Conceptual and Implementation Models for the Grid

MANISH PARASHAR, SENIOR MEMBER, IEEE AND JAMES C. BROWNE

Invited Paper

The Grid is rapidly emerging as the dominant paradigm for wide area distributed application systems. As a result, there is a need for modeling and analyzing the characteristics and requirements of Grid systems and programming models. This paper adopts the well-established body of models for distributed computing systems, which are based upon carefully stated assumptions or axioms, as a basis for defining and characterizing Grids and their programming models and systems. The requirements of programming Grid applications and the resulting requirements on the underlying virtual organizations and virtual machines are investigated. The assumptions underlying some of the programming models and systems currently used for Grid applications are identified and their validity in Grid environments is discussed. A more in-depth analysis of two programming systems, the Imperial College E-Science Networked Infrastructure (ICENI) and Accord, using the proposed definitions' structure is presented.

Keywords*—Distributed systems, Grid programming models, Grid programming systems, Grid system definition.*

I. INTRODUCTION

Goals: The goal of the Grid [39] concept is to enable a new generation of applications combining intellectual and physical resources that span many disciplines and organizations, providing vastly more effective solutions to important scientific, engineering, business and government problems. These new applications must be built on seamless and secure discovery, access to, and interactions among resources, services, and applications owned by many different organizations and institutions. One family of such applications includes scientific and engineering simulations of complex physical phenomena that symbiotically and opportunistically combine computations, experiments, observations, and real-time data. Examples include weather prediction, earthquake models, interacting black holes and neutron stars, formations of galaxies, and subsurface flows in oil reservoirs and aquifers. Another family of Grid applications includes pervasive applications that leverage the pervasive information Grid to continuously manage, adapt, and optimize our living context, crisis management applications that use pervasive conventional and unconventional information for crisis prevention and response, medical applications that use *in vivo* and *in vitro* sensors and actuators for patient management, and business applications that use anytime–anywhere information access to optimize profits.

Requirements: The requirements for attaining this goal include a conceptual framework for realization of such global scale applications, a set of standards defining and specifying the entities in this framework and their interactions, and implementations of middleware/infrastructure for realizing entities that conform to the standards. The elements[1] of the conceptual framework are the following.

- Virtual organizations: organizations composed from resources provided by multiple concrete organizations for an agreed-upon purpose.
- Programming systems: the models and abstractions to support the formulation, instantiation, and management of an application.
- Execution environments: the specific resource configurations and system services for the execution of an application.

Standards are operational definitions of the entities in the framework including policies and procedures for life-cycle management of virtual organizations, applications, and execution environments. Middleware and infrastructure services

Manuscript received March 1, 2004; revised June 1, 2004.

M. Parashar is with the Department of Electrical and Computer Engineering, Rutgers University, Piscataway, NJ 08854 USA (e-mail: parashar@caip.rutgers.edu).

J. C. Browne is with the Department of Computer Science, University of Texas, Austin, TX 78712 USA (e-mail: browne@cs.utexas.edu).

This work was supported in part by the National Science Foundation under Grants ACI-9984357, EIA-0103674, EIA-0120934, ANI-0335244, CNS-0305495, CNS-0426354, and IIS-0430826 awarded to M. Parashar and under Grants 0103725 and ACI-0305644 awarded to J. C. Browne.

Digital Object Identifier 10.1109/JPROC.2004.842780

[1]See Section II for definitions of some of these entities and the relationships among them.

support the definition, instantiation, and life-cycle management of virtual organizations, applications, and execution environments in conformance with the standards.

Current Status: A service-oriented architecture for defining virtual organizations and execution environments for applications has been formulated and is available as standards documents proposed by the Global Grid Forum [3]. Further, toolkits (middleware/infrastructure systems) enabling life-cycle management of virtual organizations in conformance to the standards are also available (e.g., Globus [4], Unicore [11]). The situation with respect to programming systems is much less satisfactory. Several programming systems have been proposed and implemented (see Section IV-D for representative systems), spanning a variety of programming models and ranging in robustness from commercially supported to research prototypes. However, these programming systems are mostly based on programming models carried over from programming systems developed for sequential or multiprocessor programming systems. As a result, these systems are based on system abstractions and assumptions that may not be realizable in a Grid environment.

Grid environments are inherently large, heterogeneous, dynamic, and unreliable. Furthermore, Grid applications are similarly complex, heterogeneous, and highly dynamic in their behaviors and interactions. Together, these challenges result in application development, configuration, and management complexities and uncertainties, which must be addressed by the Grid programming system, the Grid middleware infrastructure, or both.

This paper has three objectives: 1) to understand the programming requirements of Grid environments and the characteristics of the underlying virtual organizations; 2) to investigate the assumptions made by the abstract machines underlying existing programming models for distributed systems, and to study their validity in Grid environments; and 3) to study existing Grid programming systems that address key requirements identified in this paper.

Related Work: Recent years have seen many efforts focused on the definition and analysis of Grid system concepts [17], [40], [41], [83] and programming models and systems [64], [66], [67]. Reference[41] outlines the architecture of a virtual organization, while [40] defines the structure of a Grid application. Nemeth and Sunderam in [83] focus attention on the differences between "conventional distributed systems" and Grids and develop abstract models of both in terms of abstract state machines [52]. This work is most similar in goals to this paper. In [66], Lee *et al.* give an analysis and survey of Grid programming models from the perspective of parallel/distributed programming. In [67], Lee and Talia survey existing models which are currently used to implement applications on Grids. Their classification includes state models, message passing models, Remote Procedure Call (RPC) models, workflow models, peer-to-peer models, frameworks, component models, Web services, and coordination models. Bal *et al.* [20] survey tools for building Grid applications. They also present a survey of existing Grid programming models and systems with a viewpoint similar to Lee and Talia [67], and provide case studies of the application of some of the Grid application development tools.

The approach taken in the current paper both complements and contrasts these studies by applying a distributed systems perspective to Grid systems. In contrast to [83], we utilize the similarities between distributed systems and Grids as a starting point for our definitions and models of Grids. We incorporate in our analysis issues such as uncertainty of common knowledge, faults, unreliability of resources, and self-management of behaviors. These issues have previously received relatively little attention in the context of Grids and Grid programming models and systems.

Paper Outline: The rest of this paper is organized as follows. In Section II we present a formal view of Grid computing and define its components. In Section III we review the fundamentals of distributed systems and programming in the context of Grid computing. In Section IV we outline the challenges and requirements of programming Grid applications and discuss current research in Grid programming. In Section V we present case studies of two programming systems that address some of the core challenges of programming Grid applications identified in this paper. Section VI presents a conclusion.

II. Formal Definitions of Grid Concepts

As illustrated in Fig. 1, Grid computing builds on implementation and conceptual models. Implementation models address the virtualization of organizations which leads to Grids, the creation and management of virtual organization as goal-driven compositions of organizations, and the instantiation of virtual machines as the execution environment for an application. Conceptual models define abstract machines that support programming models and systems to enable application development. These concepts are formally defined below.

Definition: Organization

An organization ($\mathbf{o}$) is a tuple (**RS, I, PY, PL**) where

- $\mathbf{RS}$ is the set ($\{\mathbf{rs}\}$) of resources and services supported by $\mathbf{o}$;
- $\mathbf{I}$ is the interface for accessing $\mathbf{RS}$;
- $\mathbf{PY}$ is the set ($\{\mathbf{py}\}$) of policies for the operation of $\mathbf{o}$;
- $\mathbf{PL}$ is the set ($\{\mathbf{pl}\}$) of protocols for the implementation of $\mathbf{PY}$.

Definition: Interface

The interface ($\mathbf{I}$) of an $\mathbf{o}$ is the set ($\{\mathbf{i}\}$) of externally visible operations through which the resources and services of the $\mathbf{o}$ are accessed.

Definition: Policy

A policy $\mathbf{py}$ is a set of rules specifying the admissible patterns of use for some type of resource and/or service.

Definition: Protocol

A protocol $\mathbf{pl}$ defines the sequence of interactions among the resources and services of an $\mathbf{o}$ to implement a policy $\mathbf{py}$.

Discussion:

- An organization is an entity that is established to accomplish some goals and that provides resources and services to enable the goals to be achieved.
- Policies also control access to services and resources.

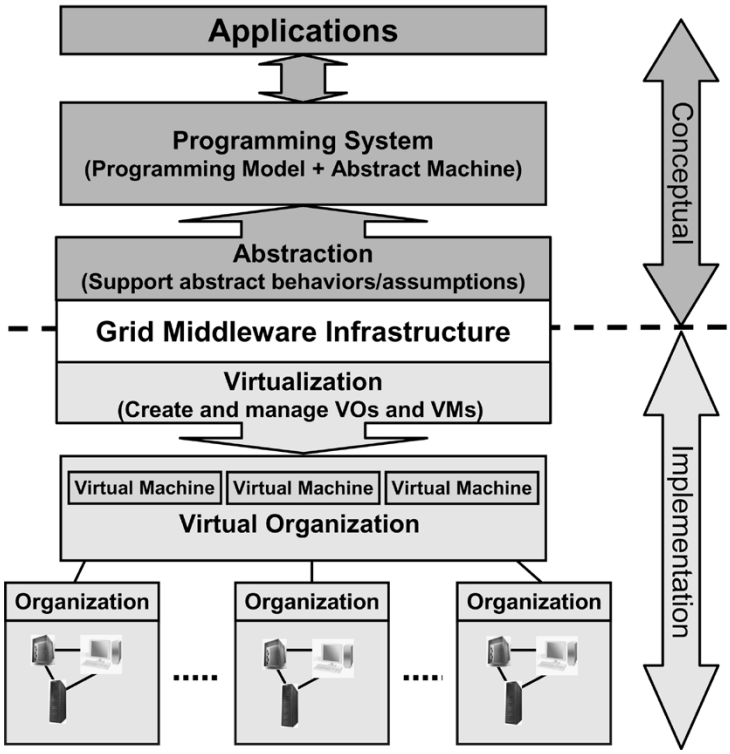

Fig. 1. Conceptual and implementation models for Grid computing.

Definition: Virtual Organization

A virtual organization ($\mathbf{vo}$) is a tuple ($\mathbf{O}, \mathbf{RS}, \mathbf{I}, \mathbf{PY}, \mathbf{PL}$) where

- $\mathbf{O}$ is a set of concrete organizations ($\{\mathbf{o}\}$) that collaborate as *peers* to form an instance of a $\mathbf{vo}$;
- $\mathbf{RS}$ is the set ($\{\mathbf{rs}\}$) of resources and services supported by $\mathbf{vo}$.
- $\mathbf{I}$ is the interface for accessing $\mathbf{RS}$;
- $\mathbf{PY}$ is the set ($\{\mathbf{py}\}$) of policies for the operation of $\mathbf{vo}$;
- $\mathbf{PL}$ is the set ($\{\mathbf{pl}\}$) of protocols for the implementation of $\mathbf{PY}$.

