A Formal Framework for Defining Grid Systems

Zsolt Németh, Vaidy Sunderam

MTA SZTAKI Computer and Automation Research Institute
P.O. Box 63., H-1518 Hungary
zsnemeth@sztaki.hu
Dept. of Math and Computer Science, Emory University
1784 North Decatur Road, Atlanta, GA 30322, USA
vss@mathcs.emory.edu

Abstract

Although there have been several attempts to create grid systems, there is no clear definition for grids. In this paper, a formal approach is presented for defining elementary functionalities of grid systems. A comparative analysis with conventional distributed systems shows that there are semantical differences, not just technical ones. The resulting abstract working model for grids can serve as a framework for defining new systems or analyzing existing ones.

1 Introduction

It is commonly accepted that through the advent of high speed network technology, high performance applications and unconventional applications emerge by sharing geographically distributed resources in a well controlled, secure and mutually fair way. Such a coordinated large scale virtual pool of resources requires an infrastructure called a grid.

Although the motivations and goals for grids are obvious, there is no clear definition for a grid system. The grid is a [framework for] "flexible, secure, coordinated resource sharing among dynamic collections of individuals, institutions, and resources" [7], "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" [11], "a widearea environment that transparently consists of workstations, personal computers, graphic rendering engines, supercomputers and non-traditional devices: e.g., TVs, toasters, etc." [12], "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" [18]. 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 the last definition ("a single coherent virtual machine") although PVM is not considered a grid system.

So far, "The Anatomy of the Grid" [7] is the only paper that presents a definition for grids in general. Its approach is entirely informal and focuses on *how* a grid system can be constructed, i.e. what components, layers, protocols and interfaces must be provided. Our approach is orthogonal to the perspective of that paper. 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 semantics of the grid (i.e. what does it mean to execute an application using a grid) rather than its actual structure.

The analysis begins with an informal comparison of the working conditions of distributed applications running in "conventional" environments (e.g. PVM [10], and some implementations of MPI like MPICH [13]), as compared to grids. Grids are relatively new (as of 1999 - "there are no existing grids yet" [8]) therefore, in the comparison and in the entire paper an *idealistic* grid is taken into consideration, not necessarily as implemented but rather as envisaged in many papers. Then a formal model is created for a distributed application assuming the working conditions of a conventional system and trying to gasp its runtime semantics. This model is transformed by adding new modules to it in order to work under assumptions made for a grid environment. Based on the formalism and the differences in working 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 Abstract State Machines (ASM). ASMs represent a mathematically well founded framework for system design and analysis [1] and were introduced by Gurevich as evolving algebras [15]. The definition of ASMs is written in [14] and [16] and a tutorial can be found in [15]. 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 mathematic reality can be represented as a first-order structure [15]. These structures are modified in ASM so that dynamics is added to them in a sense that they can be transformed. 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 [1]. 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.

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. It 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 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.

2 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 for HPC may be accomplished via traditional environments such as PVM and some implementations of MPI, or with emerging software frameworks termed computational grids. 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.

A conventional distributed application assumes a pool of computational nodes (Figure 1, left side) that form a virtual parallel machine. 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, technically it is 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, 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 her own accounts on these nodes, she 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 is of the order of 10-100 nodes.

In contrast, computational grids are based on large-scale resource sharing [7]. 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) [9] 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 [11] and atypical resources like graphical and audio input/output devices, manipulators, sensors and so on [12]. All these resources typically exist within nodes

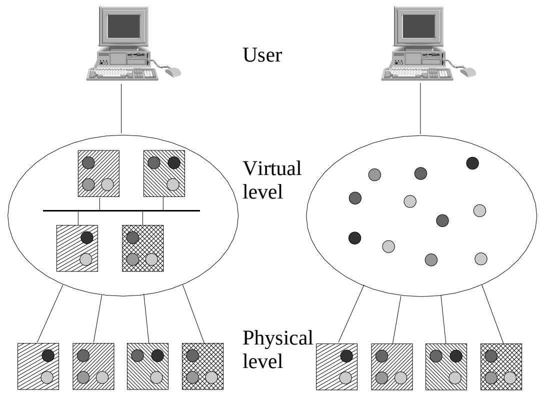

Figure 1. Conceptual schematic of conventional distributed environments (left) and grids (right). Nodes are represented by boxes, resources by circles.

