Rule-based Monitoring and Steering of Distributed Scientific Applications

Hua Liu and Manish Parashar
The Applied Software Systems Laboratory Dept of Electrical and Computer Engineering, Rutgers University, Piscataway, NJ 08854, USA

Abstract: This paper presents the design, prototype implementation and experimental evaluation of DIOS++, an infrastructure for enabling rule-based management and control of distributed scientific applications. DIOS++ provides: (1) abstractions to enhance existing application objects with sensors and actuators for runtime interrogation and control, access policies to control accesses to sensors/actuators and rule interfaces, and rule agents to enable autonomic monitoring and steering, (2) a hierarchical control network that connects and manages the distributed sensors and actuators, enables external discovery, interrogation, monitoring and manipulation of these objects at runtime, and facilitates dynamical and secure definition, modification, deletion and execution of rules for autonomic application management and control. The framework is currently being used to enable autonomic monitoring and control of a wide range of scientific applications including oil reservoir, compressible turbulence and numerical relativity simulations.

Keywords: Computational steering environment; rule; autonomic; agent.

Reference to this paper should be made as follows: Liu, H. and Parashar, M. (xxxx) 'Rule-based Monitoring and Steering of Distributed Scientific Applications', *International Journal of High Performance Computing and Networking*, Vol. x, No. x, pp.xxx–xxx.

Biographical notes: Hua Liu is Ph.D candidate in the Center for Advanced Information Processing, department of Electrical and Computer Engineering at Rutgers University. Her research interests include distributed and parallel computing (Autonomic, Grid, peer-to-peer computing), high-performance computing.

Manish Parashar is Associate Professor of Electrical and Computer Engineering at Rutgers University, where he is also director of the Applied Software Systems Laboratory. He received a BE degree in Electronics and Telecommunications from Bombay University, India, and MS and Ph.D. degrees in Computer Engineering from Syracuse University. He has received the NSF CAREER Award and the Enrico Fermi Scholarship from Argonne National Laboratory. His current research interests include parallel and distributed computing (including autonomic, Grid and peer-to-peer computing) with applications to computational science and engineering, networking and software engineering. Manish 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), and a member of ACM. He is also the co-founder of the IEEE International Conference on Autonomic Computing (ICAC). Manish has co-authored over 140 technical papers in international journals and conferences, has co-authored/edited 6 books/proceedings, and has contributed to several others in the area of computational science and parallel and distributed computing. For more information please visit http://www.caip.rutgers.edu/$\sim$ *parashar*/.

1 INTRODUCTION

High-performance parallel/distributed simulation applications are playing an increasingly important role in science and engineering and are rapidly becoming critical research modalities. These simulations and the phenomena that they model are long running, inherently complex and highly dynamic. Therefore, the simulations must be capable of dynamically managing, adapting and optimizing their behaviors to match the dynamics of the physics they are modelling and the state of their execution environment, so that they continue to meet their goals and constraints. As a result, techniques and tools for runtime interactive monitoring and steering have been developed to address this requirement. Interactive monitoring enables the information about the computation to be gathered on the fly, and interactive steering enables the state and behaviors of the computation to be changed on the fly. These systems promote a deeper understanding and insight into the behaviors of the simulations by supporting observation and experimentation of on-going applications [1]. Current systems include CAVEStudy [2], VASE [3], CUMULVS [4], VIPER [5] and SCIRun [6].

However, as the scale and complexity of these simulations grow, traditional interactive monitoring and steering by humans not only become tedious and error-prone but infeasible. This has led to the investigation of the autonomic systems where the simulations can monitor and control themselves based on the high-level rules defined by the users. This approach has been inspired by the strategies used by biological systems to deal with complexity, dynamism, heterogeneity and uncertainty. The approach, referred to as autonomic computing [7], aims at realizing computing systems and applications capable of managing themselves at runtime with minimal human interference. Therefore, users can be alleviated from time- and effort-consuming runtime monitoring, analyzing and steering routines via incorporating human knowledge with applications and steering systems to enable self-management.

In this paper we present the design, prototype implementation and experimental evaluation of DIOS++, a framework to support the rule-based autonomic monitoring and controlling of distributed and parallel applications. DIOS++ enables high-level rules/policies to be dynamically composed and securely injected into applications at runtime, allowing applications to manage and autonomically optimize their execution. Rules specify conditions to be monitored and operations that should be executed when certain conditions are detected. Rather than continuously monitoring and steering the simulations, experts can define and deploy appropriate rules that are automatically evaluated and executed at runtime to manage the computation, apply runtime corrections based on the observed state, and optimize application execution.

DIOS++ provides: (1) abstractions to enhance existing application objects with sensors and actuators for runtime interrogation and control, access policies to control accesses to sensors/actuators and rule interfaces, and rule agents to enable rule-based autonomic monitoring and steering, (2) a hierarchical control network that connects and manages the distributed sensors and actuators, enables external discovery, interrogation, monitoring and manipulation of these objects at runtime, and facilitates dynamical and secure definition, modification, deletion and execution of rules for autonomic application management and control. Rules can be dynamically composed using sensors and actuators exported by application objects. These rules are automatically decomposed, deployed into the appropriate rule agents using the control network, evaluated and executed by the rule agents in a distributed and parallel manner.