Definition: Policy

A policy $\mathbf{py}$ is a set of rules specifying the admissible patterns of use for some type of resource and/or service for some subset ($\{\mathbf{o}'\}$) of the $\mathbf{O}$ comprising the $\mathbf{vo}$.

Definition: Protocol

A protocol $\mathbf{pl}$ defines the sequence of interactions among some subset ($\{\mathbf{o}'\}$) of the $\mathbf{O}$ comprising the $\mathbf{vo}$ that implement a policy $\mathbf{py}$.

Discussion:

- A virtual organization is created to enable attainment of goals (e.g., execution of applications) or implementation of services that are not feasible within the capabilities of a single concrete organization.
- A virtual organization is a collaboration among *peers*. The collaborating organizations have agreed-upon a commonly desired set of goals. Each peer within the virtual organization may both contribute resources and services to the virtual organization and use resources and services provided through the virtual organization.

Definition: Virtual Machine

A virtual machine ($\mathbf{vm}$) is a tuple ($\mathbf{vo}, \mathbf{RS}, \mathbf{OP}, \mathbf{SR}$) where

- $\mathbf{RS}$ is the set of resources and services ($\{\mathbf{rs}\}$) belonging to the virtual organization $\mathbf{vo}$ from which the $\mathbf{vm}$ is composed;
- $\mathbf{OP}$ is the set of operations ($\{\mathbf{op}\}$) executed by a $\mathbf{vm}$. An $\mathbf{op}$ is specified as a functional interface and a behavior;
- $\mathbf{SR}$ is the set of sequencing rules ($\{\mathbf{sr}\}$) which constrain ordering and composition of operations.

Discussion:

- A virtual machine is a composition of resources within a virtual organization and defines an instance of the execution environment of an application.
- A virtual machine is implemented using the set of middleware services ($\{\mathbf{ms}\}$) made available through its containing $\mathbf{vo}$.
- The behaviors of an operation include quality of service guarantees made by the virtual machine for this operation. Quality of service guarantees should cover reliability and performance. The implementation of the virtual machine must ensure conformance to these specified behaviors. For example, if message delivery is guaranteed to be reliable, ordered and exactly once, then the implementation of the virtual machine must ensure that this behavior is followed without exception.

Definition: Programming System

A programming system ($\mathbf{ps}$) is a tuple ($\mathbf{pm}, \mathbf{l}, \mathbf{am}$) where

- $\mathbf{pm}$ is a programming model;

$\mathbf{l}$ is a language syntax for instantiating the programming model.[2]

$\mathbf{am}$ is an abstract machine on which the $\mathbf{pm}$ of the programming system is defined.

Definition: Abstract Machine

An abstract machine $(\mathbf{am})$ is a tuple $(\mathbf{OP}, \mathbf{SR})$ where

$\mathbf{OP}$ is the set of operations $(\{\mathbf{op}\})$ executed by an $\mathbf{am}$. An $\mathbf{op}$ is specified as a functional interface and a behavior;

$\mathbf{SR}$ is the set of sequencing rules $(\{\mathbf{sr}\})$ which constrain ordering and composition of operations.

Discussion:

- An abstract machine is defined independent of a virtual machine and may be realized on many different virtual machines.
- The definitions of behaviors of an operation include specification of attributes such as completion, atomicity, reliability, etc., that are assumed by the programming model.
- The operations and behaviors of the virtual machine upon which the abstract machine is realized must support realization of the behaviors specified for the operations of the abstract machine.

Definition: Programming Model

A programming model $(\mathbf{pm})$ is a tuple $(\mathbf{E}, \mathbf{OP}, \mathbf{MC}, \mathbf{am})$ where

$\mathbf{E}$ is a set of entity types $(\{\mathbf{e}\})$ which may be acted upon by the members of the set of operations $(\mathbf{OP})$ and interact, communicate, and are composed under the model of computation $(\mathbf{MC})$;

$\mathbf{e}$ is a tuple $(\mathbf{t}, \mathbf{n}, \mathbf{I}, \mathbf{S}, \mathbf{B})$ where $\mathbf{t}$ is a type, $\mathbf{n}$ is a name, $\mathbf{I}$ is an interface, $\mathbf{S}$ is a set of states $(\{\mathbf{s}\})$, and $\mathbf{B}$ is a behavior model;

$\mathbf{T}$ is a set $\{\mathbf{t}\}$ of base types;

$\mathbf{N}$ is a naming model which defines binding and resolution of names, $\mathbf{n}$;

$\mathbf{I}$ is the set of operations $(\{\mathbf{i}\})$ implemented by the entity and the set of dependencies $\{\mathbf{d}\}$ of the entity on other entities (i.e., *requires* dependencies). The operations implemented by the entity define *provides* dependencies which may satisfy *requires* dependencies of other entities. $\mathbf{I}$ may change at runtime;

$\mathbf{S}$ is a set of states $(\{\mathbf{s}\})$ where each $\mathbf{s}$ is unique for an entity;

$\mathbf{B}$ is a behavior model in which properties such as persistence, operation completion, failure modes, etc., are specified. A behavior $\mathbf{b}$ may be specified for each operation implemented by the entity.

$\mathbf{OP}$ is the set of operations $(\{\mathbf{op}\})$ which are defined upon instances of the members of $\mathbf{E}$ separately from the operations implemented by the entities;

$\mathbf{MC}$ a model of computation consists of a composition model, coordination model and communication model and is a tuple $(\mathbf{CR}, \mathbf{COORD}, \mathbf{COMM})$ where

$\mathbf{CR}$ is a set $(\{\mathbf{cr}\})$ of rules for composition of a set of entities, $\{\mathbf{e}\}$ into an entity, $\mathbf{e}'$. Composition binds a *requires* dependence of a component to one or more *provides* dependencies of other components. Bindings can occur at compile time, link-time or runtime. Runtime binding (dynamic structuring) requires a discovery mechanism;

$\mathbf{COORD}$ is a model of coordination. A model of coordination specifies allowed orderings of execution among entities;

$\mathbf{COMM}$ is a model of communication. A model of communication specifies the mechanisms through which the interactions implementing composition and coordination are implemented, e.g., point-to-point messages, broadcast, shared name spaces, etc.

$\mathbf{am}$ is the abstract machine implied by the programming model that supports the set of entities $\mathbf{E}$, their behaviors $\mathbf{I}$, and the behaviors of the set of operations $\mathbf{OP}$.

Discussion:

- Entities are the subjects of programming. Entities are the first-class citizens of a programming model. Examples of entity types include processes, threads, functions, objects, components, and services. New entity types can be recursively created by composition of existing entity types. In Grid programming models, an entity is a unit not only for composition, but also for deployment and execution.
- The set $\mathbf{OP}$ may include creation, deletion, binding and resolution of names, binding to resources, discovery, and transformations on the states and types of instances of entities depending on the functionality of the entities themselves.
- Operations have states (in progress, completed, failed) and, once initiated, the states of an operation can be detected.
- The interface of an entity may depend upon its current state.
- The abstract machine implied by the programming model defines the assumptions made by the programming model about the capabilities and guarantees provided by the execution environment of any application implemented using the model.

III. Fundamentals of Distributed Systems

In this section we highlight fundamental issues of behavior for distributed systems in general (and Grid systems in particular). Issues include system definition and control with a focus on complexity and uncertainty and programming models and frameworks for creating instances (applications)

[2]Language syntax and representation does not impact the semantics with which we are concerned and so are not further discussed here. The languages for most current Grid programming systems are extensions of the syntax of existing sequential or parallel programming languages;

of distributed systems. The motivation is that Grid systems are specific instances of distributed systems.

A. System Definition and Control

Definition: Distributed system—a distributed system is a collection of logically or physically disjoint entities which have established a process for making collective decisions.

Collective decisions depend on common knowledge [53] held by the participants in the decision process at the time a decision is made. Common knowledge may be known by the participants prior to being a part of the distributed system or may be acquired by a consensus process [102] during execution. However, consensus can be established with complete certainty only in very restricted instances of distributed systems [102] and seldom in real distributed systems. There is, therefore, intrinsic uncertainty in the common knowledge of most distributed systems. Uncertainty can be bounded through use of interaction protocols which narrow the time span of uncertainty and/or the elements of system state which are uncertain (e.g., timed behavior assumptions [32]).

The fundamental requirements for implementing a distributed system are a capability for creation and management of a system and a capability for control of the system through decision processes which operate correctly in the presence of uncertainty. The algorithms which can be used in the decision process depend upon the properties of the computer, communication, and data resources which comprise the distributed system, and particularly upon the assumed reliability, failure modes, and/or timed behavior of the computer and communication resources.

The distributed systems instantiated by the Grid middleware (called virtual organizations) implicitly assume failure modes, reliability properties, performance, etc., for the resources they assemble into a virtual organization. Often these assumptions are either not explicitly stated or are those of complete reliability. The latter case, which eliminates uncertainty in common knowledge, is clearly unrealistic for real distributed systems.

Definition: Decision Algorithms—Decisions can be abstractly formulated as

```
If (Decision(Current State, Request))
  then
State = Transition(Current State, Request);
```

Decision is a specification for execution of a change of state. *Decision* is a function which evaluates to true if a *Request* for a state change is accepted. *Transition* is a function which transforms the current state to a new state that may result in a change in the local states of the participants and/or interactions with other participants. As mentioned above, the requirement for collective decisions in the presence of uncertainty requires decision algorithms to make assumptions about behavior, reliability, and guarantees. A key objective of this paper is to identify and characterize the assumptions made by programming frameworks and infrastructures used by Grid systems.

B. Central Versus Distributed Control

Ideally the state of the system to which *Decision* is applied is complete and accurate and *Decision* is a complete function. This is straightforward in a single-site system where system state can be maintained consistent in a local data structure and *Decision* is applied to this data structure. Distributed control implies that *Decision* is partitioned among the entities composing the system and coupled by communication protocols. Central control of a distributed system can, however, be implemented by gathering the system state at a single site, executing *Decision* at this site and propagating the decision to the other sites. In fact, this is what is normally done in most Grid systems. Execution of the *Transition* function is, however, intrinsically distributed. This leads to intrinsic uncertainty in system state (common knowledge) since, for example, a resource can fail during the time span of making the decision and executing the state transition, perhaps rendering the decision invalid. Grid systems should make provision for dealing with this uncertainty in cases where complete reliability cannot reasonably be assumed.

