ACCORD: A PROGRAMMING SYSTEM FOR AUTONOMIC SELF-MANAGING APPLICATIONS

BY HUA LIU

A Dissertation submitted to the

Graduate School—New Brunswick

Rutgers, The State University of New Jersey

in partial fulfillment of the requirements

for the degree of

Doctor of Philosophy

Graduate Program in Electrical and Computer Engineering

Written under the direction of

Professor Manish Parashar

and approved by

____

____

____

____

____

New Brunswick, New Jersey

October, 2005

ABSTRACT OF THE DISSERTATION

Accord: A Programming System for Autonomic Self-managing Applications

by Hua Liu

Dissertation Director: Professor Manish Parashar

The increasing complexity, heterogeneity, and dynamism of emerging pervasive Grid environments and applications result in significant development and management challenges. This is primarily because application requirements and runtime behaviors depend on the runtime state and execution context and are typically not known a priori. Recently, autonomic computing has proposed solutions to address these challenges that draw inspiration from biological system. The goal of autonomic computing is to develop applications and systems that can manage themselves based on high level guidance from humans.

In this thesis, we develop the Accord programming system for autonomic self-managing applications. Accord builds on existing programming systems and extends them to (1) enable the definition of autonomic elements that encapsulates functional and non-functional specifications, rules, and mechanisms for self-management, (2) enable the formulation of self-managing applications as dynamic compositions of autonomic elements, and (3) provide a runtime infrastructure for the correct and efficient runtime execution of rules to enforce self-managing behaviors in response to changing requirements and execution context.

Three prototypes of the Accord programming system have been implemented and customized to support different classes of applications. The first prototype enables the rule-based self-management of objects and object-based parallel/distributed applications. The second prototype extends the Common Component Architecture Ccaffeine framework to enable self-managing component-based high-performance parallel/distributed scientific applications. This

prototype supports both function and performance driven intra- and inter-component adaptations, and enables dynamic composition and runtime component replacement. The third prototype supports self-managing service-based applications and enables runtime adaptation of service and service interactions, and decentralized and dynamic service composition. The operation of these prototypes is illustrated using a suite of scientific applications. Experimental evaluations of the prototypes are presented.

Acknowledgements

I would like to gratefully and sincerely acknowledge the supervision of Dr. Manish Parashar during my years as a graduate student. I would also like to thank Dr. Jaideep Ray for his guidance and help during my summer intern in Sandia National Labs in 2004.

I am very grateful to all my friends from Rutgers, Beijing University of Posts & Telecoms, Nortel networks (China), Bell-labs (China) for the warm smiles and encouragement.

Finally, I am forever indebted to my parents Shouxian Liu and Xiurong Feng, and my husband Xiaoxuan Li for their understanding, support, endless patience, and encouragement. Thanks for always being there for me.

Table of Contents

Abstract . . . ii

Acknowledgements . . . iv

List of Tables . . . ix

List of Figures . . . x

1. Introduction . . . 1

1.1. Motivation . . . 1

1.2. Problem Description . . . 2

1.3. Overview of Accord Programming System . . . 3

1.4. Outline . . . 6

2. Background and Related Work . . . 8

2.1. Programming Systems for Parallel and Distributed Computing . . . 8

2.2. Adaptation Technologies and Related Work . . . 12

2.2.1. Specifying Adaptation Behaviors . . . 12

2.2.2. Enforcing Adaptation Behaviors . . . 13

Enforcing Adaptation Without Source Code Modification . . . 13

Enforcing Adaptation Through Source Code Modification . . . 14

2.2.3. Conflict Detection and Resolution During Adaptation Behavior Execution 15

2.3. Accord Programming System . . . 15

3. Accord: Supporting the Development and Execution of Autonomic Self-managing Applications . . . 17

3.1. Defining Application Context . . . 18

3.2. Autonomic Element . . . 18

3.2.1. Port Definition . . . 19

3.2.2. Element Manager 20
3.3. The Accord Runtime Infrastructure 21
3.3.1. Composition Manager 22
3.3.2. Element Manager 22
3.3.3. The Rule Enforcement Engine 23
Key Concepts and Notation 23
Rule Execution Model 24
3.4. Autonomic Adaptation Behaviors in Accord 26
3.4.1. Adapting Element Behaviors 26
3.4.2. Adapting Element Composition 27
Element Composition 27
Dynamic Composition 28
3.5. Autonomic Forest Fire Application: An Illustrative Example 30
3.5.1. Defining Autonomic Element 31
3.5.2. Enabling Adaptation Behaviors 32
Adapting *DSM* Behaviors 32
Adding A New Element 32
Changing Interaction Relationships 32
3.6. Summary 33

4. DIOS++: Autonomic Object-based Accord 34
4.1. Autonomic Monitoring and Control with DIOS++ 35
4.2. DISCOVER Collaboratory 36
4.3. DIOS++ Architecture 38
4.3.1. Autonomic Object 38
Control Interface 39
Access Interface 39
Rule Interface 40
Rule Agent 40

4.3.2. Control Network 41
Initialization 42
Interaction and Rule Operation 42
4.4. The Autonomic Oil Reservoir Application: An Illustrative Example 45
4.5. Experimental Evaluation 47
4.6. Summary and Conclusion 48

5. Accord-CCA: Autonomic Component-based Accord 50
5.1. Component-Based Distributed/Parallel Scientific Applications 50
5.1.1. The Common Component Architecture (CCA) 50
5.1.2. Behavior and Performance of Component-based Scientific Applications 51
5.2. Self-management of Component-based Scientific Applications 52
5.2.1. Defining Managed Components 53
5.2.2. Enabling Runtime Self-management 54
Component Manager 56
Composition Manager 57
Rule Execution Model 57
5.2.3. Supporting Performance-driven Self-management 58
5.3. Case Studies 59
5.3.1. A Self-Managing Hydrodynamics Shock Simulation 59
Scenario 1: Self-optimization via component replacement 61
Scenario 2: Self-optimization via component adaptation 63
Scenario 3: Self-healing via component replacement 64
5.3.2. A Self-Managing CH_4 Ignition Simulation 65
5.3.3. Experimental Evaluation 65
5.4. Summary and Conclusion 67

6. Accord-WS: Autonomic Service-based Accord 68
6.1. Autonomic Services 68
6.2. The Runtime Infrastructure 70

6.2.1. Workflow Execution 70

6.2.2. Dynamic Composition 72

6.3. An Illustrative Application: The Autonomic Data Streaming Application 73

6.3.1. Service Adaptation 74

6.3.2. Application Adaptation 76

6.4. Summary 77

7. Summary, Conclusion, and Future Work 82

7.1. Summary 82

7.2. Conclusion 83

7.3. Directions For Future Work 84

References 86

Curriculum Vitae 91

List of Tables

2.1. Capabilities and limitations of current programming systems with respect to the programming requirements of Grid environments. 11

List of Figures

3.1. An autonomic element. 18

3.2. Accord runtime infrastructure for a sample application composed of three elements. 22

3.3. The three-phase rule execution model. 24

3.4. Examples of the port definition. 31

3.5. A behavior rule for *DSM*. 32

3.6. Add a new element *FFM*. 32

3.7. Change the interaction relationship between *CRM* and *DSM*. 33

4.1. DISCOVER collaboratory architecture. 37

4.2. An autonomic object. 38

4.3. A sample rule for *RandomList*. 40

4.4. The DIOS++ control network. 41

4.5. Rule1: an object rule involving only one object *RandomList*. Rule2: an application rule involving two objects *RandomList* and *SortSelector*. 43

4.6. (a): Deployment of an object rule. (b): Deployment of an application rule. . . . 44

4.7. Rules with conflicts. 45

4.8. The constraint in VFSA that maintains the probability value between 0 and 1. . 46

4.9. A sample application rule involving VFSA and IPARS. 46

4.10. DIOS++ experimental evaluations. 47

5.1. The *RulePort* specification. 53

5.2. A self-managing application composed of 5 components. The solid lines denote computational port connections between components, and the dotted lines are port connections constructing the management framework. 55

5.3. Distributed self-managing application shown in Figure 5.2 executed on three nodes. The solid lines across nodes denote the interactions among manager instances. The dotted lines are port connections constructing the management framework within one node. 55

5.4. "Wiring" diagram of the shock-hydrodynamics simulation. A second-order Runge-Kutta (**RK**2) integrator drives **InviscidFlux** component – transformation into left and right (primitive) states is done by **States** and the Riemann problem solved by **GodunovFlux**. Sundry other components for determining characteristics' speeds (u + a, u - a, u), cell-centered interpolations etc. complete the code. 61

5.5. The average execution times for **EFMFlux** and **GodunovFlux** as functions of the array size (machine effects have be averaged out). 62

5.6. Replacement of **GodunovFlux** with **EFMFlux** to decrease cache misses. . . . 63

5.7. Dynamically switch algorithms in **AMRMesh**. 64

5.8. Comparison of rule based and non rule based execution of CH_4 ignition. 65

5.9. Experimental evaluation of Ccaffeine-based Accord prototype. 66

6.1. An autonomic service. 69

6.2. Message processing in a coordination agent. 69

6.3. The runtime framework. The dashed lines represent the interactions among managers. The solid lines represent the interactions among WS-Resources. . . 70

6.4. The itinerary workflow specified using (a) BPEL4WS and (b) Accord interaction rules. 72

6.5. A new service **ParkService** is added to the itinerary workflow. The dashed lines denote the new interaction relationships created due to the addition of the new service. 73

6.6. The autonomic data streaming application based on Accord-WS. 73

6.7. The control port of **BMS**. 78

6.8. The behavior rule for **BMS**. 79

6.9. (a) Self-optimization behaviors of the Buffer Management Service - BTS switches between uniform blocking and aggregate blocking algorithms based on application data generation rates and network transfer rates and the nature of data generated. (b) Percentage overhead on simulation execution simulation with and without autonomic management. 79
6.10. The interaction rule for ADSS. 80
6.11. Effect of switching from the DSS at PPPL to the DSS ORNL in response to network congestion and/or failure. 81

Chapter 1

Introduction

1.1 Motivation

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 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, executed and managed.

1.2 Problem Description

The nature and scale of pervasive information and computational Grid environments and applications introduce new levels of development and management challenges. 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 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 software elements 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 characteristics listed above impose requirements on the programming systems for Grid applications. The programming systems must be able to specify applications that can detect and dynamically respond during execution to changes in both, the state of execution environment and the state and requirements of the application. This requirement suggests that: (1) Grid applications should be composed from discrete, self-managing elements which incorporate separate specifications for all of functional, non-functional and interaction 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 compose-able. (3) The interface definitions of these

elements should be separated from their implementations to enable the interactions between heterogeneous elements and the dynamic selection of elements. Given these features of a programming system, a Grid application requiring a given set of computational behaviors may be integrated with different interaction 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 application requirements and the environments.

However, current existing programming systems do not meet the requirements outlined above. Many current communication frameworks for parallel and distributed computing typically make very strong assumptions about the behavior of the entities, their interactions, and the underlying system. Distributed object systems provide support for parallel/distributed applications. However, the interacting objects and interaction are tightly coupled. Further, they assumes a priori (compile-time) knowledge of the syntax and semantics of interfaces as well as the interactions required by the applications. The dominant component-based and service-based programming systems have similar limitations. The orchestration and choreography of components/services must be defined a priori and the support for runtime adaptation is limited. However, they provide core mechanisms that can be extended to address the requirements of Grid applications and environments.

1.3 Overview of Accord Programming System

The challenges and requirements outlined above and the limitations of current programming systems 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.

