Cluster Comput (2006) 9: 161–174
DOI 10.1007/s10586-006-7561-5

AutoMate: enabling autonomic applications on the Grid

M. Parashar · H. Liu · Z. Li · V. Matossian · C. Schmidt · G. Zhang · S. Hariri

Received: October 2003 / Revised: May 2004 / Accepted: January 2005

Abstract The increasing complexity, heterogeneity, and dynamism of emerging pervasive Grid environments and applications has necessitated the development of autonomic self-managing solutions, that are inspired by biological systems and deal with similar challenges of complexity, heterogeneity, and uncertainty. This paper introduces Project AutoMate and describes its key components. 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. Illustrative autonomic scientific and engineering Grid applications enabled by AutoMate are presented.

Keywords Autonomic computing · Autonomic Grid applications · Programming frameworks · Middleware services

The research presented in this paper is supported in part by the National Science Foundation via grants numbers ACI 9984357, EIA 0103674, EIA 0120934, ANI 0335244, CNS 0305495, CNS 0426354 and IIS 0430826. The authors would like to acknowledge the contributions of M. Agarwal, V. Bhat and N. Jiang to this research.

M. Parashar · H. Liu · Z. Li · V. Matossian · C. Schmidt (✉) · G. Zhang
The Applied Software Systems Laboratory, Department of Electrical and Computer Engineering, Rutgers, The State University of New Jersey
e-mail: automate@caip.rutgers.edu

S.Hariri
High Performance Distributed Computing Laboratory, Department of Electrical and Computer Engineering, University of Arizona
e-mail: hariri@ece.arizona.edu

1. Introduction

The emergence of pervasive wide-area distributed computing, such as pervasive information systems and the computational Grid, has enabled a new generation of applications that are based on seamless aggregations and interactions. For example, it is possible to conceive of a new generation of scientific and engineering simulations of complex physical phenomena that symbiotically and opportunistically combine computations, experiments, observations, and real-time data, and can provide important insights into complex systems such as interacting black holes and neutron stars, formations of galaxies, and subsurface flows in oil reservoirs and aquifers, etc. Other examples include 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.

However, the underlying Grid computing environment is inherently large, complex, heterogeneous and dynamic, globally aggregating large numbers of independent computing and communication resources, data stores and sensor networks. Furthermore, emerging applications are similarly complex and highly dynamic in their behaviors and interactions. Together, these characteristics result in application development, configuration and management complexities that break current paradigms based on passive components and static compositions. Clearly, there is a need for a fundamental change in how these applications are developed and managed. This has led researchers to consider alternative programming paradigms and management techniques

that are based on strategies used by biological systems to deal with complexity, dynamism, heterogeneity and uncertainty. The approach, referred to as autonomic computing, aims at realizing computing systems and applications capable of managing themselves with minimal human intervention. An autonomic system/application has the capabilities of being contexually aware, self-defining, self-healing, self-configuring, self-optimizing and self-protecting.

Project AutoMate aims at building solutions to investigate key technologies, including programming models, frameworks, and middleware services, to enable the development of autonomic Grid applications that can address the challenges of complexity, dynamism, heterogeneity and uncertainty in Grid environments. Its overall goal is to develop conceptual models and implementation architectures that can enable the development and execution of such self-managing Grid applications. Specific issues addressed by Project AutoMate include:

Definition of Autonomic Elements: The definition of programming abstractions and supporting infrastructure that will enable the definition of autonomic elements (components/services). In addition to the interfaces exported by traditional elements, autonomic elements provide enhanced profiles (and contracts) that encapsulate their functional, operational, and control aspects. These aspects enhance the interfaces to export information and policies about their behavior, resource requirements, performance, interactivity and adaptability to system and application dynamics. Furthermore, they encapsulate sensors, actuators, access policies and a policy-engine. Together, aspects, policies, and policy engine allow autonomic elements to consistently configure, manage, adapt and optimize their execution.

Dynamic Composition of Autonomic Applications: The development of mechanisms and supporting infrastructure to enable autonomic applications to be dynamically and opportunistically composed from autonomic elments. The composition will be based on policies and constraints that are defined, deployed, and executed at run time, and will be aware of available resources (systems, services, storage, data) and elements, and their current states, requirements, and capabilities.

Autonomic Middleware Services: The design, development, and deployment of key services on top of the Grid middleware infrastructure to support the policy, content and context driven execution and management of autonomic applications. These include decentralized content-based services for coordination, messaging, discovery, peer-to-peer multi-agent substrates and deductive engines, service for security, access management and self-protection, etc.

In this paper we introduce AutoMate, and describe its underlying conceptual models and implementations. Specifically, we describe the Accord programming system, the Rudder decentralized coordination framework and agent-based deductive engine, the Meteor content-based middleware that provides support for content-based routing, discovery and associative messaging, and the SESAME access management system. We also illustrate the use of AutoMate for enabling autonomic scientific and engineering Grid applications

The rest of this paper is organized as follows. Section 2 outlines the programming requirements of Grid applications. Section 3 gives an overview of the design and architecture of AutoMate. Section 4 describes the Accord programming system and the autonomic composition engine. Section 5 describes the structure and operation of the Rudder coordination engine. Section 6 presents Squid, a P2P system providing flexible information discovery and content-based routing services. Section 7 describes the Meteor content-based middleware. Section 8 describes the PAWN messaging substrate. Section 9 presents the design and operation of the Sesame context aware access control engine. Section 10 briefly discusses the use of AutoMate in enabling autonomic science and engineering applications. Section 11 presents some concluding remarks.

2. Grid computing – challenges and requirements

The goal of the Grid concept is to enable a new generation of applications that combine intellectual and physical resources and 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. Key characteristics of Grid execution environments and applications are:

Heterogeneity: Grid environments aggregate large numbers of independent and geographically distributed computational and information resources, including supercomputers, workstation-clusters, network devices, data-storages, sensors, services, and idle personal computers on the Internet. 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 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.

2.1. Requirements for Grid programming systems

The characteristics listed above 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 state of execution environment and the state and requirements of the application. This requirement suggests that: (1) Grid applications should be composed from discrete, self-managing components that incorporate separate specifications for all functional, non-functional and interaction-coordination behaviors. (2) The specifications of computational (functional) behaviors, interaction and coordination behaviors and non-functional behaviors (e.g. performance, fault detection and recovery, etc.) should be separated so that their combinations are composable. (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 non-functional behaviors such as fault recovery and QoS to address the dynamism and heterogeneity of the application and the environments.

