Journal of Grid Computing **1:** 9–23, 2003.
© 2003 *Kluwer Academic Publishers. Printed in the Netherlands.*

Characterizing Grids: Attributes, Definitions, and Formalisms

Zsolt Németh[1] and Vaidy Sunderam[2]
[1]*MTA SZTAKI Computer and Automation Research Institute, H-1518 Budapest, P.O. Box 63, Hungary*
E-mail: zsnemeth@sztaki.hu
[2]*Math & Computer Science, Emory University, Atlanta, GA 30322, USA*
E-mail: vss@emory.edu

Key words: computational Grids, distributed computing, formal definitions

Abstract

Grid systems and technologies have evolved over nearly a decade; yet, there is still no widely accepted definition for Grids. In particular, the essential attributes that distinguish Grids from other distributed computing environments have not been articulated. Most approaches to definition adopt a static view and consider only the properties and components of, or the applications supported by, Grids. The definition proposed in this paper is based on the runtime semantics of distributed systems. Rather than attempt to simply compare static characteristics of Grids and other distributed computing environments, this paper analyzes operational differences, from the viewpoint of an application executing in both environments. Our definition is expressed formally as an Abstract State Machine that facilitates the analysis of existing Grid systems or the design of new ones with rigor and precision. This new, semantical approach proposes an alternative to the currently accepted models for determining whether or not a distributed system is a Grid.

1. Introduction

Grid technologies have attracted a great deal of attention recently, and numerous infrastructure and software projects have been undertaken to realize various visions of Grids. However, while there is universal consensus that Grids are concerned with resource sharing, there is less agreement on the unique characteristics of Grids, the functionalities that Grids support, programming models suitable to Grids, or even abstractions to model and represent Grids. Furthermore, the distinction between conventional distributed computing environments and Grids is blurred, and there is an emerging need to clarify this fuzzy boundary. This paper proposes functional attributes and a set of formalisms for characterizing Grids, with a view to evolving a common framework within which Grids may be analyzed and understood.

Computer networking has evolved over about three decades, and like many technologies, has grown exponentially in terms of performance, functionality, reliability, and widespread adoption. In these early years of the twenty-first century, high-speed, highly-reliable Internet connectivity is as commonplace as electricity in commercial, governmental, and research/educational institutions, and individual consumers are not far behind. This observation led researchers, in the mid-1990's, to propose the notion of computational Grids [11], where computing resources would be available as universally and as easily as electric power. In attempting to analyze Grids, this basic notion may be viewed from the *utility* perspective (consumers obtaining resources from utility providers) or from the *resource sharing* perspective ("virtual organizations" engage in secure sharing of resources [10] to achieve their common goals).

Currently, practitioners and researchers subscribe predominantly to the resource sharing model. However, computational resource sharing is not new; beginning with Remote Job Entry [6] systems, printer sharing, remote procedure call [6], networked file systems, heterogeneous computing systems [16], and multiprocessor emulation environments [22] have all enabled the utilization of resources on external computing devices over commodity networks. Therefore, Grids are sometimes viewed as successors to distributed computing environments, but the real differences between the two have not been clearly articulated, partly because there is no widely accepted definition for Grids. There are common views: some define it as a high-performance distributed environment; some take into consideration its geographically distributed, multi-domain feature, and others define Grids based on the number of resources they unify, and so on. The aim of this paper is to go beyond these obviously true but somewhat superficial views, and characterize the fundamental attributes of Grids.

2. Background

Grids were originally proposed in the context of large scale scientific applications that required (or could exploit) computational and data-resources distributed geographically among multiple sites. This paradigm was subsequently found to be applicable in other settings, e.g., inter-departmental or inter-company resource sharing, and in other application domains. In all scenarios, the common focus theme is the sharing of distributed resources in a well controlled, secure and mutually fair manner. A computing platform that fits such a model is referred to as a "Grid".

Although the motivations and goals for Grids are obvious, there is no clear definition for a Grid system. Various proponents have defined a Grid as a [framework for] "flexible, secure, coordinated resource sharing among dynamic collections of individuals, institutions, and resources" [10], "a single seamless computational environment in which cycles, communication, and data are shared, and in which the workstation across the continent is no less than one down the hall" [19], "a wide area environment that transparently consists of workstations, personal computers, graphic rendering engines, supercomputers and non-traditional devices: e.g., TVs, toasters, etc." [20], "a collection of geographically separated resources (people, computers, instruments, databases) connected by a high speed network [. . . distinguished by . . .] a software layer, often called middleware, which transforms a collection of independent resources into a single, coherent, virtual machine" [32]. More recent definitions [18] have not helped a consensus emerge; coordinated resource sharing [15], single-system image [21], comprehensiveness of resources [30], and utility computing [17] have been stated as key characteristics of Grids by leading practitioners. With varying degrees of precision these definitions describe the central notion of Grid systems; yet, they are unable to clearly distinguish between Grid systems and conventional distributed systems. For instance, in some cases a PVM program may conform to one definition ("a single coherent virtual machine") although PVM is not considered a Grid system.

Thus far, most attempts at defining and modeling Grids (e.g., "The Anatomy of the Grid" [10]) have been entirely informal and have focused on *how* a Grid system can be constructed, i.e. what components, layers, protocols and interfaces must be provided. Our approach is orthogonal to these perspectives and is more aligned with a recent report [14] that describes the operational characteristics of Grids and suggests methodologies for implementation. We try to present a formal definition of *what* a Grid system should provide: what functionalities are essential to creating a Grid with no regard to actual components, protocols or any other details of implementation. We focus on the dynamic, runtime semantics of Grids (i.e. what does it mean to execute an application using a Grid) rather than its actual structure or composition which is a static view found in earlier definitions. In order to grab the runtime semantics, an application *and* an environment are put together into a model revealing their interaction.

The analysis begins with an informal comparison of the working conditions of distributed applications executing within "conventional" distributed computing environments (e.g., PVM [16], and certain implementations of MPI such as MPICH [22]), as compared to Grids. Grids are relatively new and therefore, in the comparison (and in the remainder of the paper) an *idealistic* Grid is assumed – not necessarily as implemented but rather as envisaged in many papers. Subsequently a formal model is created for the execution of a distributed application, assuming the working conditions of a conventional system, with a view to distilling its runtime semantics. This model is transformed, through the addition of new modules, in order for the application to operate under assumptions made for a Grid environment. Based on the formalism and the differences in operating conditions, it is easy to trace and point out that a Grid is not just a modification of "conventional" distributed systems but fundamentally differs in semantics. The analysis identifies the functionalities that must be present in order to create a distributed environment that is able to provide Grid services.

The formal method used for modeling is the Abstract State Machine (ASM). ASMs represent a mathematically well founded framework for system design and analysis [2] and were introduced by Gurevich as evolving algebras [24]; definitions [23, 25] and a tutorial [24] may be found in the literature. In ASMs, states are represented as (modified) logician's structures, i.e. basic sets (universes) with functions and relations interpreted on them. Experience has shown that any kind of static mathematical reality can be represented as a first-order structure [24]. Structures are modified in ASM to enable state transitions for modeling dynamic systems. Applying a step of ASM M to state (structure) A will produce another state A' on the same set of function names. If the function names and arities are fixed, the only way of transforming a structure is to change the value of some functions for some arguments. Transformation may depend on conditions. Therefore, the most general structure transformation (ASM rule) is a guarded destructive assignment to functions at given arguments [2]. Readers unfamiliar with the method may simply treat the description as a set of rules written in pseudo code; the rules fire independently if their condition evaluates to true.

