A Component Based Programming Framework for Autonomic Applications *

Hua Liu and Manish Parashar
The Applied Software Systems Laboratory
Dept of Electrical and Computer Engineering, Rutgers University, Piscataway, NJ08854, USA
Email:{marialiu, parashar}@caip.rutgers.edu

Salim Hariri
Department of Electrical and Computer Engineering, University of Arizona, AZ, USA
Email:hariri@ece.arizon.edu

Abstract

The emergence of pervasive wide-area distributed computing environments, such as pervasive information systems and computational Grids, has enabled new generations of applications that are based on seamless access, aggregation and interaction. However, the inherent complexity, heterogeneity and dynamism of these systems require a change in how the applications are developed and managed. In this paper we present a component-based programming framework to support the development of autonomic self-managed applications. The framework enables the development of autonomic components and the formulation of autonomic applications as the dynamic composition and management of autonomic components. The operation of the proposed framework is illustrated using a forest fire application.

1. Introduction

The emergence of pervasive wide-area distributed computing environments, such as pervasive information systems and computational Grids, has enabled a new generation of applications that are based on seamless access, aggregation and interaction. For example, it is possible to conceive 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 (e.g., your clock estimates drive time to your next appointment based on current traffice/weather and warns you appropriately), 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 pervasive 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, these emerging applications are similarly complex and highly dynamic in their behaviors and interactions. Together, these challenges 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 the 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 minimum human intervention.

In this paper we present the Accord component-based programming framework to support the development of autonomic applications in Grid environments. The framework builds on the separation of composition (organization, interaction and coordination) aspects from computation be-

* The research presented in this paper is supported in part by the National Science Foundation via grants numbers ACI 9984357 (CAREERS), EIA 0103674 (NGS), EIA-0120934 (ITR), ANI-0335244 (NRT), CNS-0305495 (NGS) and by DOE ASCI/ASAP (Caltech) via grant number 82-1052856.

haviors that underlies the component-based paradigm, and extends it to enable the computational behaviors of components as well as their organizations, interactions and coordinations to be managed at runtime using high-level rules.

The rest of the paper is organized as follows: Section 2 investigates Grid application development challenges and related works, and introduces the Accord programming framework. A detailed discussion of the Accord framework is presented in section 3. In section 4, we illustrate the Accord framework using a forest fire application. Experiment results and current status are presented in section 5. Section 6 presents conclusions.

2. Application Development and Management in Pervasive Grid Environments

2.1. Challenges and Requirements

The inherent scale and characteristics of pervasive information and computational Grid environments and applications introduce a new level of development and management complexities. These include:

- **Heterogeneity**: Grid environments aggregate large numbers of independent and geographically distributed computational and information resources, including supercomputers, workstation-clusters, network elements, data-storages, sensors, services, and Internet networks. Similarly, applications typically combine multiple independent and distributed software elements such as components, services, real-time data, experiments and data sources.
- **Dynamism**: The Grid computation, communication and information environment is continuously changing during the lifetime of an application. This includes the availability and state of resources, services and data. Applications similarly exhibit dynamism where the runtime behaviors, organizations and interactions of components may change during execution.
- **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 and frequency of occurrence as system/application scales increase; and (3) incomplete knowledge, which is typical in large decentralized and asynchronous distributed environments.

The above challenges define key requirements for programming and managing Grid applications: (1) Interface definition should be separated from implementation to enable heterogeneous components to interact with each other. (2) Computational behaviors should be separated from interaction and coordination, and their combination should be programmable. For example, the same computational behaviors may be integrated with different interaction and coordination models or languages (and vice versa) to address the dynamism and heterogeneity of the applications and the underlying environments. (3) The computation and interaction/coordination should be context-aware and self-managing, to adapt them to the changing Grid environment and dynamic application requirements.

2.2. Existing Programming Frameworks

There has been a significant body of research on programming frameworks for parallel and distributed computing over the last few decades. While current **programming frameworks for parallel computing** (i.e., message passing models and shared memory models) and **traditional distributed object-oriented models** support parallel and distributed applications, they make very strong assumptions about the applications entities (i.e., tasks, processes, objects), their interactions, and/or the system, which limits their applicability to Grid environments.