2.2. Self-managing applications on the Grid

As outlined above, 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 programming models and the abstract machine underlying these models make strong assumptions about common knowledge, static behaviors and system guarantees that cannot be realized by Grid virtual machines and which are not true for Grid applications. Addressing these challenges requires redefining the programming framework to address the separations outlined above. 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 the rest of this paper we describe Project AutoMate, which addresses these challenges by enabling autonomic self-managing Grid applications.

3. Project AutoMate: enabling self-managing Grid applications

Project AutoMate [1] investigates autonomic solutions that are inspired by biological systems and deals with similar challenges of complexity, dynamism, heterogeneity and uncertainty. The goal is to realize systems and applications that are capable of managing (i.e., configuring, adapting, optimizing, protecting, healing) themselves. Project AutoMate aims at developing 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 (1) the definition of autonomic elements, (2) the development of autonomic applications as the dynamic and opportunistic composition of these autonomic elements, and (3) the policy, content and context driven definition, execution and management of these applications. Project AutoMate builds on three fundamental concepts:

- Separation of policy from mechanism to distill out the aspects [11, 21] of components and enable them to orchestrate a repertoire of mechanisms for responding to the heterogeneity and dynamics, both of the applications and the Grid infrastructure. The policies that drive these mechanisms are specified separately. Examples of mechanisms are alternative numerical algorithms, domain decompositions, and communication protocols; an example of a policy is to select a latency-tolerant algorithm when network load is above certain thresholds.
- Context, constraint, and aspect based composition techniques applied to applications and middleware as an alternative to the current processes for translating the application's dynamic requirements for functionality, performance, quality of service, into sets of components and Grid resource requirements.
- Dynamic, proactive, and reactive component management to optimize resource utilization and application performance in situations where computational characteristics and/or resource characteristics may change.

Fig. 1 A schematic overview of AutoMate.

A schematic overview of AutoMate is presented in Fig. 1. It is composed of the following components:

Accord Programming Layer: The Accord programming layer [14] extends existing distributed programming models and frameworks to addresses the definition, execution and runtime management of autonomic elements, as well as the formulation of autonomic applications as the dynamic and opportunistic composition of these elements.

Rudder Coordination Layer: The Rudder coordination layer [12] provides a coordination framework and agent-based deductive engine to support autonomic behaviors. The coordination framework defines protocols and mechanisms and builds on a scalable implementation of a decentralized tuple space. The deductive engine is composed of element, system and compostion agents and supports the collective decision making.

Meteor/Pawn Middleware Layer: The Meteor [7] middleware substrate provides a content-based middleware with support for content-based routing, discovery and associative messaging. It includes the Squid routing and discovery engine and the ARMS messaging substrate. The Pawn [19] peer-to-peer messaging layer provides higher level messaging abstractions in a decentralized environment.

Sesame Access Management Layer: The Sesame [30] access control engine is composed of access control agents and provides dynamic context-aware control.

Project AutoMate also includes the DAIS [29] cooperative-protection services and the Discover collaboratory [4, 15] services for collaborative monitoring, interaction and control, which are not described in this paper. Additionally, AutoMate portals provide users with secure, pervasive (and collaborative) access to the different entities. Using these portals users can access resources, monitor, interact with, and steer components, compose and deploy applications, configure and deploy rules. The key components are described in the following sections.

4. Accord, a programming framework for autonomic applications

The Accord programming system [14] 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 non-functional 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. The components of Accord are described below.

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. 2 and is defined by 3 ports:

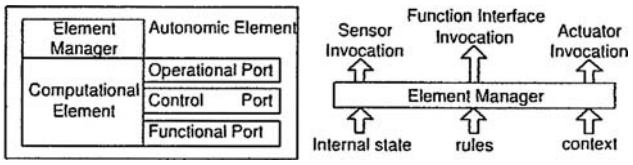

Fig. 2 An autonomic component

The **functional port** defines a set of functional behaviors that are provided and used by the element.

The **control port** is a set of tuples, that is composed of a set of sensors and actuators exported by the element, and a 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 polices.

The **operational port** defines the interface to formulate, dynamically inject and manage rules that are used for 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 is associated with an element manager (possibly embedded) 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 [13]. 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). (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 an element manager 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.

Dynamic 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) [2] (described in the following section) or using other approaches, and are expressed in XML. Element discovery uses the Meteor content-based middleware. Plan execution is achieved by a peer-to-peer control network of element managers and agents within Rudder [12]. 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 rules, which are then injected into corresponding element managers. 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. The decomposition procedure will guarantee that the local behaviors of individual elements will coordinate to achieve the application's objectives. Runtime negotiation protocols provided by Accord address runtime conflicts and conflicting decisions caused by a dynamic and uncertain environment.

Accord Implementation Issues: 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. 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 (4) 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 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. A prototype implementation and evaluation of Accord's performance is presented in [14] and shows that 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.

4.1. AutoMate autonomic composition engine

Applications are typically composed with well defined objectives. In the case of autonomic applications, however, these objectives can dynamically change based on the state of the application and/or the system. As a result, we need to dynamically select elments and compose them at runtime based on current objectives. Together, the profiles, policies, and rules allow autonomic components to consistently and securely manage and optimize their executions. Furthermore, they enable applications to be dynamically composed, configured and adapted. Dynamic application work-flows [16] can be defined to select the most appropriate elements based on user/application constraints (highest-performance, lowest cost, reservation, execution time upper bound, best accuracy), on the current applications requirements, to dynamically configure the element's algorithms and behavior based on available resources or system and/or applications state, and to adapt this behavior if necessary. Enabling dynamic composition presents significant challenges: (1) How to specify the changes in the objective to create dynamic compositions? Static composition can be described using existing languages, e.g. Petri nets or workflow definition language (WFDL), however changes in the composition requires a more flexible approach. (2) How to guarantee consistency of environment after submitting "change in plan"? Dynamic interactions and compositions can corrupt the element/application and introduce serious errors (e.g. deadlock or no termination). The AutoMate dynamic composition model may be viewed as transforming a given composition or workflow into a new one by adding or modifying interactions and participating entities. Its primary goal is to enable dynamic (and opportunistic) choreography and interactions of elements to react to the heterogeneity and dynamics of the application and underlying execution environment to produce the desired user objectives.