DIOS++ builds on the DIOS [8], a distributed object substrate for interactively monitoring and steering parallel scientific simulations. DIOS++ extends DIOS with an agent-based framework that enables rule-based autonomic management. This alleviates time- and effort-consuming interactive monitoring and control and enables richer management behaviors. DIOS++ also provides an object-level access control mechanisms. DIOS++ is part of the Discover [1] computational collaboratory. Discover enables geographically distributed clients to collabora-

[1]http://www.discoverportal.org

tively access, monitor and control Grid applications using pervasive portals. It is currently being used to enable interactive monitoring, steering and control of a wide range of scientific applications, including oil reservoir, compressible turbulence and numerical relativity simulations.

The rest of the paper is organized as follows: Section 2 investigates related work. Section 3 introduces the Discover collaboratory. Section 4 presents the architecture and operations of DIOS++. Section 5 illustrates DIOS++ using an oil reservoir simulation application. Section 6 presents the experimental evaluation. Section 7concludes the paper.

2 RELATED WORK

Related research efforts include techniques for integrating monitoring and steering capabilities with applications, systems for collaborative (multi-application) monitoring and steering, and autonomic monitoring and steering. These efforts are briefly described below.

2.1 Interactive Monitoring and Steering

Traditional debugging tools enable the tracking and altering of program variables using checkpoints. However, the overhead associated with these tools makes them unfeasible for large, complex and long-running applications. An alternate approach enables runtime steering using standard I/O (e.g. files). Systems such as CAVEStudy [2] implement this approach. This approach however cannot support fine-grained monitoring and steering.

Beazley and Lomdahl [9] demonstrated the use of a lightweight method for steering very-large molecular-dynamics simulations. They used a simplified wrapper interface generator (SWIG) that wraps existing source code with scripting language interfaces to enable external monitoring and steering. Systems such as VASE [3] also implement this approach. The script is executed when the application encounters the pre-defined breakpoints, resulting in monitoring and steering behaviors. A key drawback of this approach is that the steering behaviors have to be defined when the scripts are generated and cannot be changed at runtime. This approach also requires the knowledge of scripting languages.

Systems such as CUMULVS [4] and VIPER [5] use program instrumentation to enable computational steering. CUMULVS allows developers to declare the variables or parameters that can be modified or steered during the computation. VIPER similarly allows developers to annotate application programs to identify the data and input parameters for monitoring and steering, and associate them with synchronization points. When the application encounters a synchronization point, a server is notified, which extracts the current state of the data and parameters. However, CUMULVS and VIPER are limited in providing rich semantic steering, e.g., coordinated steering across modules.

As an improvement, problem solving environments, for example, SCIRun [6], provide mechanisms (e.g., feedback loops, cancellation, direct lightweight parameter changes, and retained state across module firings) to enable modular and dataflow-oriented systems to create a richer set of steerable parameters. However, a key limitation of CUMULVS, VIPER and SCIRun is that they only support monitoring and steering through pre-defined variables or parameters. These system cannot directly support functional or algorithmic steering - it can only be achieved indirectly through pre-defined variables, which increases the programming complexity.

The Mirror Object Steering System (MOSS) provides a high-level model for steering applications. The mirror objects, which are analogues to the application objects used for monitoring and steering, export application methods to the interaction system through which steering actions are accomplished [8]. We believe the high-level abstractions for interaction and steering provide the most general approach for enabling interactive applications. DIOS++ extends this approach to enable synchronous and asynchronous, interactive and batch-processing monitoring and steering with more semantics.

2.2 Multi-application and Collaborative Monitoring and Steering

Systems such as Magellan [3] and CSE [3] allow users to steer multiple applications simultaneously. Further, CUMULVS [4] enables multiple users to collaboratively steer the same application. The OViD [10] allows multiple users to observe and steer the same application. However, it is not a collaborative system, since data changes performed in one visualization unit are not propagated to other visualization units.

A key issue in these systems is consistency and correctness. Consistency guarantees that displays presented to the viewer represent some valid state of the computation and that steering operations are applied in a way that maintains the correctness of the computation [11].

Computational steering systems address the consistency concerns at different levels. Early steering systems leave these concerns entirely up to the users, or rely on the applications to provide necessary consistency checks [11]. Systems that use scripts to control the execution of software modules, e.g., VASE [3], limit interactions only to the invocation of modules, with no control over their inner state. The VASE ensures consistency of steering actions through entirely depending on users placing instrumentation at 'safe' points in the execution of the code [11]. The SCIRun [6] supports both inter- and intra-module steering. The consistency of intra-module steering depends on the module implementation, while the consistency of inter-module steering is achieved through cancellation and re-execution. The MOSS system assumes that objects encapsulate consistency concerns and inter-object consistency concerns are handled by applications [11].

2.3 Autonomic Monitoring and Steering

Systems such as KX(Kinesthetics eXtreme) [12] and Pathfinder [1] use mobile agents to support automated monitoring and steering. Mobile agents present power and flexibility in the specification and deployment of monitoring and steering commands [1]. For example, they are capable of executing orthogonally to the main computation of target applications in a decentralized style. Besides, mobile agents can be deployed locally with application modules, which reduces the latency when reacting to local conditions and provides corresponding actions. As another example, mobile agents can be customized to make use of application-specific information, permitting efficient solutions. However, to support the execution of mobile agents, virtual machines or milieus are required at all 'stops'. The virtual machine/mulieu serves as the hosting environment for mobile agents, providing a library of operations for agents to perform monitoring and steering actions, supporting agents' communication, migration and agent scheduling. The requirements of the hosting environment make it hard to be implemented. Besides, the behaviors of those mobile agents are restricted for security purpose. Possible security problems include masquerading, denial of service, unauthorized access, eavesdropping, alteration, repudiation [13].