This research addresses the Accord programming system that extends existing programming systems to support the development of self-managing applications in distributed environments. The system builds on the separation of the composition aspects (e.g., organization, interaction and coordination) of elements (object, components, and services) from their computational behaviors that underlies the object, component, and service based paradigms, and extends it to enable the computational behaviors of elements as well as their organizations,

interactions and coordination to be managed at runtime using high-level rules.

The Accord programming system (1) defines autonomic elements that encapsulates functional and non-functional specifications, rules, and mechanisms for self-management, (2) enables the formulation of self-managing applications as dynamic compositions of autonomic elements, and (3) provides a runtime infrastructure for correct and efficient rule execution to enforce self-management behaviors in response to changing requirements and execution context. As a result, Accord supports two levels of adaptations: (1) adaptation at the element level to monitor and control behaviors of individual elements according to their internal state and execution context, (2) adaptation at the application level to change application topologies, communication paradigms and coordination models used among elements to respond to changes in the environments and user requirements.

Three prototypes of Accord that separately extend a distributed object programming system, a component based system, and a service based system are discussed in this thesis. The design, implementation, operation, and evaluation of these prototypes are presented.

- An object based prototype of Accord, named DIOS++, has been implemented and evaluated in the context of distributed scientific/engineering simulations as part of the DISCOVER project. This prototype implements autonomic elements as autonomic objects by associating objects with sensors, actuators and rule agents, and providing a runtime hierarchical infrastructure consisting of rule agents and rule engines for the rule-based autonomic monitoring and control of distributed and parallel applications.

  DIOS++ is used to support the autonomic IPARS reservoir simulation that optimizes the placement and operation of oil wells for maximal overall revenue. DIOS++ enables direct modification of object parameters and collaborative control of multiple objects to, for example, dynamically select objects' internal algorithms based on their internal state or execution context.

- A component based prototype of Accord, named Accord-CCA, has been developed based on the DoE Common Component Architecture (CCA) and the Ccaffeine framework in the context of component-based high-performance scientific applications. This prototype extends CCA components to autonomic components by associating them with control

and operation ports and component managers, and provides a runtime infrastructure of component managers and composition managers for rule-based component adaptation and dynamic replacement of components.

Accord-CCA is used to enable (1) the self-managing shock hydrodynamics simulation that simulates the interaction of a hydrodynamic shock with a density-stratified interface, and (2) the CH_4 ignition simulation that simulates a set of chemical reactions appearing and disappearing when the fuel and oxidizer react and give rise to the various intermediate chemical species. Accord-CCA enables self-optimization and self-healing of the shock simulation, for example, by dynamically replacing components to decrease the cache misses or maintain stability for stronger shocks and larger density ratios. It also enables self-optimization of the CH_4 ignition simulation by dynamically selecting appropriate algorithms at different reaction temperatures.

- A service based prototype of Accord, named Accord-WS, is designed based on the WS-Resource specifications, the Web service specifications, and the Axis framework. Autonomic elements are implemented as autonomic service by extending traditional WS-Resources with service managers for rule-based management of runtime behaviors and interactions with other autonomic services, and coordination agents for programmable communications. A distributed runtime infrastructure is investigated to enable decentralized and dynamic compositions of autonomic services.

  An autonomic data streaming transferring application is enabled by Accord-WS to stream realtime data from a live simulation support remote runtime analysis and visualization while minimizing overheads on the simulation, adapting to network conditions, and eliminating loss of data. An example of service adaptation supported by Accord-WS is dynamically selecting algorithms within the buffer management service based on the current state of the simulation and network condition. An example of application adaptation supported by Accord-WS is dynamically switching to a local storage from remote data streaming in the cases of extreme network congestion or network failures.

Key contributions of the research include:

- Analysis of the programming requirements for autonomic self-managing applications in

heterogeneous and dynamic environments.

- Design of a programming system that addresses these requirements and enables the development of autonomic self-managing applications.

- Design, implementation, operation, and evaluation of three prototypes that extend dominant programming systems and address different classes of applications.

- Autonomic self-managing scientific applications that are capable of managing and optimizing their execution based on application and system state, user requirements, and execution context.

1.4 Outline

The rest of this thesis is organized as follows. Chapter 2 provides an overview of existing programming systems and discusses their capabilities and limitations with respect to the programming requirements. The chapter also discusses existing adaptation technologies and related efforts.

Chapter 3 describes the Accord programming system, including the definition of autonomic elements and rules, and the runtime infrastructure that executes rules to enable adaptation behaviors. An autonomic forest fire simulation is used to illustrate the operations of Accord programming system.

Chapter 4 presents an object based prototype that enables distributed scientific/engineering simulations. An experimental evaluation of the system is also presented. This prototype enables interactive and rule-based management of individual objects and applications at runtime. The autonomic IPARS reservoir simulation is used to illustrate the design, implementation, and operations of the object based Accord system.

Chapter 5 presents a component based prototype built on the DoE Common Component Architecture (CCA) Ccaffeine framework. This implementation supports autonomic component-based scientific applications. It supports both function and performance based self-management, enables dynamic composition through runtime component replacement, and provides consistent and efficient rule execution for intra- and inter-component management behaviors. The

design, implementation, and evaluation of the prototype is presented. Self-managing shock hydrodynamics simulation and CH_4 ignition simulation are presented as case studies.

Chapter 6 presents a service based prototype based on Web Services and WS-Resource specifications and the Axis framework. It supports self-managing service-based applications by enabling dynamic service composition. An autonomic data streaming transferring application is used to illustrate the adaptation operations enabled by the service based Accord.

Chapter 7 concludes the thesis with a summary of the research and lessons learned, and outlines the directions for future work.

Chapter 2

Background and Related Work

The overall goal of this research is to investigate a programming system to enable autonomic self-managing applications that address the challenges of pervasive and Grid environments. This chapter first investigates the limitations of current existing programming systems for parallel and distributed computing with respect to the programming requirements outlined in the previous chapter. It further investigates adaptation technologies that can be integrated with these programming systems. Finally, it describes the Accord programming system that extends the existing programming systems with adaptation technologies to support the development and execution of autonomic self-managing applications.

2.1 Programming Systems for Parallel and Distributed Computing

There has been a significant body of research on programming systems for parallel and distributed computing over the last few decades. Many current **communication frameworks for distributed and parallel computing**, for example message passing models and shared memory models, support interactions between distributed entities developed using conventional programming models. These systems typically make very strong assumptions about the element behaviors, element interactions, and the underlying system, especially about their static nature and reliability, which limit their applicabilities in highly dynamic and uncertain computing environments.

Researchers have also investigated the enhancement and application of traditional communication paradigms to pervasive Grid environments. For example, GridRPC [49] extends standard RPC with asynchronous coarse-grained parallel tasking, hiding the dynamics, insecurity, and instability of the Grid from the programmers. MPICH-G2 [34], a Grid-enabled implementation of the MPI [6], allows a user to run MPI programs across multiple computers, possibly across different sites, using the same abstractions that can be used on a parallel computer.

Distributed object systems: Unlike the systems described above that essentially address

only communication aspects, distributed object systems provide more support for parallel and distributed applications, including lifecycle management, location and discovery, interaction and synchronization, security, failure and reliability [17]. CORBA [1], one of the dominant distributed object systems, enables secure interactions 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). The interactions are based on remote procedure calls, method invocations and event notification. CORBA primarily addresses distribution and heterogeneity, and also provides limited support for dynamism via dynamic invocation (DSI/DII) and late binding, which enables customization at deployment time. However, interacting objects and interaction are tightly coupled. Further, the model assumes a priori (compile-time) knowledge of the syntax and semantics of interfaces and the interactions required by the applications.

Although CORBA does not directly enable dynamic adaptation of object behaviors 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" [1]. Note that these adaptation behaviors are performed by manipulating and redirecting messages using interceptors, but the direct adaptation of individual objects is not supported.

Component-based programming systems: Component models address increasing software complexity and changing requirements by enabling the construction of systems as assemblies of components. Components are reusable units of composition, deployment, execution, and lifecycle management [67], and are completely specified by their interfaces. Current component frameworks include CORBA Component Model (CCM) [1], JavaBeans [67] and Common Component Architecture(CCA) [14].

CCM [1] 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 prior knowledge about interfaces and interactions. JavaBeans [67] is a Java only component model that addresses similar issues. It also supports runtime bean customization.

The Common Component Architecture (CCA) [14] defines a component model especially for scientific applications. The model primarily addresses the heterogeneity through separating interface from implementation. One of the CCA implementation, the Ccaffeine framework, targets high-performance parallel applications and uses functional calls for inter-component interactions within a Single Component Multiple Data (SCMD) model. While it does not support runtime customization of components, it does allow components to be dynamically replaced at runtime. It does not address failures or security issues and assumes all components are trusted.

Note that component-based systems also provide some core mechanisms, such as interceptors in CORBA, the BuilderService in CCA, and the container mechanism 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 systems: Service based models, such as the Web service and Grid service [25, 46, 50] 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, for example, that it must support reliable invocation, which is not possible in the presence of failures and the lack of global knowledge. Current orchestration and choreography mechanism for Web and Grid services are static and must be defined a priori.

A huge body of research is underway to facilitate the development and management of service oriented applications. These research projects can be categorized into two types of approaches. The first approach looks at composite services mainly from the runtime perspective as functions, data and control flows [66] described in, for example BPEL4WS [15]. Workflows can be composed at runtime based on the syntactic, semantic and operational matches and end-to-end QoS analysis [12, 24], or based on the precondition and post-condition specification for

available services, as well as the pre-condition and post-condition specification for the composite service to be constructed [55]. Workflow execution involves integrating services together and executing them as specified in the workflow. This involves, for example, dynamically configuring the method calls and invoking the Web services [43], or selecting the appropriate runtime representation for each service specification in the application [32]. However, most of these workflow execution environments do not emphasize runtime adaptation and optimization of the workflow, which is essential to address changing application and system requirements, state and context. The second approach, ontology based semantic composition using OWL [9], is taken by the semantic web community. OWL supports service reasoning to facilitate service discovery and usage. However, this approach has an imprecise underlying conceptual model leading to multiple modeling possibilities and parametric polymorphism [38], which significantly increases the complexity of service composition.

The programming systems discussed above are summarized in Table 2.1.

Programming Systems	Issues addressed	Limitations
Communication frameworks (e.g., RPC, RMI, MPI, PVM)	Heterogeneity, distribution, limited dynamism	No context-awareness, assume deterministic, static, secure, and reliable environment
Distributed object systems (e.g., CORBA)	Heterogeneity (platform and languageindependence), distribution, limited dynamism	No context-awareness, assume deterministic, static, secure, and reliable environment
Component based systems (e.g., CCA, CCM, JavaBeans)	Heterogeneity, distribution, limited dynamism	No context-awareness, assume deterministic, static, secure, and reliable environment
Service oriented systems (e.g., Web service architecture, WSRF)	Heterogeneity, distribution (maybe across internet), limited dynamism	No context-awareness, assume static and reliable environment

Table 2.1: Capabilities and limitations of current programming systems with respect to the programming requirements of Grid environments.

2.2 Adaptation Technologies and Related Work

Adaptation technologies can be integrated with existing programming systems to enable autonomic self-managed applications. To enable self-managed/adaptive applications, the following issues must be addressed: (1) how to specify adaptation behaviors, (2) how to efficiently enforce adaptation behaviors, and (3) how to guarantee the correctness of adaptation behaviors.

Adaptation behaviors are typically application-specific. For example, the selection of internal variables or functions within individual components and the usage of communication paradigms across multiple components, are determined by the application logic and its execution context. However, the approaches and mechanisms that support runtime self-management and adaptation are general and independent of specific applications. These approaches and mechanisms are discussed in the following sections.