Distributed control implies that each entity makes decisions following a commonly agreed-upon process and based upon agreed-upon common knowledge. An important aspect of a distributed system which utilizes distributed control is the specification of the commonly agreed-upon processes and the common knowledge upon which distributed control is based. These aspects should be precisely specified for Grid systems which utilize distributed control.

A key objective of this paper is to characterize the models for control, specifications for commonly agreed processes, specifications and mechanisms for maintaining common knowledge, and the conditions under which consensus can be attained under the assumptions made by current Grid systems concerning reliability, performance, failure modes, etc.

C. Programming Systems for Distributed Applications

As defined in Section II, a programming system for distributed applications consists of three key components: 1) a programming model and associated language that defines a set of abstractions that the programmer uses to define the behavior of application elements and their interactions; 2) an underlying abstract machine that defines the execution context for the applications and embodies the assumptions made by the programming model about the capabilities, behaviors and qualities of services of the underlying environments; and 3) an infrastructure that provides the services necessary for creating, managing, and destroying the virtual machine upon which the abstract machine is realized and the abstractions assumed by the programming model satisfied.

The typical tradeoff in programming systems for distributed environments lies in these assumptions made by the programming model about common knowledge and the behavior of the abstract machine. While stronger assumptions (e.g., computation or communication reliability or fail-stop behaviors) reduce the responsibilities of application developers and the complexity of application development,

they also increase the complexity of the middleware which must ensure that the assumptions are satisfied. It also makes application more brittle, causing them to fail whenever an assumption is not completely satisfied by the middleware. For example, the middleware may not be able to support reliable delivery of messages or exactly once invocation semantics for RPCs as the scale and/or rate of change of a system increases, causing applications built upon these assumptions to fail. On the other hand, weakening the assumptions made in the programming model reduces the complexity of the middleware allowing it to be more robust. However, it increases the requirements of the programming model and the responsibility of the developer by exposing complexity and uncertainty.

D. Programming Grid Applications

Grid software systems typically provide capabilities for: 1) creating a transient "virtual organization" or virtual resource configuration; 2) creating virtual machines composed from the resource configuration of the virtual organization; 3) creating application programs to execute on the virtual machines; and 4) executing and managing application execution. Therefore, most Grid software systems implicitly or explicitly incorporate a programming model. These programming models implicitly or explicitly assume an underlying abstract machine with specific execution behaviors including assumptions about reliability, failure modes, etc. In previous efforts focusing on identifying and understanding requirements and models for programming Grids [83], these assumptions or their implications have often not been explicitly studied or sometimes even explicitly stated. Further, proposed Grid programming systems [67] have similarly not specified the assumptions made with respect to the underlying abstract machine. As we note above, understanding these assumptions is important, since they define capabilities and limitations of Grid applications and the core requirements for the Grid middleware infrastructure.

In the rest of this paper, we first identify the key sources of complexity and uncertainty in Grid environments. We then analyze existing distributed programming systems to extract, make explicit the underlying assumptions, and investigate their applicability and limitations as Grid programming models.

IV. Grid Computing—Challenges, Requirements and Approaches

This section begins with a catalog of the characteristics of Grid computing systems and the challenges they present with respect to the programming of applications. Distributed programming models and systems that form the basis of most existing Grid programming and application development systems are then characterized. Current Grid research in programming and application development systems is then discussed in the context of the distributed programming systems upon which they are based.

A. Characteristics of Grid Execution Environments and Applications

The characteristics of Grid execution environments and applications are as follows.

Heterogeneity: Grid environments aggregate large numbers of independent and geographically distributed computational and information resources, including supercomputers, workstation clusters, network elements, data storages, sensors, services, and Internet networks. Similarly, applications typically combine multiple independent and distributed software elements such as components, services, real-time data, experiments, and data sources.

Dynamism: The Grid computation, communication, and information environment is continuously changing during the lifetime of an application. This includes the availability and state of resources, services, and data. Applications similarly have dynamic runtime behaviors in that the organization and interactions of the components/services can change.

Uncertainty: Uncertainty in Grid environment is caused by multiple factors, including: 1) dynamism, which introduces unpredictable and changing behaviors that can only be detected and resolved at runtime; 2) failures, which have an increasing probability of occurrence and frequencies as system/application scales increase; and 3) incomplete knowledge of global system state, which is intrinsic to large decentralized and asynchronous distributed environments.

Security: A key attribute of Grids is flexible and secure hardware/software resource sharing across organization boundaries, which makes security (authentication, authorization, and access control) and trust critical challenges in these environments.

B. Requirements for Grid Programming Systems

These characteristics impose requirements on the programming systems for Grid applications. Grid programming systems must be able to specify applications which can detect and dynamically respond during execution to changes in both, the execution environment and application states. This requirement suggests that: 1) Grid applications should be composed from discrete, self-managing components which incorporate separate specifications for all of functional, nonfunctional, and interaction–coordination behaviors; 2) the specifications of computational (functional) behaviors, interaction and coordination behaviors and nonfunctional behaviors (e.g., performance, fault detection and recovery, etc.) should be separated so that their combinations are composable; and 3) the interface definitions of these components should be separated from their implementations to enable heterogeneous components to interact and to enable dynamic selection of components.

Given these features of a programming system, a Grid application requiring a given set of computational behaviors may be integrated with different interaction and coordination models or languages (and *vice versa*) and different specifications for nonfunctional behaviors such as fault recovery and

QoS to address the dynamism and heterogeneity of the application and the underlying environments.

In the rest of this section, we study these programming models and frameworks on which current Grid programming systems are built. The goal of this study is not to be critical of these systems by to analyze their underlying assumptions and to understand their capabilities and limitations in addressing the issues outlined above.

C. Distributed Programming Models/Systems

Dominant programming models and frameworks for distributed systems can be broadly classified as: 1) models based on the addition of communication/interaction models and mechanism to sequential programming models (e.g., Message Passing Interface (MPI)/Parallel Virtual Machine (PVM), RPC, Linda); 2) distributed object models [e.g., Commom Object Request Broker Architecture (CORBA)]; 3) component models [e.g., JavaBean, CORBA Component Model (CCM), Common Component Architecture (CCA)]; and 4) service models (e.g., Web Service, WRF).[3] One difference between component-based models and service-based models is that traditionally a component is defined in the context of an application and only has meaning during the lifetime of the application, while a service exists across applications. In this section we study these programming models and frameworks. The goal of this study is to analyze their underlying assumptions and to understand their capabilities and limitations in addressing the issues outlined above.

1) Communication Models and Frameworks: The communication models/frameworks supplement existing sequential programming models to enable the interactions between distributed entities. These include message passing models, RPC models, and shared-space models.

Message Passing Models: Message passing models provide messaging abstractions that enable entities defined by sequential programming models to communicate. For example systems such as MPI [7], [95] and PVM [46] provide message passing operators to sequential languages such as Fortran and C. These models primarily address distribution. They provide support for creating, operating on, and destroying virtual machines. The abstract machine model for these assumes stable common knowledge about participants, their identifiers, and their behaviors. It also assumes that this knowledge is always maintained in spite of any change in the system. Further, the abstract machine assumes the virtual machine to be reliable or at least support reliable communications.

MPI, the dominant message passing system focuses on performance, and does not support heterogeneity, dynamism or uncertainty. Further, it assumes that all interacting processes are trusted and does not address security. MPICH-G2 [59], a Grid-enabled implementation of the MPI, hides the heterogeneity using services provided by Globus toolkits. MPI-2 [51] does support dynamic creation of new processes and runtime modification of the processor set. The Harness/PVM [46], [80] effort also addresses heterogeneity and process dynamism in addition to distribution. Harness/PVM also provides a fault-tolerant extension to MPI (FT-MPI [37]) that can tolerate process failures. Other similar fault-tolerant extensions include MPI/FT [22] and MPICH-V2 [28]. These systems can tolerate process failures but assume fail-stop behavior and an abstract machine with reliable communication.

RPC: RPC [27] mechanisms also support process interactions in a distributed environment by extending the notion of a conventional procedure call to operate across the network. In addition to distribution, RPC implementations also address heterogeneity by using neutral interface description languages. However, RPC assumes that common knowledge about the name/identifier, address, and the existence of the end-points, and the syntax and semantics of the interface are known *a priori* (compile-time). Further, it assumes reliable message delivery and provides mechanisms for managing failures at the application level. The RPC model does not address dynamism or security.

Shared-Space Model: The shared-space model supports interaction/coordination by exchanging data using a shared storage space. The shared-data space implements common knowledge which can be shared by all of the entities in an application. Various implementations have different properties. The shared space may be persistent, associatively addressable, transactionally secure, and capable of exchanging executable content. The shared-space interaction/coordination model was initially proposed in the form of the Linda tuple-space [29], [47] and was more recently adapted by Jini [105] as its JavaSpace [42] service. The traditional tuple-space model has been recently extended to incorporate reactive behaviors by systems such as TuCSoN [86], [90]. A key feature of the shared-space model is that it decouples the interacting entities in space, time and synchronization. As a result, the shared-space model naturally addresses distribution and dynamic structuring among entities and can manage heterogeneity as long as the syntax, semantics, and representation of the shared data is neutrally defined. Further, this model can also address failure of the interacting entities. However, scalable, consistent, and robust distributed implementations of shared spaces remain a challenge. Current implementations of the model do not address security.

2) Distributed Object Models: Unlike the systems described above that essentially address only communication aspects, the distributed object models provide more complete support for parallel/distributed applications, including life-cycle management, location and discovery, interaction and synchronization, security, failure, and reliability [21]. CORBA [1], [23], one of the dominant distributed object models, enables the secure interactions (based on RPCs, method invocations, and event notification) between distributed and heterogeneous objects using interfaces described by a language-neutral interface definition language and through a middleware consisting of object resource brokers and interoperability protocols [e.g., General Inter-ORB Protocol (GIOP), Internet Inter-ORB Protocol (IIOP)].

[3]See [20] and [64] for similar classifications.

CORBA primarily addresses distribution and heterogeneity. CORBA also provides limited support for dynamism via dynamic invocation [Dynamic Skeleton Interface (DSI)/Dynamic Invocation Interface (DII)] and late binding, which enables customization at deployment time. However, the interacting objects and interaction are tightly coupled. Further, the model assumes *a priori* (compile-time) knowledge of the syntax and semantics of interfaces as well as the interactions required by the applications. The Java distributed object model [38] is very similar to CORBA, but supports Java objects.

3) Component Models: Component models address increasing software complexity and changing requirements by enabling the construction of systems as assemblies of reusable components. Components are reusable units of composition, deployment, and execution and life-cycle management [96]. Components are completely specified by their interfaces. Current component models include CCM, JavaBeans, and CCA.