An alternate approach to enable autonomic management is based on high-level rules/policies. Systems such as Autopilot [14] implement this approach. Rules/policies can be automatically activated when their conditions are fulfilled. Therefore, users can be relieved from continuous interactive monitoring and steering. Further, rules/policies can help the system scalability by reducing overall dataflow from sensors, help dynamically activate or deactivate sensors and actuators, and also enable coordinated actuator controls. In this approach, management behaviors are described and deployed to managed objects as rules in plain text format. These rules can be accepted and understood by objects developed independently and spanning different security domains, provided they agree on the syntax and semantics of the rules. Further, objects can easily customize rule execution, since rules only specify policy (i.e., 'what to do') and the objects implement mechanisms (i.e., 'how to do it'). This is in contrast with the mobile agent approach, where management behaviors are deployed to managed objects as executable code, which cannot be easily customized and may present security risks (may be malicious). To implement the rule-based approach, managed objects need to provide or be integrated with rule interpretation and execution mechanisms, which may increase the complexity of object development. Another disadvantage is that the rule-based approach may have larger overheads, since the rules need to be interpreted. However, this overhead can be reduced using a 'rule compiler' to compile and customize rules into executable code for direct execution.

The rule-based approach, unlike the mobile agent approach, can be applied to applications developed by multiple parties and spanning multiple security domains. DIOS++ takes the rule-based approach as we believe that scientific applications are evolving from single-user monolithic codes to the integration and assembly of componentized units developed independently by multiple parties [15].

2.4 Autonomic Monitoring and Steering with DIOS++

Rule-based autonomic monitoring and steering enhance computational steering of long-term, complex, computation- and resource-intensive applications with the capability of monitoring themselves and making corresponding actions automatically based on user-defined high-level rules. This may include automatically on the fly requesting/modifying program state, stalling program execution, calibrating runtime behaviors of applications, exploring new computational solutions for problems that are not yet well understood, adapting programs to the current computational environments, etc.

DIOS++ achieves autonomic management and control by integrating the agent-based approach with rule-based approach to strengthen the flexibility of agents with high-level rules that incorporate human knowledge. Key research issues addressed by DIOS++ include:

- Integrating monitoring and steering functionalities with applications: To realize the external monitoring and steering capacity, a small amount of modification to the applications' source code is required. The objects to be monitored and steered must explicitly expose sensors and actuators. The sensors and actuators include variables/parameters and functions/methods.

  Applications written in procedural languages need to transform their data structures to objects using, for example, a C++ wrapper. Although this requires some application modification, the wrappers are only required for those data-structures that have to be made interactive and the effort required is far less than rewriting the entire application to be interactive.

- Rich monitoring and steering capabilities: both intra-object (direct lightweight parameter changes) and inter-object (the adjustment of input parameters to objects) monitoring/steering are supported. Besides, synchronous monitoring/steering (the realtime responses to users' monitoring/steering requests) and asynchronous monitoring/steering (the monitoring/steering behaviors are specified in rules, which will be executed automatically when conditions are satisfied) are enabled.

- Consistency: Consistency of intra-object steering behaviors depends on the actuator constraints specified in the rules which are embedded inside the objects. Those constraints will automatically restrict steering behaviors within a valid range. To support inter-object consistency, the lifetime of an application is divided into iterations of computation and interaction. Steering behaviors are completed in one iteration and will automatically become effective from the next iteration. Further, a simple locking mechanism is used to ensure that applications remain in a consistent state during collaborative interactions.

- Collaboration: Users can form or join collaboration groups and interact with one or more applications based on their capabilities, which is supported by DISCOVER server [16]. Users in one collaboration group can selectively receive or broadcast application information, and disable their collaboration capacity so that their requests/responses are not broadcasted to the entire collaboration group.

3 DISCOVER COLLABORATORY

The Discover collaboratory (shown in figure 1) provides a virtual, interactive and collaborative PSE that enables geographically distributed scientists and engineers to collaboratively monitor and control high-performance parallel/distributed applications. It consists of the Discover server as the front-end and DIOS++ architecture as the back-end. Clients can connect to, collaboratively interact with registered applications, and execute rule operations using the portals. Applications register, expose sensors/actuators, receive synchronous monitoring and steering behaviors as well as asynchronous behaviors based on rules through DIOS++ architecture.

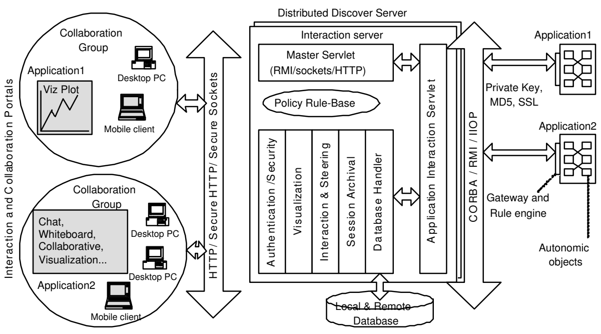

Figure 1: Discover Collaboratory Architecture.