2.2.1 Specifying Adaptation Behaviors

Adaptation behaviors can be either statically or dynamically specified. Traditional programming paradigms that use conditional branches to enable different runtime workflows can be viewed as an example of **statically specified adaptations**. Some systems extend existing programming languages to provide templates that enable adaptive scheduling for different application types [19]. Others provide adaptation classes to enrich Java classes with adaptive behaviors and a dedicated compiler that automatically generates Java code and implements these adaptive features [21]. Systems can also use scripts or languages to describe adaptation behaviors. For example, Beazley and Lomdahl [18] used a Simplified Wrapper Interface Generator (SWIG) that wraps existing source code with scripting language interfaces to enable external monitoring and steering. Systems such as VASE [47] also use this approach. The script is executed when the application encounters pre-defined breakpoints and results in adaptation behaviors. Quo [29] specifies domain-specific adaptation behaviors using a separate language and integrates them with applications as an aspect [37]. A key drawback of the above approaches is that all the possible adaptation must be known a priori and coded into the applications. If new adaptation behaviors are required or if application requirements change, the application code has to be modified and the application possibly re-compiled.

Some systems enable **dynamically specified adaptation** by allowing adaptation, 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 [62] 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 and adapt the interactions between elements. For example, ALua [70] uses the Lua language to perform interaction and adaptations in an interpretive manner, and supports the execution of dynamically defined adaptation specifications in an even-driven manner.

2.2.2 Enforcing Adaptation Behaviors

An adaptation specification needs to be integrated with applications so that can monitor and steer the application execution according to changing requirements and execution contexts. This section first introduces adaptation enforcement mechanisms for legacy applications where application source code is unaccessible. Adaptability using this approach is limited. The section then discusses adaptation enforcement requiring modification of application source code.

Enforcing Adaptation Without Source Code Modification

Systems such as KX (Kinesthetics eXtreme) [71] and Pathfinder [39] use **mobile agents** to integrate adaptation code with applications. Mobile agents provide power and flexibility in the specification and deployment of monitoring and steering commands [39]. For example, they are capable of executing orthogonally to the main computation of the target applications. Besides, mobile agents can be deployed to the same memory space where application modules reside, which reduces the latency when reacting to local conditions and provides corresponding actions. Further, mobile agents can be customized to exploit application-specific information and can efficiently perform adaptation behaviors.

However, supporting the execution of mobile agents, virtual machines or milieus are required at all ‘stops’. The virtual machine/mulieu serves as the hosting environment for mobile agents, providing a library of operations to enable agents to perform monitoring and steering actions and support agent communication, migration and scheduling. The requirement for such

a hosting environment increases implementation complexity. Besides, the behaviors of mobile agents are restricted due to security concerns. Possible security problems include masquerading, denial of service, unauthorized access, eavesdropping, alteration, and repudiation [30].

Enforcing Adaptation Through Source Code Modification

Systems such as CUMULVS [33] and VIPER [57] use program instrumentation to enable monitoring and steering. CUMULVS allows developers to declare the variables or parameters that can be modified or steered during the computation. VIPER similarly allows developers to annotate application programs to identify the data and parameters for monitoring and steering and to associate them with synchronization points. When the application encounters a synchronization point, the server is notified and extracts the current state of the data and parameters for adaptation purpose. However, CUMULVS and VIPER provide limited steering capabilities, e.g., they do not support coordinated steering across modules. As an improvement, problem solving environments such as SCIRun [53], provide mechanisms such as feedback loops, cancellation, direct lightweight parameter changes, and retained state across module firings, to enable modular and dataflow-oriented systems and create a richer set of steerable parameters. However, a key limitation of CUMULVS, VIPER and SCIRun is that these systems cannot directly support functional or algorithmic steering.

The component-based programming paradigm [67, 14] views applications as a composition of individual components. Therefore, adaptation behaviors can be systematically enforced at two separate levels - (1) intra-component: components can expose internal variables or parameters as sensors and actuators for external monitoring and steering, and (2) inter-component: the interactions among components can be adapted.

If component source code is inaccessible, sensors and actuators can be obtained from component interfaces using techniques such as (1) superimposition [22] that enables the software engineer to impose predefined but configurable functionalities on a component, and (2) wrapping [69] that enables components to be refined at runtime using wrappers to introduce new behaviors. Alternately, components can directly expose their internal variables, data, parameters, functions, etc., as sensors and actuators for external monitoring and steering. To adapt

component interactions, filters [55, 13] or proxies [62, 59] can be interposed between interacting components to manipulate messages or re-direct the messages to different components, and consequently introduce new behaviors to the application.

2.2.3 Conflict Detection and Resolution During Adaptation Behavior Execution

The correctness of adaptation behaviors that are described using templates [19] and adaptation classes [21], and compiled together with application code, can be checked by compilers. This section focuses on adaptation behaviors described separately using description languages, scripts, and rules, and weaved with application execution at runtime.

Conflicts can occur between multiple adaptation behaviors, and between adaptation behaviors and application execution. Some conflicts can be detected and resolved **statically**. For example, the conflicts among typed authorization policies can be statically detected as part of the policy specification process [42], and resolved by having users revise the conflicting policies. Specification time conflict detection is analogous to compile time type checking for programming languages. A limitation of static analysis is that it may not detect conflicts that depend on runtime state. As a result, it can be used to detect potential rather than actual conflicts.

Some conflicts can only be detected and resolved **at runtime**. For example, applications may require services to be delivered in different ways when requested in different execution contexts [23]. In this case, conflicts occur and can be detected at runtime when a conflicting service is invoked, and then resolved synchronously using auctions based on microeconomic techniques. As an another example, the execution of business policies depends on runtime business logic and information. Therefore the conflicts can only be detected at runtime when overlapping queries and inconsistent provisions are found, and resolved asynchronously by using legal reasoning based on the status of individual obligations in a database [11].

2.3 Accord Programming System

Accord extends object, component, and service based programming systems to support dynamically defined adaptation behaviors using high-level rules. These rules can be directly defined

and injected by users through Accord portals, or they can be automatically generated from application requirements and workflows specified by users. Users can add new rules, delete rules, and modify existing rules.

Accord enforces adaptation behaviors using a runtime distributed rule execution infrastructure. It senses the internal state of elements and applications as well as execution context, evaluates rule conditions, and performs the corresponding actions.

Accord provides different mechanisms to detect and resolve conflicts for scientific and business applications. In the scientific computing domain, the most interesting use of rules is to directly control the behaviors and performance of applications. Therefore, simple reaction rules and efficient and scalable parallel rule execution with minimal performance impact is critical. Rules in business applications are more complex and typically used to create business obligations, provide recommendations and decision support, or describe entity privileges. The component-based and service-based prototype implementations of Accord supports these two classes of applications.

Chapter 3

Accord: Supporting the Development and Execution of Autonomic Self-managing Applications

Accord extends existing programming systems to support the development and execution of autonomic self-managing applications. Specifically, Accord builds on the separation of the composition aspects (e.g., organization, communication, and coordination) of elements (e.g., objects, components, and services) and applications from their computational behaviors that underlies the distributed object, component, and service based paradigms. Further, Accord extends it to enable element behaviors and element composition to be managed at runtime using high-level rules.

To support runtime self-management, Accord relaxes static (defined at the compilation or instantiation time) application requirements and system/application behaviors, and allows them to be dynamically specified using high-level rules. Further, it enables the behaviors of elements and applications to be sensitive to changing system state and application requirements, and adapt to these changes at runtime. This is achieved by extending computational elements into autonomic elements with the specifications of high-level rules and mechanisms for self-management, and providing a distributed runtime infrastructure that consistently and efficiently enforces these rules to enable autonomic self-managing behaviors to respond to current requirements, state and execution context.

The Accord programming system consists of four concepts. The first is an application context that defines a common semantic basis for applications, discussed in Section 3.1. The second is the definition of autonomic elements as the building blocks of autonomic applications, and the next is the definition of rules and mechanisms to enable autonomic elements, presented in Section 3.2. The final is a runtime infrastructure that enforces rules to realize adaptation behaviors, described in Section 3.3.

3.1 Defining Application Context

Autonomic elements should agree on a common syntax and semantics for defining and describing ontologies, namespaces, sensors, actuators, function interfaces and events that 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 programming systems with adaptation capabilities, it uses the mechanisms provided by these systems to define application context. Current implementations of Accord extend a distributed object system [48], a component based system [14], and a service based system [31], 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 adapt element behaviors and interactions at runtime.

3.2 Autonomic Element

An autonomic element is the fundamental building block for self-managing applications in the Accord system. It extends a computational element, including object, component, and services, to define a self-contained modular software unit of composition with specified interfaces and explicit context dependencies. Additionally, an autonomic element encapsulates rules and mechanisms for self-management, and dynamically interacts with other autonomic elements.

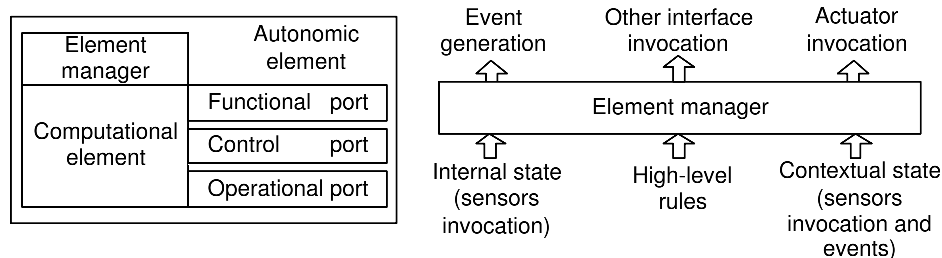


Figure 3.1: An autonomic element.

3.2.1 Port Definition

The structure of an autonomic element is shown in Figure 3.1. It is defined by three classes of ports:

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 actuator 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 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 element. Rules incorporate high-level guidance and practical human knowledge, describing the *actions* to be taken when a certain *condition* is met or *events* are received. *Events* are messages generated by elements and/or systems. *Condition* is a logical combination of element (and environment) sensors. *Actions* consist of a sequence of invocations of elements and/or system actuators, and other interfaces. Two types of rules are defined.

   - *Behavior rules* control the runtime functional and non-functional behaviors of elements. For example, behavior rules can control the algorithms, data representations or input/output formats used by elements. Behavior rules can be directly defined by users or automatically generated from application requirements or objectives.

   - *Interaction rules* control the interactions between elements, between elements and their environments, and the coordination within an application. For example, an interaction rule may define where an element will get inputs and forward outputs,

define the communication mechanisms used, and specify when the element interacts with other elements. Interaction rules can be either directly specified by users or automatically generated from the primary workflow.

Computational elements have to implement and export appropriate sensors and actuators so that their behaviors and interactions can be monitored and controlled. Adding sensors and actuators requires modification/instrumentation of the element source code. Sensors and actuators can be implemented either as new methods that monitor or modify internal parameters and behaviors of an element, or defined in terms of existing methods. Accord provides corresponding programming abstractions that can be used to specify sensors and actuators and register them in element managers. The instrumented computational elements have to be re-compiled.

In case of third-party and legacy elements where such a modification may not be possible or feasible, proxies [62, 59] can be used to collect relevant element information from interacting messages. A proxy is interposed between the caller and callee elements to monitor and control, for example, all the method invocations for the callee element. It may also collect performance information (e.g., response time, memory usage, cache misses) and input parameter values for each interfaces in the functional port and control the invocations by modifying the input parameters for these interfaces. In this case, elements may not have to be re-compiled, but the adaptability of the elements is limited.

3.2.2 Element Manager

As shown in Figure 3.1, each computational element is associated with an element manager that implements the operation port. The element manager performs one or more of the following management functions.

- *Functional management*: The manager controls the functional behaviors of the managed element based on changing requirements, state and execution context. For example, the manager may generate customized functional ports that activate or deactivate operations provided by the element based on the security of the execution environment and the access privileges of the user. As another example, the manager may dynamically select internal algorithms or data representations for current inputs.