Component-based programming frameworks (e.g., CORBA [7], CCA [1] and JavaBeans [7]) have addressed heterogeneity and dynamism to a certain extent by separating interface from implementation (e.g., interfaces defined using IDL in CORBA, and using SIDL in CCA) and separating interaction/coordination from computation (e.g., stub/skeleton and DII/DSI in CORBA, and InfoBus in JavaBeans). However, these frameworks do not address the context-aware self-management of individual components/beans. Further, the interaction patterns (i.e., RMI, message sending, shared space) among components/beans have to be defined a priori and cannot be changed at runtime.

Service-based models (e.g., Web service and Grid service [3] models) have been proposed in recent years to address the requirements of loosely coupled wide-area distributed environments. These models require very little or no prior knowledge of the service before invocation. The decoupling between application entities provided by these models allows applications to be constructed in a more flexible and extensible way. However, the runtime behaviors of services and applications themselves are still rigid and they implicitly assume that context does not change during the lifetime of applications, i.e., services can only be customized during their instantiation. Further, services in the Web services model are assumed to be stateless. While the Grid service model allows stateful services, it makes strong assumptions about the underlying system, i.e., it must support reliable invocation, which is not possible in the presence of failures and the lack of global knowledge. Current orchestration and choreography for Web and Grid services are static and must be defined a priori.

2.3. Accord Programming Framework

The challenges and requirements outlined above, and the limitations of current programming frameworks, have led researchers to investigate an alternative approach, i.e., autonomic computing, to develop applications that are capable of managing themselves using high-level rules with minimal human intervention.

The Accord programming framework presented in this paper supports the development of autonomic Grid applications that can address the challenges described above. Accord enables the definition of autonomic components with programmable behaviors and interactions. Further, it enables runtime composition and autonomic management of these components using dynamically defined rules.

3. The Accord Programming Framework for Autonomic Applications

The Accord programming framework consists of 4 concepts. The first is an application context that defines a common semantic basis for the application. The second is the definition of autonomic components as the building blocks of autonomic applications. The next is the definition of rules and mechanisms for the dynamic composition of autonomic components. And the final is an agent infrastructure to support rule enforcement to realize self-managing and dynamic composition behaviors. Accord builds on the AutoMate [2] middleware infrastructure that provides the essential services required to support the development and execution of autonomic applications. These include naming service, discovery service, lifecycle management service, and registration service.

3.1. Defining Application Context

Autonomic components should agree on a common syntax and semantics for defining and describing ontologies, namespaces, sensors, actuators, function interfaces and/or events to enable components to understand and interact with each other. Using such a common context allows definition of rules for autonomic management of components and dynamic composition and interactions between components. As Accord builds on and extends existing frameworks with autonomic capabilities, it uses the mechanisms provided by these frameworks to define application context. Current implementations of Accord extend CCA [1] and OGSA [3], and use SIDL and WSDL respectively to define functional interfaces, sensors and actuators. Further, these functional interfaces, sensors and actuators are used to define if-then-else rules that specify a component's runtime behaviors and its interaction relationships with other components.

3.2. Defining Autonomic Components

An autonomic component is the fundamental building block for autonomic applications in the Accord framework. It extends traditional components [1, 7] to define a self-contained modular software unit of composition with specified interfaces and explicit context dependencies. Additionally, an autonomic component encapsulates rules, constraints and mechanisms for self-management, and dynamically interacts with other autonomic components. The structure of an autonomic component is shown in Figure 1. It is defined by three classes of ports:

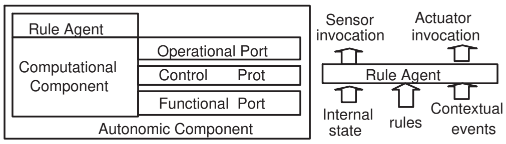

Figure 1. An Autonomic component

1. The **functional port** (Γ) defines a set of functionalities γ provided and used by the autonomic component. $\gamma \in \Omega \times \Lambda$, where Ω is the set of inputs and Λ is the set of outputs of the components, and γ defines a valid input-output set.
2. The **control port** (Σ) is the set of tuples (σ, ξ), where σ is a set of sensors and actuators exported by the component, and ξ is the constraint set that controls access to the sensors/actuators. Sensors are interfaces that provide information about the component while actuator are interfaces for modifying the state of the component. Constraints are based on state, context and/or high-level access polices, and can control who invokes the interface, when and how they are invoked.
3. The **operational port** (Θ) defines the interfaces to formulate, inject and manage rules, and encapsulates a set of rules that are used to manage the runtime behaviors of the autonomic component. Rules incorporate high-level guidance and practical human knowledge in the form of conditional if-then expressions, i.e., *IF condition THEN actions*. *Condition* is a logical combination of component (and environment) sensors and events. *Actions* consist of a sequence of invocations of components and/or system sensors/actuators, and other interfaces. A rule fires when its condition expression evaluates to be true which causes the corresponding actions to be executed. Each rule is assigned a priority level,

which is used to manage multiple firings and resolve conflicts [6]. Two types of rules are defined.

- *Behavior rules* control the runtime functional behaviors of autonomic components and applications. For example, behavior rules can control the algorithms, data representations or input/output formats used by a component and an application.
- *Interaction rules* control the interactions between components, between components and their environments, and the coordination within an autonomic application. For example, an interaction rule may define where a component will get inputs and forward outputs, define the communication mechanisms used, and specify when the component interacts with other components.

Each autonomic component also embeds a **rule agent** (see Figure 1) that is delegated to manage its execution. The rule agent monitors the state of the component and its context, and controls the firing of rules. Note that rule agents cooperate to fulfill application objectives as described in the following subsections.

3.3. Dynamic Composition of Autonomic Components

3.3.1. Definition of Dynamic Composition. The composition of autonomic components consists of defining an *organization of components* and the *interactions among these components*. The organization of components is based on the composition of functional ports (Γ), and can be defined as:

$C_0 \propto_\Gamma \bigcup C_i, \exists \Gamma_{C_0,u} \subseteq \bigcup \Gamma_{C_i,p}$

where, C_0 is an autonomic component, $\bigcup C_i$ is a set of one or more autonomic components, $\propto_\Gamma$ denotes the relation "be functionally composeable with", $\Gamma_{C_0,u}$ is the functions used by component C_0, and $\bigcup \Gamma_{C_i,p}$ represents the functions provided by the component set $\bigcup C_i$. This definition says that component C_0 is functionally composeable with components $\bigcup C_i$, when $\bigcup C_i$ can provide all the functions required by C_0. This is similar to the composition defined by component-based frameworks such as the CORBA [7] and Web services.

Interactions among components defines how and when components interact, the interaction mechanism (messaging, shared-memory, tuple-space) and coordination model (data-driven or control-driven). For example, Ccaffeine [1] defines interactions as function calls, CORBA [7] uses remote method invocations, and Web services and Grid services [3] communicate using XML messages. Interactions may be triggered by an event or may be actively initiated by a component.

Dynamic composition introduces dynamism and uncertainty into both aspects of composition described above, i.e., "which components are composed" and "how and when they interact" are defined only at runtime. Compositions are often represented as workflow graphs where nodes represent components and edges represent interaction relationships between the components. Using such a workflow graph representation of composition, dynamic composition consists of (a) node (component) dynamism - components are replaced, added to or deleted from the workflow, and (b) edge (interaction) dynamism - interaction relationships are changed, added to or deleted from the workflow.

3.3.2. Dynamic Composition in Accord. In Accord, dynamic composition is performed by a multi-agent system [5] consisting of peer rule agents embedded within autonomic components, and a composition agent.

Programmers submit the primary application workflow to the composition agent, which decomposes the workflow into interaction rules. This decomposition process consists of mapping control statements in the workflow into corresponding rule templates, and defining the required parameters for the templates. The composition agent injects these interaction rules into corresponding rule agents, which then execute the rules to appropriately configure the components and establish interaction relationships. Note that there is no centrally controlled orchestration. While the interaction rules are defined by the composition agent, the actual interactions are managed by rule agents in a decentralized manner. The Accord framework supports component and interaction dynamism as described below.