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 1000 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 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.

3 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 nonempty 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, 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.

An application (universe $APPLICATION$) consists 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 $uses : PROCESS \times$

	Conv. 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 sites
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 knowledge about each site
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 1000-10000, dynamic

Table 1. Comparison of conventional distributed environments and grids

$RESOURCE \rightarrow \{true, false\}$ 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.

4 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.

4.1 Initial state

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

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: $\forall u \in USER, \exists n_1, n_2, ...n_l \in NODE : CanLogin(u, n_i) = true, 1 \leq i \leq l$

- The tasks of the processes have been preinstalled on some of the nodes (or accessible from some nodes via NFS or other means): $\forall p_i, 1 \leq i \leq k : \exists n \in NODE : installed(task(p_i), n) = true$ 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 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 is not specified 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 it can be up to a scheduler or a load balancer layer, but at this level of abstraction it is irrelevant. 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, 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 the 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 it is not mapped to any node. It requests for 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[13]) 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 the low level they are all based on some form of send and receive communication primitives. This model restricts its scope to (blocking and nonblocking 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 is 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 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 [2]). 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 $(\exists msg \in MESSAGE) : to(msg) = p$
         & $\{from(msg) = p'| \quad \}$
         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
   $PROCESS(p) := false$
```

5 Rules for a grid

5.1 Initial state

The initial state is exactly the same as in the case of conventional distributed systems except for the specific items (see Section 4.1) 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: $\forall u \in USER, \exists r_1, r_2, ...r_m : CanUse(u, r_i) = true, 1 \leq i \leq m$

As is evident, the initial state of an application is very similar to that of conventional distributed systems, and once applications start execution there are few differences between 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 4 onto an environment representing a grid according to the characteristics in Table 1. We then try to achieve grid-like behaviour 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.

5.2 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 process' needs in terms of resource types and attributes in general, e.g. 64M of memory or a processor of a given architecture or 200M of storage, etc. These needs are satisfied by certain physical resources, e.g. 64M 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. 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. 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.

5.3 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 she can login to the nodes to which the resources belong (assumption 2 in Table 1) : $\forall u \in USER, \forall r \in PRESOURCE, \forall n \in NODE : CanUse(u, r) \nrightarrow CanLogin(u, n)$ 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 [12].)

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[17].) 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
      $request(p, ar) := false$
      $uses(p, mappedresource(ar)) := true$
```

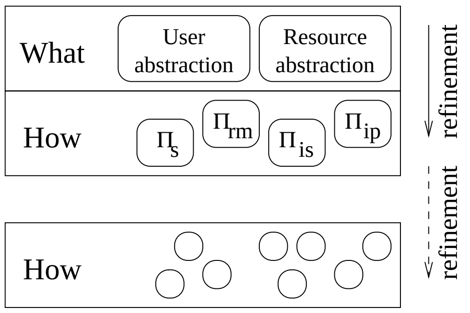

Figure 2. The concept of the formal framework

5.4 Model 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 2. 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.

6 From abstract functionalities to real services

The rules in section 5 describe a grid-like behaviour of a distributed system. Since 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. Those 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. There is a well established procedure called model refinement [1] 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.)

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, it is not specified how the actual selection is made but it must 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.

User abstraction 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.

These additional modules represent the minimal support needed to realize the functionality of resource and user abstraction. For example, they exist in both Globus [5] [6] and Legion [3] [17] although in different forms. By asking additional *how* questions their realization can be defined, but would differ significantly from system to system.

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 cannot be defined universally.

7 Conclusions

There are many solutions to grid security, resource management, information system, staging, resource brokering, grid protocols, interfaces, and so on. Yet, there is no clear definition for grids. We argue that neither the 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 the structure of applications are similar, it is shown in this paper that a conventional distributed system cannot 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.