CCM [106] extends the CORBA distributed object model and similarly supports distribution, heterogeneity, and security. It also supports dynamic instantiation and runtime customization of components. However, CCM inherits some of the limitations of CORBA including the requirement for a prior knowledge about interfaces and interactions. JavaBeans [35] is a Java-only component model which addresses similar issues. JavaBeans also support runtime bean customization.

CCA [15] defines a component model especially for scientific applications. The model primarily addresses the heterogeneity and the separation of interface and implementation. CCA targets high-performance parallel applications and uses functional calls for intercomponent interactions. While it does not support runtime customization of components, it does allow components to be replaced. It does not address failure or security and assumes all components are trusted.

4) Service Models: Service-based models such as Web Services [12], [30] and Open Grid Services Architecture (OGSA)/Web Service Resource Framework (WSRF) [33], [34] target loosely coupled and highly dynamic systems and support "just-in-time" integration of applications without requirements for *a priori* knowledge of services or interfaces. The Web service model supports heterogeneity through XML-described (e.g., WSDL) interfaces. It addresses dynamism by enabling service customization based on application requirements and the execution environment. The Web service model, however, assumes that services are stateless and that the execution environment remains relatively static during service execution.

The WSRF [34] builds on Web Services and experiences from the OGSA [40] is concerned primarily with the creation, addressing, inspection, and lifetime management of stateful resources. WSRF supports transient services and addresses dynamism by supporting customization during service instantiation. It also addresses security by building on Grid protocols and mechanisms. However, WSRF assumes that the execution environment remains static during service execution. Further, it makes strong assumptions about the reliability of the underling mechanism for remote service invocation.

An important aspect related to component- and service-based models is composition. Composition has been addressed by systems such as Symphony [73], METEOR [81], COSMOS [49], Aurora [76], SWORD [89], and DySCo [88]. Symphony is a Java-based static composition framework based on JavaBeans. METEOR addresses runtime adaptability and management of a composed workflow. COSMOS and Aurora are similar systems focused on services in electronic commerce applications. SWORD uses a rule-based expert system to find composition plans for informational services. DySCo enables dynamic service composition for stateless services and is based on the idea of functional incompleteness and multiparty orchestration.

Composition and flow specification languages for Web and Grid services include BPEL [18], Grid Services Flow Language (GSFL) [104], Web Services Flow Language (WSFL) [68], XLANG [100], ebXML [2], and Web Service Choreography Interface (WSCI) [19].

D. Grid Computing Research

Grid computing research efforts over the last decade can be broadly divided into efforts addressing the realization of virtual organizations and those addressing the development of Grid applications. The former set of efforts have focused on the definition and implementation of the core services that enable the specification, construction, operation, and management of virtual organizations and instantiation of virtual machines that are the execution environments of Grid applications. Services include: 1) security services to enable the establishment of secure relationships between a large number of dynamically created subjects and across a range of administrative domains, each with its own local security policy; 2) resource discovery services to enable discovery of hardware, software and information resources across the Grid; 3) resource management services to provide uniform and scalable mechanisms for naming and locating remote resources, support the initial registration/discovery and ongoing monitoring of resources, and incorporate these resources into applications; 4) job management services to enable the creation, scheduling, deletion, suspension, resumption, and synchronization of jobs; and 5) data management services to enable accessing, managing, and transferring of data, and providing support for replica management and data filtering. Efforts in this class include Globus [4], Unicore [11], Condor [99] and Legion [50]. Other efforts in this class include the development of common application programming interfaces (APIs), toolkits, and portals that provide high-level uniform and pervasive access to these services. These efforts include the Grid Application Toolkit (GAT) [16], Distributed Virtual Computer (DVC) [97], and the Commodity Grid Kits (CoG Kits) [65]. These systems often incorporate programming models or capabilities for utilizing programs written in some distributed programming model. For example, Legion implements an object-oriented programming model, while

Globus provides a capability for executing programs utilizing message passing.

The second class of research efforts, which is also the focus of this paper, deals with the formulation, programming, and management of Grid applications. These efforts build on the Grid implementation services and focus on programming models, languages, tools and frameworks, and application runtime environments. Research efforts in this class include GrADS [24], GridRPC [94], GridMPI [56], Harness [80], Satin/IBIS [84], [85], XCAT [48], [63], Alua [103], G2 [60], J-Grid [77], Triana [98], and the Imperial College E-Science Networked Infrastructure (ICENI) [43]. Very brief sketches of the programming models underlying these systems are presented below. ICENI is covered in more detail in the next section.

These systems have essentially built on, combined, and extended existing models for parallel and distributed computing. For example, GridRPC extends the traditional RPC model to address system dynamism. It builds on Grid system services to combines resource discovery, authentication/authorization, resource allocation, and task scheduling to remote invocations. Similarly, Harness and GridMPI build on the message passing parallel computing model. Satin supports divide-and-conquer parallelism on top of the IBIS communication system. GrADS builds on the object model and uses reconfigurable object and performance contracts to address Grid dynamics. XCAT and Alua extend the component-based model. G2, J-Grid, Triana, and ICENI build on various service-based models. G2 builds on .Net [8], J-Grid builds on Jini [6], and current implementations of Tirana and ICENI build on JXTA [10]. While this is natural, it also implies that these systems implicitly inherit the assumptions and abstractions that underlie the programming models of the systems upon which they are based and, thus, in turn inherit their assumptions, capabilities and limitations.

V. Self-Managing Applications on the Grid

As outlined in the earlier sections of this paper, the inherent scale, complexity, heterogeneity, and dynamism of emerging Grid environments result in application programming and runtime management complexities that break current paradigms. This is primarily because the abstract machine underlying these models makes strong assumptions about common knowledge, static behaviors and system guarantees that cannot be realized by Grid virtual machines. Addressing these challenges requires redefining the programming framework to address the separations outlined in Section IV. Specifically, it requires: 1) static (defined at the time of instantiation) application requirements and system and application behaviors to be relaxed; 2) the behaviors of elements and applications to be sensitive to the dynamic state of the system and the changing requirements of the application and be able to adapt to these changes at runtime; 3) required common knowledge be expressed semantically (ontology and taxonomy) rather than in terms of names, addresses and identifiers; and 4) the core enabling middleware services (e.g., discovery, messaging) be driven by such a semantic knowledge. In this section we describe two recent Grid programming systems that attempt to address these challenges by enabling self-managing Grid applications.

A. ICENI

ICENI [43]–[45], [54], [79] meets many of the requirements given in Section IV-B for Grid programming systems. Applications are composed from components which are semiautonomous, specifications for functional and nonfunctional behaviors[4] are separated, and interface definitions are separated from implementations. Capabilities for instantiating virtual machines and for optimization of performance are provided.

1) ICENI Architecture: ICENI is a package of middleware built on top of middleware systems for creating virtual organizations. The ICENI middleware layer enables creation of applications and management and optimization of the execution of these applications on computational Grids realized within virtual organizations or "computational communities." Fig. 2, taken from [5], is a schematic of the architecture of ICENI. ICENI was originally implemented upon the virtual organization capability or private domain capability implemented using JINI [6]. It has subsequently been implemented with JXTA [10] providing the virtual organization capability and also with an implementation of OGSI [101] providing the virtual organization capability.

2) Programming System: The elements of the ICENI programming system are summarized in the subsections below following the definition of a programming system given in Section II. The programming system consists of a definitional system for components in which each of the computational behaviors, the interface and the "meaning or semantics" of a component are separately described (see Fig. 3). Capabilities for application instantiation, management, and optimization through performance models are also provided. We have tried, where explicit specification of an element of the definitions in Section II was not available, to infer the specification. The specifications given herein were extracted from papers found on the ICENI Web site, and in particular [43]–[45], [54], and [79].

Programming Model: ***Entities***: The entities of the programming model are components. Each component has a meaning and may have several behaviors each of which may have several implementations. The meaning of a component includes its types, its semantic constraints, and its functionality. A component may implement many different behaviors in terms of its interaction with other components. Finally, each behavior may have many implementations which may have different performance properties. These specifications are bound to input and output ports which constitute the interface of the component.

Operations: Operations on components include registration, discovery, and instantiation. A component is registered through a script called the Software Resource Factor Script

[4]The definition of "behavior" used in the ICENI system is different from the definition of "behavior" used in the definitions of programming systems in Section II of this paper. ICENI behaviors specify what we call interaction/coordination models.

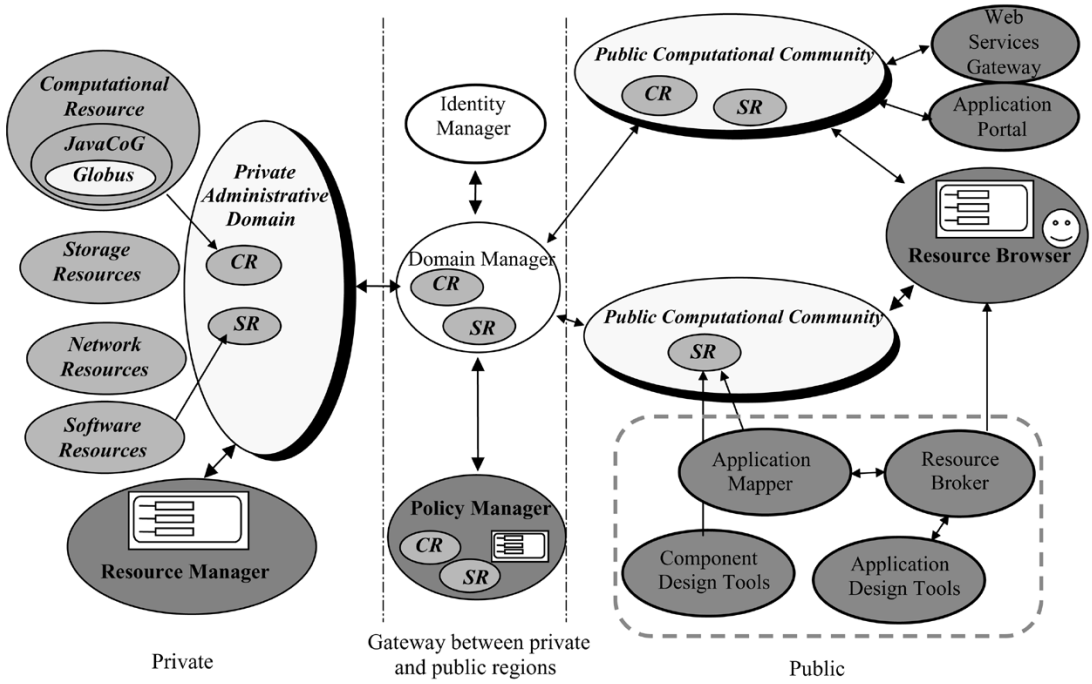

Fig. 2. Architecture of ICENI.