The AutoMate dynamic composition model is context aware and is based on policies and constraints that are defined, deployed and executed at runtime. Composition policies and constraints are defined as simple rules and execute on the distributed deductive engine (Section 5) – i.e. there is no central authority that manages the composition process. These rules are defined in terms of the ports [11] exported by Accord elements, the current context of the scenario and the overall objective of the application. Rules are simple and non-recursive, and can be composed and aggregated in a consistent way – based on logic and constraint based programming techniques [17]. Users can define and deploy rules at runtime provided they have the required privileges, and the rules inherit the priorities and privileges of their owners [13]. Rules execute in a distributed fashion on a peer-to-peer deductive shell exported by the autonomic middleware as described below. Firing of rules causes the elements to adapt, optimize, interact and compose. Composition metadata [16] is defined locally at the component level or globally at the application or the middleware level using a standard representation.

5. Rudder, an agent-based coordination middleware

Rudder [12] is an agent-based middleware infrastructure for autonomic Grid applications. Rudder effectively supports the Accord programming framework and enables autonomic self-managing applications. The overall objective of Rudder is to provide the core capabilities for supporting autonomic compositions, adaptations, and optimizations. 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 policy definition and its distributed execution. The overall architecture builds on two concepts:

Agent framework: Context-aware agents manage context information at different system and application levels to trigger autonomic behaviors. A context-aware agent is a processing unit that performs tasks to automate the control and coordination of the autonomic elements. The context consists of the information such as device profiles (e.g., CPU, memory, physical location, domain), network resources (e.g., bandwidth, latency, and disconnection rate), and software components (e.g., reliability, processing capability). These agents do not have the complex symbolic

reasoning capability, while they act and interact using predefined preferences and policies to select a plan that optimizes an appropriate measurement. Agents can control, compose and manage autonomic components, monitor and analyze system runtime state, sense changes in environment and application requirements, and dynamically define and enforce rules to locally enable component self-managing behaviors. The Rudder agent framework consists of three types of peer agents: Component Agent (CA), System Agent (SA), and Composition Agent (CSA). CA and SA exist as system services, while composition agents are transient and are generated to satisfy specific application requirements. CAs manage the computations performed locally within components and SAs are embedded within Grid resource units, exist at different levels of the system, and represent their collective behaviors. CSAs enable dynamic composition of autonomic components by defining and executing workflow-selection and component-selection rules.

Decentralized tuple space: A robust decentralized reactive tuple space can scalably and reliably support distributed agent-based system coordination. It provides the core semantic resource discovery and event notification services to enable the dynamic system deployment, composition, monitoring and management. The Rudder decentralized reactive tuple space extends the traditional tuple space with flexible matching mechanisms and simple reactivity to enable global coordination for dynamic and ad hoc agent communities. Runtime adaptive polices or constrains defined by the context-aware agents or administrators can be inserted and executed using reactive tuples to achieve coordinated application execution and optimized computational resource allocation and utilization.

The Rudder tuple space adopts a fully decentralized architecture consisting of the following layers implemented at each peer in the system: (1) a tuple space layer that implements a persistent tuple repository and extends the strict definition of tuples by exploiting XML strings with flexible matching mechanisms and coordination interfaces; (2) a content-based routing layer, which efficiently maps the tuples to peers nodes; and (3) a resilient self-organizing peer-to-peer content-based overlay. Programming reactive behaviors enable the definition and execution of coordination policies. Transient tuple spaces, which are context and content specific, can be dynamically created and destroyed to support local coordination needs. The Rudder decentralized tuple space builds on the Meteor infrastructure.

6. Squid: decentralized discovery service

A fundamental problem in large, decentralized, distributed resource sharing environments such as the Grid, is the efficient discovery of information, in the absence of global knowledge of naming conventions. For example a document is better described by keywords than by its filename, a computer by a set of attributes such as CPU type, memory, operating system type than by its host name, and a component by its aspects than by its instance name. The heterogeneous nature and large volume of data and resources, their dynamism (e.g. CPU load) and the dynamism of the Grid make the information discovery a challenging problem. An ideal information discovery system has to be efficient, fault-tolerant, self-organizing, has to offer guarantees and support flexible searches (using keywords, wildcards, range queries). Decentralized peer-to-peer (P2P) systems, by their inherent properties (self-organization, fault-tolerance, scalability), provide an attractive solution.

Squid [25] supports decentralized information discovery in AutoMate. It is a P2P system that supports complex queries containing partial keywords, wildcards, and range queries, and guarantees that all existing data elements that match a query will be found with bounded costs in terms of number of messages and number of nodes involved. The key innovation is a dimension reducing indexing scheme that effectively maps the multidimensional information space to physical peers.

The overall architecture of Squid is a distributed hash table (DHT), similar to typical data lookup systems [22, 28]. The key difference is in the way we map data elements[1] to the index space. In existing systems, this is done using consistent hashing to uniformly map data element identifiers to indices. As a result, data elements are randomly distributed across peers without any notion of locality. Our approach attempts to preserve locality while mapping the data elements to the index space. In our system, all data elements are described using a sequence of keywords (common words in the case of P2P storage systems, or values of globally defined attributes - such as memory and CPU frequency - for resource discovery in computational grids). These keywords form a multidimensional keyword space where the keywords are the coordinates and the data elements are points in the space. Two data elements are "local" if their keywords are lexicographically close or they have common keywords. Thus, we map documents that are local in this multi-dimensional index space to indices that are local in the 1-dimensional index space, which are then mapped to the same node or to nodes that are close together in the overlay network. This mapping is derived from a locality-preserving mapping called Space Filling Curves (SFC) [23]. In the current implementation, we use the Hilbert SFC [23] for the mapping, and Chord [28] for the overlay network topology.

[1] The term 'data element' is used to represent a piece of information that is indexed and can be discovered. A data element can be a document, a file, an XML file describing a resource, an URI associated with a resource, etc.