The Discover server builds on a traditional web server and extends its functionality to handle real-time application information and client requests with "handlers" servlets that provide interaction, collaboration and rule services. The Discover server provides each registered client with a unique client-id, and each registered application with a unique application-id. The client-id along with an application-id (corresponding to the application to which the client is connected) is used to identify each session. To start interaction behaviors, users must be authenticated by the authentication handler, which builds a customized interaction interface for each valid client to match his/her access capabilities. This ensures that the client can only access, interact with and steer applications in an authorized way. One thing needs to be noted is the Discover capability to avoid the situation in which multiple clients try to steer the same variable/parameter/operation. In Discover, clients must explicitly request and release locks before and after their steering behaviors. In the back-end DIOS++ architecture, the similar lock mechanism is used to avoid multiple rule agents invoking the same steering access points simultaneously. Rules with high priority will lock those steering points when the conditions specified in the rules are satisfied. The locks are released when rules with higher priority disable the rules or the conditions are no longer fulfilled.

Discover enables multiple users to collaboratively interact with and steer applications. All clients connected to a particular application form a collaboration group by default. Global updates (e.g. current application status) are automatically broadcasted to this group. The clients can selectively broadcast application information to the group. Also, they can select the type of information that they are interested in. Finally, clients can disable all collaboration so that their requrests/responses are not broadcasted to the entire collaboration group. In addition, each application on the client portal is provided with chat and whiteboard tools to further assist collaboration. Detailed description of the Discover server is presented in [16].

4 DIOS++ ARCHITECTURE

DIOS++ is composed of 3 key components: (1) autonomic objects that extend computational objects with sensors to monitor the state of the objects, actuators to modify the state, access policies to control accesses to sensors/actuators and rule interfaces, and rule agents to enable rule-based autonomic monitoring and steering, (2) a hierarchical control network that is dynamically configured to enable runtime access to and management of the autonomic objects including their sensors, actuators, access policies and rules, and to enable dynamical and secure definition, modification, deletion and execution of rules.

4.1 Autonomic Object

In addition to its functional interfaces, an autonomic object (shown in figure 2) exports three interfaces: (1) a *control interface*, which defines sensors and actuators to allow the object's state to be externally monitored and controlled, (2) an *access interface*, which controls access to the sensors/actuators and rule interfaces, and describes users' access privileges based on their roles and the object's state, and (3) a *rule interface*, which contains rules used to autonomically monitor and control the object, and provides methods for adding, modifying and deleting rules. Rule operations are handled by the rule agent embedded within the autonomic object. These interfaces and the rule agent are described in the following sections. A sample object that generates a list of random integers (*RandomList*) is used as a running example. The number of integers and their range are allowed to be set at runtime.

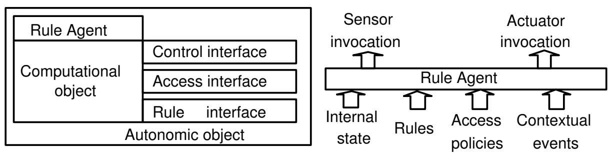

Figure 2: An autonomic object.

4.1.1 Control Interface

The *control interface* specifies the sensors and actuators exported by an object. Sensors provide methods for viewing the current state of an object, and actuators provide methods for processing commands to modify the object's state. For example, a *RandomList* object provides sensors to query the current length of the list or the maximum value in the list, and an actuator for deleting the current list. Note that sensors and actuators must be co-located in memory with the computational objects and must have access to their internal state, since computational objects may be distributed across multiple processors and can be dynamically created, deleted, migrated and redistributed.

DIOS++ provides programming abstraction to enable application developers to define and deploy sensors and actuators. This is achieved by deriving computational objects from a virtual base object provided by DIOS++. The derived objects can then selectively overload the base object methods to specify their sensors and actuators. This process requires minimal modification to the original computational objects and has been successfully used by DIOS++ to support interactive steering.

4.1.2 Access Interface

The *access interface* addresses security and application integrity. It controls the accesses to an object's sensors/actuators and rule interfaces, and constrains them to the authorized users. The role-based access control model is used, where users are mapped to roles and each role is granted specific access privileges defined by access policies.

The DIOS++ defines three roles: owner, member, and guest. Each user is assigned a role based on her/his credentials. The owner can define/modify access policies, and enable or disable external access to sensors/actuators and rule interfaces. The polices define which roles can access a sensor, actuator and rule interface, and in what way. Access polices can be defined statically during object creation using the DIOS++ APIs, or can be injected dynamically by the owner at runtime via secure Discover portals. Objects can dynamically change their access policies based on their current state without affecting other objects. Therefore, a user may be denied of access in one object, but his/her privileges are not changed in another object.

4.1.3 Rule Interface

The DIOS++ framework uses user-defined rules to enable autonomic management and control of applications. The *rule interface* contains rules that define actions to be executed when specified conditions are satisfied, and provides methods for dynamically defining/modifying/deleting rules. The conditions and actions are defined in terms of the *control interface*, e.g., sensors and actuators provided by the object. A rule consists of 3 parts: (1) the condition part, defined by the keyword "IF" and composed of conditions which are conjoined by logical relationships (AND, OR, NOT, etc.), (2) the then action part, defined by the keyword "THEN" and composed of operations that are executed when the corresponding condition is true, and (3) the else action part, defined by the keyword "ELSE" and composed of operations to be executed when condition is not fulfilled.