There are numerous formal methods accepted for modeling, yet a relatively new method, ASM has been chosen for two reasons. First, it is able not just to model a working mechanism precisely but also to reveal the highly abstract nature of a system, i.e. grasp the semantics. ASM is a generalized machine that can very closely and faithfully model any algorithm no matter how complex and abstract it is [27]. In this sense it is similar to Turing machines. Second, ASMs – unlike Turing machines – can easily be tailored to the required level of abstraction. There are many methods that operate on states and state transitions and thus, could have been a potential candidate for modeling Grids. Yet, methods that apply a fixed and low level representation of states (e.g., symbols, tokens, bits, etc.) are less suitable to model a system of extremely high complexity. Logician's structures applied in ASMs offer a much more expressive, flexible and complete way of state description. The basic sets and the functions interpreted on them can be freely chosen to the required level of complexity and precision. ASM has been successfully applied in various scientific and industrial projects [4].

The outcome of our analysis is a highly abstract declarative model. The model is declarative in the sense that it does not specify *how* to realize or decompose a given functionality, but rather *what* it must provide. Our model also specifies formally what the most fundamental differences are between conventional distributed environments and Grids. Without any restriction on the actual implementation, if a certain distributed environment conforms to the definition, i.e. it provides the necessary functionalities, it can be termed a Grid system. In this paper the most elementary and inevitable services are defined. It is a minimal set: without them no application can be executed under assumptions made for Grids although, a number of applications may also require additional services.

Our model adopts an architectural/system developer's point of view. The resulting formal model can be applied in several ways. First, it is a precise formal definition that enables checking or comparing existing Grids to determine if they provide the necessary functionalities. Furthermore it can serve as a basis for high level specification of a new system or components, or for modification of an existing one. Finally, the model is also useful in reasoning about the properties of Grids.

3. Conventional Distributed Environments Versus Grids

Distributed applications are comprised of a number of cooperating processes that exploit resources of loosely coupled computer systems. An application may be distributed simply due to its nature, in order to gain performance, or to utilize resources that are not locally present. Distributed computing, in the high performance computing domain, for example, may be accomplished via traditional environments (e.g., PVM, MPICH) or with emerging software frameworks termed computational Grids. As we will show in this paper, the essential semantical difference between these two categories of environments centers around the manner in which they establish a virtual, hypothetical concurrent machine from the available resources.

An application in a conventional distributed environment assumes a pool of computational nodes (Figure 1, left side) from (a subset of) which a virtual concurrent machine is formed. The pool consists of PCs, workstations, and possibly supercomputers, provided that the user has access (valid login name and password) to all of them. Login to the virtual machine is realised by login (authentication) to each node, although it is technically possible to avoid per-node authentication if at least one node accepts the user as authentic. In general it can be assumed that once a user is logged on to a node, he or she is allowed to use essentially all the resources belonging to, or attached to the node without further authorization procedures. On the other hand the user is restricted to using the local resources at a given node; only in rare cases is there support for using remote resources. Since the user has his or her own accounts on these nodes, the user is aware of their features: architecture type, computational power and capacities, operating system, security concerns, usual load, etc. Furthermore, the virtual pool of nodes can be considered static, since the set of nodes to which the user has login access changes very rarely. The size of such systems as deployed and used in practice, is typically of the order of 10–100 nodes.

In contrast, computational Grids are based on large-scale resource sharing [10]. Grids assume a virtual pool of resources rather than computational nodes (right side of Figure 1). Although current systems mostly focus on computational resources (CPU cycles + memory) [12] that basically coincide with the notion of nodes, Grid systems are expected to operate on a wider range of resources like storage, network, data, software [19] and atypical resources like graphical and audio input/output devices, manipulators, sensors and so on [20]. All these resources typically exist within nodes that are geographically distributed, and span multiple administrative domains. The virtual machine is constituted of a set of resources taken from the pool.

In Grids, the virtual pool of resources is dynamic and diverse, since the resources can be added and withdrawn at any time according to their owner's discretion, and their performance or load can change frequently over time. The typical number of resources in the pool is of the order of several thousand or more. For all these reasons, the user has very little or no *a priori* knowledge about the actual type, state and features of the resources constituting the pool.

Access to the virtual machine means that the user has some sort of credential that is accepted by the owners of resources in the pool. A user may have the right to use a given resource; however, it does not mean that he or she has login access to the node hosting the resource. Access to the nodes cannot be controlled based on login access due to the large number of resources in the pool and the diversity of local security policies, and it is unrealistic that a user has login access to thousands of nodes simultaneously.

Table 1 summarises the differences between conventional distributed and Grid systems.

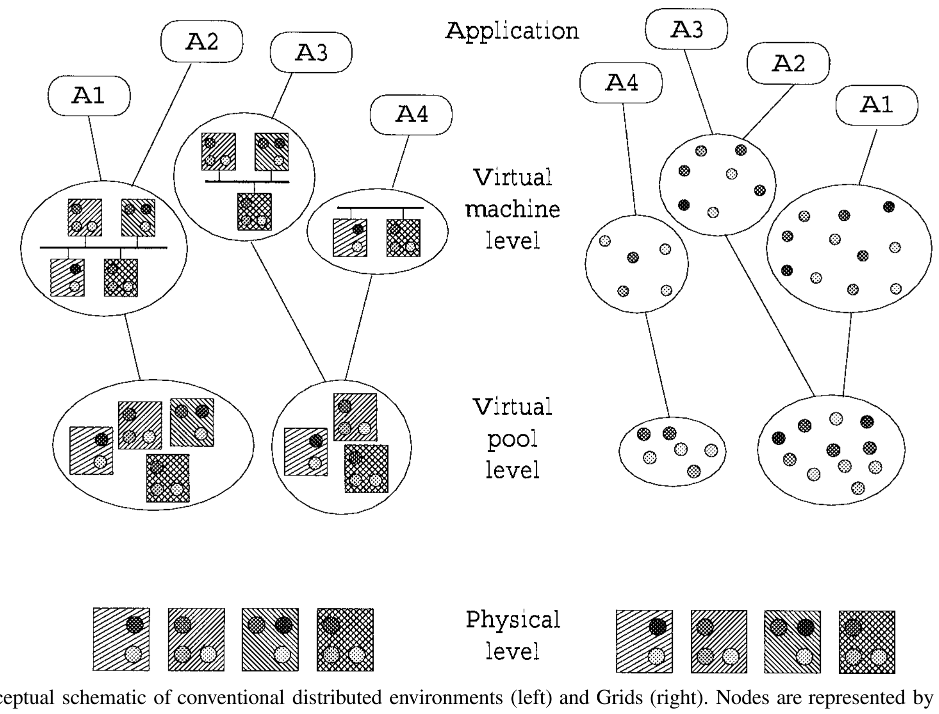

Figure 1. Conceptual schematic of conventional distributed environments (left) and Grids (right). Nodes are represented by boxes, resources by circles.