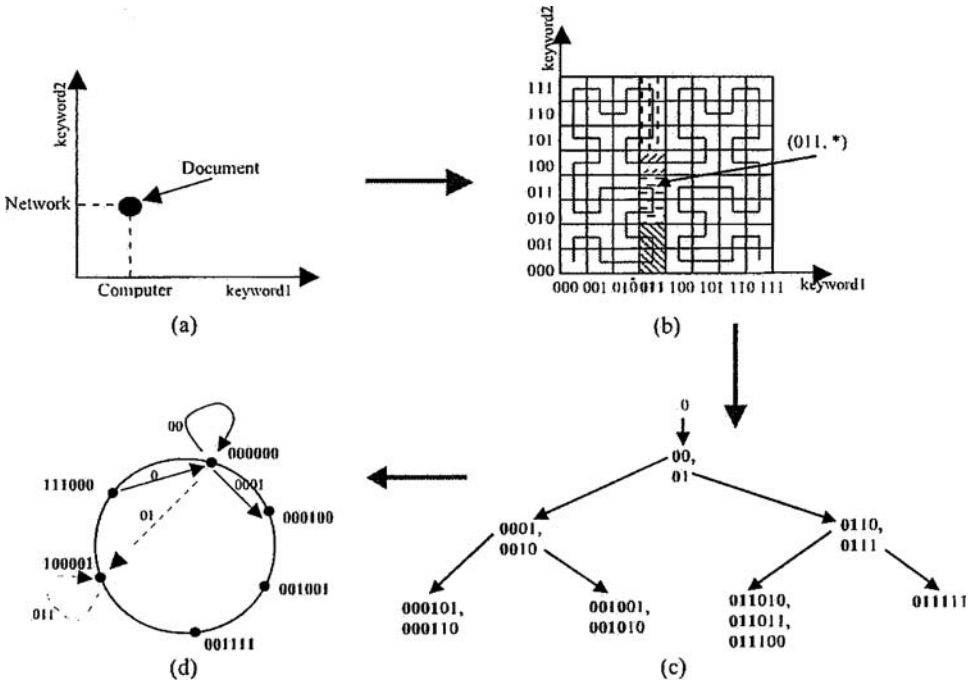

Fig. 3 (a) A 2-dimensional keyword space. The data element "*Document*" is described by keywords "*Computer*" and "*Network*"; (b) Mapping the 2-dimensional space to a curve. The query (011, *) defines clusters on the curve (segments); (c) Recursive refinement of query (011, *) viewed as a tree. Each node is a cluster, and the bold characters are the cluster's prefixes; (d) Solving the query: embedding the leftmost tree path (solid arrows) and the rightmost path (dashed arrows) onto the overlay network topology.

Note that locality is not preserved in an absolute sense in this keyword space; documents that match the same query (i.e. share a keyword) can be mapped to disjoint fragments of the index space, called clusters. These clusters may in turn be mapped to multiple nodes so a query will have to be efficiently routed to these nodes. Squid optimizes the querying process using successive refinement and pruning of the queries. These optimizations significantly reduce the number of clusters that need to be generated for a query, and as a consequence, the number of messages sent. The overall operation of Squid is presented in Fig. 3.

Unlike the consistent hashing mechanisms, SFC does not necessarily result in uniform distribution of data elements in the index space - certain keywords may be more popular and hence the associated index subspace will be more densely populated. As a result, when the index space is mapped to nodes load may not be balanced. Squid provides a suite of relatively inexpensive load-balancing optimizations and experimentally demonstrate that they successfully reduce the amount of load imbalance.

7. Meteor: content-based middleware for decoupled interactions in pervasive environments

Meteor is a content-based middleware infrastructure for decoupled interactions in pervasive Grid environments based on the Associative Redendezvous model. A schematic overview of the Meteor stack is presented in Fig. 4. It consists of 3 key components: (1) a self-organizing overlay, (2) a content-based routing infrastructure (Squid), and (3) the Associative

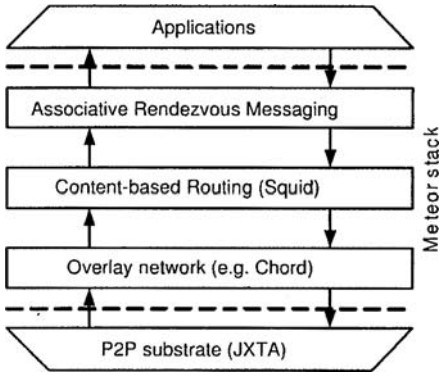

Fig. 4 A schematic overview of the Meteor stack.

Rendezvous Messaging substrate (ARMS). Squid was discussed in Section 6. The other two components are described below.

The Overlay Network Layer

The Meteor overlay network is composed of Rendezvous Peers (RP) nodes, which may be access points or message forwarding nodes in ad-hoc sensor networks and servers or end-user computers in wired networks. RP nodes can join or leave the network at any time.

The current Meteor overlay network is largely built over Chord [28]. Peer nodes in the Chord overlay form a ring topology. Every node in the Chord overlay is assigned a unique identifier ranging from 0 to $2^m - 1$. While in the original Chord implementation this identifier is obtained using consistent hashing [9], in Meteor it is obtained using Squid. The identifiers are arranged as a circle modulo 2^m. Each node maintains information about (at most) m neighbors, called

fingers, in a *finger table*. The finger table is used for efficient routing and enables data lookup with O(log N) cost [28], where N is the number of nodes in the system. The finger table is constructed when a node joins the overlay, and it is updated any time a node joins or leaves the system. The cost of a node join/leave is $O(\log^2 N)$.

Advantages of Chord include its guaranteed performance, logarithmic in number of messages, and its ease of implementation. Drawbacks include the cost of node join and leave operations (i.e. key reallocation) and the fact that constant periodic messages are required to maintain the ring (i.e. update propagation).

The overlay network layer of the Meteor stack provides a simple abstraction to the layers above, consisting of a single operation: **lookup**(identifier). Given an identifier, this operation locates the node that is responsible for it, i.e., the node with an identifier that is the closest identifier greater than or equal to the queried identifier. Application names can be mapped to identifiers using hashing mechanisms, and then mapped to nodes in the overlay network.

Associative Rendezvous Messaging Substrate

The matching engine component is essentially responsible for matching profiles. An incoming message profile is matched against existing interest and/or data profiles depending on the desired reactive behavior. If the result of the match is positive, then the action field of the incoming message is executed first and then the action field of the matched profile is evaluated.

The ARMS layer implements the Associative Rendezvous interaction model. At each RP, ARMS consists of two components: the *profile manager* and the *matching engine*. The profile manager manages locally stored profiles. Profiles are implemented as XML files. The managers monitor message credentials and contexts and ensures that related constraints are satisfied. For example, a client cannot retrieve data that it is not authorized to. The profile manager is also responsible for garbage collection. It maintains a local timer and purges interest and data profiles when their TTL fields have expired. Finally, the profile manager executes the action corresponding to a positive match.