For example, consider the *RandomList* object with 2 sensors: (1) *getLength()* to get the current length of the list and (2) *getMaxValue()* to get the maximal value in the list, and 1 actuator *append(length, max, min)* that creates a list of size *length* with random integers between *max* and *min*, and appends it to the current list.

```
IF RandomList.getLength()<10
   AND RandomList.getMaxValue()<=50
THEN RandomList.append(10, 50, 0)
```

Note that rules are separated from the application logic. And rules can be created, deleted and modified at runtime orthogonal to the application execution. This provides the flexibility of allowing users to on the fly monitor and adjust the application execution without stopping and restarting the application. Rules are handled by rule agents and the rule engine, which are part of the control network (described in the following sections) and are responsible for storing, evaluating and executing rules.

4.1.4 Rule Agent

There is a rule agent embedded within each autonomic object. The rule agent receives rules from the rule engine through rule interfaces, authorizes and authenticates the user who defines the rules, evaluates and executes the rules based on the internal and contextual state to dynamically monitor and steer its host object via invoking sensors and actuators. Multiple rule agents may coordinate with each other to provide collaborative steering behaviors on multiple autonomic objects. Rule agents and the rule engine will be discussed in section 4.2.

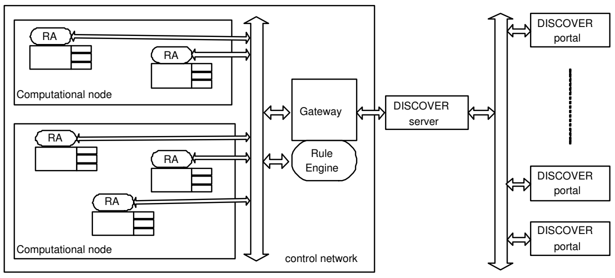

Figure 3: The DIOS++ control network.

4.2 Control Network

The DIOS++ control network (see Figure 3 in which the Discover collaboratory consists of the control network as the back end, the Discover server and portals as the front-end) is a hierarchical structure consisting of the rule engine and Gateway, and computational nodes. It is automatically configured at runtime using the underlying messaging environment (e.g. MPI) and the available processors.

The lowest level of the control network hierarchy consists of computational nodes. Each node maintains a local object registry containing references to all autonomic objects currently active and registered. At the next level of hierarchy, the Gateway represents a management proxy for the entire application. It combines the registries exported by the nodes and manages a registry of the interaction interfaces (sensors and actuators) for all the objects in the application. It also maintains a list of access policies related to each exported interface and coordinates the dynamic injection of rules. The Gateway interacts with external interaction servers or brokers such as those provided by Discover.

Co-located with Gateway, the rule engine accepts and maintains the rules for the application. It decomposes these rules and distributes them to the corresponding rule agents, collects rule execution results from rule agents and reports them to the users. Each rule agent executes its rules based on an execution script, and reports the rule execution results to the rule engine. The execution script is defined by the rule engine to specify the rule execution sequence and the rule agent's runtime behaviors. The specification and execution of scripts and the coordination between the rule engine and rule agents are illustrated in the following sections.

In DIOS++, rules are evaluated and executed by rule agents in a parallel and distributed fashion. The decomposition of rules, collection of rule execution results, and management of rule execution are assumed by the rule engine. This central-control and distributed-execution mechanism has the following advantages: (1) Rule execution which can be compute-intensive is done in parallel by rule agents. This reduces the rule execution time as compared to a sequential rule execution. (2) A rule agent's behavior is based on script that is defined and modified at runtime by the rule engine, allowing it to adapt to the current execution environment and the rules to be executed.

The operation of the control network is explained below using an example. Consider a simple application that generates a list of integers and then sorts them. This application contains two objects: (1) *RandomList* that provides a list of random integers, and (2) *SortSelector* that provides several sorting algorithms (bubble sort, quick sort, etc.) to sort integers.

4.2.1 Initialization

During initialization, the application uses the DIOS++ APIs to create and register its objects, and export its interfaces and access policies to the local computational node. Each node exports these specifications of all its objects to the Gateway. The Gateway then updates its registry. Since the rule engine is co-located with Gateway, it has access to the Gateway's registry. The Gateway interacts with the external access environment (Discover servers in our prototype) and coordinates accesses to the application's sensor/actuators, policies and rules.

4.2.2 Interaction and Rule Operation

The lifetime of an application is divided into iterations of computation and interaction. Users' requests (realtime interaction requests or rule operation requests) received at computation iterations will be queued for execution in the next interaction iteration. Steering actions are completed in one interaction iteration and will automatically become effective from the next computation iteration.

At runtime the Gateway may receive incoming interaction or rule requests from users. The Gateway first checks the user's privileges based on her/his role, and refuses any invalid access. It then transfers valid interaction requests to corresponding objects and transfers valid rule requests to the rule engine. Finally, the responses to the user's requests or the rule execution results are combined, collated and forwarded to the user. Once again we use the example to describe this process.

Rule definition: Suppose *RandomList* exports 2 sensors: *getLength()* and *getList()*. *SortSelector* exports no sensors, and 2 actuators: *sequentialSort()* and *quickSort()*. The owner can access all these interfaces. Members can only access *getLength()* and *getList()* in *RandomList*, and *sequentialSort()* in *SortSelector*. Guests can only access *getLength()* in *RandomList*.

Using DIOS++, users can view, add, delete, modify and temporarily disable rules at runtime using a graphical rule interface integrated with the Discover portal. An application's sensors, actuators and rules are exported to the Discover server and can be securely accessed by authorized users (based on access control polices) via the portal. Authorized users can compose rules using the sensors and actuators. Note that rules may be defined for individual objects, or for the entire application and span multiple objects. Users specify a priority for each rule, which is then used to resolve rule conflicts.

