Neubauer, A. Hoheisel, J. Geiler, "Workflow-based Grid applications", *Future Generation Computer Systems*, Vol.22, no.1-2, pp.6–15, 2006.

[31] Z. Guan, F. Hernandez, P. Bangalore, J. Gray, A. Skjellum, V. Velusamy, Y. Liu, "Grid-Flow: a Grid-enabled scientific workflow system with a Petri-net-based interface", *Concurrency and Computation: Practice and Experience*, Vol. 18, no. 10, pp. 1115–1140, 2006.

[32] W.D. Liu, J.X. Song, C. Lin, "Modeling and Analysis of Grid Computing Application Based Price Timed Petri Net", *Acta Electronica Sinica*, 2005-08.

[33] Y. Gurevich, B. Rossman, W. Schulte, "Semantic essence of AsmL", *Theoretical Computer Science*, Vol.343, no.3, pp.370 – 412, 2005.

[34] Y. Gurevich, "Sequential Abstract State Machines capture Sequential Algorithms", *ACM Transactions on Computational Logic*, Vol.1, no.1, pp. 77-111, 2000.

[35] R. Milner, *Communicating and Mobile Systems: the π-calculus*, Cambridge University Press, 1999.

[36] J. Zhou, G. Zeng, "Describing and reasoning on the composition of grid services using pi-calculus", *Proceedings of the 6th IEEE International Conference on Computer and Information Technology*, Seoul, Korea, pp.48-53, September 2006.

[37] R. Eshuis, R. Wieringa, "Comparing Petri Net and Activity Diagram Variants for Workflow Modelling – A Quest for Reactive Petri Nets", *Petri Net Technology for Communication Based Systems*, LNCS vol.2472, Springer, 2003, pp 321-351.

[38] R. Farahbod, V. Gervasi, U. Glässer, "Executable formal specifications of complex distributed systems with CoreASM", *Science of Computer Programming*, 2012.

[39] P. Andreetto, S. Andreozzi, A. Ghiselli, M. Marzolla, V. Venturi, L. Zangrando, "Standards-based Job Management in Grid Systems", *Journal of Grid Computing*, Vol.8, no.1, pp.19-45, 2010.

[40] W. van der Aalst, C. Bratosin, N. Sidorova, and N. Trcka, "A Reference model for Grid Architectures and its Validation", *Concurrency and Computation: Practice and Experience*, Vol.22, no.11, pp. 1365-1385, 2010.

[41] J. Atlas, M. Swany, K.S. Decker, "Flexible Grid Workflows Using TAEMS", Proceedings of the Workshop on Exploring Planning and Scheduling for Web Services, Grid and Autonomic Computing, at AAAI05, Pittsburgh, Pennsylvania, pp. 24-31, July2005.