8. Pawn: a P2P messaging substrate

Pawn[19] is a peer-to-peer messaging substrate that builds on project JXTA[8] to support peer-to-peer interactions on the Grid. Pawn provides a stateful and guaranteed messaging to enable key application-level interactions such as synchronous/asynchronous communication, dynamic data injection, and remote procedure calls. It exports these interaction modalities through services at every step of the scientific investigation process, from application deployment, to interactive monitoring and steering, and group collaboration.

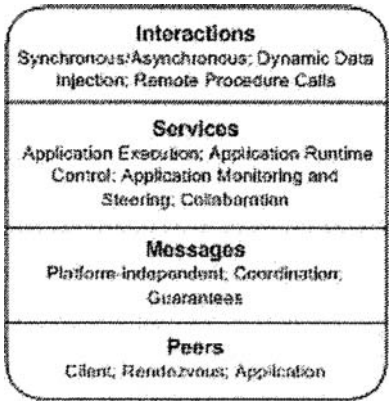

Fig. 5 Pawn requirements stack

A conceptual overview of the Pawn P2P substrate is presented in Fig. 5 and is composed of peers (computing, storage, or user peers), network and interaction services, and mechanisms. These components are layered to represent the requirements stack enabling interactions in a Grid environment. The figure can be read from bottom to top as: "Peers compose messages handled by services through specific interaction modalities".

JXTA defines unicast pipes that provide a communication channel between two endpoints, and propagate pipes that can multicast a message to a peergroup. It also defines the Resolver Service that sends and receives messages in an asynchronous manner. The recipient of the message can be a specific peer or an entire peergroup. The pipe and resolver service use one or any of the available underlying transport protocols (TCP, HTTP, TLS, etc...) to transport messages from point to point. Pawn extends the pipe and resolver services to provide stateful and guaranteed messaging. This messaging is then used to enable the key application-level interactions such as synchronous/asynchronous communication, dynamic data injection, and remote procedure calls.

Stateful Messages: In Pawn, messages are platform-independent, and are composed of source and destination identifiers, a message type, a message identifier, a payload, and a handler tag. The handler tag uniquely identifies the service that will process the message. State is maintained by making every message a self-sufficient and self-describing entity that, in case of a link failure, can be resent to its destination by an intermediary peer without the need to be recomposed by its original sender. In addition, messages can include system and application parameters in the payload to maintain application state.

Message Guarantees: Pawn implements application-level communication guarantees by combining stateful messages and a per-message acknowledgment table maintained at every peer. Message queues are used to handle all incoming and outgoing messages. Every outgoing message that expects a response is flagged in the table as awaiting acknowledgment.

This flag is removed once the message is acknowledged. Messages contain a default timeout value representing an upper limit on the estimated response time. If an acknowledgment is not received and/or the timeout value expires, the message is resent. The message identifier is a composition of the destination and sender's unique peer identifiers. It is incremented for every transaction during a session (interval between a peer joining and leaving a peergroup) to provide application-level message ordering guarantees.

Synchronous/Asynchronous communication: Pawn combines JXTA communication semantics, synchronous (using blocking pipes) or asynchronous (using non-blocking pipes) interactions, with its stateful messaging and message guarantees mechanisms to provide reliable messaging enabling the desired higher-level application interactions.

Dynamic Data Injection: In Pawn, every peer advertisement contains a pipe advertisement, which uniquely identifies an input and output communication channel to the peer. This pipe is used by other peers to create an end-to-end channel to dynamically send and receive messages.

Every interacting peer implements a message handler that listens for incoming messages on the peer's input pipe channel. The message payload is passed to the application/service identified by the handler tag field at runtime.

Remote Method Calls (PawnRPC): The PawnRPC mechanism provides the low-level constructs for building applications interactions across distributed peers. Using PawnRPC, a peer can dynamically invoke a method on a remote peer by passing its request as an XML message through a pipe. The interfaces for the methods that are exported by a peer are published as part of the peer advertisement during peer discovery. The PawnRPC XML message is a composition of the destination address, the remote method name, the arguments of the method, and the arguments associated types. Upon receiving a PawnRPC message, a peer locally checks the credentials of the sender, and if the sender is authorized, the peer invokes the appropriate method and returns a response to the requesting peer. The process may be done in a synchronous or asynchronous manner. PawnRPC uses the messaging guarantees to assure delivery ordering, and stateful messages to tolerate failure.

9. SESAME: dynamic role-based access control engine

A key requirement of autonomic applications is the support for dynamic, seamless and secure interactions between the participating entities, i.e. components, services, applications, data, instruments, resources and users. Ensuring interaction security requires a fine grained access control mechanism. Furthermore, in the highly dynamic and heterogeneous Grid environment, the access rights of an entity depends on the entity's privileges, capabilities, context and state. For example, the ability of a user to access a resource or steer a component depends on users' privileges (e.g. owner), current capabilities (e.g. resources available), current context (e.g. secure connection) and the state of the resource or component. The AutoMate Access Control Engine addresses these issues and provides dynamic access control to users, applications, services, components and resources. The engine is composed of access control agents associated with various entities in the system. The underlying dynamic role based access control mechanism extends the RBAC (Role Based Access Control) model [6, 24] to make access control decisions based on dynamic context information. The access control engine dynamically adjusts *Role Assignments* and *Permission Assignments* as illustrated in Fig. 6.

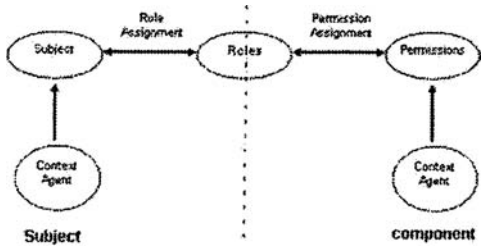

Fig. 6 Dynamic Access Control Model

The subject is the entity which requests service from another entity. In AutoMate, the subject may be a user, application, service or component. The respective context agent is responsible for collecting an entity's current context information such as the state and current execution environment of a component or an application. Based on this context information, the access control agent dynamically adjusts the *user-role* and *role-permission* relationships to dynamically grant appropriate access permissions. Note that the access control agent (and context agent) is authenticated and delegated by the authority service (e.g. a Grid Authority Service). In our approach, each component is assigned a role subset (by the authority service) from the entire role set. Similarly the component has permission subsets for each role that will access the component. During a secure interaction, state machines are maintained by the access control agent at the subject (*Role State Machine*) to navigate the role subset, and the object (*Permission State Machine*) to navigate the permission subset for each active role. The state machine consists of state variables (role, permission), which encode its state, and commands, which transform its state. These state machines define the currently active role and its assigned permissions and navigate the role/permission subsets to react to changes in the context.