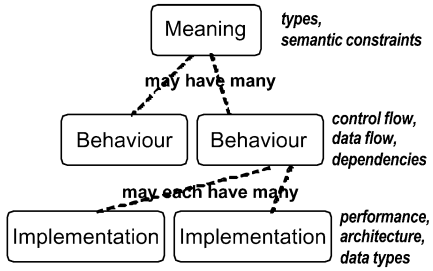

Fig. 3. Component definition and structure.

which generates the data to be stored in an instance of a JINI lookup service. Lookup or discovery is implemented through either the JINI (JXTA or Globus) discovery services depending on the implementation. Instantiation is done through the Jini factory service using Java proxy objects.

Behavior: Performance properties are based on resource specifications which are incorporated into components. No explicit fault behavior model is specified.

Model of Computation: The model of computation which is inferred from the implicit definition of component composition is as follows. ***Composition Rules***: An application is a composition of components. Allowable compositions of components are defined by the type specifications for input/output ports and dependency relations specified in the behavior specification. **Communication Model**: Communication is unidirectional point-to-point defined by bindings of output ports of one component to input ports of another component. The assumed semantics of communication appear to be reliable, ordered, once and only once delivery. **Coordination Model:** The coordination model allows both asynchronous parallel and serial executions of components.

Language: ICENI language facilities are three XML-languages. A component is defined by three programs. The meaning of a component is described in the Component Description Language. Behaviors are described using the Behavioral Definition Language, and the implementation definition uses the Interface Definition Language. These are XML documents from which skeletons for the components are compiled as Java objects.

Context Object: A helper class for the middleware, which provides certain component specific support within the middleware.

Advice: The context object implements this Java interface. This is called "advice," as it should be used by the developer to see what methods the context object offers that can be used within their code. Typically these are methods that allow control to leave the component and other methods by which the component can interact with the middleware.

Interface: The developer's implementation must implement this interface. The middleware will, at times, call methods given in this interface, so they must be supported.

Implementation: This extends the *GridComponent* class which is a part of the ICENI system library—it must possess the methods given in the interface above, and also *initialize*() and *execute*() methods. *Initialize*() passes a context object into the code, which may then be used by the developer. *Execute*() will be called after *initialize*(), and acts as the "main" for each component.

A component is completed by adding an implementation in Java or C/C++ to the implementation skeleton for the component. GUI support for preparing these programs and metadata is provided.

An application is generated by composing components through a simple GUI which enables the establishment of connections between instances of components.

Abstract Machine: We were not able to find an explicit specification of the semantics of the abstract machine

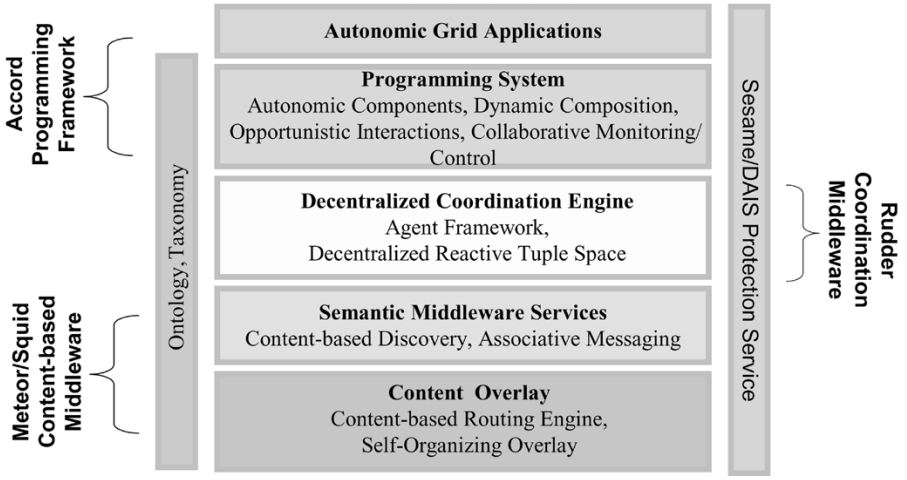

Fig. 4. A schematic overview of AutoMate.

assumed for the execution of ICENI programs. Implementations in Java and Jini result in an implied definition of abstract machine semantics. Jini communicates using Java RMI, which assumes reliable (once-and-only-once semantics) communication and execution. Therefore, this is the default assumption for abstract machine behavior. A Jini implementation could use Jini leasing specifications as a basis for implementation of fault detection and access control.

3) Optimization and Adaptation: ICENI provides an optimization service [45] for the execution of computations on computational Grids. Each component can have associated with it a performance model which can be executed by executing the application with each component represented by its performance model. This can then be used to determine an optimal implementation within a given resource configuration, virtual organization, or "computational community."

The capabilities for semantically based adaptation for ICENI are described in [55] and [79]. The semantic specifications are based on OWL [9], use rule based matching, and are implemented though Jini.

4) Analysis: The ICENI system meets many of the criteria for a complete programming system for Grid applications. The most significant capability ICENI does not address is that of dealing with uncertainty, i.e., faults and monitoring for faults in the environment or failures in application components.

In principle, the ability to deal with faults could be realized in the Behavior Description Language and presumably administered through an additional management component in the ICENI runtime.

B. Project AutoMate: Enabling Self-Managing Grid Applications

Project AutoMate [13], [87] investigates solutions that are based on the strategies used by biological systems to deal with similar challenges of complexity, dynamism, heterogeneity, and uncertainty. This approach, referred to as autonomic computing [61], aims at realizing systems and applications that are capable of managing (i.e., configuring, adapting, optimizing, protecting, healing) themselves. The overall goal of Project AutoMate is to investigate conceptual models and implementation architectures that can enable the development and execution of such self-managing Grid applications. Specifically, it investigates programming models, frameworks, and middleware services that support the definition of autonomic elements, the development of autonomic applications as the dynamic and opportunistic composition of these autonomic elements, and the policy, content, and context driven definition, execution, and management of these applications.

A schematic overview of AutoMate is presented in Fig. 4. AutoMate builds on JXTA [36] and uses OGSI Grid middleware [101] services to define and manage virtual organizations. Components of AutoMate include the Accord [70], [72] programming system, the Rudder [69] decentralized coordination framework and agent-based deductive engine, and the Meteor [57], [58] content-based middleware providing support for content-based routing, discovery, and associative messaging. Project AutoMate additionally includes the Sesame [109] context-based access control infrastructure, the DAIS [107], [108] cooperative-protection services, and the Discover collaboratory [26], [74], [75], [82] services for collaborative monitoring, interaction, and control. The Accord programming system is described below.

Accord: A Programming System for Autonomic Grid Applications: The Accord programming system [70]–[72] addresses Grid programming challenges by extending existing programming systems to enable autonomic Grid applications. Accord realizes three fundamental separations: 1) a separation of computations from coordination and interactions; 2) a separation of nonfunctional aspects (e.g., resource requirements, performance) from functional behaviors; and 3) a separation of policy and mechanism—policies in the form of rules are used to orchestrate a repertoire of mechanisms to achieve context-aware adaptive runtime computational behaviors and coordination and interaction relationships based on functional, performance, and QoS requirements. Using the definitions in Section II, the components of Accord are described below.

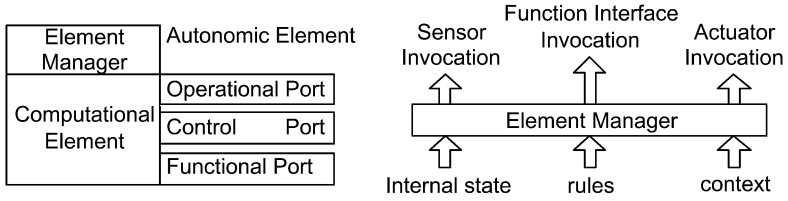

Fig. 5. An autonomic element in Accord.

Accord Programming Model: Accord extends existing distributed programming models, i.e., object, component and service-based models, to support autonomic self-management capabilities. Specifically it extends the entities and composition rules defined by the underlying programming model to enable computational and composition/interaction behaviors to be defined at runtime using high-level rules. The resulting *autonomic elements* and their *autonomic composition* are described below. Note that other aspects of the programming model, i.e., operations, model of computation, and rules for composition, are inherited and maintained by Accord.

Autonomic Elements: An autonomic element extends programming elements (i.e., objects, components, services) to define a self-contained modular software unit with specified interfaces and explicit context dependencies. Additionally, an autonomic element encapsulates rules, constraints, and mechanisms for self-management and can dynamically interact with other elements and the system. An autonomic element is illustrated in Fig. 5 and is defined by three ports.

1) The **functional port** (Γ) defines a set of functional behaviors γ provided and used by the element $\gamma \in \Omega \times \Lambda$, where Ω is the set of inputs and Λ is the set of outputs of the element, and γ defines a valid input–output set.
2) The **control port** $(\sum)$ is the set of tuples (σ, ξ), where σ is a set of sensors and actuators exported by the element, and ξ is the constraint set that controls access to the sensors/actuators. Sensors are interfaces that provide information about the element while actuators are interfaces for modifying the state of the element. Constraints are based on state, context and/or high-level access policies.
3) The **operational port** (Θ) defines the interfaces to formulate, dynamically inject, and manage rules that are used to manage the runtime behavior of the elements and the interactions between elements, between elements and their environments, and the coordination within an application.

The control and operational ports (specified in XML) enhance element interfaces to export information about their behaviors and adaptability to system and application dynamics. Each autonomic element also embeds an element manager that is delegated to manage its execution. The element manager monitors the state of the element and its context and controls the execution of rules. Note that element managers may cooperate with other element managers to fulfill application objectives.

Rules in Accord: Rules incorporate high-level guidance and practical human knowledge in the form of if–then expressions, i.e., IF *condition* THEN *actions*, similar to production rule, case-based reasoning, and expert systems. *Condition* is a logical combination of element (and environment) sensors, function interfaces, and events. *Actions* consist of a sequence of invocations of element and/or system sensors/actuators and other interfaces. A rule fires when its condition expression evaluates to be true and causes the corresponding actions to be executed. A priority-based mechanism is used to resolve conflicts [71]. Two classes of rules are defined: 1) *behavioral rules* that control the runtime functional behaviors of an autonomic element (e.g., the dynamic selection of algorithms, data representation, input/output format used by the element) and 2) *interaction rules* that control the interactions between elements, between elements and their environment, and the coordination within an autonomic application (e.g., communication mechanism, composition and coordination of the elements). Note that behaviors and interactions expressed by these rules are defined by the model of computation and the rules for composition of the underlying programming model.

Behavioral rules are executed by a rule agent embedded within a single element without affecting other elements. Interaction rules define interactions among elements. For each interaction pattern, a set of interaction rules are defined and dynamically injected into the interacting elements. The coordinated execution of these rules results in the desired interaction and coordination behaviors between the elements.