Rule deployment: Consider the following rules defined by a user. Let Rule1 have a higher priority than Rule2:

```
Rule1: IF RandomList.getLength()<100
       THEN RandomList.getList()
       ELSE RandomList.getLength()
Rule2: IF RandomList.getLength()<50
       THEN SortSelector.sequentialSort()
       ELSE SortSelector.quickSort()
```

Rule1 is an object rule, which means that the rule only applies to one object. Rule2 is an application rule, which means that the rule can affect several objects. When the Gateway receives the two rules, it will first check the user's privileges. If the rules are defined by member users, Rule2 will be rejected since member users do not have the privilege to access *quickSort()* interface in *SortSelector.*

The Gateway transfers valid rules to the rule engine. The rule engine dynamically decomposes the rules and inject them into corresponding rule agents. It then composes a script for each agent, which defines the rule agent's lifetime and rule execution sequence based on rule priorities. For example, the script for the rule agent in *RandomList* may specify that this agent will terminate itself when it has no rules, and Rule1 is executed first. Note that this script is extensible.

In the case of an object rule, the rule engine just injects the object rule into its corresponding rule agent. In the case of an application rule, the rule engine will first decompose the rule into triggers and then inject triggers into corresponding agents. The application rule 'Rule2' is decomposed into 3 triggers: (1) *SortSelector.sequentialSort()*, (2) *SortSelector.quickSort()*, and (3) *RandomList.getLength() < 50.* These triggers are injected into corresponding agents as shown in Figure 4. The left figure shows the rule engine deploys an object rule to its corresponding rule agent, and the right one shows the rule engine decomposes an application rule into triggers and injects triggers to corresponding rule agents.

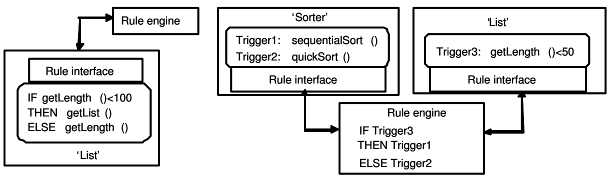

Figure 4: Rule deployment.

Rule execution and conflicts resolution: During interaction iterations, the rule engine fires all the rule agents at the same time, and then those rule agents work in parallel. Rule agents execute object rules and return the results to the rule engine. The rule engine then reports them to the user. Rule agents also execute triggers, which are part of application rules, and report corresponding results to the rule engine. The rule engine collects the required trigger results, evaluates condition combinations, and then issues corresponding actions to be executed in parallel by rule agents if the conditions are fulfilled. Application rule results are also reported to the user.

While typical rule execution is straightforward (actions are issued when their required conditions are fulfilled), the application dynamics and user interactions make things unpredictable. As a result, rule conflicts must be detected at runtime. In DIOS++, rule conflicts are detected at runtime and are handled by grouping rules based on their priority and disabling the conflicting rules with lower priorities. This is done by locking the required sensors/actuators. For example, suppose that a user defines two rules for the object instance *RandomList.* Rule3 requires setting the minimal integer value to 5 when the list length is less than 100 and larger than 50, and Rule4 requires the minimal value to be 6 when the list length is larger than 30 and less than 70. Rule3 has higher priority than Rule4. The two rules conflict with each other, for example, when the list length is 60.

```
Rule3: IF RandomList.getLength()>50
          AND RandomList.getLength()<100
       THEN RandomList.setMinInt() = 5
Rule4: IF RandomList.getLength()>30
          AND RandomList.getLength()<70
       THEN RandomList.setMinInt() = 6
```

The script asks the rule agent to fire Rule3 first. After Rule3 is executed, the interface of *setMinInt()* is locked during the period when the length is less than 100 and larger than 50. When Rule4 is issued, it cannot be executed as the required interface is locked. The interface will be unlocked when the length value is not within the range of 50 to 100.

The rule execution model is being enhanced (see [17]) to handle conflicts among rules with the same priority and between newly added rules and existing rules, using a three-phase model. Specifically, the rule conflicts are resolved by relaxing the rule conditions until a non-empty intersection between the actions of conflicting rules is found. Users define the condition relaxation strategies and the model executes these strategies during rule enforcement process.

5 AN ILLUSTRATIVE EXAMPLE

In this section, we use the oil reservoir simulation application [18] to illustrate the ideas described in section 4. The application optimizes the placement and operation of oil wells to maximize overall revenue. The application consists of the instances of distributed multi-model, multi-block reservoir simulation components provided by the IPARS, simulated annealing based optimization services provided by the VFSA, economic modelling services, real-time services providing current economic data (e.g. oil prices), historical data archives, and experts (scientists, engineers) connected via collaborative portals. In the initialization period, experts configure and launch the IPARS factory and the VFSA optimization service. In the iterative optimization phase, the IPARS factory gets initial guess from the VFSA and launches an IPARS instance, which uses the Economic Model along with current market parameters to estimate the current revenue. This revenue is normalized and then communicated to the VFSA service, which in turn uses this value to generate an updated guess of the well parameters and sends this to the IPARS Factory. The IPARS Factory now configures a new instance of IPARS with the updated well parameters and deploys it. This process continues until the required terminating condition is reached (e.g. revenue stabilizes).

The IPARS instance exposes its input parameters (well parameters) for external modification, and also makes its internal physical models changeable from outside. Similarly, the VFSA exposes its input parameters (the revenue) for external modification, and makes its internal probability value (This value is used to determine whether to accept the new model whose energy is larger than the initiate state) changeable from outside.

