Accord: A Programming Framework for Autonomic Applications

Hua Liu and Manish Parashar, *Senior Member, IEEE*

***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 programming framework that extends existing programming models/frameworks to support the development of autonomic self-managed applications. The framework enables the development of autonomic elements and the formulation of autonomic applications as the dynamic composition of autonomic elements. The operation of the proposed framework is illustrated using a forest fire management application.**

***Index Terms*—Programming framework, self-management, autonomic, dynamic composition**

I. Introduction

THE emergence of 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, 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 distributed 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 [15], aims at realizing computing systems and applications capable of managing themselves with minimal human intervention.

In this paper we present the Accord programming framework that extends existing programming models/frameworks to support the development of autonomic applications in wide-area distributed environments. The framework builds on the separation of the composition aspects (e.g., organization, interaction and coordination) of elements from their computational behaviors that underlies the component- and service-based paradigm, and extends it to enable the computational behaviors of objects/components/services as well as their organizations, interactions and coordination to be managed

Manuscript received January, 2005. 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.

Hua Liu (e-mail: marialiu@caip.rutgers.edu), TASSL, Dept. of Electrical and Computer Engineering, Rutgers Univ., Piscataway, NJ 08854.

Contact author: Manish Parashar (e-mail: parashar@caip.rutgers.edu), TASSL, Dept. of Electrical and Computer Engineering, Rutgers Univ., Piscataway, NJ 08854. Phone: (732) 445-5388. Fax: (732) 445-0593

at runtime using high-level rules.

Accord is part of project AutoMate [2], which investigates autonomic solutions to deal with the challenges of complexity, dynamism, heterogeneity and uncertainty in Grid environments. The overall goal of project AutoMate is to develop conceptual models and implementation architectures that can enable the development and execution of such self-managing Grid applications.

The rest of the paper is organized as follows: Section II investigates programming challenges, discusses related work, and introduces the Accord programming framework. A detailed discussion of the Accord framework is presented in Section III. In Section IV, we illustrate the operation of Accord framework using a forest fire management application. Section V describes the prototype implementations and presents an experimental evaluation of Accord. Section VI presents a conclusion.

II. Application Development and Management in Pervasive Grid Environments

A. Challenges and Requirements

The nature and scale of pervasive information and computational Grid environments and applications introduce new levels of development and management complexities. These include:

- **Heterogeneity**: The 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 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 these environments 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 the scale and complexity of systems/applications increase, and (3) incomplete knowledge, which is typical in large decentralized and asynchronous distributed environments.

The above challenges impose requirements on programming frameworks to enable applications that can address the challenges. This section studies existing programming models and frameworks and their ability to address the challenges listed above. It then, introduces autonomic computing as means for addressing these challenges and presents related autonomic computing systems. Finally, the Accord programming framework is introduced.

B. Programming Frameworks for Distributed System

There has been a significant body of research on programming frameworks for parallel and distributed computing over the last few decades. Many current **communication frameworks for distributed and parallel computing** (i.e., message passing models and shared memory models) supplement existing sequential programming systems to support interactions between distributed entities. These systems typically make very strong assumptions about the behavior of the entities, their interactions, and the underlying system, especially about their static nature and reliable behaviors, which limit their applicability to highly dynamic and uncertain computing environments.

Distributed Object Frameworks: Unlike the systems described above that essentially address only communication aspects, the distributed object frameworks provide more complete support for parallel/distributed applications, including lifecycle management, location and discovery, interaction and synchronization, security, failure and reliability [8]. CORBA [9], one of the dominant distributed object models, enables the secure interactions (based on remote procedure calls, method invocations and event notification) between distributed and heterogeneous objects using interfaces described by a language-neutral interface definition language and through a middleware consisting of object resource brokers and interoperability protocols (e.g., GIOP, IIOP). CORBA primarily addresses distribution and heterogeneity. CORBA also provides limited support for dynamism via dynamic invocation (DSI/DII) and late binding, which enables customization at deployment time. However, the interacting objects and interaction are tightly coupled. Further, the model assumes a priori (compile-time) knowledge of the syntax and semantics of interfaces as well as the interactions required by the applications.

While CORBA does not directly enable dynamic adaptation of the behaviors of objects or their interactions, it does have the potential to support adaptive runtime behaviors by providing portable request interceptors that "intercept the flow of a request/reply sequence through the ORB at specific points so that services can query the request information and manipulate the service contexts that are propagated between clients and servers" [9]. Such extensions are discussed in Section C. Note that these adaptations are performed by manipulating and redirecting messages using interceptors, but the direct adaptation of individual objects is not provided.

Component-based programming frameworks: Component models address increasing software complexity and changing requirements by enabling the construction of

systems as assemblies of reusable components. Components are reusable units of composition, deployment and execution and lifecycle management [7]. Components are completely specified by their interfaces. Current component models include CCM, JavaBeans and CCA.