Autonomic Composition in Accord: Dynamic composition enables relationships between elements to be established and modified at runtime. Operationally, dynamic composition consists of a composition plan or workflow generation and execution. Plans may be created at runtime, possibly based on dynamically defined objectives, policies and applications, and system context and content. Plan execution involves discovering elements, configuring them, and defining interaction relationships and mechanisms. This may result in elements being added, replaced, or removed or the interaction relationships between elements being changed.

In Accord, composition plans may be generated using the Accord Composition Engine (ACE) [14] or using other approaches (e.g., [89]), and are expressed in XML. Element discovery uses the Meteor content-based middleware and specifically the Squid discovery service [92], [93]. Plan execution is achieved by a peer-to-peer control network of element managers and agents within Rudder [69]. A composition relationship between two elements is defined by the control structure (e.g., loop, branch) and/or the communication mechanism (e.g., RPC, shared space) used. A composition agent translates this into a suite of interaction

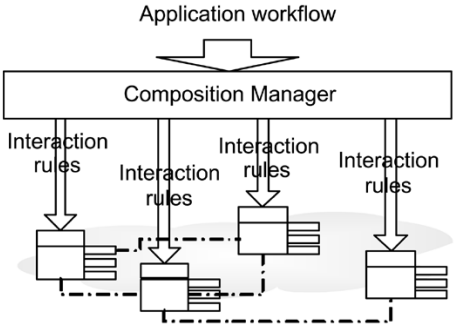

Fig. 6. Autonomic application execution.

rules, which are then injected into corresponding element managers, as illustrated in Fig. 6. Element managers execute the rules to establish control and communication relationships among these elements in a decentralized manner. Rules can be similarly used to add or delete elements. Note that the interaction rules must be based on the core primitives provided by the system. Accord defines a library of rule sets for common control and communications relationships between elements.

Accord Programming Language: As mentioned above, Accord extends existing programming systems and, thus, uses the language(s) defined by the system it extends. It does, however, define an XML-based language for specifying control and operational ports. These definitions are translated into the interface specification language used by the native system. It also defines an XML-based language for specifying rules.

The initial prototype of Accord extended an object-oriented framework based on C++ and MPI. The current implementation extends the Department of Energy (DOE) CCA [15], and we are exploring the extension of the XCAT [63] programming system.

Accord Implementation Issues: Accord decouples interaction and coordination from computation and enables both these behaviors to be managed at runtime using rules. This enables autonomic elements to change their behaviors and to dynamically establish/terminate/change interaction relationships with other elements. Deploying and executing rules does impact performance; however, it increases the robustness of the applications and their ability to manage dynamism. Further, our observations indicate that the runtime changes to interaction relationships are infrequent and their overheads are relatively small. As a result, the time spent to establish and modify interaction relationships is small as compared to typical computation times. A prototype implementation and evaluation of its performance overheads is presented in [71].

Accord Abstract Machine: The Accord abstract machine assumes the existence of common knowledge in the form of an ontology and taxonomy that defines the semantics for specifying and describing application namespaces, and element interfaces, sensors and actuators, and system/application context and content. This common semantics is used for formulating rules for autonomic management of elements and dynamic composition and interactions between the elements. Further, the abstract machine assumes time-asynchronous system behavior with fail-stop failure modes as described in [32]. Finally, Accord assumes the existence of an execution environment that provides: 1) an agent-based control network; 2) support for associative coordination; 3) service for content-based discovery and messaging; 4) support of context-based access control; and 5) services for constructing and managing virtual machines for a given virtual organization. These requirements are addressed respectively by Rudder, Meteor, Sesame/DAIS, and the underlying Grid middleware on which it builds.

Accord Application Infrastructure: As mentioned above, AutoMate provides infrastructure and services for supporting autonomic behaviors and the Accord programming system.[5] Key components of AutoMate include the following.

- *Rudder Decentralized Coordination Framework:* Rudder [69] is an agent-based decentralized coordination framework for enabling autonomic Grid applications. It provides the core capabilities for supporting autonomic compositions, adaptations, optimizations, and fault tolerance. Specifically, Rudder employs context-aware software agents and a decentralized tuple space coordination model to enable context and self-awareness, application monitoring and analysis, and rule definition and its distributed execution.
- *Meteor Content-Based Middleware:* Meteor [57], [58] is a scalable content-based middleware infrastructure that provides services for content routing, content discovery, and associative interactions. The Meteor stack includes: 1) a self-organizing content overlay; 2) the Squid [91] content-based routing engine and decentralized information discovery service supporting flexible routing and querying with guarantees and bounded costs; and 3) the Associative Rendezvous Messaging Substrate (ARMS) [57] supporting content-based decoupled interactions with programmable reactive behaviors.

Current Status: The core components of AutoMate have been prototyped and are being currently used to enable self-managing applications in science and engineering (e.g., autonomic oil reservoir optimizations [25], [78], autonomic runtime management of adaptive simulations [31], [62], etc.), and sensor-based pervasive applications [57].[6]

VI. Conclusion

The definitions of Grids and Grid programming systems reported in this paper derive from the established domain of distributed systems and are based on explicitly stated axioms and assumptions. These definitions establish a basis for characterizing and classifying Grid research, and enable the Grid community to utilize the knowledge base from distributed systems to establish requirements for and evaluate the state

[5]Note that Accord relies on the frameworks that it extends to provide services not addressed by AutoMate, such as life-cycle management, etc.

[6]Further information about AutoMate and its components can be obtained from http://automate.rutgers.edu/

of the art in Grid infrastructures, middleware and programming models, and systems. Evaluation of current Grid programming models and systems using this definition structure show that these efforts are understandably in an early state of development, and that applications based on these systems are likely to be fragile, since the programming systems are largely based on assumptions which are unlikely to be universally realized in Grid environments. Nonetheless, there are systems that address a substantial number of the issues identified as important in this paper. ICENI and AutoMate are two examples of such systems, which we have evaluated in detail. We believe that a sound basis for future development of Grid programming models and systems is emerging. However, we also believe that a focus on formalizing and modeling Grids and Grid programming models and systems is essential to this effort.

REFERENCES

[1] Common Object Broker Resource Architecture (CORBA) [Online]. Available: http://www.corba.org

[2] ebXML Requirements Specification, Version 1.06 (2001). [Online]. Available: http://www.ebxml.org/specs/ebREQ.pdf

[3] Global Grid Forum [Online]. Available: http://www.gridforum.org

[4] The Globus Alliance [Online]. Available: http://www.globus.org

[5] ICENI—Imperial College e-Science Networked Infrastructure (2004). [Online]. Available: http://www.lesc.ic.ac.uk/iceni/

[6] Jini Network Technology [Online]. Available: http://wwws.sun.com/software/jini/

[7] Message Passing Interface [Online]. Available: http://www.mcs.anl.gov/mpich/

[8] Microsoft.Net [Online]. Available: http://www.microsoft.com/net/

[9] OWL Web Ontology Language [Online]. Available: http://www.w3.org/TR/owl-semantics

[10] Project JXTA [Online]. Available: www.jxta.org

[11] Unicore Forum [Online]. Available: http://www.unicore.org

[12] Web Services Activity [Online]. Available: http://www.w3.org/2002/ws/

[13] M. Agarwal, V. Bhat, Z. Li, H. Liu, V. Matossian, V. Putty, C. Schmidt, G. Zhang, M. Parashar, B. Khargharia, and S. Hariri, "AutoMate: Enabling autonomic applications on the grid," in *Proc. Autonomic Computing Workshop, 5th Annu. Int. Workshop Active Middleware Services (AMS 2003)*, pp. 365–375.

[14] M. Agarwal and M. Parashar, "Enabling autonomic compositions in grid environments," in *Proc. 4th Int. Workshop Grid Computing (Grid 2003)*, pp. 34–41.

[15] B. A. Allan, R. C. Armstrong, A. P. Wolfe, J. Ray, D. E. Bernholdt, and J. A. Kohl, "The CCA core specifications in a distributed memory SPMD framework," in *Concurrency Comput. Pract. Exp.*, 2002, vol. 14, pp. 323–345.

[16] G. Allen, K. Davis, K. N. Dolkas, N. D. Doulamis, T. Goodale, T. Kielmann, A. Merzky, J. Nabrzyski, J. Pukacki, T. Radke, M. Russell, E. Seidel, J. Shalf, and I. Taylor, "Enabling applications on the Grid: A GridLab overview," *Int. J. High Perform. Comput. Appl. (Special Issue on Grid Computing: Infrastructure and Applications)*, vol. 17, no. 4, pp. 449–466, Winter 2003 , to be published.

[17] G. Aloisio and D. Talia, "Grid computing: Toward a new computing infrastructure," *Future Gener. Comput. Syst.*, vol. 18, no. 8, 2002.

[18] T. Andrews, F. Curbera, H. Dholakia, Y. Goland, J. Klein, F. Leymann, K. Liu, D. Roller, D. Smith, S. Thatte, I. Trickovic, and S. Weerawarana. (2003) Business Process Execution Language for Web Services Version 1.1. [Online]. Available: http://www-106.ibm.com/developerworks/library/ws-bpel/

[19] A. Arkin, S. Askary, S. Fordin, W. Jekeli, K. Kawaguchi, D. Orchard, S. Pogliani, K. Riemer, S. Struble, P. Takacsi-Nagy, I. Trickovic, and S. Zimek. (2002) Web Service Choreography Interface (WSCI) 1.0. [Online]. Available: http://www.w3.org/TR/wsci/

[20] H. Bal, H. Casanova, J. Dongarra, and S. Matsuoka, "Application level tools," in *The Grid 2: Blueprint for a New Computing Environment*, I. Foster and C. Kesselman, Eds. San Francisco, CA: Morgan Kaufmann, 2004, pp. 463–490.

[21] H. E. Bal, J. G. Steiner, and A. S. Tanenbaum, "Programming languages for distributed computing systems," *ACM Comput. Surv.*, vol. 21, no. 3, pp. 261–322, 1989.

[22] R. Batchu, A. Skjellum, Z. Cui, M. Beddhu, J. P. Neelamegam, Y. Dandass, and M. Apte, "MPI/FT: Architecture and taxonomies for fault-tolerant, message-passing middleware for performance-portable parallel computing," in *Proc. 1st Int. Symp. Cluster Computing and the Grid*, 2001, pp. 26–33.

[23] R. Ben-Natan, *CORBA: A Guide to Common Object Request Broker Architecture*. New York: McGraw-Hill, 1995.