- *Performance management*: The manager monitors the performance of the element, selectively collects relevant performance data (e.g., response time and throughput), and adapts it behaviors to meet performance requirements at runtime. For example, the managers may dynamically decrease message frequency during network congestion.

- *Interaction management*: Managers may negotiate with each other to dynamically establish or change interaction relationships. For example, the manager may modify communication patterns and coordination sequence between managed elements and respond to changing application requirements and execution context. For example, the managers may automatically establish interaction relationships between newly introduced elements with the rest of an application.

As shown in Figure 3.1, an element manager monitors the internal state of its associated element using element sensors, and senses the execution context using system sensors and events. It further controls the firing of rules and performs rule actions by generating events, invoking actuators or other interfaces exposed by the element and the system. To support the management behaviors outlined above, elements and the system should expose appropriate sensors and events. For example, runtime element performance data should be exposed as sensors or events to support performance based management. Further, elements should support different interaction relationships and enable the element managers to select among these at runtime for interaction management. Wrappers [69], filters [55, 13], or proxies [62, 59] can be used to introduce these capabilities in managed elements that do not directly provide them.

3.3 The Accord Runtime Infrastructure

The Accord runtime infrastructure, shown in Figure 3.2, consists of a portal, composition managers, and peer element managers.

The infrastructure is constructed at runtime using services provided by the underlying programming system. For example, the MPI runtime architecture [6] is used to construct the infrastructure in the object based prototype of Accord presented in Chapter 4. In the component based prototype presented in Chapter 5, the Ccaffeine framework [14] provides the required services. The service based prototype presented in Chapter 6 builds on the Axis framework [16].

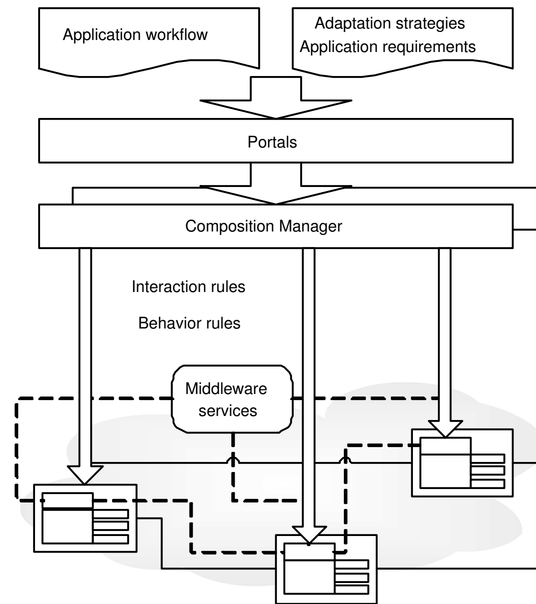


Figure 3.2: Accord runtime infrastructure for a sample application composed of three elements.

The infrastructure can also be built on AutoMate middleware services including content-based discovery service, associative messaging service, and decentralized reactive tuple space.

3.3.1 Composition Manager

Application workflows are defined by users or generated by a workflow engine, for example [12], and expressed in XML. The workflows are then decomposed by the composition manager into interaction rules and injected into corresponding element managers during initialization period. At runtime, the composition manager analyzes and decomposes adaptation strategies, dynamically defined by users or generated from application requirements, into behavior rules and interaction rules, and injects them into corresponding element managers.

3.3.2 Element Manager

Element managers can be statically associated with elements at compilation time, or created by the composition manager at runtime. During initialization, element managers register the sensors and actuators exposed by associated elements, and register element and system events required by the rules. Managers communicate with each other using XML on top of the underlying communication system such as MPI messaging in the object and component based

prototypes, and sockets and SOAP in the service based prototype. Element managers coordinate with each other and construct a distributed rule enforcement engine to enable consistent and efficient rule execution. This is discussed below.

3.3.3 The Rule Enforcement Engine

Key Concepts and Notation

All specified rules for an application define a **rule space**, denoted as $R = \{R_i\}$, where $R_i =$ "$IF\ S_i\ THEN\ A_i$". S_i represents the set of sensors and/or events in the rule condition, denoted as $S_i = \{s_i\}$. A_i is a list of actuators in the rule action, denoted as $A_i = \{a_i\}$. An **active rule space**, denoted as $\bar{R}$, is the set of rules in the rule space whose conditions are currently satisfied.

The **pre-condition** of the rule space R consists of the sensors used by R and their current values, and is represented as $\{S, VS\}$, where $S = \bigcup S_i$ and $VS = Value(S)$. The pre-condition is defined by the state of the computation and the execution environment, and changes during the lifetime of the application. As a result, the pre-conditions is known only at runtime.

The **post-condition** of the rule space R consists of the set of actuators and the values that they should take, and is represented as $\{A, VA\}$, where $A = \bigcup A_i$, $VA = Value(A)$.

In traditional rule-based systems, firing of rules consists of sequentially executing the action part of all the rules triggered by pre-condition and producing consequence. This makes it difficult to detect and resolve rule execution conflicts and guarantee consistent rule execution (as described in more detail below). To address this, we define post-condition, which combines the action part of all the rules triggered by pre-condition and can be used to detect and resolve conflicts and inconsistencies before producing the consequences.

Consequence is the change in application state and behaviors caused by invoking the actuators defined by the post-condition. Changes can affect both component behaviors and the application process, and are applied in the next computation phase.

Two kinds of **rule conflicts** are defined: (1) A **sensor-actuator (**$S - A$**) conflict** occurs when $SA = S \bigcap A \neq \phi$, i.e., a parameter/variable is exposed both as a sensor and as an

actuator in a set of rules. (2) An **actuator-actuator ($A - A$) conflict** occurs when the post-condition for a rule set contains multiple instances of an actuator with different values. Note that these conflicts have to be detected at runtime as rules can be dynamically defined. They need to be resolved only if the conflicting rules are simultaneously triggered.

Reconciliation is required in SCMD parallel applications, as different processing nodes may generate different post-conditions based on their local states and contexts. The goal of reconciliation is to produce a unique post-condition across all nodes.

Rule Execution Model

As mentioned above, traditional rule-based systems directly invoke actions when rules fire [11]. However, this approach aggravates rule conflicts when multiple rules are simultaneously triggered. If $S - A$ conflicts exist, directly invoking actions when the condition of a rule is met will change the values of the sensors in the set SA, which in turn will change the pre-condition. This can have two consequences. First, the modified pre-condition may invalidate already executed rules. Second, it may trigger rules whose conditions previously evaluated as false. If $A - A$ conflicts exist, actions invoked later will 'overwrite' the results of perviously invoked actions. Together, these conflicts can produce both uncertainty and inconsistency in rule execution. To address this, we employ a three-phase rule execution model, consisting of (1) batch condition inquiry, (2) condition evaluation and conflict resolution and reconciliation, and (3) batch action invocation. This is illustrated in Figure 3.3.

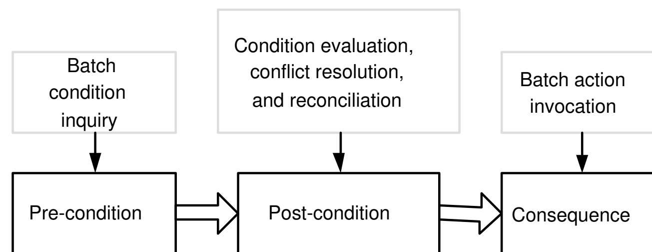


Figure 3.3: The three-phase rule execution model.

The batch condition inquiry phase queries all the sensors in S in parallel, gets their values VS, and then generates the pre-condition. Based on this pre-condition, rules whose conditions are satisfied form the active rule space $\bar{R}$. In next phase, condition evaluation for all the rules

in $\bar{R}$ is performed in parallel. The overall evaluation time in this case will be determined by the longest evaluation time for an individual rule. Conflict resolution and reconciliation then takes place and the post-condition is generated.

In the final phase, the actuators in the post-condition are invoked to produce the consequence. This may also be done in parallel, since the actuators in the post-condition are independent and free of conflicts. Note that as the rule base becomes larger, the conflict resolution time will increase. However the time required for sensor queries, condition evaluations and actuator invocations will not change too much. Therefore, efficient conflict resolution is especially important for high-performance parallel scientific applications using this model.

Conflict Resolution

Typical conflict resolution approaches are based on rule priorities defined through implicit textual ordering of rules or explicit precedence relationships. However, this approach can introduce additional logic complexities and overheads, making it difficult to determine rule priorities and to make general inferences about the behaviors of conflicting rules under various circumstances [11]. The Accord programming system enables users to explicitly set rule priorities, and further provides mechanisms to assist in resolving conflicts among rules with the same priority.

Detecting and resolving $S - A$ conflicts:

if $SA \neq \phi$ *then*

- *for each rule* $R_i \in \bar{R}$
  - *if* $SA \bigcap A_i \neq \phi$, *and* $\exists s_i \in SA \bigcap S_i$ *and* $a_i \in SA \bigcap A_i$ *and* $a_i = s_i$, $Value(s_i) \neq Value(a_i)$, *then delete* R_i *from* $\bar{R}$.

In Accord, one variable or parameter can be exposed both as a sensor and an actuator. $S-A$ conflicts happen when at least one variable or parameter belongs to both the *pre-condition* and *post-condition* but with different values. Rules that contain this variable or parameter in the action part will be disabled to avoid changing the *pre-condition*.

Detecting and resolving $A - A$ conflicts:

$A - A$ conflicts happen when an actuator will be invoked with different values. Resolving $A-A$ conflicts requires users to define a sequence of sensors (i.e., $CS = \{cs_i\}$). The resolution

algorithm relaxes the *pre-condition* by incrementally 'deleting' sensors in CS from the *pre-condition*, until each actuator in the *post-condition* has one value or all the sensors in CS are exhausted.

if $\bigcap A_i \neq \phi, \forall R_i \in \bar{R}'$, $\exists a \in \bigcap A_i, \bigcap Value_i(a) = \phi$ *then*

- *repeat*
  - *read the next* cs *from* CS
  - *relax* cs *in the pre-condition*
  - *re-evaluate rules*
- *until* $\forall a \in \bigcap A_i, Value(a)$ *has at least one value, or* CS *is exhausted.*
- *if* CS *is exhausted, an error is reported to users for further instructions; otherwise, the post-condition* $\{A, VA\}$ *is constructed by randomly selecting a value for those actuators having multiple values.*

Reconciliation

Reconciliation is required to generate a consistent post-condition in parallel SCMD applications as each processing node may independently generate a different post-condition based on its local computation, data and execution context. Different reconciliation strategies are defined for behavior rules and interaction rules. Reconciliation will be discussed in Chapter 5.

3.4 Autonomic Adaptation Behaviors in Accord

Accord enables two levels of adaptation behaviors, behavior adaptation and composition adaptation, discussed in the following sections. They can be used separately or in combination to enable the autonomic self-configuring, self-optimizing and self-healing behaviors of elements and applications.

3.4.1 Adapting Element Behaviors

At the element level, Accord treats functions, variables, and parameters as adaptation units. Element managers monitor behaviors of individual elements by invoking sensors and listening to events, evaluate rule conditions, and perform actions through actuator invocation and

event generation. The behavior adaptation of one element is functionally transparent to other elements, since the changes do not effect its functional syntax and semantics. However, the changes will effect its non functional behaviors such as execution time, memory usage, and bandwidth consumption.

3.4.2 Adapting Element Composition

At the application level, Accord treats elements as adaptation units. It enables dynamic composition of autonomic elements, which 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.

Element Composition

The composition of autonomic elements consists of *selection of elements* and *definition of interactions among these elements*.