CORBA Component Model (CCM) [9] extends the CORBA distributed object model and similarly supports distribution, heterogeneity and security. It also supports dynamic instantiation and runtime customization of components. However, CCM inherits some of the limitations of CORBA including the requirement for a prior knowledge about interfaces and interactions. JavaBeans [7] is a Java only component model which addresses similar issues. JavaBeans also supports runtime bean customization.

The Common Component Architecture (CCA) [1] defines a component model especially for scientific applications. The model primarily addresses the heterogeneity and the separation of interface and implementation. CCA targets high-performance parallel applications and uses functional calls for inter-component interactions. While it does not support runtime customization of components, it does allow components to be replaced dynamically. It does not address failure or security and assumes all components are trusted.

Note that component-based frameworks also provide core mechanisms, such as interceptors in CORBA, the BuilderService in CCA, and the container in JavaBeans, which can be extended to support dynamic runtime adaptation. However the communication pattern between components and their coordination are statically defined.

Service-based models: Service based models, such as 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 services 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.

C. Autonomic Computing Systems

Addressing these challenges requires applications to be autonomic. The essence of autonomic computing is self-management, which is manifested in four aspects, self-configuration, self-optimization, self-healing, and self-protection [15]. Therefore, applications should be capable of detecting and dynamically responding to changes in both the state of the system and the requirements of the applications, to dynamically configure themselves, continuously improve their performance, detect, diagnose, and repair problems, and defend themselves against attacks. This imposes key requirements on the programming systems: (1) the applications should be composed from discrete, self-managing elements which incorporate separate specifications for all of 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 composeable; (3) The interface definitions of these elements should be separated from their implementations to enable dynamic selection of elements and interactions among heterogeneous elements.

Given these features of a programming system, an autonomic 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 underlying environments.

The challenges and requirements outlined above and the limitations of current programming frameworks have led researchers to investigate alternate approaches that enable the development of applications that are capable of managing themselves using high-level rules, with minimal human intervention. These autonomic applications are context aware and self-adapting.

Existing research in autonomic application system can be divided into two categories. Systems in the first category either extend existing programming languages/systems (for example [10]) or defining new adaptation languages (for example [11]), and enable adaptive applications where the adaptations are statically specified at compile time. These systems require that all the possible adaptation must be known a priori and must be coded into the application. If new adaptations are required or if application requirements change, the application code has to be modified and the application is re-compiled.

Systems in the second category enable dynamically-defined adaptation by allowing adaptations, in the form of code, scripts or rules, to be added, removed and modified at runtime. Many existing projects in this category directly utilize and extend the capabilities of existing programming frameworks to enable dynamic adaptation. For example, ACT [12] extends CORBA by using a rule-based interceptor to dynamically weave new adaptive code into the ORB as applications execute. Other projects investigate specific coordination languages to describe/adapt the

interactions between elements. For example, ALua [13] uses the Lua language to perform interaction/coordination and adaptation in an interpretive manner and supports the execution of dynamically defined adaptation specification in an even-driven manner. In both these types of projects, the mechanisms of adaptations include (1) interposition — filters [14, 17] or proxies [12, 18] may be interposed between interacting elements to change their interaction relationships and to introduce dynamism to the execution of an application, (2) wrapping [19] — the interactions may be refined at runtime using wrappers to introduce new behaviors into existing elements, and (3) superimposition [20] — it enables the software engineers to impose pre-defined but configurable types of functionalities on individual elements.

D. Accord Programming Framework

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

The prototype implementations of Accord extend an object oriented framework based on C++ and MPI, and the CCAFFEINE CCA Framework. These implementations and their evaluation are presented in this paper.

III. 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 elements (objects, components, services) as the building blocks of autonomic applications. The next is the definition of rules and mechanisms for the dynamic composition of autonomic elements. And the final is an agent infrastructure to support rule enforcement to realize self-managing and dynamic composition behaviors. Accord builds on the AutoMate 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.

A. Defining Application Context

Autonomic elements should agree on a common syntax and semantics for defining and describing ontologies, namespaces, sensors, actuators, function interfaces and/or events to enable elements to understand and interact with each other. Using such a common context allows definition of rules for autonomic management of elements and dynamic composition and interactions between elements. 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 an element's runtime behaviors and its interaction relationships with other elements.

B. Defining Autonomic Elements

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

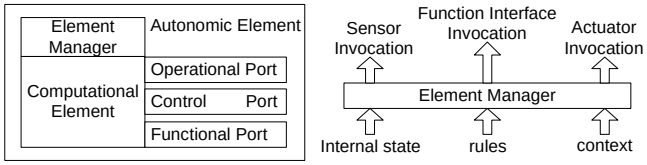

Fig. 1. An autonomic element.