- *Dynamically replacing components*: An existing component can be replaced by another component as long as the functional ports of the two components are compatible. The replacement is managed by the composition agent and is achieved as follows. First, the new component is registered in the composition agent using registration service provided by AutoMate, and the old component is notified by the composition agent to transition to a quiescent state. In this state, the old component does not respond to invocations or requests and does not produce any responses. While, it transfers its rule set to the new component and notifies related components to update their interaction rules. The execution of these updated interaction rules will establish the interactions between the new component and those related components. The old component is then deleted, as described in deleting a component in below. If the old component crashes, the replacement process is handled entirely by the composition agent.

  Two tasks are required to enable the transfer of state information. First, the component should expose sensors and actuators to enable its state to be externally queried and modified. Second, rules should be defined to direct the rule agent to periodically query the state of the component and store it at the composition agent.

- *Dynamically adding/deleting components*: To add a new component, the composition agent loads the new component with interaction rules defined by users, and injects corresponding rules to related components. The execution of these rules will establish interactions between the new component and the existing components. To delete a component, the composition agent notifies related components to delete corresponding interaction rules. Once the component is no longer active in this application, it will be terminated by the lifecycle service provided by AutoMate.
- *Establishing/deleting/changing interaction relationships*: Interaction rules will instruct the autonomic components to establish or delete interaction relationships at runtime. The composition agent may inject new rules and modify existing rules, which will be executed by corresponding rule agents to dynamically change the interaction relationships to cope with the dynamism and uncertainty.

The decomposition of the primary application workflow into rules enables users to adjust the workflow at runtime without recompiling/restarting the applications. The interaction relationships are managed and autonomically adapted to the dynamic context by rule agents according to interaction rules. As a result, the applications can be autonomically re-configured to manage the dynamism and uncertainty of the applications and the environment.

4. Autonomic Forest Fire Application: An Illustrative Example

In this section, we use forest fire application [4] to illustrate the Accord programming framework. The application predicts the speed, direction and intensity of the fire front as the fire propagates using static and dynamic environment and vegetation conditions. The application is composed of 5 components listed below.

- *DSM (Data Space Manager)*: The forest is represented as a 2D space composed of cells. The function of *DSM* is to divide the data space into sub spaces based on current system resources using load-balancing algorithms, and to send the divided 2D space to *Rothermel*.
- *CRM (Computational Resource Manager)*: *CRM* provides *DSM* with system resource information, including the number of current available computation resources and their usages.
- *Rothermel*: *Rothermel* generates the processes required to simulate the fire spread on each subspace in parallel. Each subspace consists of a group of adjacent cells. A cell is programmed to undergo state changes from *unburned* to *burning* and finally to *burned* when the fire line propagates through it. The direction and value of maximum fire spread is computed using Rothermel's fire spread model.
- *WindModel*: *WindModel* simulates the wind direction and intensity.
- *GUI*: Experts interact with the above components using the *GUI* component.

DSM partitions the 2D space based on the currently available computational resources detected by *CRM*. *Rothermel* then simulates the fire propagation in this 2D space according to the current wind information obtained from *WindModel*. When the load on computational nodes is unbalanced, *DSM* will re-partition the 2D space and restart the process. The process continues until no *burning* cells remain.

4.1. Defining Autonomic Components

We use the *Rothermel* and *CRM* as examples to illustrate the definition of functional, control and operational ports.

```
(1) Functional port
<function name=``getSpaceState">
    <out name=``space" type=``tns:SpaceDes"/>
</function>
(2) Control port
addSensor(``getDirection", ``string");
/* sensor name is getDirection, return type is string */
addActuator(``setCellState",``cellState", ``string",``void")
/* actuator name is setCellState */
/* the name of input is cellState and type is string */
/* this actuator has no return */
(3) Operation port
IF isMaxUsageDiff() > 0.5 THEN setLoadBalanced(false)
```

Figure 2. Examples of the port definition

Functional Port: *Rothermel* simulates the propagation of the fire in the subspace. An example of its functional port definition is shown in Figure 2. The function *getSpaceState* generates information about the space. The namespace *tns* defines the context of this application and describes the data structures used. For instance, the data structure *tns:SpaceDes* describes the space information for this application, including the direction and value of maximum fire spread, the vegetation type and the terrain type.