- *Selection of elements* describes who is interacting, based on the composition of functional ports (Γ), and can be defined as:

  $E_0 \propto_\Gamma \bigcup E_i, \exists \Gamma_{E_0,u} \subseteq \bigcup \Gamma_{E_i,p}$

  where, E_0 is an autonomic element, $\bigcup E_i$ is a set of autonomic elements, $\propto_\Gamma$ denotes the relation “be functionally compose-able with”, $\Gamma_{E_0,u}$ is the functions used by element E_0, and $\bigcup \Gamma_{E_i,p}$ represents the functions provided by the element set $\bigcup E_i$. This definition says that element E_0 is functionally compose-able with elements $\bigcup E_i$, when $\bigcup E_i$ can provide all the functions required by E_0. Function composition can be based on semantics, which is currently being investigated in semantic web community using OWL [9], or based on syntax description, which is used by component frameworks such as Ccaffeine [14] and service frameworks such as the Web service architecture.

- *Interactions among elements* define how and when elements interact such as the interaction mechanism (messaging, shared-memory, tuple-space) and coordination model (data-driven or control-driven). For example, CCAFFEINE [14] defines interactions as function calls, CORBA [1] uses remote method invocations, and Web and Grid services [31]

communicate using XML messages. Interactions may be triggered by an event or actively initiated by an element.

Dynamic Composition

Once a workflow has been generated (e.g., using the mechanism in [12]) and the elements have been discovered using middleware services, the composition manager decomposes the workflow into interaction rules. This decomposition process consists of mapping workflow patterns [72] in the workflow into corresponding rule templates, and defining the required parameters for the templates. 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 decentralized composition enables autonomic elements to fully exploit explicit and implicit parallelism in the application workflow. The execution sequence among autonomic elements is caused by data dependency when elements have to wait for the data inputs from their interacting elements, and/or control dependency when there is no data exchange and the waiting is required by control flow, e.g., synchronization. Elements without these dependencies can be executed in parallel. Workflow decomposition and decentralized execution is especially useful for large scale applications whose parallelism cannot be completely discovered a priori and/or manually, and therefore enables the efficient execution of their workflow.

The Accord framework supports dynamic composition 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 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 related elements. The old element is then deleted, as described 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 and deleting elements: To add a new element, the composition manager creates a new element manager, initializes it with the 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, and 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 interaction rules enables users to adjust the workflow at runtime without recompiling and 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 and optimized to manage the dynamism and uncertainty of the applications and environments.

3.5 Autonomic Forest Fire Application: An Illustrative Example

In this section, we use the autonomic forest fire application [36] to illustrate the Accord programming system. 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 notify *Rothermel* of the divided 2D space.

- *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 processes 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.

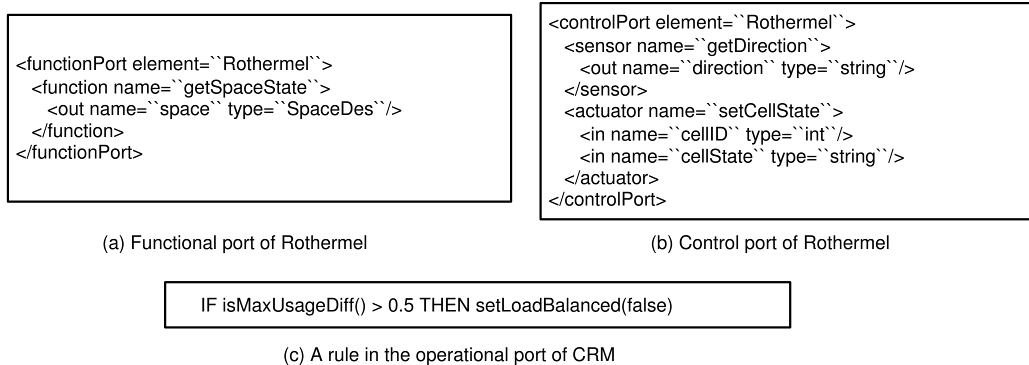
- *WindModel*: *WindModel* simulates the wind direction and intensity.

- *GUI*: Experts interact with the above 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. The process continues until no *burning* cells remain.

3.5.1 Defining Autonomic Element

We use the *Rothermel* and *CRM* as examples to illustrate the definition of functional, control and operational ports, shown in Figure 3.4.



(a) Functional port of Rothermel

(b) Control port of Rothermel

(c) A rule in the operational port of CRM

Figure 3.4: 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 3.4 (a). The function *getSpaceState* generates information about the space. The data structure *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 forwarded to the user. An example of control port is shown in Figure 3.4 (b).

Operational Port: The operational port contains the rules that are used to manage the runtime behavior of a element. The rules may be defined at runtime and injected into the element, and will be executed by the rule agent embedded in the autonomic element. An example *behavior rule* in *CRM* may be shown in Figure 3.4 (c). 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.

3.5.2 Enabling Adaptation Behaviors

The adaptation behaviors for the autonomic forest fire application enabled by Accord are illustrated bellow.

Adapting *DSM* Behaviors

DSM has two 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 3.5.

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

Figure 3.5: A behavior rule for *DSM*.

Adding A New Element

A new element, *FFM* (*Fire Fighter Model*)that models the behaviors of the fire fighters, may be added into the primary workflow. This element dynamically changes the cell state and informs *Rothermel*. This addition of a new element at runtime is achieved by the composition manager inserting interaction rules into both *FFM* and *Rothermel*, shown in Figure 3.6. The two elements will automatically establish the interaction based on the rules.

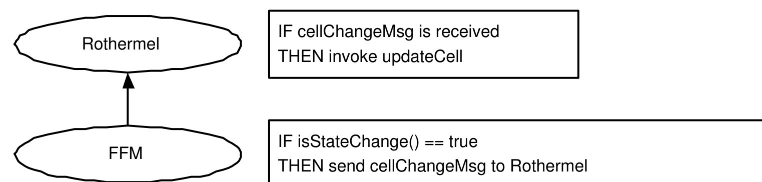


Figure 3.6: Add a new element *FFM*.

Changing Interaction Relationships

CRM can dynamically decrease the frequency of notifications to *DSM* when the communication network is congested. This adaptation behavior can be enabled by the rules shown in Figure 3.7.

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

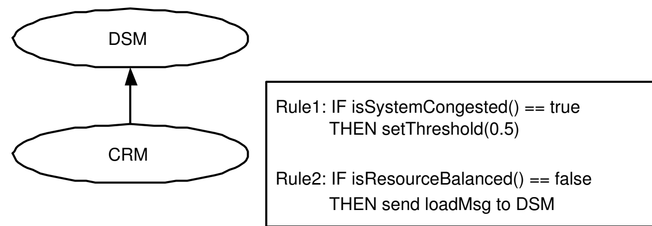


Figure 3.7: Change the interaction relationship between *CRM* and *DSM*.

3.6 Summary

In this chapter we presented the Accord programming system that extends existing programming systems with adaptation mechanisms to support the development and execution of autonomic self-managing applications. Accord includes the common semantic basis, the definition of autonomic elements as the building blocks of autonomic applications, the definition of rules and mechanisms to enable autonomic elements, and the runtime infrastructure that enforces rules to realize adaptation behaviors. Further, the adaptation behaviors enabled by Accord are presented and illustrated using an autonomic forest fire application.

Chapter 4

DIOS++: Autonomic Object-based Accord

An object based prototype of Accord, named DIOS++, has been implemented and evaluated in the context of distributed scientific/engineering simulations as part of the DIOS++/DISCOVER project [1]. DISCOVER enables geographically distributed clients to collaboratively access, monitor, and control Grid applications using pervasive portals. It is currently being used to enable interactive monitoring, steering and control of a wide range of scientific applications, including oil reservoir, compressible turbulence and numerical relativity simulations.

DIOS++ supports the rule-based autonomic monitoring and control of distributed and parallel applications. It enables high-level rules to be dynamically composed and securely injected into applications at runtime, allowing applications to manage and autonomically optimize their execution. Rules specify conditions to be monitored and operations that should be executed when certain conditions are detected. Rather than continuously monitoring and steering the simulations, experts can define and deploy appropriate rules that are automatically evaluated and executed at runtime to manage the computation, apply runtime corrections based on the observed state, and optimize application execution.

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

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

agents in a distributed and parallel manner.

DIOS++ builds on the DIOS [48], a distributed object substrate for interactively monitoring and steering parallel scientific simulations. DIOS++ extends DIOS with an agent-based framework that enables rule-based autonomic management. This alleviates time- and effort-consuming interactive monitoring and control and enables richer self-management behaviors. DIOS++ also provides an object-level access control mechanisms.

Note that in this prototype, autonomic elements are implemented as autonomic objects, the operational port is implemented as the access interface and rule interface, the role of composition manager is taken by the Gateway and the rule engine, and the Accord portal is implemented by the DISCOVER server and portals. Only behavior rules are defined and used. Interaction rules are not supported.

4.1 Autonomic Monitoring and Control with DIOS++

Rule-based autonomic monitoring and control enhances traditional computational steering and enables long-term, complex, computation- and resource-intensive applications to monitor and steer themselves based on user-defined high-level rules. This may include requesting or modifying program state, pausing program execution, calibrating the runtime behaviors of the application, exploring new computational solutions for problems that are not yet well understood, adapting programs to the current execution environments, etc.

DIOS++ enables autonomic monitoring and control by enhancing the agent based approach with high-level rules that incorporate human knowledge. Key research issues addressed by DIOS++ include:

- Integrating monitoring and steering functionalities with applications: To realize external monitoring and steering capabilities, a small amount of modification to the application source code is required. The objects to be monitored and steered must explicitly expose sensors and actuators. In case of object oriented applications, this consists of invoking the APIs provided by DIOS++ to expose their internal variables, parameters, and functions.

  Applications written in procedural languages need to transform their data structures to

objects using, for example, C++ wrappers. Although this requires some application modification, the wrappers are only required for those data-structures that need to be managed and the effort required is far less than rewriting the entire application.

- Rich monitoring and steering capabilities: Synchronous and asynchronous monitoring and steering are enabled. In the case of synchronous control, DIOS++ performs real-time management behaviors responding to users' runtime requests. In the case of asynchronous control, users define and submit rules to DIOS++, and DIOS++ performs management behaviors when rule condition is met.

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

- Collaboration: Using the DISCOVER server, users can form or join collaboration groups and interact with one or more applications based on their capabilities. Users in one collaboration group can selectively receive or broadcast application information.

4.2 DISCOVER Collaboratory

The DISCOVER collaboratory (shown in Figure 4.1) provides a virtual, interactive, and collaborative Problem Solving Environment (PSE) that enables geographically distributed scientists and engineers to collaboratively monitor and control high-performance parallel/distributed applications. It consists of the DISCOVER server as the front-end and DIOS++ architecture as the back-end.