1. The **functional port** (Γ) defines a set of functionalities γ provided and used by the autonomic element. $\gamma \in \Omega \times \Lambda$, where Ω is the set of inputs and Λ is the set of outputs of the elements, 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 element, and ξ is the constraint set that controls access to the sensors/actuators. Sensors are interfaces that provide information about the element while actuators are interfaces for modifying the state of the element. Constraints are based on state, context and/or high-level access polices, and can control who invokes the interfaces, 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 element. 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 element (and environment) sensors and events. *Actions* consist of a sequence of invocations of elements 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. Two types of rules are defined.
   - *Behavior rules* control the runtime functional behaviors of autonomic elements and applications. For example, behavior rules can control the

algorithms, data representations or input/output formats used by an element and an application.

- *Interaction rules* control the interactions between elements, between elements and their environments, and the coordination within an autonomic application. For example, an interaction rule may define where an element gets inputs and forwards outputs, define the communication mechanisms used, and specify when the element interacts with other elements.

As shown in Figure 1, each computational element is associated with an element manager that is delegated to manage its execution. The element manager monitors the state of the element and its context, and controls the firing of rules. Further, element managers cooperate to fulfill application objectives as described in the following subsections.

Note that computational elements have to implement and export appropriate "sensor" and "actuator" interfaces so that their behaviors can be monitored and controlled. Adding sensors requires modification/instrumentation of the element source code. In case of third-party and legacy elements where such a modification may not be possible or feasible, proxies [12, 18] can be used to collect relevant element information. The element manager implements the proxy functions and is interposed between the caller and callee elements to monitor, for example, all the method invocations for the callee. Actuators can be similarly implemented either as new methods that modify internal parameters and behaviors of an element, or defined in terms of existing methods if the element cannot be modified. The adaptability of the elements will be limited in the latter case.

C. RULE EXECUTION MODEL

A three-phase rule execution model [24] is used to ensure correct and efficient parallel rule execution. This model provides mechanisms to dynamically detect and handle rule conflicts for both, behavior and interaction rules.

Rule execution proceeds as follows. After the evaluation, a pre-condition is constructed. Rule conflicts are detected at runtime when rule execution changes the pre-condition (a sensor-actuator conflict), or the same actuator will be invoked multiple times with different values (an actuator-actuator conflict). Sensor-actuator conflicts are resolved by disabling the rules that change the pre-condition. Actuator-actuator conflicts are resolved by relaxing the pre-condition according to user-defined strategies until no actuator is invoked multiple times with different values.

For example, consider element C1 with 3 algorithms: algorithm 1 has better cache performance but consumes a large communication bandwidth, algorithm 2 has comparatively more cache misses but only consumes a small bandwidth, and algorithm 3 demonstrates an acceptable cache miss and communication delay but has lower precision. It is possible that under certain conditions, rule evaluation may results in the selection of algorithm 1 and 2 at the same time to simultaneously decrease cache misses and communication delay, and maintain high-precision. This conflict is detected and resolved by relaxing the high-precision requirement, and therefore algorithm 3 can be selected.

The Accord framework also provides mechanisms for reconciliation among manager instances, which is required to ensure consistent adaptations. For example, in parallel SCMD (Single Component Multiple Data) applications, since each processing node may independently propose different and possible conflicting adaptation behaviors based on its local state and execution context. Rules are statically assigned one of two priorities. A high priority means that the execution of the rule is necessary, for example, to avoid an application crash. A low priority means that the execution of the rule is optional. During reconciliation, actions associated with the rule with high priority are propagated to all the managers. If there are multiple high priority rules, a runtime error is generated and reported to the user. If only low priority rules involved, reconciliation uses cost functions to select the most appropriate action at all involved managers. Details of the design and operation of the Accord rule engine can be found in [24].

D. Dynamic Composition of Autonomic Elements

1) Definition of Dynamic Composition

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

$$C_0 \propto_\Gamma \bigcup C_i, \exists \Gamma_{c_0,u} \subseteq \Gamma_{c_i,p}$$

Where, C_0 is an autonomic element, $\bigcup C_i$ is a set of one or more autonomic elements, $\propto_\Gamma$ denotes the relation "be functionally composeable with", $\Gamma_{c_0,u}$ is the functions used by element C_0, and $\Gamma_{c_i,p}$ represents the functions provided by the element set $\bigcup C_i$. This definition says that element C_0 is functionally composeable with elements $\bigcup C_i$, when $\bigcup C_i$ can provide all the functions required by C_0. This is similar to the composition defined by distributed object frameworks such as CORBA [7] and service-based models such as Web Services [3].

Interactions among elements define how and when elements 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 an element.

Dynamic composition introduces dynamism and uncertainty into both aspects of composition described above, i.e., "which elements are composed" and "how and when they interact" are defined only at runtime. Compositions are often represented as workflow graphs where nodes represent elements and edges represent interaction relationships between the elements. Using such a workflow graph representation of composition, dynamic composition consists of (a) node (element) dynamism – elements 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.

2) Dynamic Composition in Accord

In Accord, dynamic composition is performed by a multi-agent infrastructure [5] consisting of peer element managers associated with computational elements, and a composition manager, as shown in Figure 2.