Control Port: In *Rothermel*, the sensor *getDirection* is used to get the spread direction of the fire line that has the maximal intensity, and the actuator *setCellState* is used to modify the state of a specified cell. The value of the input parameter *cellState* of the actuator *setCellState* can be one of *burning*, *unburned* or *burned*. This constraint is handled by the implementation of *setCellState*, by either providing no response to an invalid input value or returning an error. If an error is returned, it will be captured by the *Rothermel* rule agent to generate an exception, which is handled by the AutoMate middleware or forwarded to the user. An example of control port is shown in Figure 2.

Operational Port: The operational port contains the rules that are used to manage the runtime behavior of a component. The rules may be defined at runtime and injected into the component, and will be executed by the rule agent embedded in the autonomic component. An example *behavior rule* in *CRM* may be shown in Figure 2. When this rule fires, *CRM* will deduce that the load is unbalanced. Note that the threshold (0.5 in this example) that triggers the rules can be modified at run time.

4.2. Dynamic Composition in the Forest Fire Application

The primary workflow of the forest fire application is decomposed to interaction rules shown in Figure 3 and 4. In

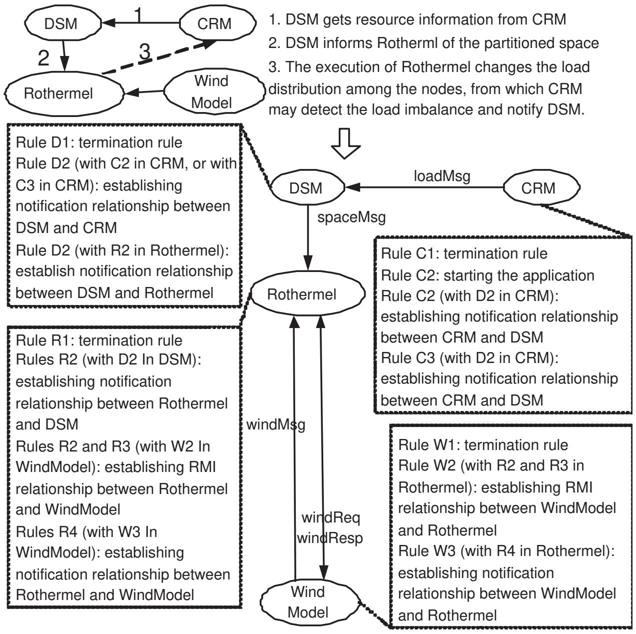

Figure 3. Decomposing the primary workflow for the forest fire application into interaction rules

this example, We illustrate interaction rules (shown in Figure 3 and Figure 4) for:

- Establishing a "while" loop among *Rothermel*, *DSM*, and *CRM*, which will be terminated when there are no *burning* cells in the data space. The loop control flow is established by executing R1, D1 and C1.
- Establishing synchronous RMI between *Rothermel* and *WindModel* by executing R2, R3 and W2. In this interaction, *Rothermel* will be blocked until it receives a response from *WindModel*.
- Establishing notification relationship between (1) *CRM* and *DSM* by executing C2 and D2, or C3 and D2, (2) *DSM* and *Rothermel* by executing D2 and R2, and (3) *WindModel* and *Rothermel* by executing W3 and R4.

```
----------Rules for Rothermel----------
R1: IF getNumOfBurningCells()==0 THEN
        send terminationMsg to DSM, WindModel, CRM; invoke stop;
R2: IF spaceMsg is received THEN assign spaceMsg to input;
        invoke setSpace with input;
        send windReq to getWindInfor in WindModel; block;
R3: IF windResp is received THEN nonblock; assign windMsg to input;
        invoke simulate with input;
R4: IF windMsg is received THEN assign windMsg to input; invoke simulate with input;

----------Rules for DSM----------
D1: IF terminationMsg is received THEN invoke stop;
D2: IF loadMsg is received THEN assign loadMsg to input;
        invoke partition with input to output;
        assign output to spaceMsg; send spaceMsg to Rothermel;

----------Rules for CRM----------
C1: IF terminationMsg is received THEN invoke stop;
C2: IF startSignal is received THEN send loadMsg to DSM;
C3: IF isResourceBalanced()==false THEN send loadMsg to DSM;

----------Rules for WindModel----------
W1: IF terminationMsg is received THEN  invoke stop;
W2: IF windReq is received THEN assign windReq to input; invoke getWindInfor to output;
        assign output to windResp; send windResp to Rothermel;
W3: IF isWindChanged()==true THEN invoke getWindInfor to output;
        assign output to windMsg; send windMsg to Rothermel;
```