The operation of dynamic access control engine at the component layer is illustrated in Fig. 7. This figure shows

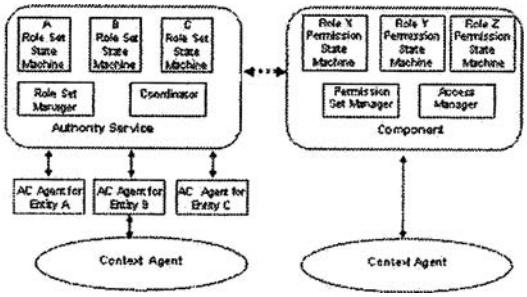

Fig. 7 Dynamic Access Control in AutoMate

three AutoMate components, A, B, and C, each with their own access control agents and state machines assigned to them by the authority service. The access control agent maintains the role state machine for each component and defines its active role based on its current context. When the subject component accesses another component, it will first get its current role from its role state machine, and then use this role to access the component. At the accessed component, a permission state machine is defined (if it does not already exist) for the active role. For example, active roles X, Y, and Z have their own permission state machines at the component. The access control agent at the accessed component will maintain this permission state machine to define the current permissions for a role based in its current context and state.

10. Enabling autonomic applications in science and engineering using AutoMate

10.1. Project AutoMate: current status

The core components of AutoMate have been prototyped and are currently being used to enable self-managing applications in science and engineering. The initial prototype of Accord extended an object-oriented framework based on C++ and MPI. The current implementation extends the DoE Common Component Architecture (CCA) [3] and we are working on extending an OGSA-based programming system. Current prototypes of Rudder and Meteor build on the JXTA [8] platform and use existing Grid middleware services. Current applications include autonomic oil reservoir optimizations [18, 20], autonomic forest-fire management [10], autonomic runtime management of adaptive simulations [5], and enabling sensor-based pervasive applications [7]. The first two application are briefly described below. Further information about AutoMate and its components and applications can be obtained from http://automate.rutgers.edu/.

10.2. Autonomic oil-reservoir optimization

One of the fundamental problems in oil reservoir production is determining the optimal locations of the oil production and injection wells. However, the selection of appropriate optimization algorithms, the runtime configuration and invocation of these algorithms and the dynamic optimization of the reservoir remains a challenging problem. In this research we use AutoMate to support autonomic aggregations, compositions and interactions and enable an autonomic self-optimizing reservoir application. The application consists of: (1) sophisticated reservoir simulation components that encapsulate complex mathematical models of the physical interaction in the subsurface, and execute on distributed computing systems on the Grid; (2) Grid services that provide secure and coordinated access to the resources required by the simulations; (3) distributed data archives that store historical, experimental and observed data; (4) sensors embedded in the instrumented oilfield providing real-time data about the current state of the oil field; (5) external services that provide data relevant to optimization of oil production or of the economic profit such as current weather information or current prices; and (6) the actions of scientists, engineers and other experts, in the field, the laboratory, and in management offices.

The main components of the autonomic reservoir framework [18] are (i) instances of distributed multi-model, multi-block reservoir simulation components, (ii) optimization services based on the Very Fast Simulated Annealing (VFSA) [26] and Simultaneous Perturbation Stochastic Approximation (SPSA) [27], (iii) economic modeling services, (iv) real-time services providing current economic data (e.g. oil prices) and, (v) archives of data that has already been computed, and (vi) experts (scientists, engineers) connected via pervasive collaborative portals.

The overall oil production process is autonomic in that the peers involved automatically detect sub-optimal oil production behaviors at runtime and orchestrate interactions among themselves to correct this behavior. Further, the detection and optimization process is achieved using policies and constraints that minimize human intervention. Policies are used to discover, select, configure, and invoke appropriate optimization services to determine optimal well locations. For example, the choice of optimization service depends on the size and nature of the reservoir. The SPSA algorithm is suited for larger reservoirs with relatively smooth characteristics. In case of reservoirs with many randomly distributed maxima and minima, the VFSA algorithm can be employed during the initial optimization phase. Once convergence slows down, VFSA can be replaced by SPSA. Similarly, policies can also be used to manage the behavior of the reservoir simulator, or may be defined to enable various optimizers to execute concurrently on dynamically acquired Grid resources, and

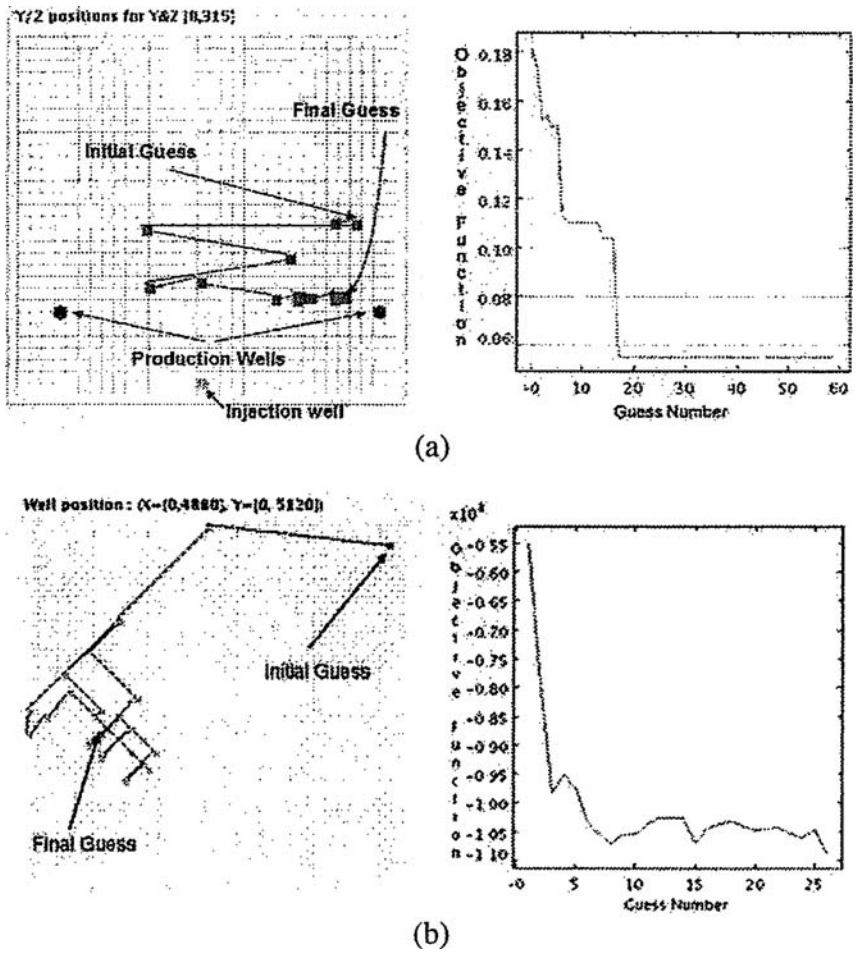

Fig. 8 Convergence history for the optimal well placement in the Grid using (a) VFSA algorithm on the left and (b) SPSA algorithm on the right.