Table 1. Comparison of conventional distributed environments and Grids.

	Conventional distributed environments	Grids
1	A virtual pool of computational nodes	A virtual pool of resources
2	A user has access (credential) to all the nodes in the pool	A user has access to the pool but not to individual nodes
3	Access to a node means access to all resources on the node	Access to a resource may be restricted
4	The user is aware of the capabilities and features of the nodes	The user has little or no knowledge about each resource
5	Nodes belong to a single trust domain	Resources span multiple trust domains
6	Elements in the pool 10–100, more or less static	Elements in the pool $\gg$100, dynamic

4. Universes and the Signature

To formally model Grid systems we begin by defining key ASM terms. In ASM, the signature (or vocabulary) is a finite set of function names, each of fixed arity. Furthermore, it also contains the symbols *true*, *false*, *undef*, $=$ and the usual Boolean operators. A state A of signature Υ is a non-empty set X together with interpretations of function names in Υ on X.

As an extremely simple example let us consider a dataflow node that adds 1 to the value of the incoming token if its colour is red and changes its colour to blue. The superuniverse X is composed of the following universes: *TOKEN*, *COLOUR*, *VALUE*. The signature Υ consists the function names: $val : TOKEN \rightarrow VALUE$, $col : TOKEN \rightarrow COLOUR$. For instance, if *TOKEN* contains a single element t that is red and has the value 2, state A can be represented by X with the following interpretations: $col(t) = red$, $val(t) = 2$. The rule

```
if $col(t) = red$ then
    $val(t) := val(t) + 1$
    $col(t) := blue$
```

describes the activity of the node. The condition evaluates to true in state A. After firing the rule, the functions of Υ can be interpreted on X as $col(t) = blue$, $val(t) = 3$ that represents a new state, B.

The definition of the universes and the signature places the real system to be modeled into a formal framework. Certain objects of the physical reality are modeled as elements of universes and relationships between real objects are represented as functions and relations. These definitions also highlight what is *not* modeled by circumscribing the limits of the formal model and keeping it reasonably simple.

When using the modeling scheme in the realm of distributed computing we consider an application (universe *APPLICATION*) as consisting of several processes (universe *PROCESS*) that cooperate in some way. Their relationship is represented by the function $app : PROCESS \rightarrow APPLICATION$ that identifies the specific application a given process belongs to. Processes are owned by a user (universe *USER*). Function $user : PROCESS \rightarrow USER$ gives the owner of a process. Processes need resources (universe *RESOURCE*) to work. A distinguished element of this universe is $resource_0$ that represents the computational resource (CPU cycles, memory) that is essential to run a process. $request : PROCESS \times RESOURCE \rightarrow \{true, false\}$ yields true if the process needs a given resource, whereas $uses : PROCESS \times RESOURCE \rightarrow \{true, false\}$ is true if the process is currently using the resource. Note that the *uses* function does not imply either exclusive or shared access, but only that the process can access and use it during its activity. Processes are mapped to a certain node of computation (universe *NODE*). This relationship is represented by the function $mapped : PROCESS \rightarrow NODE$ which gives the node the process is mapped on. On the other hand, resources cannot exist on their own, they belong to nodes, as characterized by relation $BelongsTo : RESOURCE \times NODE \rightarrow \{true, false\}$. Processes execute a specified task represented by universe *TASK*. The physical realisation of a task is the static representation of a running process, therefore it must be present on (or accessible from) the same node ($installed : TASK \times NODE \rightarrow \{true, false\}$) where the process is.

Resources, nodes, and tasks have certain attributes (universe *ATTR*) that can be retrieved by function $attr : \{RESOURCE, NODE, TASK\} \rightarrow ATTR$. (Also, *user*, *request* and *uses* can be viewed as special cases of *ATTR* for processes.) A subset of *ATTR* is the architecture type represented by *ARCH* ($arch : RESOURCE \rightarrow ARCH$) and location (universe *LOCATION*, $location : RESOURCE \rightarrow LOCATION$). Relation $compatible : ATTR \times ATTR \rightarrow \{true, false\}$ is true if the two attributes are compatible according to a reasonable definition. To keep the model simple, this high level notion of attributes and compatibility is used instead of more precise processor type, speed, memory capacity, operating system, endian-ness, software versions and so on, and the appropriate different definitions for compatibility.

Users may login to certain nodes. If $CanLogin : USER \times NODE \rightarrow \{true, false\}$ evaluates to true it means that user has a credential that is accepted by the security mechanism of the node. It is assumed that initiating a process at a given node is possible if the user can log in to the node. $CanUse : USER \times RESOURCE \rightarrow \{true, false\}$ is a similar logic function. If it is true, the user is authentic and authorized to use a given resource. While *CanLogin* directly corresponds to the login procedure of an operating system, *CanUse* remains abstract at the moment.

Processes are at the center of the model. To maintain generality, they are modeled according to the following assumptions involving a minimal set of states, communication procedures and events but are yet able to describe the entire process lifecycle. In modern operating systems processes have many possible states but there are three inevitable ones: *running*, *ready to run* and *waiting*. In our model the operating system level details are entirely omitted. States *ready to run* and *running* are treated evenly assuming that processes in the *ready to run* state will proceed to *running* state in finite time. Therefore, in this model processes have essentially two states, but for purely technical reasons, "waiting for communication" adds another state. These states can be retrieved by function $state : PROCESS \rightarrow \{running, waiting, receive\_waiting\}$.

Processes interact via messages (*MESSAGE*) that has a sender and a receiver process ($from : MESSAGE \rightarrow PROCESS, to : MESSAGE \rightarrow PROCESS$). The actual content of the message is irrelevant in this model. Blocking communication is controlled by the function $expecting : PROCESS \rightarrow PROCESS + \{any\}$.

During the execution of a task, different events may occur represented by the external function $event : TASK \rightarrow \{req\_res, spawn, send, receive, terminate\}$. Events are defined here as a point where the state of one or more processes is changed. They may be prescribed in the task itself or may be external, independent from the task – at this level of abstraction there is no difference.

5. Rules for a Conventional Distributed System

The model presented here is a distributed multi-agent ASM where agents are processes, i.e. elements from the *PROCESS* universe. The nullary *Self* function represented here as p allows an agent to identify itself among other agents. It is interpreted differently by different agents. The following rules constitute a module, i.e. a single-agent program that is executed by each agent. Agents have the same initial state as described below.

5.1. *Initial State*

Let us assume k processes belonging to an application and a user: $\exists p_1, p_2, \ldots, p_k \in PROCESS, \forall p_i, 1 \leqslant i \leqslant k : app(p_i) \neq undef; \forall p_i, 1 \leqslant i \leqslant k : user(p_i) = u \in USER$. Initially they require certain resources ($\forall p_i, 1 \leqslant i \leqslant k : \exists r \in RESOURCE : request(p_i, r) = true$) but do not possess any of them ($\forall p_i, 1 \leqslant i \leqslant k : \forall r \in RESOURCE : uses(p_i, r) = false$). All processes have their assigned tasks ($\forall p_i, 1 \leqslant i \leqslant k : task(p_i) \neq undef$) but no processes are mapped to a node ($\forall p_i, 1 \leqslant i \leqslant k : mapped(p_i) = undef$).