Figure 4. Interaction rules

The Accord programming framework decouples interaction and coordination from computation, and enables both these behaviors to be managed at runtime using rules. This enables autonomic components to change their behaviors, and to dynamically establish/terminate/change interaction relationships with other components. Users are responsible for the correctness of rules, however, Accord resolves runtime rule conflicts using a priority and dynamic locking mechanism [6]. Conflicting rules with low priority are forced to wait for locks held by rules with high priority. Deploying and executing rules does impact performance, however, it increases the robustness of the applications and their ability to manage dynamism and uncertainty. 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. An evaluation of performance overheads is presented in Section 5.

4.3. Self-managing Behaviors For the Forest Fire Application

The self-managing behaviors for the forest fire application enabled by the Accord programming framework are illustrated bellow.

Examples of Autonomic Behaviors: Autonomic behaviors are achieved using simple or compound behavior rules.

- *Simple behavior rules*: These rules affect an individual component. For example, *DSM* has 2 partitioning algorithms: a *greedyBlockAlgorithm*, which is fast but consumes more resources, and a *graphAlgorithm*, which is slow but needs less resources. *DSM* needs to dynamically select an appropriate algorithm based on current system state. The behavior rule is shown in Figure 5.

```
IF isSystemOverLoaded()==true THEN invoke graphAlgorith();
                               ELSE invoke greedyBlockAlgorithm();
```

Figure 5. An example of a simple behavior rule for *DSM*

- *Compound behavior rules*: These rules may affect several components and need to be executed collaboratively by corresponding rule agents embedded within these components. For example, a rule is defined to notify the user when the fire is propagating towards an important building located at cell X. This rule is decomposed by the *Rothermel* rule agent based on the sensors, actuators and component names contained in the rule, shown in Figure 6. The generated sub rules are injected into the corresponding rule agents in the components that expose these sensors and actuators. The *Rothermel* rule agent will collect *windNotif* from *WindModel*, evaluate the rule and notify the GUI when the condition is true.

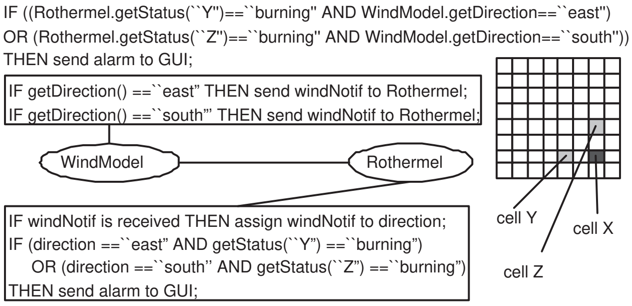

Figure 6. The execution of a compound behavior rule

Examples of Autonomic Interactions

- Adding new components: A new component, *Fire Fighter Model*, which models the behaviors of the fire fighters may be added into the primary workflow. This component dynamically changes the state of cells that it is associated with and informs *Rothermel*. The interaction behavior of the *Fire Fighter Model* is defined by Rule1, and the responding interaction behavior of the *Rothermel* is defined by Rule2, shown in Figure 7.

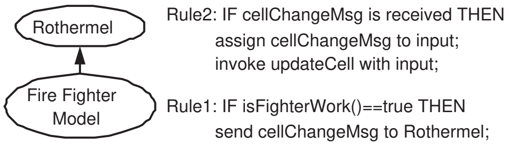

Figure 7. A new component *Fire Fighter Model* is added.

- Changing interaction relationships: *CRM* needs to dynamically decrease the frequency of notifications to *DSM* when the communication network is congested. This self-adapting behavior can be achieved by the combination of a behavior rule Rule1 and an interaction rule Rule2 injected to *CRM*, as shown in Figure 8. Rule1 increases the threshold value to 0.5 when the network is congested. When the maximal difference in resource usages among the nodes is larger than the threshold, *CRM* will set *isResourceBalanced* to return false. When the load is imbalanced, Rule2 will be triggered and will send the *loadMsg* to *DSM*. Note that, once the rules, Rule1 and Rule2 in this example, have been defined, the changes of interactions occur in an automatical manner without human intervention. Further, this change is local to the components involved,

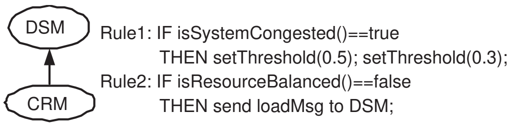

Figure 8. The interaction relationship between *CRM* and *DSM* is changed.