select the best well location among these based on some metric (e.g., estimated revenue, time or cost of completion).

Figure 8 illustrate the optimization of well locations using the VFSA and SPSA optimization algorithms for two different scenarios. The well positions plots (on the left in 8(a) and (b)) show the oil field and the positions of the wells. Black circles represent fixed injection wells and a gray square at the bottom of the plot is a fixed production well. The plots also show the sequence of guesses for the position of the other production well returned by the optimization service (shown by the lines connecting the light squares), and the corresponding normalized cost value (plots on the right in 8(a) and (b)).

10.3. Autonomic forest fire management simulation

The autonomic forest fire simulation, composed of *DSM* (*Data Space Manager*), *CRM* (*Computational Resource Manager*), *Rothermel, WindModel*, and *GUI elements*, predicts the speed, direction and intensity of the fire front as the fire propagates using static and dynamic environment and vegetation conditions. *DSM* partitions the forest represented by a 2D data space into sub spaces based on current system resources information provided by *CRM*. Under the circumstance of load imbalance, *DSM* re-partitions the data space. *Rothermel* generates processes to simulate the fire spread on each subspace in parallel based on current wind direction and intensity simulated by the *WindModel*, until no *burning* cells remain. Experts interact with the above elements using the *GUI* element.

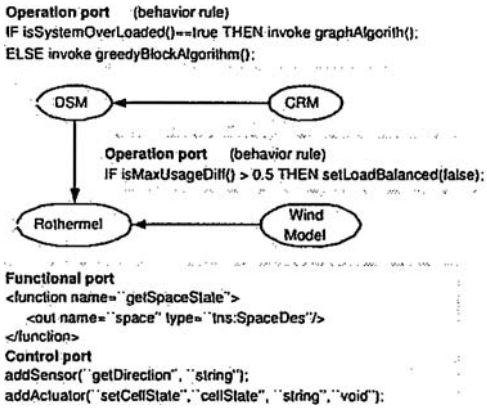

Fig. 9 Examples of the port definition

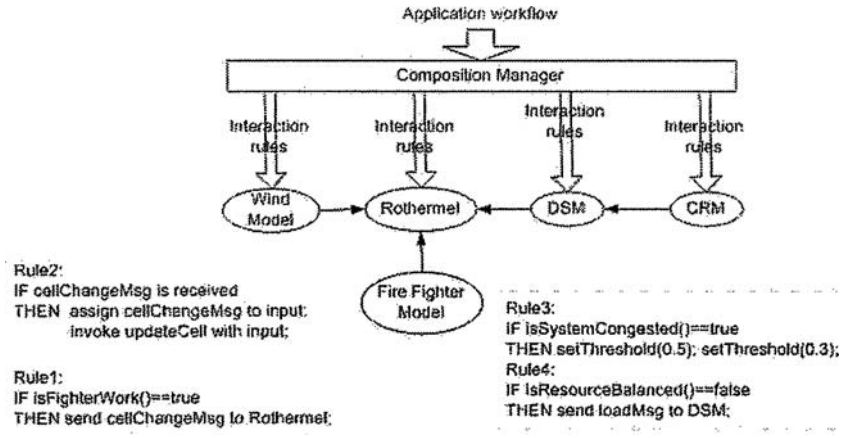

Fig. 10 Add a new component *Fire Fighter Model* and change the interaction relationship between *CRM* and *DSM*

We use the *Rothermel, DSM*, and *CRM* as examples to illustrate the definition of the Accord functional, control and operational ports, as shown in Fig. 9. *Rothermel*, for example, provides *getSpaceState* to expose space information as part of its **Functional Port**, and provides the sensor *getDirection* to get the fire spread direction and the actuator *setCellState* to modify the state of a specified cell as part of its **Control Port**. The *DSM* and *CRM* receive rules to manage their runtime behaviors through the **Operation Port**.

Behavior rules can be defined at compile time or at runtime and injected into corresponding element managers to dynamically manage the computational behaviors of elements. As illustrated in Fig. 9, *DSM* dynamically selects an appropriate algorithm based on the current system load and *CRM* will detect load imbalance when the maximal difference among resource usage exceeds the threshold according to the behavior rules shown.

The application workflow is decomposed by the *Composition Manager* into interaction rules, which are injected into individual elements. Therefore, addition, deletion and replacement of elements can be achieved using corresponding interaction rules. For example, a new element, *Fire Fighter Model*, modelling the behaviors of the fire fighters, is added to the application as shown in Fig. 10, by inserting Rule1 into *Fire Fighter Model* and Rule2 into *Rothermel*. Similarly, changing an interaction relationship can be achieved by replacing the existing interaction rules with new rules. As shown in Fig. 10, *CRM* dynamically decreases the frequency of notifications to *DSM* when the communication network is congested based on Rule3 and Rule4.

11. Conclusion

In this paper, we presented Project AutoMate and described its key components. Project AutoMate investigates solutions that are based on the strategies used by biological systems to deal with the challenges of Grid environment, including complexity, dynamism, heterogeneity and uncertainty. This approach, referred to as autonomic computing, 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. Illustrative autonomic scientific and engineering Grid applications enabled by AutoMate were presented.