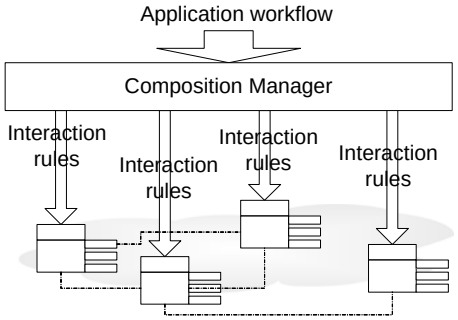

Fig. 2. Dynamic composition in Accord.

Programmers submit the primary application workflow to the composition manager, which decomposes the workflow into interaction rules. This decomposition process consists of mapping workflow patterns [25] in the workflow into corresponding *rule templates*, and defining the required parameters for the templates [26]. The composition manager injects these interaction rules into corresponding element managers, which then execute the rules to appropriately configure the elements and establish interaction relationships. Note that there is no centrally controlled orchestration. While the interaction rules are defined by the composition manager, the actual interactions are managed by element managers in a decentralized manner.

The Accord framework supports element and interaction dynamism as described below.

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

  Two tasks are required to enable the transfer of state information. First, the element should expose sensors and actuators to enable its state to be externally queried and modified. Second, rules should be defined to direct the element manager to periodically query and store the state of the element.
- *Dynamically adding / deleting elements*: To add a new element, the composition manager creates a new element manager, initializes it with interaction rules defined by users, and injects corresponding rules into managers of related elements. The execution of these rules will establish interactions between the new element and the existing elements. To delete an element, the composition manager notifies related element managers to delete corresponding interaction rules. Once the element is no longer active in this application, it will be terminated by the lifecycle service provided by AutoMate or the underlying framework.
- *Establishing / deleting / changing interaction relationships*: Interaction rules will instruct the autonomic elements to establish or delete interaction relationships at runtime. The composition manager may inject new rules and modify existing rules, which will be executed by corresponding element managers to dynamically change the interaction relationships to cope with the dynamism and uncertainty of applications and systems.

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 automatically adapted to the dynamic context by element managers according to interaction rules. As a result, applications can be automatically re-configured to manage the dynamism and uncertainty of the applications and environments.

E. Accord Implementation Issues

The Accord framework 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, it assumes time-asynchronous system behavior with fail-stop failure modes [23]. Finally, Accord assumes the existence of an execution environment that provides (1) an agent-based control network, (2) support for associative coordination, (3) services for content-based discovery and messaging, (4) support of context-based access control and (5) core services for managing distributed computing environment. These requirements are addressed respectively by Rudder, Meteor, Sesame/DAIS and the underlying Grid middleware on which it builds.

F. Enabling Self-managing Behaviors using Accord

In this section we demonstrate how Accord can be used to enable autonomic self-managing behaviors in applications by dynamically changing the computational behaviors of individual elements, by changing their interaction relationships, and by adding/deleting/replacing elements. All the behaviors described below are defined by experts using rules, which may be specified at compile time and/or dynamically specified at runtime. Note that many these behaviors have been implemented and demonstrated, and are described in the publications cited [6, 16].

Self-configuration: Autonomic applications can dynamically configure themselves in accordance with high-level functional and nonfunctional requirements in a dynamical execution environment. At the element level, element managers sense the current execution context of the managed elements using underlying context services (e.g. NWS), and customize their computational behaviors (e.g., select the algorithms or data representatives) and their interaction behaviors (e.g., negotiate with related element managers to construct the appropriate interaction relationship). For example, if a user is working on a PDA that typically has poor graphics resolution as well as limited memory capacity, a visualization element may dynamically adapt its behavior to the current display capacities [16]. If two interacting elements detect that the current communication channel is congested, they may negotiate to decrease the interaction frequency within some tolerance to reduce congestion.

At the application level, when an element is instantiated within an application, it will integrate itself seamlessly, and the rest of the application will adapt to its presence. For example, in the forest fire application described in this paper, a new element is dynamically introduced into the application to simulate the fire fighters. When it is introduced, the element registers itself with the composition manger, which then injects corresponding interaction rules into the new element and related existing elements. This enables the elements to establish interaction relationship and the application to continue without interruption.

Self-optimizing: Elements and applications continually seek opportunities to improve their own performance and efficiency. At the element level, for example, if minimizing execution time is specified as an optimization objective, an individual element will select the fastest algorithm for the current execution context, application state and its inputs. Alternately, if decreasing memory usage is the optimization objective, the element will select the algorithm that requires the minimal core memory for current execution context. These optimizations are specified by experts in the form of rules, possibly at runtime based on observed behaviors.

At the application level, the composition manager may decide to replace an element with a new version that offers better performance for the current execution context. The new version must provide all the active functions as the one being replaced (active functions are those functions that are currently used by interacting elements in the application) so that the replacement is transparent to interacting elements.