[24] F. Berman, A. Chien, K. Cooper, J. Dongarra, I. Foster, D. Gannon, L. Johnsson, K. Kennedy, C. Kesselman, J. Mellor-Crummey, D. Reed, L. Torczon, and R. Wolski, "The GrADS project: Software support for high-level grid application development," *Int. J. High Perform. Comput. Appl.*, vol. 15, no. 4, pp. 327–344, 2001.

[25] V. Bhat, V. Matossian, M. Parashar, M. Peszynska, M. Sen, P. Stoffa, and M. F. Wheeler, "Autonomic oil reservoir optimization on the grid," *Concurrency Comput. Pract. Exp.*, to be published.

[26] V. Bhat and M. Parashar, "Discover middleware substrate for integrating services on the grid," in *Proc. 10th Int. Conf. High Performance Computing (HiPC 2003)*, 2003, pp. 373–382.

[27] A. D. Birrell and B. J. Nelson, "Implementing remote procedure calls," *ACM Trans. Comput. Syst.*, vol. 2, no. 1, pp. 39–59, 1984.

[28] A. Bouteiller, F. Cappello, T. Hérault, G. Krawezik, P. Lemarinier, and F. Magniette, "MPICH-V2: A fault tolerant MPI for volatile nodes based on pessimistic sender based message logging," presented at the ACM/IEEE SC2003 Conf., Phoenix, AZ.

[29] N. Carriero and D. Gelernter, "Linda in context," in *Commun. ACM*, vol. 32, 1989, pp. 444–458.

[30] E. Cerami, *Web Services Essentials*. Sebastopol, CA: O'Reilly, 2002.

[31] S. Chandra, M. Parashar, and S. Hariri, "GridARM: An autonomic runtime management framework for SAMR applications in grid environments, new frontiers in high-performance computing," in *Proc. Autonomic Applications Workshop, 10th Int. Conf. High Performance Computing (HiPC 2003)*, pp. 286–295.

[32] F. Cristian and C. Fetzer, "The timed asynchronous distributed system model," in *IEEE Trans. Parallel Distrib. Syst.*, Jun. 1999, vol. 10, pp. 642–657.

[33] K. Czajkowski, D. Ferguson, I. Foster, J. Frey, S. Graham, T. Maguire, D. Snelling, and S. Tuecke. (2004) From Open Grid Services Infrastructure to WS Resource Framework: Refactoring and evolution. [Online]. Available: http://www-106.ibm.com/developerworks/library/ws-resource/ogsi_to_wsrf_1.0.pdf

[34] K. Czajkowski, D. F. Ferguson, I. Foster, J. Frey, S. Graham, I. Sedukhin, D. Snelling, S. Tuecke, and W. Vambenepe. (2004) The WS-Resource Framework. [Online]. Available: http://www-106.ibm.com/developerworks/library/ws-resource/ws-wsrf.pdf

[35] R. Englander, *Developing Java Beans*. Sebastopol, CA: O'Reilly, 1997.

[36] D. H. J. Epema, M. Livny, R. V. Dantzig, X. Evers, and J. Pruyne, "A worldwide flock of condors: Load sharing among workstation clusters," *J. Future Gener. Comput. Syst.*, vol. 12, pp. 53–65, 1996.

[37] G. E. Fagg and J. Dongarra, "FT-MPI: Fault tolerant MPI, supporting dynamic applications in a dynamic world," in *Proc. 7th Eur. PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, 2000, pp. 346–353.

[38] J. Farley, *Java Distributed Computing*. Sebastopol, CA: O'Reilly, 1998.

[39] I. Foster and C. Kesselman, Eds., *The Grid 2: Blueprint for a New Computing Infrastructure*. San Francisco, CA: Morgan Kaufmann, 2004.

[40] I. Foster, C. Kesselman, J. M. Nick, and S. Tuecke. (2002) The physiology of the grid: An open grid services architecture for distributed systems integration. *Open Grid Service Infrastructure WG, Global Grid Forum* [Online]. Available: http://www.globus.org/research/papers/ogsa.pdf

[41] I. Foster, C. Kesselman, and S. Tuecke, "The anatomy of the grid: Enabling scalable virtual organizations," *Int. J. High Perform. Comput. Appl.*, vol. 15, no. 3, pp. 200–222, 2001.

[42] E. Freeman, S. Hupfer, and K. Arnold, *JavaSpaces Principles, Patterns, and Practice*. Reading, MA: Addison-Wesley, 1999.

[43] N. Furmento, J. Hau, W. Lee, S. Newhouse, and J. Darlington, "Implementations of a service-oriented architecture on top of Jini, JXTA, and OGSA," in *Proc. UK e-Science All Hands Meeting*, 2003, pp. 703–710.

[44] N. Furmento, W. Lee, A. Mayer, S. Newhouse, and J. Darlington, "ICENI: An open grid service architecture implemented with Jini," presented at the Supercomputing Conf. 2002, Baltimore, MD.

[45] N. Furmento, A. Mayer, S. McGough, S. Newhouse, T. Field, and J. Darlington, "ICENI: Optimization of component applications within a grid environment," *J. Parallel Comput.*, vol. 28, no. 12, pp. 1753–1772, 2002.

[46] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam, *PVM: Parallel Virtual Machine—A Users' Guide and Tutorial for Networked Parallel Computing*. Cambridge, MA: MIT Press, 1994.

[47] D. Gelernter, "Generative communication in Linda," *ACM Trans. Program. Lang. Syst. (TOPLAS)*, vol. 7, no. 1, pp. 80–112, 1985.

[48] M. Govindaraju, S. Krishnan, K. Chiu, A. Slominski, D. Gannon, and R. Bramley, *XCAT 2.0: A Component Based Programming Model for Grid Web Services*. Bloomington: Indiana Univ., 2002.

[49] F. Griffel, M. Boger, H. Weinreich, W. Lamersdorf, and M. Merz, "Electronic contracting with COSMOS—How to establish, negotiate and execute electronic contracts on the internet," in *2nd Int. Workshop Enterprise Distributed Object Computing (EDOC'98)*, San Diego, CA.

[50] A. S. Grimshaw and W. A. Wulf, "The legion vision of a worldwide virtual computer," in *Commun. ACM*, vol. 40, 1997, pp. 39–45.

[51] W. Gropp, E. Lusk, and R. Thakur, *Using MPI-2—Advanced Features of the Message Passing Interface*. Cambridge, MA: MIT Press, 1999.

[52] Y. Gurevich, "Sequential abstract state machines capture sequential semantics," *ACM Trans. Comput. Logic*, vol. 1, no. 1, pp. 77–111, 2000.

[53] J. Y. Halpern and Y. Moses, "Knowledge and common knowledge in a distributed environment," *J. ACM*, vol. 37, no. 3, pp. 549–587, 1990.

[54] J. Hau, W. Lee, and S. Newhouse, "Autonomic service adaptation using ontological annotation," presented at the 4th Int. Workshop Grid Computing, Grid 2003, Phoenix, AZ, 2003.

[55] ——, "The ICENI service adaptation framework," in *Proc. U.K. e-Science All Hands Meeting*, 2003, pp. 79–86.

[56] Y. Ishikawa, M. Matsuda, T. Kudoh, H. Tezuka, and S. Sekiguchi. The design of a latency-aware MPI communication library. *Proc. SWOPP03* [Online]. Available: http://www.gridmpi.org/publications

[57] N. Jiang, C. Schmidt, V. Matossian, and M. Parashar, "Content-based middleware for decoupled interactions in pervasive environments," Wireless Inf. Netw. Lab., Rutgers Univ. (WINLAB), Piscataway, NJ, Tech. Rep. WINLAB-TR-252, 2004.

[58] ——, "Project METEOR: A content-based middleware for decoupled interactions in pervasive environments," *CAIP Update*, vol. 15, no. 3, p. 2, 4, May 2004.

[59] N. Karonis, B. Toonen, and I. Foster, "MPICH-G2: A grid-enabled implementation of the message passing interface," *J. Parallel Distrib. Comput.*, vol. 63, pp. 551–563, 2003.

[60] W. Kelly, P. Roe, and J. Sumitomo, "G2: A grid middleware for cycle donation using .NET," in *Proc. 2002 Int. Conf. Parallel and Distributed Processing Techniques and Applications*, pp. 699–705.

[61] *Proc. 1st Int. Conf. Autonomic Computing*, J. Kephart, M. Parashar, V. Sunderam, and R. Das, Eds., 2004.

[62] B. Khargharia, S. Hariri, B. Kim, M. Zhang, P. Vadlamani, and M. Parashar, "vGrid: A framework for development and execution of autonomic grid applications, new frontiers in high-performance computing," in *Proc. Autonomic Applications Workshop, 10th Int. Conf. High Performance Computing (HiPC 2003)*, pp. 269–285.

[63] S. Krishnan and D. Gannon, "XCAT3: A framework for CCA components as OGSA services," in *Proc. HIPS 2004, 9th Int. Workshop High-Level Parallel Programming Models and Supportive Environments*, pp. 90–97.

[64] D. Laforenza, "Grid programming: Some indications where we are headed," *Parallel Comput.*, vol. 28, no. 12, pp. 1733–1752, 2002.

[65] G. V. Laszewski, I. Foster, and J. Gawor, "CoG kits: A bridge between commodity distributed computing and high-performance grids," in *Proc. ACM 2000 Conf. Java Grande*, pp. 97–106.

[66] C. Lee, S. Matsuoka, D. Talia, A. Sussman, M. Mueller, G. Allen, and J. Saltz. (2001) A grid programming primer. Adv. Program. Models Res. Group, Global Grid Forum. [Online]. Available: http://www.eece.unm.edu/~apm/docs/APM_Primer_0801.pdf

[67] C. Lee and D. Talia, "Grid programming models: Current tools, issues and directions," in *Grid Computing: Making the Global Infrastructure a Reality*, F. Berman, G. Fox, and T. Hey, Eds. New York: Wiley, 2003, pp. 555–578.

[68] F. Leymann. (2001) Web Services Flow Language (WSFL) 1.0. [Online]. Available: http://www-3.ibm.com/software/solutions/ webservices/pdf/\\WSFL.pdf

[69] Z. Li and M. Parashar, "Rudder: A rule-based multi-agent infrastructure for supporting autonomic grid applications," in *Proc. 1st IEEE Int. Conf. Autonomic Computing (ICAC-04)*, pp. 10–17.

[70] H. Liu, "A programming framework for autonomic grid applications," Dept. Elect. Comput. Eng., Rutgers Univ., Piscataway, NJ, 2004.

[71] H. Liu and M. Parashar, "DIOS++: A framework for rule-based autonomic management of distributed scientific applications," in *Lecture Notes in Computer Science, Euro-Par 2003 Parallel Processing*. Heidelberg, Germany: Springer-Verlag, 2003, vol. 2790, pp. 66–73.