Specifically, the following holds for conventional systems (but not for Grids) in the initial state:

– There is a virtual pool of l nodes for each user. The user has a valid login credential for each node in her pool: $\forall u \in USER, \exists n_1, n_2, \ldots, n_l \in NODE : CanLogin(u, n_i) = true, 1 \leqslant i \leqslant l$
– The tasks of the processes have been preinstalled on some of the nodes (or accessible from some nodes via NFS or other means): $\forall p_i, 1 \leqslant i \leqslant k : \exists n \in NODE : installed(task(p_i), n) = true$ in such a way that the format of the task corresponds to the architecture of the node: $compatible(arch(task(p_i)), arch(n))$.

Rule 1: Mapping

The working cycle of an application in a conventional distributed system is based on the notion of a pool of computational nodes (assumptions 1 and 2 in Table 1). Therefore, first all processes must be mapped to a node chosen from the pool. Other rules cannot fire until the process is mapped. Rule 1 will fire exactly once.

```
if $mapped(p) = undef$ then
   choose $n$ in $NODE$ satisfying $CanLogin(user(p), n)$
                                   & $installed(task(p), n)$
          $mapped(p) := n$
   endchoose
```

Note the declarative style of the description: it does not specify how the appropriate node is selected; any of the nodes where the conditions are true can be chosen. The selection may be done by the user, prescribed in the program text, or may be left to a scheduler or a load balancer layer, but at this level of abstraction it is irrelevant. It is possible because the user (application) has information about the state of the pool (assumption 4 in Table 1). Actually, the conditions listed here (login access and the presence of the binary code) are the absolute minimal conditions and in a real application there may be others with respect to the performance of the node, the actual load, user's priority and so on.

Rule 2: Resource grant
Once a process has been mapped, and there are pending requests for resources, they can be satisfied if the requested resource is on the same node as the process. If a specific type of resource is required by the process, it is the responsibility of the programmer or user to find a mapping where the resource is local with respect to the process. Furthermore, assumption 2 in Table 1 is applied here: if a user can login to a node, he or she is authorized to use all resources belonging to or attached to the node: $\forall u \in USER, \forall r \in RESOURCE : CanLogin(u, n) \rightarrow CanUse(u, r)$ where $BelongsTo(r, n) = true$. Therefore, at this level of abstraction it is assumed realistically that resources are available or will be available within a limited time period. The model does not incorporate information as to whether the resource is shared or exclusive.

```
if $(\exists r \in RESOURCE) : request(p, r) = true$
   & $BelongsTo(r, mapped(p))$
   then
        $uses(p, r) := true$
        $request(p, r) := false$
```

Rule 3: State transition
If all the resource requests have been satisfied and there is no pending communication, the process can enter the *running* state.

```
if $(\forall r \in RESOURCE) : request(p, r) = false$
   & $expecting(p) = undef$
   then
        $state(p) := running$
```

The running state means that the process is performing activities prescribed by the *task*. This model is aimed at formalizing the mode of distributed execution and not the semantics of a given application.

Rule 4: Resource request
During execution of the *task*, events can occur represented by the external *event* function. The event in this rule represents the case when the process needs additional resources during its work. In this case process enters the *waiting* state and the *request* relation is raised for every resource in the *reslist*.

```
if $state(p) = running$ & $event(task(p)) = req\_res(reslist)$ then
   $state(p) := waiting$
   do forall $r \in RESOURCE : r \in reslist$
      $request(p, r) := true$
   enddo
```

Rule 5: Process spawning
This rule describes another possible event during execution: the currently running process creates another process. The newly created process belongs to the same user and application, and is not mapped to any node. It requests, but holds no resources. Obviously, the new process will fire the mapping rule and then the resource grant rule subsequently.

```
if $state(p) = running$ & $event(task(p)) = spawn(reslist)$ then
   extend $PROCESS$ by $p'$ with
          $user(p') := user(p)$
          $app(p') := app(p)$
          $state(p') := waiting$
          $mapped(p') := undef$
          do forall $r \in RESOURCE : r \in reslist$
             $request(p', r) := true$
          enddo
   endextend
```

Distributed environments with a more static system model (e.g., MPICH [22]) do not necessarily have an implementation of this rule.

Rule 6: Send (communication)
Processes of a distributed application interact with each other via message passing. Although in modern programming environments there are higher level constructs, e.g., RPC, RMI, virtual object spaces, etc., and sophisticated message passing libraries like MPI provide a rich set of various communication patterns for virtually any kind of data, at a low level they are

all based on some form of send and receive communication primitives. This model restricts its scope to (blocking and non-blocking versions of) message passing. In the following, code fragments for blocking versions are bracketed, and are supplementary to the non-blocking code.

Upon encountering a send instruction during the execution of the task, a new message is created with the appropriate sender and receiver information. If it is a blocking send and the communication partner p' is not waiting for this message, the process goes to the waiting state and expects p' to receive.

```
if $state(p) = running$ & $event(task(p)) = send(p')$ then
   extend $MESSAGE$ by $msg$ with
         $to(msg) := p'$
         $from(msg) := p$
   endextend
   [if $expecting(p') \neq p$ & $expecting(p') \neq any$ then
       $expecting(p) := p'$
       $state(p) := waiting$
   endif]
```

Rule 7: Receive (communication)
Normally, receive procedures explicitly specify the source process for expected messages. However, message passing systems must be able to handle indeterminacy, i.e. in some situations there is no way to specify the order in which messages are accepted. For this purpose receive instructions may use a special symbol *any* for expressing that the first arriving message from any process will be accepted. For compact notation a single rule is presented here with alternative parts in brackets, separated by a vertical line (e.g., `{code for specific sender|code for any sender}`).