Self-healing: Systems automatically detect, diagnose, and repair problems. For example, in a scientific simulation, if a solver is detected not to converge, an alternate, possibly less accurate solver may be used to prevent the application from failing. This can be achieved in two ways. If a managed element provides multiple solvers, it may switch the solvers based on current state and execution context. However, if the solvers are provided by separate elements, the composition manager will select and instantiate the alternative element, and replace the current element with the new one as described previously. Similarly, if an adaptive simulation runs out of memory (i.e., a memory allocation request fails), rather than allowing the application to crash, the application can survive but execute at a lower resolution. Once again, individual elements may adapt their behavior to reduce their core memory requirements, and/or elements may be replaced or moved to other nodes.

Self-protecting: Autonomic systems will defend themselves against problems arising from malicious attacks. They also will anticipate problems and take steps to avoid or mitigate them. In Accord, individual elements contain constraints or guards controlling accesses to their interfaces, and they may disable certain interfaces by modifying these constraints. For example, an element may shut down some of its sensors and actuators when its state is not conducive to the actions of those sensors and actuators. Further, interface constraints in Accord may specify credential checks during invocation and an element may reject invocations from elements that don't have the required credentials. For example, element A may deny access from element B to some critical internal state when element B is in an insecure environment.

Note that self-healing behaviors are used to recover from problems that have occurred, while self-protecting behaviors try to anticipate and avoid problems before they actually happen. Consequently, techniques discussed in self-healing can be used for self-protecting. Self-protecting behaviors in Accord may use the Sesame/DAIS services provided by AutoMate [22].

IV. Autonomic Forest Fire Application: An Illustrative Example

In this section, we use the 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 elements 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 computational 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 maximal fire spread is computed using Rothermel's fire spread model.
- *WindModel*: *WindModel* simulates the wind direction and intensity.
- *GUI*: Experts interact with the above elements using the *GUI* element.

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 continue the process. The process continues until no *burning* cells remain.