The paper provides 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 analysis or comparison of existing systems.

Acknowledgments

The authors wish to thank Z. Balaton and N. Podhorszki for their valuable help and comments on the paper. The work presented in this paper was partially supported by US Dept. of Energy grant DE-FG02-99ER25379 and OTKA grant No.032226.

References

[1] 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, 1999, pp. 1-43.

[2] E. Börger and U. Glässer: Modelling and Analysis of Distributed and Reactive Systems using Evolving Algebras. In Y. Gurevich and E. Börger, "Evolving Algebras Mini-Course", Technical Report BRICS-NS-95-4, University of Aarhus, July 1995. http://www.eecs.umich.edu/gasm/papers/pvm.html

[3] S.J. Chapin, D. Karmatos, J. Karpovich, A. Grimshaw: The Legion Resource Management System. 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

[4] K. Czajkowski, I. Foster, N. Karonis, C. Kesselman, S. Martin, W. Smith, S. Tuecke: A Resource Management Architecture for Metacomputing Systems. Proc. IPPS/SPDP '98 Workshop on Job Scheduling Strategies for Parallel Processing, 1998

[5] K. Czajkowski, S. Fitzgerald, I. Foster, C. Kesselman: Grid Information Services for Distributed Resource Sharing. Proc. 10th IEEE International Symposium on High-Performance Distributed Computing (HPDC-10), IEEE Press, 2001.

[6] I. Foster, C. Kesselman, G. Tsudik, S. Tuecke: A Security Architecture for Computational Grids. In Proc. of the 5th ACM Conference on Computer and Communication Security, November 1998.

[7] I. Foster, C. Kesselman, S. Tuecke: The Anatomy of the Grid. International Journal of Supercomputer Applications, 15(3), 2001.

[8] I. Foster, C. Kesselman: The Grid: Blueprint for a New Computing Infrastructure. Morgan Kaufmann Publishers, 1999.

[9] I. Foster, C. Kesselman: The Globus Toolkit. In [8] pp. 259-278.

[10] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, B. Manchek, V. Sunderam: PVM: Parallel Virtual Machine - A User's Guide and Tutorial for Network Parallel Computing. MIT Press, Cambridge, MA, 1994

[11] A. S. Grimshaw, W. A. Wulf, J. C. French, A. C. Weaver, P. F. Reynolds: Legion: The Next Logical Step Toward a Nationwide Virtual Computer. Technical report No. CS-94-21. June, 1994.

[12] A. S. Grimshaw, W. A. Wulf: Legion - A View From 50,000 Feet. Proceedings of the Fifth IEEE International Symposium on High Performance Distributed Computing, IEEE Computer Society Press, Los Alamitos, California, August 1996

[13] W. Gropp and E. Lusk and N. Doss and A. Skjellum: A high-performance, portable implementation of the MPI message passing interface standard. Parallel Computing, Vol. 22 No. 6, 1996, pp. 789-828

[14] Y. Gurevich: Evolving Algebras 1993: Lipari Guide. In: Specification and Valdation Methods, Ed. E. Börger, Oxford University Press, 1995. pp 9-36.

[15] Y. Gurevich: Evolving Algebras: An Attempt to Discover Semantics. In: Current Trends in Theoretical Computer Science. Eds. G. Rozenberg, A. Salomaa, World Scientific, 1993. pp. 266-292.

[16] Y. Gurevich: May 1997 Draft of the ASM Guide. http://www.eecs.umich.edu/gasm/papers/guide97.html

[17] M. Humprey, F. Knabbe, A. Ferrari, A. Grimshaw: Accountability and Control of Process Creation in the Legion Metasystem. Proc. of the 2000 Network and Distributed System Security Symposium NDSS2000, San Diego, California, February 2000.

[18] G. Lindahl, A. Grimshaw, A. Ferrari, K. Holcomb: Metacomputing - What's in it for me. White paper. http://legion.virginia.edu/papers.html