If the task reaches the receive instruction and there exists a message that can be accepted, it is removed from the universe *MESSAGE* and the process resumes its work. *MESSAGE*(msg) := *false* means that *msg* is not part of the *MESSAGE* universe anymore. It is assumed that the content of the message (not specified or modeled here) is in the possession of the recipient. The concept of message is like a container: the information held by the sender is transformed into a message and the message exists until the receiver extracts the information. The actual handling of the message (queued, buffered or transmitted) is up to the lower levels of abstraction (e.g., the PVM model in [3]). If the expected message does not exist and the operation is a blocking call, the process goes into the *receive_waiting* state and updates the *expecting* function.

```
if $state(p) = running$
   & $event(task(p)) = receive(\{p'|any\})$
   then
     if $(\exists msg \in MESSAGE) : to(msg) = p$
        & $\{from(msg) = p'|true\}$
        then
             $MESSAGE(msg) := false$
             $[expecting(from(msg)) := undef$
        else
             $expecting(p) := \{p'|any\}$
             $state(p) := receive\_waiting]$
        endif
```

The blocking version of *receive* has an additional rule: whenever the process is suspended in *receive_waiting* state, an appearing message that can be accepted will transit the process to running state. The *expecting* function is updated accordingly.

```
[if $state(p) = receive\_waiting$
    & $(\exists msg \in MESSAGE) : to(msg) = p,$
    $\{from(msg) = expecting(p)|\quad\}$
    then
         $MESSAGE(msg) := false$
         $state(p) := running$
         $expecting(from(msg)) := undef$
         $expecting(p) := undef]$
```

Rule 8: Termination
This rule represents the event of termination. *PROCESS*(p) := *false* means that process p is removed from universe *PROCESS*: it does not exist anymore.

```
if $state(p) = running$ & $event(task(p)) = terminate$ then
   $PROCESS(p) := false$
```

6. Rules for a Grid

6.1. *Initial State*

The initial state is exactly the same as in the case of conventional distributed systems except for the specific items (see Section 5.1) that is

– There exist a virtual pool of resources and the user has a credential that is accepted by the owners of resources in the pool: $\forall u \in USER, \exists r_1, r_2, \ldots, r_m : CanUse(u, r_i) = true$, $1 \leqslant i \leqslant m$

As is evident, the initial state is very similar to that of conventional distributed systems, and once applications start execution there are few differences in the runtime model of conventional and Grid systems. The principal differences that do exist pertain mainly to the acquisition of resources and nodes. Conventional systems try to find an appropriate node to map processes onto, and then satisfy resource needs locally. In contrast, Grid systems assume an abundant pool of resources; thus, first the necessary resources are found, and then they designate the node onto which the process must be mapped.

Rule 9: Resource selection
To clarify the above, we superimpose the the model for conventional systems from Section 5 onto an environment representing a Grid according to the characteristics in Table 1. We then try to achieve Grid-like behavior by minimal changes in the rules. The intention here is to swap the order of resource and node allocation while the rest of the rules remain intact. If an authenticated and authorized user requests a resource, it may be granted to the process. If the requested resource is computational in nature, (resource type *resource*$_0$) then the process must be placed onto the node where the resource is located. Let us replace Rules 1 and 2 by Rule 9 while keeping the remaining rules constant.

```
if $(\exists r \in RESOURCE) : request(p, r) = true$
   & $CanUse(user(p), r)$
   then
        if $type(r) = resource_0$ then
            $mapped(p) := location(r)$
            $installed(task(p), location(r)) := true$
        endif
        $request(p, r) := false$
        $uses(p, r) := true$
```

For obvious reasons, this first model will not work due to the slight but fundamental differences in working conditions of conventional distributed and Grid systems. The formal description enables precise reasoning about the causes of malfunction and their elimination. In the following, new constructs are systematically added to this simple model in order to realize the inevitable functionalities of a Grid system.

6.2. *Resource Abstraction*

The system described by Rules 3, 4, 5, 6, 7, 8 and 9 would not work under assumptions made for Grid environments. To see why, consider what r means in these models. r in $request(p, r)$ is abstract in that it expresses the process' needs in terms of resource types and attributes in general, e.g., 64 M of memory or a processor of a given architecture or 200 M of storage, etc. These needs are satisfied by certain physical resources, e.g., 64 M memory on machine foo.somewhere.edu, an Intel PIII processor and a file system mounted on the machine. In the case of conventional distributed systems there is an *implicit* mapping of abstract resources onto physical ones. This is possible because the process has been (already) assigned to a node and its resource needs are satisfied by local resources present on the node. *BelongsTo* checks the validity of the implicit mapping in Rule 2.

This is not the case in Grid environments. A process' resource needs can be satisfied from various nodes in various ways, therefore $uses(p, r)$ cannot be interpreted for an abstract r. There must be an *explicit* mapping between abstract resource needs and physical resource objects that selects one of the thousands of possible candidate resources that conforms to abstract resource needs. Let us split the universe *RESOURCE* into abstract resources *ARESOURCE* and physical resources *PRESOURCE*. Resource needs are described by abstract resources, whereas physical resources are those granted to the process. Since the user (and the application) has no information about the exact state of the pool (assumption 4 in Table 1), a new agent executing module $\Pi_{resource\_mapping}$ must be introduced that can manage the appropriate mapping between them by asserting the *mappedresource* : $PROCESS \times ARESOURCE \rightarrow PRESOURCE$ function as described by the following rule:

$\Pi_{resource\_mapping}$

```
if $(\exists ar \in ARESOURCE, pr \in PROCESS)$ :
   $mappedresource(pr, ar) = undef$
   & $request(pr, ar) = true$
   then
       choose $r$ in $PRESOURCE$
         satisfying $compatible(attr(ar), attr(r))$
             $mappedresource(pr, ar) := r$
         endchoose
```

This rule does not specify how resources are chosen; such details are left to lower level implementation oriented descriptions. Just as in the case of node selection (Rule 1), this is a minimal condition, and in an actual implementation there will be additional conditions with respect to performance, throughput, load balancing, priority and other issues. However, the selection must yield relation *compatible* : $ATTR \times ATTR \rightarrow \{true, false\}$ as true, i.e. the attributes of the physical resource must satisfy the prescribed abstract attributes. Based on this, **Rule 9** is modified as:

```
let $r = mappedresource(p, ar)$
if $(\exists ar \in ARESOURCE)$ :
    $request(p, ar) = true$
    & $r \neq undef$
    & $CanUse(user(p), r)$
    then
        if $type(r) = resource_0$ then
            $mapped(p) := location(r)$
            $installed(task(p), location(r)) := true$
        endif
        $request(p, ar) := false$
        $uses(p, r) := true$
```

This rule could be modified so that if $CanUse(user(p), r))$ is false, it retracts $mappedresource(p, ar)$ to $undef$ allowing $\Pi_{resource\_mapping}$ to find another possible mapping. Accordingly, the signature must be changed: $request : PROCESS \times ARESOURCE \rightarrow \{true, false\}$, $uses : PROCESS \times PRESOURCE \rightarrow \{true, false\}$, $BelongsTo : PRESOURCE \times NODE \rightarrow \{true, false\}$, $attr : \{ARESOURCE, PRESOURCE, NODE, TASK\} \rightarrow ATTR$, $location : PRESOURCE \rightarrow LOCATION$, $CanUse : USER \times PRESOURCE \rightarrow \{true, false\}$.

Subsequently Rules 3, 4, 5, 8 must be modified to differentiate between abstract and physical resources. This change is purely syntactical and does not affect their semantics; therefore, their new form is omitted here.

6.3. *Access Control Mechanism (User Abstraction)*

Rule 9 is still missing some details: accessing a resource needs further elaboration. $uses(p, r) := true$ is a correct and trivial step in case of conventional distributed systems, because resources are granted to a local process and the owner of the process is an authenticated and authorized user. In Grids however, the fact that the user can access resources in the virtual pool (i.e. can login to the virtual machine) does not imply that he or she can login to the nodes to which the resources belong (assumption 2 in Table 1): $\forall u \in USER$, $\forall r \in PRESOURCE$, $\forall n \in NODE : CanUse(u, r) \not\rightarrow CanLogin(u, n)$ where $BelongsTo(r, n) = true$.