A. Defining Autonomic Elements

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);
```

Fig. 3. Examples of port definitions in Accord.

Functional Port: *Rothermel* simulates the propagation of the fire in the subspaces. An example of its functional port definition is shown in Figure 3. 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 maximal 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* in the actuator *setCellState* can be one of *burning*, *unburned*, or *burned*. This constraint is handled by the implementation of *setCellState*, through 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* element manager 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 3.

Operational Port: The operational port contains the rules that are used to manage the runtime behavior of an element. The rules may be defined at runtime and injected into the element, and will be executed by the element manager associated with the computational element. An example *behavior rule* in *CRM* may be shown in Figure 3. 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.

B. Dynamic Composition in the Forest Fire Application

The primary workflow of the forest fire application is decomposed to interaction rules shown in Figure 4 and 5. In this example, we illustrate interaction rules 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 *WindMode*l by executing R2, R3 and W2. In this interaction, *Rotherme*l 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.

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 elements to change their behaviors, and to dynamically establish/terminate/change interaction relationships with other elements. Users are responsible for the correctness of rules. However, Accord resolves runtime

rule conflicts using the three-phase rule execution model [24] described in Section III C. Deploying and executing rules impact performance, but 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 time. An evaluation of performance overheads is presented in Section V.

C. 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 element. For example, DSM has 2 partitioning algorithms: a greedyBlockAlgorithm, which is fast but consumes more resources, and a graphAlgorithm, which is slow but needs fewer resources. DSM needs to dynamically select an appropriate algorithm based on current system state. The behavior rule is shown in Figure 6.

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

Fig. 6. An example of a simple behavior rule for *DSM*..

- *Compound behavior rules*: These rules may affect several elements and need to be executed collaboratively by corresponding element managers associated with these elements. 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* element manager based on the sensors, actuators and element names contained in the rule, shown in Figure 7. The generated sub rules are injected into the corresponding element managers in the elements that expose these sensors and actuators. The *Rothermel* element manager will collect *windNotif* from *WindModel*, evaluate the rule and notify the GUI when the condition is true.

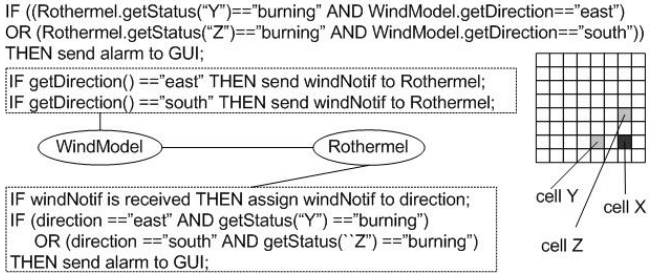

Fig. 7. The execution of a compound behavior rule.

Examples of Autonomic Interactions

- Adding new elements: A new element, *Fire Fighter Model*, which models the behaviors of the fire fighters, may be added into the primary workflow. This element 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 8.

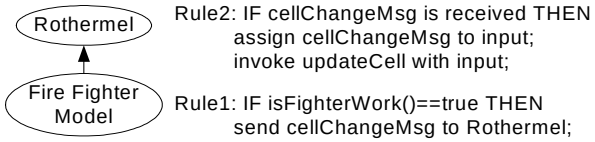

Fig. 8. A new element *Fire Fighter Model* is added.

- 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 9. 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 automatic manner without human intervention. Further, this change is local to the elements involved, *CRM* and *DSM* in the example above, and does not affect other elements.

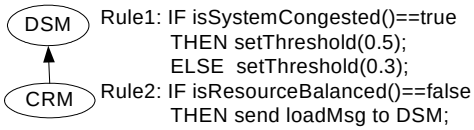

Fig. 9. The interaction relationship between *DSM* and *CRM* is changed.

V. PROTOTYPE IMPLEMENTATION AND EVALUATION

A. Prototype implementation based on a distributed object framework

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]. This prototype implementation was based on a distributed object framework developed using C++ and the Message Passing Interface (MPI). In this prototype, computational objects were enhanced with sensors, actuators, and behavior rules forming their functional, control and operational interfaces. 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 objects across multiple processors. This prototype however did not implement dynamic composition.

Autonomic application based on this implementation included an autonomic oil reservoir and subsurface simulator [21], an autonomic feature-based visualization system [16], and an autonomic runtime management system for adaptive scientific and engineering simulations.

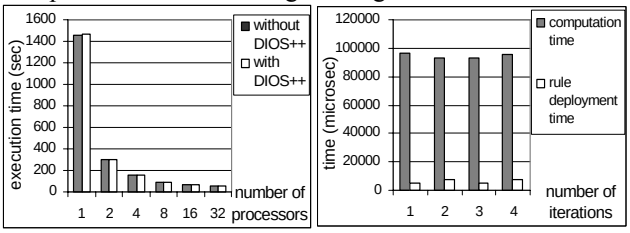

Fig. 10. (a) Runtime overheads introduced in the minimal mode. (b) Comparison of computation and rule deployment times.

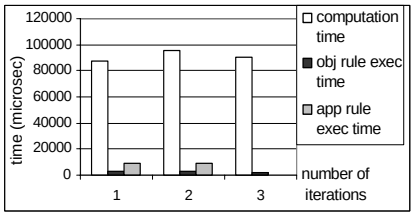