DIOS++ provides intra-object steering through directly modifying parameters exported by objects. For instance, modification of the probability value of the VFSA will increase or decrease the process time to find a global minimum. Through testing on the probability value for several iterations, the VFSA can determine the best value for optimization. Consistency of intra-object steering behaviors is guaranteed through the actuator constraints specified in the rules which are embedded inside the objects. Those constraints will automatically restrict steering behaviors within a valid range. For instance, a constraint is defined to prevent the probability value from falling out of the range between 0 and 1. When a user tries to set the probability to an invalid value, the constrain will reject the request and send an error message to the user.

```
IF probability <0 OR probability>1
THEN exception(VFSA, probability, error_message)
```

DIOS++ provides inter-object steering through modifying the input parameters to the objects, which are similar to intra-object steering. Let us examine a more complex case that combines intra- and inter-object monitoring and steering. Suppose IPARS provides two algorithms, algorithms1 that generates a result with higher precision but is resource-consuming, and algorithm2 that generates a result with lower precision but consumes less resources. IPARS begins with algorithm2 and then use algorithm1 when the revenue approaches some pre-defined threshold to achieve the best performance in terms of precision under the condition of limited computational resources. The rules are specified as follows:

```
IF VFSA.revenue < threshold THEN IPARS.algorithm2()
                            ELSE IPARS.algorithm1()
```

This rule is decomposed into sensor1 “VFSA.revenue < threshold”, actuator1 “IPARS.algorithm2()” and actuator2 “IPARS.algorithm1()”. the sensor1 is injected into the VFSA rule agent; The actuator1 and actuator2 are injected into the IPARS rule agent. When sensor1 is triggered, IPARS rule agent will be notified and corresponding actuator1 or actuator2 will be executed. This complex monitoring/steering behavior is performed locally within the VFSA and IPARS, without the supervision from the rule engine.

After defined and submitted to the system, the rules will be automatically evaluated and executed to configure the IPARS without human intervene.

Intra- and inter-object monitoring/steering can be synchronous and asynchronous. Synchronous monitoring/steering is a one-time behavior (an example could be the modification of probability value in VFSA); while asynchronous monitoring/steering requires to set up a series of requests and commands which will be executed in a batch when specified conditions are fulfilled (an example could be the complex case discussed above). Rule agents ensure that requests and commands within one rule are executed as one atomic operation.

It is assumed that a typical application proceeds through interaction and computation phases alternately, and an interaction phase occurs after one or more steps (iterations) of computation. Modification happens at iteration N will become effective at iteration N+1. Rules are evaluated at each interaction iteration and fired when the conditions are satisfied. Similarly, rule execution results will be effective from the next iteration. Monitoring/steering requests arrive at computation iteration will be queued and processed at the next interaction iteration. The maximum number of requests to be serviced in an interaction session is a programmable parameter and can be changed to vary the degree of interaction being handled by the application.

6 EXPERIMENTAL EVALUATION

DIOS++ has been implemented as a C++ library. This section summarizes an experimental evaluation of the DIOS library using the IPARS reservoir simulator framework on a 32 node beowulf cluster. IPARS is a Fortran-based framework for developing parallel/distributed reservoir simulators. Using DIOS++/Discover, engineers can interactively feed in parameters such as water/gas injection rates and well bottom hole pressure, and observe the water/oil ratio or the oil production rate. The evaluation consists of 3 experiments:

Experiment 1 (Figure 5): This experiment measures the runtime overhead introduced to the application in DIOS++ minimal rule execution mode. In this experiment, the application automatically updates the Discover server and its connected clients with current state of autonomic objects and rules. Explicit interaction and rule execution are disabled during the experiment. The application's runtime with and without DIOS++ are plotted Figure 5. It can be seen that the runtime overhead due to DIOS++ is very small and within the error of measurements.

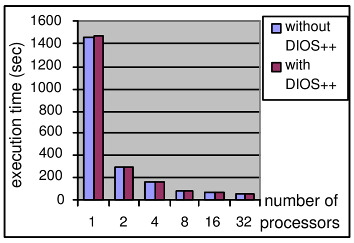

Figure 5: Minimal overhead.

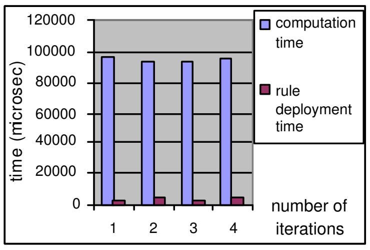

Figure 6: Rule deployment overhead.

Experiment 2 (Figure 6): This experiment compares computation time and the average rule deployment time for successive iterations. In this experiment, we deployed object rules in the first and third iterations, and application rules in the second and fourth iterations. The experiment shows that object rules need less deployment time than application rules. This is true since the rule engine only has to inject object rules to corresponding rule agents, while it has to decompose application rules to triggers, and inject triggers to corresponding rule agents. Since in most cases, rules are deployed at startup and runtime injection is relatively rare, the impact of the rule deployment overhead is not significant.

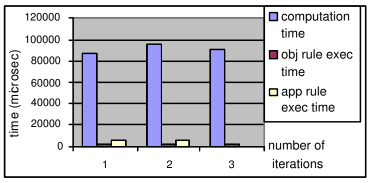

Figure 7: Rule execution overhead.