At a high level of abstraction $uses(p, r) := true$ assigns any resource to any process. However, at lower levels, resources are granted by operating systems to local processes. Thus, a process of the application must be on the node to which the resource belongs, or an auxiliary, handler process ($handler : PRESOURCE \rightarrow PROCESS$) must be present. In the latter case the handler might be already running or might be installed by the user when necessary. (For instance, the notion of handler processes appear in Legion as object methods [20] or as services [13].)

Thus by adding more low level details (refinements, from a modeling point of view) Rule 9 becomes:

```
let $r = mappedresource(p, ar)$
if $(\exists ar \in ARESOURCE) : request(p, ar) = true$
    & $r \neq undef$
    & $CanUse(user(p), r)$
    then
        if $type(r) = resource_0$ then
            $mapped(p) := location(r)$
            $installed(task(p), location(r)) := true$
        else if$(\nexists p' \in PROCESS) : handler(r) = p'$
            extend $PROCESS$ by $p'$ with
                $mapped(p') := location(r)$
                $installed(task(p'), location(r)) := true$
                $handler(r) := p'$
                do forall $ar \in ARESOURCE$
                    $request(p', ar) := false$
                enddo
            endextend
        endif
    endif
    $request(p, ar) := false$
    $uses(p, r) := true$
```

This refined rule indicates that granting a resource involves starting or having a local process on behalf of the user. Obviously, running a process is possible for local account holders. In the initial state there exists a user who has valid access rights to a given resource. However, users are not authorized to log in and start processes on the node to which the resource belongs. To resolve this contradiction let user be split into global user and local user as $globaluser, localuser : PROCESS \rightarrow USER$. Global user identifies the user (a real person) who has access credentials to the resources, and for whom the processes work. A local user is one (not necessarily a real person) who has a valid account and login rights on a node. A Grid system must provide some functionality that finds a proper mapping between global users and local users $usermapping : USER \times PRESOURCE \rightarrow USER$, so that a global user temporarily has the rights of a local user for placing and running processes on the node. Therefore, another agent is added to the model that performs module $\Pi_{user\_mapping}$.

$\Pi_{user\_mapping}$

```
let $r = mappedresource(pr, ar)$
if $(\exists ar \in ARESOURCE,\ pr \in PROCESS) :$
                                    $request(pr, ar) = true$
    & $r \neq undef$
    & $CanUse(user(pr), r)$
    then
```

```
if $type(r) = resource_0$
   or $(\nexists p' \in PROCESS) : handler(r) = p'$
   then
     choose $u$ in $USER$
                  satisfying $CanLogin(u, location(r))$
       $usermapping(globaluser(pr), r) := u$
     endchoose
   else
     if $(\exists p' \in PROCESS) : handler(r) = p'$ then
        $usermapping\ (globaluser(pr), r) :=$
                     $localuser(handler(r))$
     endif
   endif
```

If the process is going to be placed onto the node (directly or via a handler process), then a valid local login name is chosen to be mapped. The choice mechanism is undefined at this level. If the resource is used by an existing handler process, the chosen local user name is the owner of the handler process. In other words, the handler process owned by a local account holder will temporarily work on behalf of another user. (This, again, corresponds to the Legion security mechanism [28].) To include this aspect into **Rule 9**, a valid mapping is required instead of a check for authenticity and authorization.

```
if $(\exists ar \in ARESOURCE) : request(p, r) = true$
  & $usermapping(globaluser(p), mappedresource(ar)) \neq undef$
  then
      $request(p, ar) := false$
      $uses(p, mappedresource(ar)) := true$
```

6.4. *Definition for Grid*

Rules 3, 4, 5, 6, 7, 8 and 9 together with $\Pi_{resource\_mapping}$ and $\Pi_{user\_mapping}$ constitute a reference model for distributed applications under assumptions made for Grid systems in Section 3. A Grid must minimally provide *user* and *resource* abstractions. A system is said to be a Grid if it can provide a service equivalent to $\Pi_{resource\_mapping}$ and $\Pi_{user\_mapping}$ according to some reasonable definition of equivalence (the issue of equivalence is explained in [26]).

7. Discussion and Conclusions

7.1. *From Abstract Functionalities to Real Services*

The rules in Section 6 describe a Grid-like behaviour of a distributed system. Since they portray a very high-level abstraction and are declarative, some clarifications are needed to suggest how they can be turned into practical services for use by system developers. The goal of this project was to create a formal *framework* and to find the most general elementary functionalities based on a comparison between Grids and conventional systems. These elementary functionalities answer the question of *what* a Grid must provide minimally to be semantically different from conventional environments. The obvious question of *how* they can be realized can be answered within the formal framework. One approach is to follow the well established procedure called model refinement [2] where an "abstract" model can be transformed into a "more concrete" one, i.e. hidden details at a higher level of abstraction can be elaborated and specified at a lower level. In such a way by successive refinement the components of a system can be separated, specified and the functional equivalence between two refinement steps can be ensured (see Figure 2). An exact refinement step is beyond the scope of this paper but an informal example is presented here to show how the framework can serve system design.

By asking the *how* question, the following services can be separated at the next level of abstraction. The key in resource abstraction is the selection of available physical resources. According to general principles in Section 6.2, the actual selection method is not specified, but should yield relation $compatible(attr(ar), attr(r))$ true. In the model this relation is external and acts like an oracle: it can tell if the selection is acceptable or not. In practice however, a mechanism must be provided that implements the functionality expressed by the relation. Resource abstraction in a real implementation must be supported at least by two components: a *local information provider* that is aware of the features of local resources, their current availability, load, etc. – in general, a module Π_{ip} that can update $attr(r)$ functions either on its own or by a request, and an *information system* Π_{is} that can provide the information represented by $attr(r)$ upon a query (Figure 2).

User abstraction defined in Section 6.3 is a mapping of valid credential holders to local accounts. A fundamental, highly abstract relation of this functionality is $CanUse(globaluser(p), r)$. It expresses the following: the user $globaluser(p)$ has a valid credential, it is accepted through an authentication procedure and the authenticated user is authorized to use resource r. Just as in case of resource abstraction, this oracle-like statement assumes other assisting services: a *security mechanism* (module Π_s) that accepts global users' certificates and authenticates users and a *local resource management* (module Π_{rm}) that authorizes authentic users to use certain resources (Figure 2).

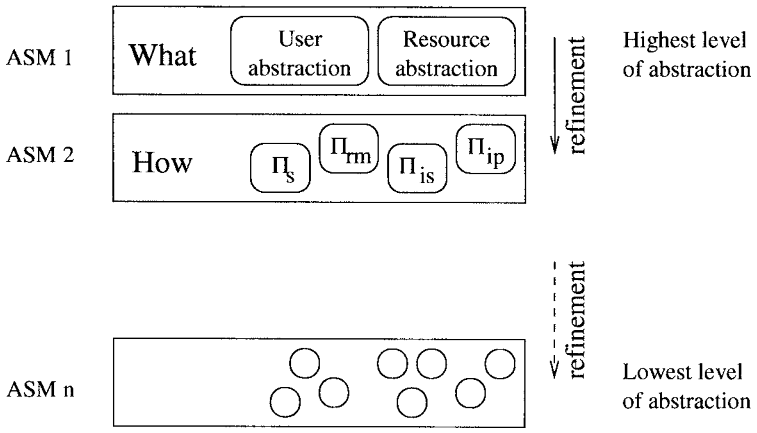

Figure 2. The concept of the formal framework.