Fig. 11. Comparison of computation and rule execution times.

The evaluation of the prototype is presented in Figure 10 and 11. The experiments were conducted on a 32 node Linux cluster using an oil-reservoir simulation application. The left plot in Figure 10 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 10 compares the rule deployment time to the computational time for successive iterations. Figure 11 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.

B. Prototype implementation based on the CCA component framework

We are implementing a prototype autonomic component framework based on Accord and the CCAFFEINE CCA framework. The framework supports the development and execution of self-managing autonomic scientific applications. It allows CCA components to instantiate and export control ports composed of sensors and actuators. It also introduces two specialized types of components: (1) component manager that monitors and manages the computational behaviors of individual components, for example, by selecting the appropriate algorithms and modifying parameters, and (2) composition manager that manages, adapts, and optimizes the execution of an application at runtime, for example, by dynamically replacing a component that is executing sub-optimally or has failed. The two manager components encapsulate the Accord operational ports.

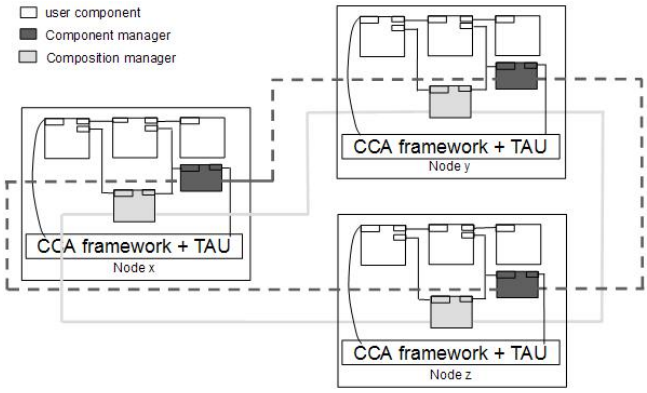

Fig. 12. The architecture of an application using the autonomic component framework based on Accord and CCAFFEINE.

Both, the component manager and the composition manager components are peers of the managed components and other system components, providing and/or using ports that are connected to other ports by the CCAFFEINE framework. The two manager components are not part of the CCAFFEINE framework, and consequently provide the programmers the flexibility to integrate them into their applications only as needed.

A sample application using the Accord and CCAFFEINE based autonomic component framework is illustrated in Figure 12. As shown in the figure, instances of the component managers and composition managers on different nodes independently evaluate and execute the rules to manage and possibly change the computational and interaction behaviors of the managed component instances based on their local state and execution contexts. However, as the CCAFFEINE framework employs the SCMD model of computation, all instances of the managed element must be adapted in exactly the same way, i.e., the corresponding managers must enforce the same actions. As a result, reconciliation (discussed in the rule execution model in section III C) is used by the managers to select actions that are acceptable to all the instances.

VI. SUMMARY AND CONCLUSION

As the scale, complexity, heterogeneity and dynamism of distributed environments and applications increase, conventional paradigms based on passive elements 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 that supports the development of autonomic self-managed applications. It enables the development of autonomic elements and the formulation of autonomic applications as the dynamic composition of autonomic elements, where the runtime computational behavior of the

elements as well as their compositions and interactions can be managed at runtime using dynamically injected rules. Prototype implementations and an evaluation of the framework were also presented. The operation of the proposed framework is illustrated using a forest fire management application.

The programming overhead and requirements of adaptive applications have been widely investigated, both in academia and industry. Component software [7, 9] and service oriented architecture [3, 27] have been proposed as new programming paradigms that separate the development and compilation of elements (components and services) and applications. These paradigms have been widely accepted and integrated with other technologies to enable the programming of adaptive applications. Accord builds on and extends the paradigms with rules/policies to enable adaptations based on both, application/element internal state and execution context.

We believe that extending widely used programming paradigms and using rules and polices to support self-managing behaviors will drive the realization of the vision of autonomic computing, leading to the definition of community wide open standards and their widespread adoption by industry and academia.

REFERENCES

[1] Common Component Architecture Tutorial, http://acts.nersc.gov/events/workshop2003/slides/cca/.

[2] 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 *Proc. of the Autonomic Computing Workshop*, Seattle, WA, 2003, pp. 48-57.

[3] I. Foster, C. Kesselman, J. M. Nick and S. Tuecke, "The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration," Open Grid Service Infrastructure Work Group, Global Grid Forum, Tech. Rep., 2002.

[4] B. Khargharia, S. Hariri, M. Parashar, L. Ntaimo and B. U. Kim, "vGrid: A framework for building autonomic applications," In *Proc. of the 1st International Workshop on Heterogeneous and Adaptive Computing–CLADE 2003*, Seattle, Washington, 2003.