References

1. M. Agarwal, V. Bhat, Z. Li, H. Liu, B. Khargharia, V. Matossian, V. Putty, C. Schmidt, G. Zhang, S. Hariri and M. Parashar, AutoMate: Enabling Autonomic Applications on the Grid. In: *Proceedings of the Autonomic Computing Workshop, 5th Annual International Active Middleware Services Workshop* (AMS2003). Seattle, WA, USA (2003) pp. 48–57.
2. M. Agarwal and M. Parashar, Enabling Autonomic Compositions in Grid Environments. In: *Proceedings of the 4th International Workshop on Grid Computing*. Phoenix, AZ (2003) pp. 34–41.
3. 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. *Concurrency and Computing:Practice and Experience* 14(5) (2002) 323–345.
4. V. Bhat and M. Parashar, Discover Middleware Substrate for Integrating Services on the Grid. In: T. P. Prasanna and V.K. (eds.): *10th International Conference on High Performance Computing (HiPC 2003)*. pp. 373–382.
5. S. Chandra, M. Parashar and S. Hariri, GridARM: An Autonomic Runtime Management Framework for SAMR Applications in Grid Environments. In: *New Frontiers in High-Performance Computing, Proceedings of the Autonomic Applications Workshop, 10th*

International Conference on High Performance Computing (HiPC 2003). Hyderabad, India, pp. 286–295.
6. M. J. Covington, M. J. Moyer and M. Ahamad, Generalized Role-Based Access Control for Securing Future Applications. In: *In Proceedings of the 23rd National Information Systems Security Conference (NISSC 2000)* .
7. N. Jiang, C. Schmidt, V. Matossian and M. Parashar, Enabling Applications in Sensor-based Pervasive Environments. In: *Proceedings of the 1st Workshop on Broadband Advanced Sensor Networks (BaseNets 2004)*. San Jose, CA, USA.
8. JXTA: 2001, Project JXTA. http://www.jxta.org.
9. D. Karger, E. Lehman, T. Leighton, M. Levine, D. Lewin and R. Panigrahy, Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web. In: *ACM Symposium on Theory of Computing*. (1997) pp. 654–663.
10. B. Khargharia, S. Hariri and M. Parashar, vGrid: A Framework for Building Autonomic Applications, In: *Proceedings of 1st International Workshop on Heterogeneous and Adaptive Computing-Challenges of Large Applications in Distributed Environments (CLADE 2003)*. pp. 19–26.
11. G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier and J. Irwin, Aspect-Oriented Programming. In: M. Akşit and S. Matsuoka (eds.): *Proceedings European Conference on Object-Oriented Programming*, Vol. 1241. Berlin, Heidelberg, and New York (1997) pp. 220–242.
12. Z. Li and M. Parashar, Rudder: A Rule-based Multi-agent Infrastructure for Supporting Autonomic Grid Applications. In: *Proceedings of the International Conference on Autonomic Computing*. New York, NY (2004).
13. H. Liu and M. Parashar, DIOS++: A Framework for Rule-Based Autonomic Management of Distributed Scientific Applications. In: *Proceedings of the 9th International Euro-Par Conference (Euro-Par 2003), Lecture Notes in Computer Science*. Klagenfurt, Austria, pp. 66–73.
14. H. Liu, M. Parashar and S. Hariri, A Component-based Programming Framework for Autonomic Applications. In: *Proceedings of the 1st IEEE International Conference on Autonomic Computing (ICAC-04), IEEE Computer Society Press*. New York, NY (2004) pp. 278–279.
15. V. Mann, V. Matossian, R. Muralidhar and M. Parashar, DISCOVER: An Environment for Web-based Interaction and Steering of High-Performance Scientific Applications. *Concurrency and Computation: Practice and Experience* 13(8–9) (2001) 737–754.
16. D. C. Marinescu, *Internet-Based Workflow Management: Towards a Semantic Web* (John Wiley & Sons, 2002).
17. K. Marriott and P. J. Stuckey, *Programming with Constraints: an Introduction* (MIT Press, 1999).
18. V. Matossian and M. Parashar, Autonomic Optimization of an Oil Reservoir using Decentralized Services. In: *Proceedings of the 1st International Workshop on Heterogeneous and Adaptive Computing-Challenges for Large Applications in Distributed Environments (CLADE 2003)*. Seattle, WA, USA (2003a) pp. 2–9.
19. V. Matossian and M. Parashar, Enabling Peer-to-Peer Interactions for Scientific Applications on the Grid. In: H. H. H. Kosch, L. Boszormenyi (ed.): *Proceedings of the 9th International Euro-Par Conference (Euro-Par 2003)*, Vol. 2790. Klagenfurt, Austria (2003b) pp. 1240–1247.
20. V. Matossian, M. Parashar, W. Bangerth, H. Klie and M. F. Wheeler, An Autonomic Reservoir Framework for the Stochastic Optimization of Well Placement. *Cluster Computing: The Journal of Networks, Software Tools, and Applications* (2004).
21. A. Popovici, T. Gross and G. Alonso, Dynamic weaving for aspect-oriented programming. In: *Proceedings of the 1st international conference on Aspect-oriented software development*. (2002) pp. 141–147.
22. S. Ratnasamy, P. Francis, M. Handley, R. Karp and S. Schenker, A scalable content-addressable network. In: *Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*. (2001) pp. 161–172.
23. H. Sagan, *Space-Filling Curve*. Springer Verlag (1995).
24. R. Sandhu, D. Ferraiolo and R. Kuhn, The NIST model for role-based access control: towards a unified standard. In: *Proceedings of the fifth ACM workshop on Role-based access control*. (2000) pp. 47–63.
25. C. Schmidt and M. Parashar, Flexible Information Discovery in Decentralized Distributed Systems. In: *Proceedings of the 12th International Symposium on High Performance Distributed Computing*. Seattle, WA (2003) pp. 226–235.
26. M. Sen and P. Stoffa, *Global Optimization Methods in Geophysical Inversion* (Elsevier, 1995).
27. J. C. Spall, Adaptive stochastic approximation by the simulateous perturbation method. *IEEE Trans. Autom. Contr*. 45 (2000) 1839–1853.
28. I. Stoica, R. Morris, D. Karger, M. F. Kaashoek and H. Balakrishnan, Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. In: *Proceedings of the ACM SIGCOMM'01 Conference*. San Diego, California (2001) pp. 149–160.
29. G. Zhang and M. Parashar, Cooperative Mechanism against DDoS Attacks. In: *Proceedings of IEEE International Conference on Information and Computer Science (ICICS 2004)*.
30. G. Zhang, and M. Parashar, November Dynamic Context-aware Access Control for Grid Applications. In: *Proceedings of the 4th International Workshop on Grid Computing (Grid 2003)*. Phoenix, AZ, USA (2003) pp. 101–108.