These additional modules represent the minimal support needed to realize the functionality of resource and user abstraction. This example is not the only way to decompose the system, but it is a very straightforward one since, for example, these modules exist in both Globus [7, 9] and Legion [5, 28] albeit in different forms. If the highest level model presented in this paper is represented by an abstract state machine ASM1 and the decomposed system by ASM2 then it can be formally checked if ASM2 is operationally equivalent to ASM1 despite their obviously different appearence. By asking further "*how*" questions, the realization of the four modules of ASM2 can be defined in ASM3, then that can be refined to ASM4, etc. They are closer and closer to a technical realization (and thus, would differ significantly from system to system), yet they must be operationally equivalent to ASM1; i.e. provide the Grid functionalities defined at the highest level.

As can be seen, by lowering the level of abstraction, further technical details may be added to the model. For instance, the function of staging is expressed by the $installed(task(p'), location(r)) :=$ *true* statement in the extended version of Rule 9 that must be specified in a "more concrete" model. Some applications may be sensitive to the simultaneous availability of resources thus, co-allocation services may be necessary. While user and resource abstractions represent intrinsic differences, other services are technical differences and need not be defined universally.

7.2. *Applying the Criteria*

It is a common misconception that Grids are used primarily for high performance computing. However, the goal of Grids is to support large-scale resource sharing in any application domain. Conventional distributed environments differ form Grids principally in that they are based on resources the user owns. Sharing and owning in this context are not necessarily related to the ownership in a usual sense. *Sharing* refers to *temporarily* utilizing resources where the user has no direct (login) access otherwise. Similarly, *owning* means having *permanent* and unrestricted access to the resource.

The intent of this paper is to evolve a definiton for clearly distinguishing between systems, to determine whether or not they provide Grid functionalities. The definition is semantically based, and can place existing systems in different contexts. In this discussion we try to show that a Grid is not defined by its hardware, software, or infrastructure; rather, it is a semantically different way of resource usage across ownership domains. Several well-established Grid initiatives clearly meet this definition. However, according to this definition, there are some systems that qualify as Grids even though they are not classified as such, and others that do not meet the criteria for qualification despite their use of the term in their descriptions. We discuss a few examples below.

The SETI@home effort is aimed at harnessing the computing power of millions of (otherwise idle) CPUs for analyzing radio signals [31]. Although it has no specially constructed infrastructure, it is semantically equivalent to a Grid – there is a single user and millions of resources that are owned by entities distinct from the user. Contrary to common scenarios, the job (process) assignment is initiated by the (owners of) resources, but otherwise, it is true that the user specifies the required abstract resource (by providing different versions of the same code); this requirement is satisfied by allocating physical resources at the moment of downloading and installing the code. The user has no *a priori* knowledge about the actual physical resource; yet there is a way to realize an explicit mapping. Similarly, the user has no login access on the individual resource nodes at all. By downloading the code, resource owners authorize the user to use some portion of the resource's CPU cycles. There is an implicit security mechanism: there is a single user with known tasks thus, resource owners can easily determine if the user is trusted or not. Although it was not a deliberate intention, the SETI@home project demonstrated a new computing paradigm that is semantically equivalent to Grids. By providing the functionalities of resource and user abstraction, it realizes a Grid through geographically distributed large-scale resource sharing.

Condor is a workload management mechanism aimed at supporting high-throughput computing [1]. Its primary goal is an effective resource management within a so called Condor pool which, in most cases coincides with a cluster. By a matchmaking mechanism it tries to find appropriate resources for jobs. The entire scheme is based on classads expressing the features of the offered resources and the requirements of jobs [33]. It clearly realises resource abstraction. The capabilities of Condor however, can be extended beyond the boundaries of a single pool. The owners of different pools may have an agreeement that under certain circumstances jobs may be transferred form one cluster to another. This mechanism is called flocking [8]. It means that a job submitted to a cluster by a local user may end up in another cluster where the user has no login access at all. It is governed by the rules of resource owners agreement and their mutual trust. Although technically this solution is far from the security required by Grids, semantically it realises the user abstraction. From this point Condor can be considered as a Grid.

The Open Grid Services Architecture (OGSA) [13] is an attempt to redefine the architecture of Grids in terms of standard services. At first glance OGSA is a more refined and well structured formulation of the same principles. While one of the results of this paper is revealing the semantics of virtualisation hidden in current Grid systems, it is one of the main features of the new approach. "A service-oriented view simplifies the virtualisation through encapsulation of diverse implementations behind a common interface" [13]. In fact, virtualisation through services introduces another level of abstraction: instead of physical resources, services are discovered and a given service can be provided by physical resources in various ways.

Although the main feature of Grids is resource sharing, its realization alone does not make a system a Grid. For example, by deploying frameworks like the Sun Grid Engine [34] or InnerGrid technologies [29], any organization may use its PC intranet for distributed computing by allocating jobs to idling or underloaded processors. Sharing of resources is an obvious feature of these systems that are targeted at cluster- and enterprise-sized distributed computing with a well defined boundary. In these scenarios, the number of resources and users are known and limited, and they belong to a single administrative domain. While resource abstraction is present in limited form, user abstraction is either not necessary or not realized ("all Sun Grid Engine, Enterprise Edition users have the same user names on all submit and execution hosts" [35]). As a consequence, they do not provide a universal, scaleable solution for large scale resource sharing and are semantically not equivalent to Grids, according to the definitions developed above.

7.3. *Concluding Remarks*

Currently, a number of projects and software frameworks are termed "Grid" or "Grid-enabled". Undoubtedly, they have provided many solutions to resource sharing issues in the areas of security, resource management, information services, staging, resource brokering, communication protocols, interfaces, and others. Yet, there is no clear definition for Grids. We argue that neither geographical extent, performance, nor simply the presence of any of the afore mentioned "Grid services" make a distributed system Grid-like. Rather, Grids are semantically different from other, conventional, distributed environments. Although applications executed in these environments are structurally similar, it is shown in this paper that a conventional distributed system cannot provide the necessary functionalities that enable the applications to work under assumptions made for Grids. While in conventional distributed systems the virtual layer is just a *different view* of the physical reality, in Grid systems both *users and resources* appear differently at the virtual and at physical levels and an appropriate mapping must be established between them. Semantically, the inevitable functionalities that must be present in a Grid system are resource and user abstraction. Technically, these two functionalities are realized by various services like resource management, information system, security, staging and so on. Based on the central notions of resource and user abstraction, this paper has attempted to provide a high level semantical model for Grid systems formalized by the ASM method. From the abstract definition, concrete practical systems can be derived by model refinement. This formal framework also enables the analysis or comparison of existing systems.