The DISCOVER server builds on a traditional web server and extends its functionality to

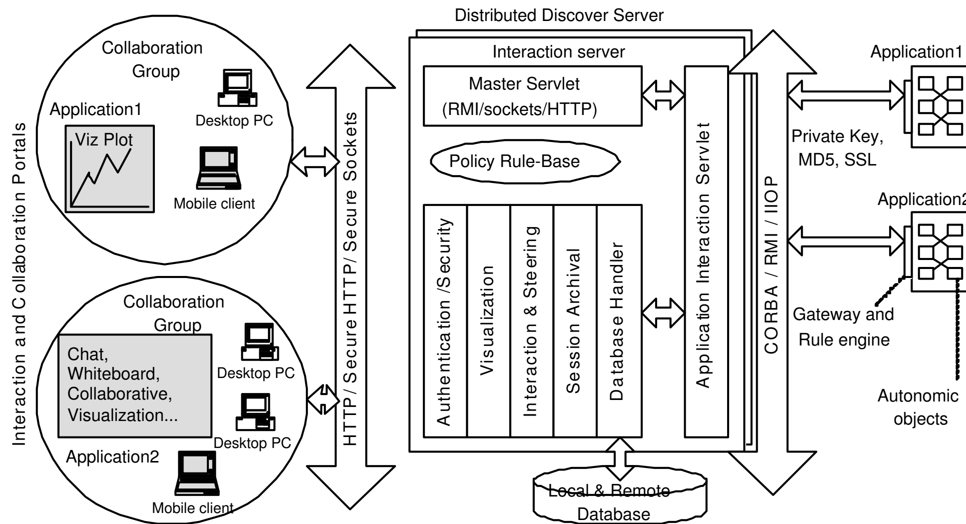


Figure 4.1: DISCOVER collaboratory architecture.

handle realtime application information and client requests using "handler" servlets that provide interaction, collaboration, and rule services. The DISCOVER server provides each registered client with a unique client-id, and each registered application with a unique application-id. The client-id along with an application-id is used to identify each session. To start interaction behaviors, users must be authenticated by the authentication handler, which builds a customized interaction interface for each valid client to match his/her access capabilities. This ensures that the client can only access, interact with, and steer applications in an authorized way. In DISCOVER, clients must explicitly request and release locks before and after steering behaviors. In the back-end DIOS++ architecture, a similar locking mechanism is used to protect multiple rule agents from invoking the same actuators simultaneously. Rules with high priority will lock these actuators when the conditions specified in the rules are satisfied. The locks are released when rules with higher priority disable these rules, or the conditions are no longer satisfied.

DISCOVER enables multiple users to collaboratively interact with and steer applications. All clients connected to a particular application form a collaboration group by default. Global updates (e.g. current application status) are automatically broadcast to this group. Clients can selectively broadcast application information to the group. Further, they can select the type of information that they are interested in. In addition, each application portal is provided with chat and whiteboard tools to further assist collaboration.

4.3 DIOS++ Architecture

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

4.3.1 Autonomic Object

In addition to its functional interface, an autonomic object (shown in Figure 4.2) exports three interfaces: (1) a *control interface*, which defines sensors and actuators to allow the object's state to be externally monitored and controled, (2) an *access interface*, which controls access to the sensors/actuators and rule interfaces, and describes users' access privileges based on their roles and the object's state, and (3) a *rule interface*, which contains rules used to autonomically monitor and control the object, and provides methods for adding, modifying and deleting rules. Rule operations are handled by the rule agent embedded within the autonomic object. These interfaces and the rule agent are described in the following sections. A sample object that generates a list of random integers (*RandomList*) is used as a running example. The number of integers and their range can be set at runtime.

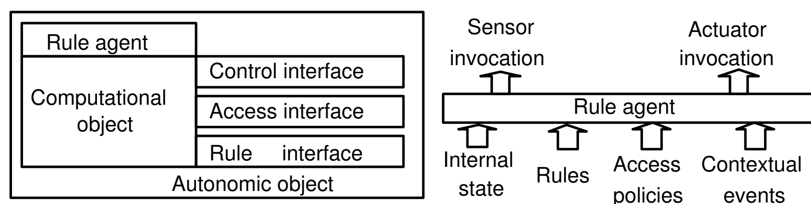


Figure 4.2: An autonomic object.

Control Interface

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

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

Access Interface

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

The DIOS++ defines three roles: owner, member, and guest. Each user is assigned a role based on her/his credentials. The owner can define/modify access policies, and enable or disable external access to sensors/actuators and rule interfaces. The polices define which roles can access a sensor, actuator and rule interface, and in what way. Access polices can be defined statically during object creation using the DIOS++ APIs, or can be injected dynamically by the owner at runtime using the secure DISCOVER portal. Objects can dynamically change their access policies based on their current state without affecting other objects. Therefore, a user may be denied of access in one object, while maintaining access privileges for another object.

Rule Interface

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

For example, as shown in Figure 4.3, consider the *RandomList* object with 2 sensors: (1) *getLength()* to get the current length of the list, and (2) *getMaxValue()* to get the maximal value in the list, and an actuator *append(length, max, min)* that creates a list of size *length* with random integers between *max* and *min*, and appends it to the current list.

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

Figure 4.3: A sample rule for *RandomList*.

Note that rules are separated from the application logic and can be created, deleted and modified at runtime orthogonal to the application execution. This provides flexibility, allowing users to monitor and control the application execution, without stopping and restarting the application. Rules are handled by rule agents and the rule engine, which are part of the control network described in the following section.

Rule Agent

A rule agent is embedded within each autonomic object. The rule agent receives rules from the rule engine through rule interfaces, authenticates the user defining the rules, evaluates and executes the rules based on the internal and contextual state to dynamically monitor and steer

its host object by invoking appropriate sensors and actuators. Multiple rule agents may coordinate with each other to provide collaborative steering behaviors accessing multiple autonomic objects.

4.3.2 Control Network

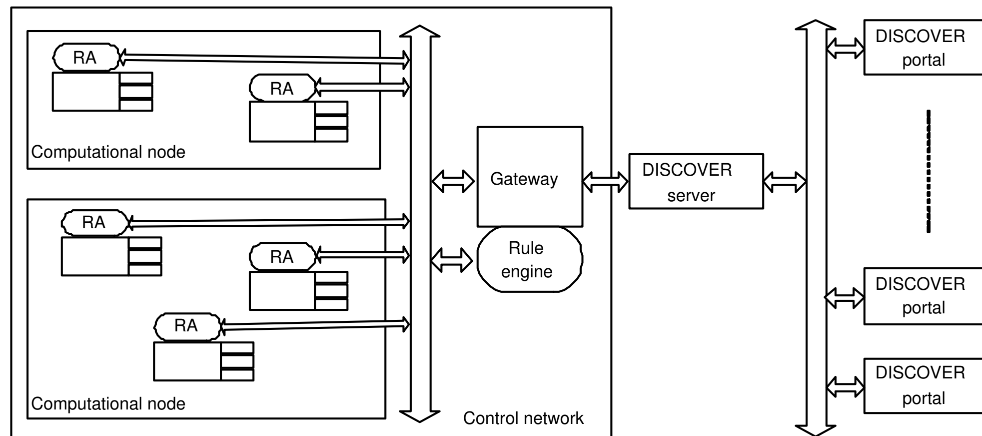


Figure 4.4: The DIOS++ control network.

The DIOS++ control network (see Figure 4.4) is a hierarchical structure consisting of the rule engine and Gateway, and computational nodes. It is automatically configured at runtime using the underlying messaging environment (e.g. MPI) and the available processors.

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

Co-located with Gateway, the rule engine accepts and maintains the rules for the application. It decomposes these rules and distributes them to the corresponding rule agents, collects rule execution results from rule agents and reports them to the users. Each rule agent executes its rules based on an execution script, and reports the rule execution results to the rule engine.

The execution script is defined by the rule engine to specify the rule execution sequence and the rule agent's runtime behaviors. The specification and execution of scripts and the coordination between the rule engine and rule agents are illustrated in the following sections.

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

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

Initialization

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

Interaction and Rule Operation

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

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

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

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

Rule deployment: Consider the rules in Figure 4.5. Let Rule1 have a higher priority than Rule2:

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

Figure 4.5: Rule1: an object rule involving only one object *RandomList*. Rule2: an application rule involving two objects *RandomList* and *SortSelector*.

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

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

In the case of an object rule, the rule engine just injects the object rule into its corresponding rule agent, as shown in Figure 4.6 (a). In the case of an application rule, the rule engine will first decompose the rule into triggers and then inject triggers into corresponding agents. For example, the application rule ‘Rule2’ is decomposed into 3 triggers: (1) *SortSelector.sequentialSort()*, (2) *SortSelector.quickSort()*, and (3) *RandomList.getLength()* < 50. These triggers are injected into corresponding agents as shown in Figure 4.6 (b).

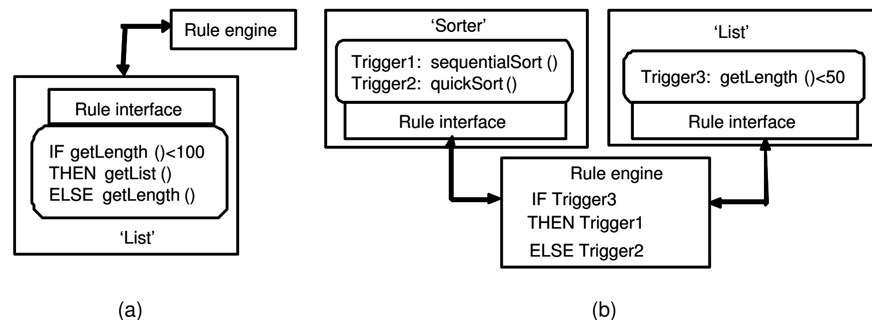


Figure 4.6: (a): Deployment of an object rule. (b): Deployment of an application rule.

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

While typical rule execution is straightforward (actions are issued when their required conditions are fulfilled), the application dynamics and user interactions make things unpredictable. As a result, rule conflicts must be detected at runtime. In DIOS++, rule conflicts are detected

at runtime and are handled by grouping rules based on their priority and disabling conflicting rules with lower priorities. This is done by locking the required sensors and actuators. For example, suppose that a user defines two rules for the object instance *RandomList* shown in Figure 4.7. Rule3 requires setting the minimal integer value to 5 when the list length is less than 100 and larger than 50, and Rule4 requires the minimal value to be 6 when the list length is larger than 30 and less than 70. Rule3 has higher priority than Rule4. The two rules conflict with each other, for example, when the list length is 60.

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

Figure 4.7: Rules with conflicts.

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

4.4 The Autonomic Oil Reservoir Application: An Illustrative Example

In this section, we use the oil reservoir simulation application [45] to illustrate the ideas described in this chapter. The application optimizes the placement and operation of oil wells to maximize overall revenue. The application consists of the instances of distributed multi-model, multi-block reservoir simulation components provided by the IPARS, simulated annealing based optimization services provided by the VFSA, economic modelling services, real-time services providing current economic data (e.g. oil prices), historical data archives, and experts (scientists, engineers) connected via collaborative portals. During initialization, experts configure and launch the IPARS factory and the VFSA optimization service. In the iterative optimization phase, the IPARS factory gets initial guess from the VFSA and launches an IPARS instance, which uses the Economic Model along with current market parameters to estimate the current revenue. This revenue is normalized and then communicated to the VFSA service,

which in turn uses this value to generate an updated guess of the well parameters and sends this to the IPARS Factory. The IPARS Factory now configures a new instance of IPARS with the updated well parameters and deploys it. This process continues until the required terminating condition is reached (e.g. revenue stabilizes).

The IPARS instance exposes its input parameters (well parameters) and physical models as actuators. Similarly, the VFSA exposes its input parameters (the revenue) and probability value as actuators.

DIOS++ enables directly modifying parameters exported by objects. For instance, modification of the probability value of the VFSA will increase or decrease the process time required to find a global minimum. Consistency of these steering behaviors is guaranteed through the actuator constraints specified in the rules that are embedded inside the objects. These constraints will automatically restrict the values to be within a valid range. For instance, a constraint is defined to maintain the probability value between 0 and 1. When a user tries to set the probability to an invalid value, the constraint will reject the request and send an error message to the user, shown in Figure 4.8.

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

Figure 4.8: The constraint in VFSA that maintains the probability value between 0 and 1.