Experiment 3 (Figure 7): This experiment compares computation time, average object rule execution time and average application rule execution time for successive iterations. The experiment shows that application rules require longer execution time than object rules, since the rule engine has to collect results from all the triggers, check whether the conditions are fulfilled and invoke corresponding actions. The execution of both application rules and object rules involves querying sensors, evaluating conditions, resolving conflicts, and invoking actuators. Sensor queries, conditional evaluation and actuator invocation can be done in parallel. As a result, these overheads are not significantly impacted by the size of the rule base. However, conflict resolution overhead does increase with the size of the rule base. We are currently exploring more efficient conflict resolution algorithms that can improve the performance and scalability of the rule framework.

7 SUMMARY AND CONCLUSION

This paper presented the design, prototype implementation and experimental evaluation of DIOS++, a framework for supporting the rule-based management and control of distributed scientific applications. DIOS++ extends computational objects with control, access and rule interfaces, and embedded rule agents to allow secure external monitoring and steering behaviors with rich semantics. DIOS++ enables a synchronous management via direct interactions between users and application sensors/actuators as well as asynchronous and automatic management via user defined rules.

Rules can be defined, modified and deleted at runtime. They are evaluated and executed in a distributed and parallel manner by rule agents embedded within autonomic objects, to automatically adjust the runtime behaviors of applications. Besides, these rules are defined in a simple “IF-THEN-ELSE” format and can be used with many different applications. The experimental evaluation presented in the paper demonstrates that DIOS++ overheads are small and the framework is scalable.

DIOS++ is currently being used, along with DISCOVER, to enable autonomic monitoring and control of a wide range of scientific applications, including oil reservoir, compressible turbulence and numerical relativity simulations.

ACKNOWLEDGEMENTS

The research presented in this paper is supported in part by the National Science Foundation through grants ACI 9984357, EIA 0103674, EIA 0120934, ANI 0335244, CNS 0305495, CNS 0426354 and IIS 0430826.

REFERENCES

1 B. Kohn, E. Kraemer, D. Hart, and D. Miller, ‘An agent-based approach to dynamic monitoring and steering of distributed computations’, *Proceedings of IASTED*, Las Vegas, Nevada, 2000.

2 L. Renambot, H. E. Bal, D. Germans, and H. J. Spoelder, ‘Cavestudy: an infrastructure for computational steering in virtual reality environments’, *Proceedings of the Ninth IEEE International Symposium on High Performance Distributed Computing*, Pittsburgh, PA, 2000, pp. 5761.

3 J. D. Mulder, ‘Computational steering with parametrized geometric objects’, *Ph.D. dissertation*, Universiteit van Amsterdam, 1998.

4 G. A. Geist, J. A. Kohl, and P. M. Papadopoulos, ‘Cumulvs: Providing fault-tolerance, visualization and steering of parallel applications’, *Proceedings of Environment and Tools for Parallel Scientific Computing Workshop*, Lyon, France, 1996.

5 S. Rathmayer and M. Lenke, ‘A tool for on-line visualization and interactive steering of parallel hpc applications’, *Proceedings of 11th International Parallel Processing Symposium (IPPS’97)*, Geneva, Switzerland, 1997.

6 S. Parker and C. Johnson, ‘An integrated problem solving environment: The scirun computational steering environment’, *Proceedings of HICCS-31*, 1998.

7 J. O. Kephart and D. M. Chess, ‘The vision of autonomic computing’, *Computer magazine*, Vol. 36, No. 1, pp. 4152, 2003.

8 R. Muralidhar and M. Parashar, ‘A distributed object infrastructure for interaction and steering’, *Concurrency and Computation: Practice and Experience*, 2003.

9 D. Beazley and P. Lomdahl, ‘Controlling the data glut in large-scale molecular-dynamics simulations’, *Computers in Physics*, Vol. 11, No. 3, pp. 230238, 1997.

10 S. Rathmayer, ‘Visualization and computational steering in heterogeneous computing environments’, *Proceedings of Euro-Par 2000 - Parallel Processing: 6th International Euro-Par Conference*, Munich, Germany, 2000.

11 D. Hart and E. Kraemer, ‘Consistency considerations in the interactive steering of computations’, *international journal of parallel and distributed networks and systems*, 1999.

12 G. Valetto and G. Kaiser, ‘Using process technology to control and coordinate software adaptation’, *Proceedings of the 25th international conference on Software engineering*, 2003.

13 S. Fischmeister, ‘Mobile code paradigms’, 2002, http://library.mobrien.com/downloads/mobile agent.ppt.

14 J. S. Vetter and D. A. Reed, ‘Real-time performance monitoring, adaptive control, and interactive steering of computational grids’, *The International Journal of High Performance Computing Applications*, Vol. 14, Nno. 4, pp. 357366, 2000.

15 B. A. Allan and et al., ‘The CCA core specification in a distributed memory SPMD framework’, *Concurrency Computation*, Vol. 14, No. 5, pp. 323345, 2002.

16 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*, Vol. 13, No. 8-9, p. 737 754, 2001.

17 H. Liu, ‘A programming framework for autonomic grid applications’, 2004, *Ph.D. proposal*, CAIP, Rutgers University, NJ, USA.

18 V. Matossian and M. Parashar, ‘Autonomic optimization of an oil reservoir using decentralized services’, *Proceedings of the 1st International Workshop on Heterogeneous and Adaptive Computing Challenges for Large Applications in Distributed Environments (CLADE 2003)*, Seattle, WA, USA, 2003.