[72] H. Liu, M. Parashar, and S. Hariri, "A component-based programming framework for autonomic applications," in *Proc. 1st IEEE Int. Conf. Autonomic Computing (ICAC-04)*, pp. 278–279.

[73] M. Lorch and D. Kafura, "Symphony: A java-based composition and manipulation framework for computational grids," in *Proc. 2nd IEEE/ACM Int. Symp. Cluster Computing and the Grid*, 2002, pp. 136–143.

[74] V. Mann, V. Matossian, R. Muralidhar, and M. Parashar, "DISCOVER: An environment for Web-based interaction and steering of high-performance scientific applications," *Concurrency Comput. Pract. Exp.*, vol. 13, no. 8–9, pp. 737–754, 2001.

[75] V. Mann and M. Parashar, "Engineering an interoperable computational collaboratory on the grid," *Concurrency Comput. Pract. Exp. (Special Issue on Grid Computing Environments)*, vol. 14, no. 13–15, pp. 1569–1593, 2002.

[76] M. Marazakis, D. Papadakis, and C. Nikolaou, "Aurora: An architecture for dynamic and adaptive work sessions in open environments," in *Proc. Int. Conf. Database and Expert System and Applications (DEXA'98)*, pp. 480–491.

[77] J. Mathe, K. Kuntner, S. Pota, and Z. Juhasz, "The use of Jini technology in distributed and grid multimedia systems," in *Proc. MIPRO 2003, Hypermedia and Grid Systems*, pp. 148–151.

[78] V. Matossian, M. Parashar, W. Bangerth, H. Klie, and M. F. Wheeler, "An autonomic reservoir framework for the stochastic optimization of well placement," *Cluster Comput. (Special Issue on Autonomic Computing)*, to be published.

[79] A. Mayer, S. McGough, M. Gulamali, L. Young, J. Stanton, S. Newhouse, and J. Darlington, "Meaning and behavior in grid oriented components," in *Lecture Notes in Computer Science, Grid Computing—GRID 2002*. Heidelberg, Germany: Springer-Verlag, 2002, vol. 2536, pp. 100–111.

[80] M. Migliardi and V. Sunderam, "The harness metacomputing framework," presented at the 9th SIAM Conf. Parallel Processing for Scientific Computing, San Antonio, TX, 1999.

[81] J. Miller, D. Palaniswami, A. Sheth, K. Kochut, and H. Singh, "WebWork: METEOR's Web-based workflow management system," *J. Intell. Inf. Syst.*, vol. 10, no. 2, pp. 185–215, 1998.

[82] R. Muralidhar and M. Parashar, "A distributed object infrastructure for interaction and steering," *Concurrency Comput. Pract. Exp.*, vol. 15, no. 10, pp. 957–977, 2003.

[83] Z. Nemeth and V. Sunderam, "Characterizing grids: Attributes, definitions, and formalisms," *J. Grid Comput.*, vol. 1, no. 1, pp. 9–23, 2003.

[84] R. V. V. Nieuwpoort, J. Maassen, G. Wrzesinska, R. Hofman, C. Jacobs, T. Kielmann, and H. E. Bal, "Ibis: A flexible and efficient java-based grid programming environment," *Concurrency Comput. Pract. Exp.*, to be published.

[85] R. V. V. Nieuwpoort, J. Maassen, G. Wrzesinska, T. Kielmann, and H. E. Bal, "Satin: Simple and efficient java-based grid programming," *J. Parallel Distrib. Comput. Pract.*, to be published.

[86] A. Omicini and F. Zambonelli, "Tuple centres for the coordination of internet agents," in *Proc. ACM Symp. Applied Computing*, 1999, pp. 183–190.

[87] M. Parashar, Z. Li, H. Liu, C. Schmidt, V. Matossian, and N. Jiang, "Enabling autonomic applications: Models and infrastructure," presented at the Eur. Commission–U.S. National Science Foundation Strategic Research Workshop Unconventional Programming Paradigms, Mont Saint-Michel, France, 2004.

[88] G. Piccinelli and L. Mokrushin, "Dynamic E-service composition in DySCo," in *Proc. 21st Int. Conf. Distributed Computing Systems Workshops (ICDCSW'01)*, pp. 88–96.

[89] S. R. Ponnekanti and A. Fox, "SWORD: A developer toolkit for Web service composition," presented at the 11th World Wide Web Conf. (Web Engineering Track), Honolulu, HI, 2002.
[90] A. Ricci, A. Omicini, and E. Denti, "The TuCSoN coordination infrastructure for virtual enterprises," in *Proc. IEEE 10th Int. Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (WET ICE 2001), 3rd Int. Workshop Web-Based Infrastructures and Coordination Architectures for Collaborative Enterprises*, pp. 348–353.
[91] C. Schmidt and M. Parashar, "Enabling flexible queries with guarantees in P2P systems," *IEEE Internet Comput.*, vol. 8, no. 3, pp. 19–26, May/Jun. 2004.
[92] ——, "Flexible information discovery in decentralized distributed systems," in *Proc. 12th Int. Symp. High Performance Distributed Computing*, 2003, pp. 226–235.
[93] C. Schmidt and M. Parashar, "A peer-to-peer approach to Web service discovery," *World Wide Web*, vol. 7, no. 2, pp. 211–229, Jun. 2004.
[94] K. Seymour, H. Nakada, S. Matsuoka, J. Dongarra, C. Lee, and H. Casanova, "Overview of GridRPC: A remote procedure call API for grid computing," in *Lecture Notes in Computer Science, Grid Computing—GRID 2002*. Heidelberg, Germany: Springer-Verlag, 2002, vol. 2536, pp. 274–278.
[95] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra, *MPI: The Complete Reference*. Cambridge, MA: MIT Press, 1998, vol. 1, The MPI Core.
[96] C. Szyperski, D. Gruntz, and S. Murer, *Component Software Beyond Object-Oriented Programming*. New York: Addison-Wesley/ACM, 2002.
[97] N. Taesombut and A. Chien, "Distributed virtual computer (DVC): Simplifying the development of high performance grid applications," in *Proc. Workshop Grids and Advanced Networks (GAN'04), IEEE Cluster Computing and the Grid (CCGrid2004) Conf.*, 2004, pp. 715–722.
[98] I. Taylor, M. Shields, I. Wang, and R. Philp, "Distributed P2P computing within triana: A galaxy visualization test case," presented at the Int. Parallel and Distributed Processing Symp. (IPDPS'03), Nice, France.
[99] D. Thain, T. Tannenbaum, and M. Livny, "Condor and the grid," in *Grid Computing: Making the Global Infrastructure a Reality*, F. Berman, G. Fox, and T. Hey, Eds. New York: Wiley, 2002.
[100] S. Thatte. (2001) XLANG: Web services for business process design. [Online]. Available: http://www.gotdotnet.com/team/xml_wsspecs/xlang-c
[101] S. Tuecke, K. Czajkowski, I. Foster, J. Frey, S. Graham, C. Kesselman, D. Snelling, and P. Vanderbilt. (2003) Open Grid Services Infrastructure (OGSI), Global Grid Forum. [Online]. Available: http://www.ggf.org/ogsi-wg
[102] J. Turek and D. Shasha, "The many faces of consensus in distributed systems," *IEEE Computer*, vol. 25, no. 6, pp. 8–17, Jun. 1992.
[103] C. Ururahy and N. Rodriguez, "Programming and coordinating grid environments and applications," *Concurrency Comput. Pract. Exp.*, vol. 16, no. 5, pp. 543–549, 2004.
[104] P. Wagstrom, S. Krishnan, and G. V. Laszewski, "GSFL: A workflow framework for grid services," in *Proc. SC'2002*, pp. 11–16.
[105] J. Waldo and K. Arnold, *The Jini Specifications*. Upper Saddle River, NJ: Pearson, 2000.
[106] N. Wang, D. C. Schmidt, and C. O'Ryan, "An overview of the CORBA component model," in *Component-Based Software Engineering: Putting the Pieces Together*, G. T. Heineman and W. T. Councill, Eds. Reading, MA: Addison-Wesley, 2001.
[107] G. Zhang, "Cooperative mechanisms for protection against distributed network attacks," Dept. Elect. Comput. Eng., Rutgers Univ., Piscataway, NJ, 2004.
[108] G. Zhang and M. Parashar, "Cooperative mechanism against DDoS attacks," presented at the IEEE Int. Conf. Information and Computer Science (ICICS 2004), Dhahran, Saudi Arabia.
[109] ——, "Dynamic context-aware access control for grid applications," in *Proc. 4th Int. Workshop Grid Computing (Grid 2003)*, pp. 101–108.

Manish Parashar (Senior Member, IEEE) received the B.E. degree in electronics and telecommunications from Bombay University, Bombay, India, in 1988, and the M.S. and Ph.D. degrees in computer engineering from Syracuse University Syracuse, NY, in 1994.

He is Associate Professor of Electrical and Computer Engineering at Rutgers University, Piscataway, NJ, where he also is director of the Applied Software Systems Laboratory. He has coauthored over 130 technical papers in international journals and conferences, has coauthored/edited five books/proceedings, and has contributed to several others in the area of parallel and distributed computing. His current research interests include autonomic computing, parallel, distributed and grid computing, networking, scientific computing, and software engineering.

Dr. Parashar is a Member of the Association for Computing Machinery. He has received the National Science Foundation (NSF) CAREER Award (1999) and the Enrico Fermi Scholarship from Argonne National Laboratory (1996). He is a Member of the Executive Committee of the IEEE Computer Society Technical Committee on Parallel Processing (TCPP), part of the IEEE Computer Society Distinguished Visitor Program (2004–2006). He is also the Cofounder of the IEEE International Conference on Autonomic Computing (ICAC).

James C. Browne received the Ph.D. degree in chemical physics at the University of Texas, Austin, in 1960.

He taught in the Physics Department at the University of Texas from 1960 to 1964. He was, from 1965 through 1968, Professor of Computer Science at Queens University, Belfast, Ireland and directed the Computer Laboratory. He joined the University of Texas in 1968 as Professor of Physics and Computer Science. He served as Department Chair for Computer Science in 1968–1969, 1971–1975, and 1984–1987. He is currently Professor of Computer Science and Physics and holds the Regents Chair #2 in Computer Sciences at the University of Texas. He has published approximately 100 papers in computational physics and 250 papers in computer science. His current research interests span parallel programming and computation with a focus on applications in science and engineering, distributed and grid computing methods, performance measurement and analysis, software engineering and formal methods including model checking of software systems.

Prof. Browne is a Fellow of the Association for Computing Machinery (ACM), of the British Computer Society, the American Physical Society and of the American Association for the Advancement of Science. He was Chairman of the ACM Special Interest Group on Operating Systems from 1973 to 1975.