[5] M. Parashar, Z. Li, H. Liu, V. Matossian and C. Schmidt, "Enabling Autonomic Grid Applications: Requirements, Models and Infrastructures," *Self-Star Properties in Complex Information Systems, Lecture Notes in Computer Science*, Springer Verlag. Editors: O. Babaoglu, M. Jelasity, A. Montresor, C. Fetzer, S. Leonardi, A. van Moorsel, and M. van Steen, Vol. 3460, 2005.

[6] H. Liu and M. Parashar, "Rule-based Monitoring and Steering of Distributed Scientific Applications," *International Journal of High Performance Computing and Networking (IJHPCN)*, issue 1, Inderscience, 2005.

[7] C. Szyperski, "Component Software Beyond Object-Oriented Programming," 2nd ed., Component Software Series, Addison-Wesley, Great Britain, 2002.

[8] H. E. Bal, J. G. Steiner and A. S. Tanenbaum, "Programming Languages for Distributed Computing Systems," *ACM Computing Surveys*, 21(3), 1989, 261-322.

[9] "Common Object Broker Resource Architecture (CORBA)," Object Management Group (OMG), http://www.corba.org.

[10] P. Boinot, R. Marlet, J. Noyé, G. Muller, and C. Cosell, "A declarative approach for designing and developing adaptive components," In *Proc. Of the 15th IEEE International Conference on Automated Software Engineering*, pages 111-119, IEEE, 2000.

[11] G. Duzan, J. Loyall, and R. Schantz, "Building adaptive distributed applications with middleware and aspects," In *Proc. of the 3rd International Conference on Aspect-oriented Software Development*, pages 66-73, Lancaster, UK, 2004, ACM.

[12] S. M. Sadjadi, and P. K. McKinley, "ACT: An Adaptive CORBA Template to Support Unanticipated Adaptation," In *Proc. of the 24th International Conference on Distributed Computing Systems (ICDCS'04)*, Hachioji, Tokyo, Japan, pages 74-83, 2004.

[13] C. Ururahy, N. Rodriguez, and R. Ierusalimschy, "ALua: Flexibility for parallel programming," *Computer Languages*, 28(2):155-180, 2002.

[14] M. Aksit and Z. Choukair, "Dynamic, adaptive and reconfigurable systems overview and prospective vision," In *Proc. of the 23rd international conference on distributed computing systems workshops*, pages 84-89, Providence, Rhode Island, 2003, IEEE.

[15] M. Parashar and S. Hariri, "Autonomic Computing: An Overview," *Unconventional Programming Paradigms, Lecture Notes in Computer Science*, Springer Verlag, 2005.

[16] H. Liu, L. Jiang, M. Parashar, and D. Silver, "Rule-based Visualization in the Discover Computational Steering Collaboratory," *the Journal of Future Generation Computer System*, Elsevier Science, Jan 2005.

[17] S. R. Ponnekanti and A. Fox, "Sword: A developer toolkit for building composite web services," http://mortimer.law.uga.edu/jesse/4900/review10 presentation.ppt, 2004.

[18] S. M. Sadjadi and P. K. McKinley, "Transparent self-optimization in existing corba applications," In *Proc. of the first international conference on autonomic computing*, NYC, NY, 2004.

[19] E. Truyen,W. Joosen, P. Verbaeten, and B. N. Jorgensen, "On interaction refinement in middleware," in *Proc. of the 5th International Workshop on Component-Oriented Programming*, 2000.

[20] J. Bosch, "Superimposition: A component adaptation technique," *Information and Software Technology*, 1999.

[21] V. Bhat, V. Matossian, M. Parashar, M. Peszynska, M. Sen, P. Stoffa, and M. F. Wheeler, "Autonomic Oil Reservoir Optimization on the Grid," *Concurrency and Computation: Practice and Experience*, John Wiley and Sons, accepted October 2003.

[22] G. Zhang, and M. Parashar, "Context-aware Dynamic Access Control for Pervasive Applications," In *Proc. of the Communication Networks and Distributed Systems Modeling and Simulation Conference (CNDS 2004)*, San Diego, CA, USA, 2004

[23] F. Cristina, and C. Fetzer, "The Timed Asynchronous Distributed System Model," *IEEE Transactions on Parallel and Distributed Systems, IEEE Computer Society Press*, 10(6), 1999, 642-657.

[24] H. Liu and M. Parashar. "A framework for rule-based autonomic management of parallel scientific applications," In *Proc. Of the 2nd IEEE International Conference on Autonomic Computing (ICAC-05)*, Seattle, Washington, 2005.

[25] W.M.P.Van Der Aalst and et al., "Workflow patterns," *distributed and parallel databases*, 14(3), pages 5-51, 2003.

[26] H. Liu and M. Parashar, "A Component-based Programming Framework for Autonomic Grid Applications," PhD proposal, Rutgers University, 2004.

[27] "Decentralized Orchestration of Composite Web Services," http://www.research.ibm.com/irl/projects/decentralized.shtml.

[28] H. Liu and M. Parashar, "Enabling Self-management of Component-based High-Performance Scientific Applications," In *Proc. of the14th IEEE international symposium on high performance distributed computing (HPDC-14),* Research Triangle Park, NC, 2005.

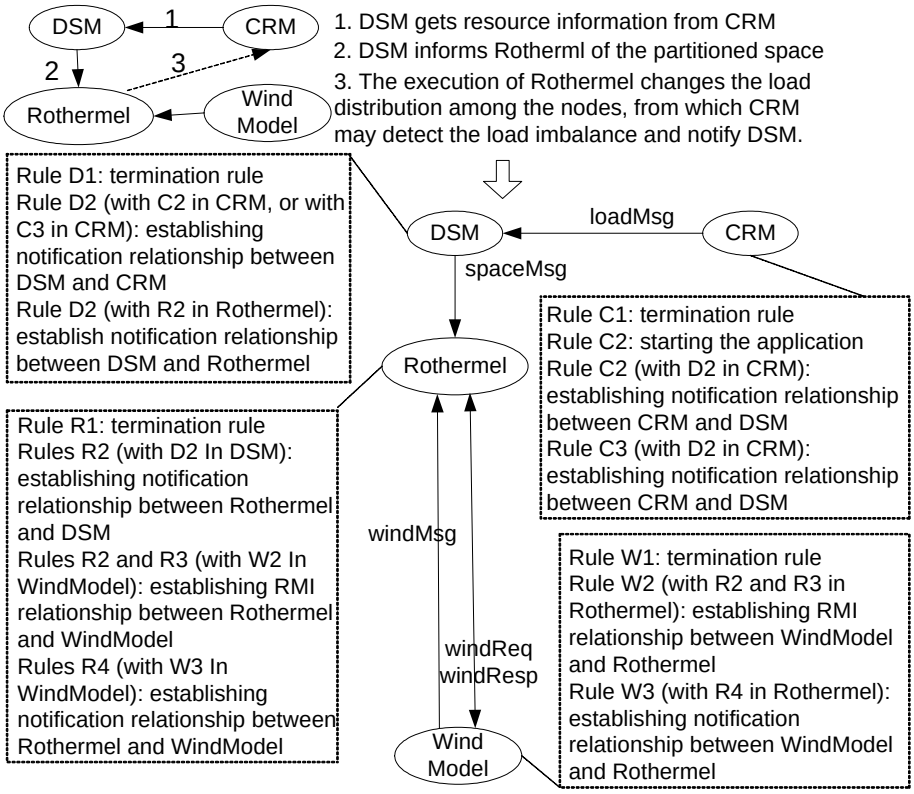

Fig. 4. Decomposing the primary workflow for the forest fire application into interaction rules.

```
----------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;
```

Fig. 5. Interaction rules