Let us examine a more complex case that involves multiple objects. Suppose IPARS provides two algorithms, algorithms1 that generates a result with higher precision but is resource-consuming, and algorithm2 that generates a result with lower precision but consumes less resources. IPARS begins with algorithm2 and then use algorithm1 when the revenue approaches some pre-defined threshold to achieve the best performance in terms of precision under conditions of limited computational resources. The rule is specified in Figure 4.9:

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

Figure 4.9: A sample application rule involving VFSA and IPARS.

This rule is decomposed into one sensor and two actuators: *sensor1* "VFSA.revenue < threshold", *actuator1* "IPARS.algorithm2()" and *actuator2* "IPARS.algorithm1()". *sensor1* is injected into the VFSA rule agent; *actuator1* and actuator2 are injected into the IPARS rule agent. When *sensor1* is triggered, IPARS rule agent will be notified and *actuator1* or *actuator2* will be executed. The rule will be automatically evaluated and executed to configure IPARS.

In DIOS++, monitoring and steering behaviors may be synchronous or asynchronous. Synchronous monitoring and steering is a one-time behavior (an example could be the modification of probability value in VFSA) responding to users' realtime requests, while in asynchronous monitoring, steering behaviors are performed whenever condition is satisfied during the life time of an application (an example could be the complex case discussed above).

4.5 Experimental Evaluation

This section summarizes the experimental evaluation of the DIOS++ library using the IPARS reservoir simulator framework on the beowulf cluster. The cluster contains 64 Linux-based computers connected by 100 Mbps full-duplex switches. Each node has an Intel(R) Pentium-4 1.70GHz CPU with 512MB RAM and is running Linux 2.4.20-8 (kernel version). IPARS is a Fortran-based framework for developing parallel/distributed reservoir simulators. Using DIOS++/DISCOVER, engineers can interactively feed in parameters such as water/gas injection rates and well bottom hole pressure, and observe the water/oil ratio or the oil production rate. The evaluation consists of 3 experiments:

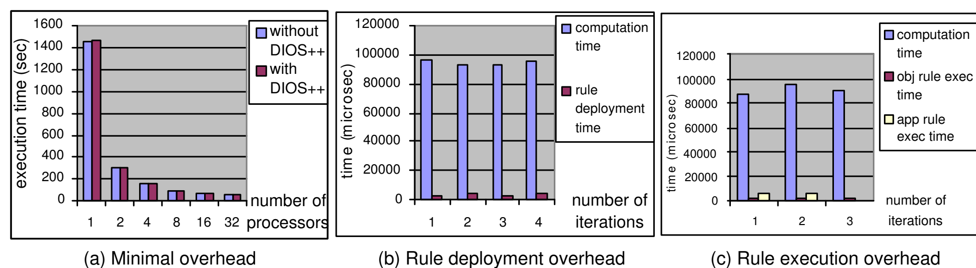


Figure 4.10: DIOS++ experimental evaluations.

Experiment 1 (shown in Figure 4.10 (a)): This experiment measures the runtime overhead introduced by DIOS++ in minimal rule execution mode. In this experiment, the application automatically updates the DISCOVER server and its connected clients with the current state

of autonomic objects and rules. Explicit interaction and rule execution are disabled during the experiment. The application's runtime with and without DIOS++ are plotted in Figure 4.10 (a). It can be seen that the runtime overhead due to DIOS++ is very small and within the error of measurement.

Experiment 2 (shown in Figure 4.10 (b)): This experiment compares computation time and the average rule deployment time for successive iterations. In this experiment, we deployed object rules in the first and third iterations, and application rules in the second and fourth iterations. The experiment shows that object rules need less time than application rules. This is true since the rule engine only has to inject object rules to corresponding rule agents, while it has to decompose application rules to triggers, and inject triggers to corresponding rule agents.

Experiment 3 (shown in Figure 4.10 (c)): This experiment compares computation time, average object rule execution time and average application rule execution time for successive iterations. The experiment shows that application rules require longer execution time than object rules, since the rule engine has to collect results from all the triggers, check whether the conditions are fulfilled and invoke corresponding actions. The execution of both application rules and object rules involves querying sensors, evaluating conditions, resolving conflicts, and invoking actuators. Sensor queries, condition evaluation and actuator invocation can be done in parallel. As a result, these overheads are not significantly impacted by the size of the rule base. However, conflict resolution overhead does increase with the size of the rule base.

4.6 Summary and Conclusion

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

Rules can be defined, modified and deleted at runtime. They are evaluated and executed

in a distributed and parallel manner by rule agents embedded within autonomic objects, to automatically adjust the runtime behaviors of applications. Besides, these rules are defined in a simple "IF-THEN-ELSE" format and can be used with many different applications. The experimental evaluation presented in the chapter demonstrates that DIOS++ overheads are small and the architecture is scalable.

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

Chapter 5

Accord-CCA: Autonomic Component-based Accord

This chapter presents the component-based prototype of Accord based on the DoE Common Component Architecture (CCA) and the Ccaffeine framework [14] in the context of component-based high-performance scientific applications. Specific contributions include: (1) extension of CCA to enable the definition of self-managing components and applications; (2) design and implementation of a runtime framework to support self-management behaviors using dynamically defined rules; (3) implementation of the three-phase rule execution model to enable consistent and efficient rule execution for distributed/parallel scientific applications; and (4) support for performance driven self-management using the TAU framework [5]. Compared to DIOS++, which enables function oriented adaptation at object and application levels, this prototype supports both function and performance (using TAU utilities [5]) oriented adaptation, enables dynamic composition by replacing components at runtime, and provides consistent and efficient rule execution for intra- and inter-component adaptation behaviors. The self-managing shock hydrodynamics simulation and CH_4 ignition simulation are presented as case studies.

Note that in this prototype, autonomic elements are implemented as autonomic components, the control port is constructed by exposing sensors and actuators via the CCA RulePort, the operational port is implemented by component managers. The behavior rules are named component rules, and the interaction rules are named composition rules.

5.1 Component-Based Distributed/Parallel Scientific Applications

5.1.1 The Common Component Architecture (CCA)

Component-based software architectures address some of the key requirements of emerging high-performance parallel/distributed scientific applications. Specifically, the DoE Common Component Architecture (CCA) and its implementation, the Ccaffeine framework [14], have been successfully used by a number of applications [35, 41, 40]. CCA supports the provides-uses design pattern. Components *provide* functions and *use* other components' functions via

ports. Components are peers and independently developed. Further, CCA employs the *Single Component Multiple Data (SCMD)* model, where all processing nodes execute the same program structure.

Ccaffeine [14], developed at Sandia National Laboratories, implements the CCA core specification and provides the fast and lightweight glue to integrate external and portable peer components into a SCMD style parallel application. Components are created and exist within the Ccaffeine framework. They register themselves and their ports with the framework and are dynamically loaded and connected. As a result, the Ccaffeine framework maintains complete knowledge about an application. Further, all the components on the same processor reside in the same address space and these components interact with each other using method calls. Component interaction across processors use MPI [6].

5.1.2 Behavior and Performance of Component-based Scientific Applications

The component-based programming approach not only reduces the burden of developing scientific applications, but also benefits their runtime management. With componentization [14], the behavior and performance of an application can be interpreted as a composition of individual components. For example, the composite performance of a component assembly is determined by the performance of the individual components and the efficiency of their interaction [59]. Therefore, management behaviors can be systematically enforced at two separate levels - intra-component and inter-component.

The execution of scientific applications typically consists of a series of computational phases. Between two successive phases, computations within components and communications between components are paused, and the components are reconfigured for the next phase. This pause between phases has been called a *quiet interval*. Runtime management is usually performed during these *quiet intervals* to ensure the integrity of the numerical computations. Changes made to components/applications during *a quiet interval* are automatically applied in the next computational phase.

Finally, in case of the Ccaffeine framework, due to the underlying SCMD model, connections between components can be made by directly passing ports (i.e., pointers to pure virtual interfaces), which incur negligible overheads [14]. As a result, the overall performance

of an application can be simply viewed as a function of the performance of its constituent components. Further, in case of scientific applications, the performance of a component is dominated by the cache performance of its implementation and the cost of inter-processor communications [59]. Cache performance is defined by the degree of data locality in computation algorithms and is affected by the cache size and cache management strategies used by the execution environment. Inter-processor communication costs are defined by software and algorithmic strategies used by the implementation (e.g., combining communication steps, minimizing/combining global reductions and barriers, overlapping communications with computations, etc.), and are affected by factors such as load-balance and communication channel congestion (due to competing application or possibly malicious attacks).

5.2 Self-management of Component-based Scientific Applications

As mentioned in Chapter 1, addressing the challenges of emerging high-performance scientific applications requires a programming system that enables the specification of applications, which can detect and dynamically respond, during their execution to changes in both the execution environment and application state. This requirement suggests that: (1) applications should be composed from discrete self-managing components, 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 compose-able; (3) the interface definitions of these components should be separated from their implementations to enable heterogeneous components to interact and to enable dynamic selection of components.

Component-based scientific simulations and the CCA architecture address some of these requirements and support application maintainability and extensibility. The capability of dynamically swapping components has been incorporated into the CCA specification and implemented by the Ccaffeine framework. However, enabling self-managing components/applications requires extending CCA to enable components that can adapt their behaviors and interactions based on their current state and execution context in an autonomic manner. In this section we

describe an extension of the CCA architecture, and specifically the Ccaffeine framework [14] using Accord, to support self-management. This consists of extending CCA components (including legacy components) to support monitoring and control, and extending the Ccaffeine framework to support consistent and efficient rule-based intra-component and inter-component self-management behaviors.

5.2.1 Defining Managed Components

In order to monitor and control the behaviors and performance of CCA components, the components must implement and export appropriate “sensor” and “actuator” interfaces. Note that the sensor and actuator interfaces are similar to those used in monitoring/steering systems [33, 60, 61]. However, these systems focus on interactive management where users manually invoke sensors and actuators, while this research focuses on automatic management based on user-defined rules. Adding sensors requires modification/instrumentation of the component source code. In case of third-party and legacy components, where such a modification may not be possible or feasible, proxy components [59] are used to collect relevant component information. A proxy provides the same interfaces as the actual component and is interposed between the caller and callee components to monitor, for example, all the method invocations for the callee component. Actuators can similarly be implemented either as new methods that modify internal parameters and behaviors of a component, or defined in terms of existing methods if the component cannot be modified. The adaptability of the components may be limited in the latter case. In the CCA based implementation, both sensors and actuators are exposed by invoking the ‘addSensor’ or ‘addActuator’ methods defined by a specialized *RulePort*, which is shown in Figure 5.1.

```
class RulePort: public virtual Port {
public:
	RulePort(): Port() { }
	virtual ~RulePort() { }
	virtual void loadRules(const char* fileName) throw(Exception) = 0;
	virtual void addSensor(Sensor *snr) throw(Exception) = 0;
	virtual void addActuator(Actuator *atr) throw(Exception) = 0;
	virtual void setFrequency() throw(Exception) = 0;
	virtual void fire() throw(Exception) = 0;
};
```

Figure 5.1: The *RulePort* specification.

Management and adaptation behaviors can be dynamically specified by developers in the form of rules. Two classes of rules are defined:

- *Component rules* address intra-component management. These rules manage the runtime behaviors of individual components, including dynamic selection of algorithms, implementations, data representation, input/output format used by the components, etc., based on the current state and execution context of the component.

- *Composition rules* address inter-component management. These rules manage the structure of the application and the interaction relationships among components based on the current application/system state, changing requirements, and changing execution context. Intra-component management behaviors include dynamic composition of components, definition of coordination relationships and selection of communication mechanisms. For example, composition rules can be used to add, delete or replace a component.

Management rules in this prototype incorporate high-level guidance and practical human knowledge in the form of conditional if-then expressions, i.e., IF *condition* THEN *action*. This simple construction of rules is deliberately used to enable efficient execution and minimize impact on the performance of the application. The *condition* is a logical combination of sensors (exposed by components) and performance data, and the *action* consists of a sequence of invocations of actuators exposed by components. The rules are interpreted and executed by the runtime framework, as discussed in the next section.