Acknowledgements

This research was supported in part by U.S. DoE grant DE-02-FG02-23557, NSF grant ACI-0220183, and OTKA No. 032226. The authors are grateful to Z. Balaton for his valuable comments that resulted in significant improvements to the paper.

References

1. J. Basney and M. Livny, “Deploying a High Throughput Computing Cluster”, in R. Buyya (ed.), *High Performance Cluster Computing*, Vol. 1, Chapter 5, Prentice Hall PTR, May 1999.
2. E. Börger, “High Level System Design and Analysis using Abstract State Machines”, in D. Hutter et al. (eds.), *Current Trends in Applied Formal Methods (FM-Trends 98)*, LNCS 1641, Springer, pp. 1–43, 1999.
3. E. Börger and U. Glässer, “Modelling and Analysis of Distributed and Reactive Systems using Evolving Algebras”, in Y. Gurevich and E. Börger (eds.), *Evolving Algebras Mini-Course*, Technical Report BRICS-NS-95-4, University of Aarhus, July 1995. http://www.eecs.umich.edu/gasm/papers/pvm.html.
4. E. Börger, “The Origins and the Development of the ASM Method for High Level System Design and Analysis”, *Journal of Universal Computer Science*, Vol. 8, pp. 2–74.
5. S.J. Chapin, D. Karmatos, J. Karpovich and A. Grimshaw, “The Legion Resource Management System”, in *Proc. of the 5th Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP’99)*, in conjunction with the *International Parallel and Distributed Processing Symposium (IPDPS’99)*, April 1999.
6. G. Coulouris, J. Dollimore and T. Kindberg, *Distributed Systems: Concepts and Design*. Addison-Wesley, Pearson Education, 2001.
7. K. Czajkowski, S. Fitzgerald, I. Foster and C. Kesselman, “Grid Information Services for Distributed Resource Sharing”, in *Proc. of the 10th IEEE International Symposium on High-Performance Distributed Computing (HPDC-10)*, IEEE Press, 2001.
8. D.H. J. Epema, M. Livny, R. van Dantzig, X. Evers and J. Pruyne, “A Worldwide Flock of Condors: Load Sharing among Workstation Clusters”, *Journal on Future Generations of Computer Systems*, Vol. 12, 1996.
9. I. Foster, C. Kesselman, G. Tsudik and S. Tuecke, “A Security Architecture for Computational Grids”, in *Proc. of the 5th ACM Conference on Computer and Communication Security*, November 1998.
10. I. Foster, C. Kesselman and S. Tuecke, “The Anatomy of the Grid”, *International Journal of Supercomputer Applications*, Vol. 15, No. 3, 2001.
11. I. Foster and C. Kesselman: *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers, 1999.
12. I. Foster and C. Kesselman, “The Globus Toolkit”, in [11], pp. 259–278.
13. I. Foster, C. Kesselman, J.M. Nick and S. Tuecke, “Grid Services for Distributed System Integration”, *IEEE Computer*, pp. 37–46, June 2002.
14. I. Foster, C. Kesselman, J.M. Nick and S. Tuecke, “Physiology of the Grid”, Draft version 2.9 22/06/02. http://www.globus.org/research/papers/ogsa.pdf.
15. I. Foster, “What is the Grid? A Three Point Checklist”, *Grid Today*, Vol. 1, No. 6, 22 July 2002. http://www.gridtoday.com/02/0722/100136.html.
16. A. Geist, A. Beguelin, J. Dongarra, W. Jiang, B. Manchek and V. Sunderam, *PVM: Parallel Virtual Machine – a User’s Guide and Tutorial for Network Parallel Computing*. MIT Press, Cambridge, MA, 1994.
17. W. Gentzsch, “Response to Ian Foster’s “What is the Grid?” ” Grid Today, Vol. 1, No. 8, 5 August 2002. http://www.gridtoday.com/02/0805/100191.html.
18. Grid Today. http://www.gridtoday.com.
19. A.S. Grimshaw, W.A. Wulf, J.C. French, A.C. Weaver and P.F. Reynolds, “Legion: The Next Logical Step Toward a Nationwide Virtual Computer”, Technical report No. CS-94-21, June 1994.
20. A.S. Grimshaw and W.A. Wulf, “Legion – a View From 50,000 Feet”, in *Proceedings of the Fifth IEEE International Symposium on High Performance Distributed Computing*. IEEE Computer Society Press: Los Alamitos, California, August 1996.
21. “Grid Computing – Today and Tomorrow: Another View”, *Grid Today*, Vol 1, No. 9, 12 August 2002. http://www.gridtoday.com/02/0812/100221.html.
22. W. Gropp, E. Lusk, N. Doss and A. Skjellum, “A High-Performance, Portable Implementation of the MPI Message Passing Interface Standard”, *Parallel Computing*, Vol. 22, No. 6, pp. 789–828 (1996).
23. Y. Gurevich, “Evolving Algebras 1993: Lipari Guide”, in E. Börger (ed.), *Specification and Valdation Methods*, Oxford University Press, pp. 9–36, 1995.
24. Y. Gurevich, “Evolving Algebras: An Attempt to Discover Semantics”, in G. Rozenberg and A. Salomaa (eds.), *Current Trends in Theoretical Computer Science*, World Scientific, pp. 266–292, 1993.
25. Y. Gurevich, May 1997 Draft of the ASM Guide. http://www.eecs.umich.edu/gasm/papers/guide97.html.
26. Y. Gurevich and J.K. Huggins, “Equivalence is in the Eye of Beholder”, *Theoretical Computer Science*, pp. 353–380, 1997.
27. Y. Gurevich, “Sequential Abstract State Machines Capture Sequential Algorithms”, *ACM Transactions on Computational Logic*, Vol. 1, No. 1, pp. 77–111, July 2000.
28. M. Humprey, F. Knabbe, A. Ferrari and A. Grimshaw, “Accountability and Control of Process Creation in the Legion Metasystem”, in *Proc. of the 2000 Network and Distributed System Security Symposium NDSS2000*, San Diego, California, February 2000.
29. InnerGrid. http://www.gridsystems.com/serv/serv.jsp.
30. W.E. Johnston, “A Different Perspective on the Question of what Is a Grid?” *Grid Today*, Vol. 1, No. 9, 12 August 2002. http://www.gridtoday.com/02/0812/100217.html.

31. E. Korpela, D. Werthimer, D. Anderson, J. Cobb and M. Lebofsky, “SETI@home: Massively Distributed Computing for SETI”, *Computing in Science and Engineering*, No. 1, 2001.
32. G. Lindahl, A. Grimshaw, A. Ferrari and K. Holcomb, “Metacomputing – what’s in it for me”, White paper. http://legion.virginia.edu/papers.html.
33. R. Raman and M. Livny, “High Throughput Resource Management”, Chapter 13 in [11].
34. Sun Grid Engine. http://wwws.sun.com/software/gridware/.
35. Sun Grid Engine, Enterprise Edition 5.3, Administration and User’s Guide. April 2002, revision 01. http://www.sun.com/products-n-solutions/hardware/docs/pdf/816-4739-10.pdf.