CRM and *DSM* in the example above, and does not affect other components.

5. Prototype implementation, evaluation, and current status

The key concepts underlying the Accord programming framework have been prototyped and evaluated in the context of distributed scientific/engineering simulations as part of the DIOS++/Discover project [6]. In this prototype, computational objects were enhanced with sensors, actuators and behavior rules. The overheads associated with dynamic injection and runtime execution of these rules were evaluated. Note that the objects could be partitioned across multiple processors and could be dynamically created, deleted or migrated. Rules could span multiple object across multiple processors. This prototype however did not implement dynamic composition.

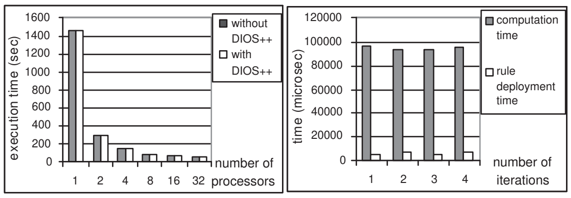

Figure 9. Left: Runtime overheads introduced in the minimal mode. Right: Comparison of computation and rule deployment time.

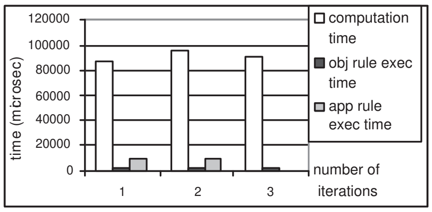

Figure 10. Comparison of computation and rule execution time.

The evaluation of the prototype is presented in Figure 9 and 10. It was conducted on a 32 node Linux cluster using an oil-reservoir application. The left plot in Figure 9 shows the runtime overhead due to the introduction of sensors, actuators and rules into computational objects. No rules were evaluated in this experiment. The right plot in Figure 9 compares the rule deployment time to the computational time for successive iterations. Figure 10 compares rule execution time to the computational time for successive iterations. An object rule is a rule that manages a single object while an application rule manages a set of objects across multiple processors. It can be seen from these experiments that the overheads are quite tolerable.

6. Summary and Conclusion

As the scale, complexity, heterogeneity and dynamism of Grid environments and applications increase, conventional paradigms based on passive components and static compositions quickly become insufficient. This has led researchers to consider alternative autonomic approaches, where applications are context aware and self-managing. In this paper we presented the Accord programming framework to enable the development and management of autonomic applications in pervasive information and computational Grids. The Accord builds on the separation of composition (configuration, interaction and coordination) aspect from computation behaviors provided by component/service based models, and extends these models to enable both components and compositions dynamically managed using runtime injected rules. The programming framework is illustrated using an autonomic forest fire application. Further, an experimental evaluation of the overheads associated with Accord using a prototype implementation was presented.

References

[1] Common component architecture tutorial, http://acts.nersc.gov/events/workshop2003/slides/cca/.

[2] M. Agarwal and et al. Automate: Enabling autonomic applications on the grid. In *Proceedings of the Autonomic Computing Workshop*, Seattle, WA, 2003.

[3] I. Foster and et al. The physiology of the grid: An open grid services architecture for distributed systems integration. Technical report, Open Grid Service Infrastructure WG, Global Grid Forum, 2002.

[4] B. Khargharia and et al. vgrid: A framework for building autonomic applications. In *Proceedings of the 1st International Workshop on Heterogeneous and Adaptive Computing–CLADE 2003*, Seattle, Washington, 2003.

[5] 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.

[6] 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)*, Klagenfurt, Austria, 2003.

[7] C. Szyperski and et al. *Component Software Beyond Object-Oriented Programming*. Component Software Series. Addison-Wesley, Great Britain, 2 edition, 2002.