5.2.2 Enabling Runtime Self-management

To enable runtime self-management in this prototype, two specialized component types are defined (see Figures 5.2 and 5.3): (1) Component manager that monitors and manages the behaviors of individual components, e.g., selecting the optimal algorithms or modifying internal states, and (2) Composition manager that manages, adapts and optimizes the execution of an application at runtime. Both, component and composition managers are peers of user components and other system components, providing and/or using ports that are connected to other ports by the Ccaffeine framework. The two managers are not part of the Ccaffeine framework, and

consequently provide the programmers the flexibility to integrate them into their applications only as needed.

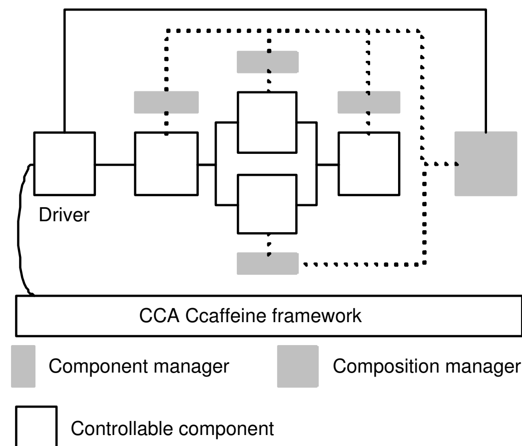


Figure 5.2: A self-managing application composed of 5 components. The solid lines denote computational port connections between components, and the dotted lines are port connections constructing the management framework.

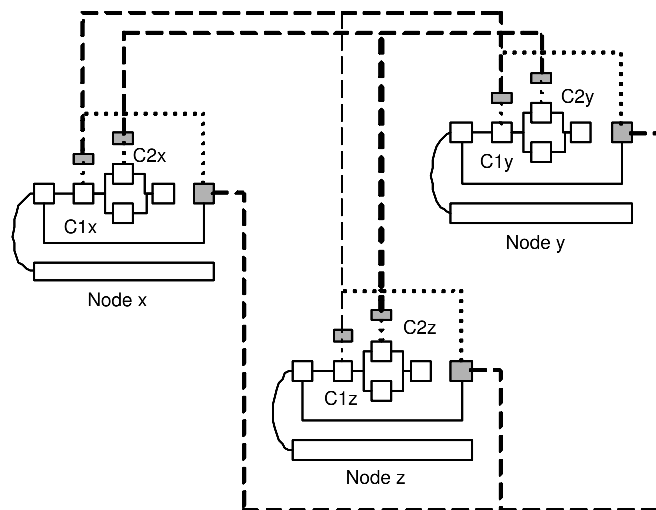


Figure 5.3: Distributed self-managing application shown in Figure 5.2 executed on three nodes. The solid lines across nodes denote the interactions among manager instances. The dotted lines are port connections constructing the management framework within one node.

The design of the component manager and composition manager components are based on the following observations and considerations.

- Scientific applications may contain tens of components, but only a few of them need to be dynamically monitored and controlled. Therefore, the manager functionalities are

encapsulated into two component types and provide programmers with the flexibility of integrating them with other components in the applications. For example, in Figure 5.3, only component $C1$ and $C2$ are associated with component managers for dynamic management.

- The manager functionalities are provided by components instead of being integrated within the Ccaffeine framework. This prevents the framework from being 'overweight' and thus avoids the resulting performance and maintenance implications.

- By encapsulating the manager functionality into these components and providing abstract interfaces for invoking this functionality, the manager functionality can be modified and improved without affecting other components and the framework. Additional functionality can be added into the manager components, and other components that deal with specific management functions can be created and integrated with the manager components using the 'uses-provides design pattern' [14].

Component Manager

Component managers provide the *RulePort* shown in Figure 5.1. They are instantiated only after the other application components are composed together. Their instantiation consists of two steps: first, instances of managed components expose their sensors and actuators to their respective component manager instances by invoking the 'addSensor' and 'addActuator' methods, and second, component rules are then loaded into component manager instances, possibly from disk files, by invoking the 'loadRules' method. This initialization of component manager instances is a one-time operation.

Management operations are performed during application *quiet intervals*. The managed components (or their proxies) invoke the 'fire' method of the *RulePort* to inform the component managers that they have entered into a quiet interval. This behavior must be explicitly programmed, possibly at the beginning/end of a computation phase or once every few phases, to establish the self-management frequency. Adaptations made during a quiet interval will be applied during the next computation phase.

Composition Manager

The composition manager also provides the *RulePort* (shown in Figure 5.1). Composition manager instances are initialized by the CCA driver component to load in composition rules (possibly from a disk file) using the 'loadRules' method. These rules are then decomposed into sub rules, and delegated to corresponding component managers. The driver component notifies composition manager instances of quiet intervals by invoking the 'fire' method. During execution of the composition rules, composition manager instances collect results of sub rule execution from component manager instances, evaluate the combined rule, and notify component managers of actions to be performed. Possible actions include adding, deleting, or replacing components. When replacing a managed component, the new component does not have to provide and use the exact same ports as the old one. However, the new component must at least provide all the active ports (those used by other components in the application) that are provided by the old component.

Rule Execution Model

The three-phase rule execution model discussed in Chapter 3 is used by the component managers to ensure consistent and efficient parallel rule execution.

During the batch condition inquiry phase, each component manager queries all the sensors used by the rules in parallel, gets their current values, and then generates the *pre-condition*. During the next phase, condition evaluation for all the rules is performed in parallel. Rule conflicts are detected at runtime when rule execution changes the *pre-condition* (defined as sensor-actuator conflicts), or the same actuator will be invoked with different values (defined as actuator-actuator conflicts). Sensor-actuator conflicts are resolved by disabling those rules that will change the *pre-condition*. Actuator-actuator conflicts are resolved by relaxing the pre-condition according to user-defined strategies until no actuator will be invoked with different values.

For example, consider component $C1$ 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 computation. This conflict is detected and resolved by relaxing the high-precision requirement, and therefore algorithm 3 can be selected.

Further, the framework also provides mechanisms for reconciliation among manager instances, which is required to ensure consistent adaptations in parallel SCMD applications, since each processing node may independently proposes different adaptation behaviors based on its local state and execution context. The reconciliation for component rules consists of identifying and propagating the actions proposed by a majority of the nodes. If a majority is not found, an error is reported to the user. Composition rules are statically assigned one of two priorities. A high priority means that the re-composition is necessary, while a low priority means the re-composition is optional. Actions associated with composition rules with high priority are propagated to all the nodes. If there are multiple high priority rules with collisions, a runtime error is generated and reported to the user. In case of actions associated with composition rules with low priority, a cost model is used to approximate the performance gain of each action set and the action set with the best overall gain is selected and applied by all the nodes.

After conflict resolution and reconciliation, the *post-condition*, consisting of a set of actuators and their new values, is generated. The *post-condition* is enforced by appropriately invoking the actuators in parallel during the batch action invocation phase.

Note that the rule execution model presented here focuses on correct and efficient execution of rules and provides mechanisms to detect and resolve conflicts at runtime. However, correctness of rules and conflict resolution strategies are responsibilities of the users.

5.2.3 Supporting Performance-driven Self-management

The TAU [5] framework provides support for monitoring the performance of components and applications, and is used to enable performance-driven self-management. TAU can record inclusive and exclusive wall-clock time, process virtual time, hardware performance metrics such

as data cache misses and floating point instructions executed, as well as a combination of multiple performance metrics, and help track application and runtime system level atomic events. Further, TAU is integrated with external libraries such as PAPI [3] or PCL [4] to access low-level processor-specific hardware performance metrics and low latency timers.

In our framework, TAU APIs are directly instrumented into the computational components, or into proxies in case of third-party and legacy computational components, and performance data is exported as sensors to component managers. Optimizations are used to reduce the overheads of performance monitoring. For example, as the cache-hit rate will not change unless a different algorithm is used or the component is migrated to another system with a different cache size and/or cache policies, monitoring of cache-hit rate can be deactivated after the first a few iterations and only re-activating when an algorithm is switched or the component is migrated. Similarly, inter-processor communication time is measured per message by default but this can be modified using the 'setFrequency' method in the *RulePort* to reduce overheads. Another possibility is to restrict monitoring to only those components that significantly contribute to the application performance. Composition managers can identify these components at runtime using mechanisms similar to those proposed in [68] and enable or disable monitoring as required. Finally, in case of homogeneous execution environments only a subset of nodes may be monitored.

5.3 Case Studies

The operation of the component based prototype is illustrated using two applications, (1) a self-managing hydrodynamics shock simulation and (2) a self-managing CH_4 ignition simulation.

5.3.1 A Self-Managing Hydrodynamics Shock Simulation

This application simulates the interaction of a hydrodynamic shock with a density-stratified interface. The system is modelled using the 2D Euler equation (inviscid Navier-Stokes). Details of the equations used and the interaction are presented in [58, 63, 64]. The governing equations (the compressible Euler equations) in conservative form are:

$$\mathbf{U}_t + \mathcal{F}(\mathbf{U})_x + \mathcal{G}(\mathbf{U})_y = 0 \tag{5.1}$$

where

$$\begin{aligned}
\mathbf{U} &= \{\rho, \rho u, \rho v, \rho e, \rho\zeta\}^T, \\
\mathcal{F}(\mathbf{U}) &= \{\rho u, \rho u^2 + p, \rho uv, (\rho e + p)u, \rho\zeta u\}^T, \\
\mathcal{G}(\mathbf{U}) &= \{\rho v, \rho uv, \rho v^2 + p, (\rho e + p)v, \rho\zeta v\}^T,
\end{aligned}$$

ρe is the total energy, related to the pressure p by $p = (\gamma - 1)(\rho e - \frac{1}{2}\rho(u^2 + v^2))$ and ζ is an interface tracking function. We have used the conservative level set formulation of Mulder et. al [51] to track the interface. The basic idea is as follows: Consider a function $\zeta(\mathbf{x}, t)$, which is defined everywhere in the domain. Then a particular value defines the interface. In our case, we initially use $\zeta(\mathbf{x}, 0) = +1(0)$ in the incident (transmitted) gas. We define the interface as $\zeta(\mathbf{x}, t) = 0.5$. The function $\zeta(\mathbf{x}, t)$ is governed by the partial differential equation $D\zeta/Dt = 0$, resulting in the last equation in the system above. We use the ideal gas law as the equation of state. The equations are solved on a uniform cell-centered mesh i.e. the mesh divides the domain into small rectangular cells and fluid variables are defined and indexed at the cell centers. In 1D, the equation would be solved as

$$\mathbf{U}^{n+1} = \mathbf{U}^n + \frac{\Delta t}{\Delta x}\left(\mathcal{F}^{n+1/2}_{i+1/2} - \mathcal{F}^{n+1/2}_{i-1/2}\right) \tag{5.2}$$

The Godunov method is used to determine $\mathcal{F}^{n+1/2}_{i+1/2}$ at the cell interfaces in order to evaluate the RHS. This involves transforming the equation at each cell into Riemann Invariants in the X and Y directions; constructing the states on the left and right of a cell interface using slope-limiters and upwinding. Since the left and right states are not identical, a Riemann problem [65] is setup, which is solved (iteratively) to obtain the fluxes $\mathcal{F}^{n+1/2}_{i+1/2}$. The construction of left and right states holds true for most finite volume methods; solving an exact Riemann problem could be substituted by a gas-kinetics scheme (*e.g.* Equilibrium Flux Method [56]).

Figure 5.4 shows the assembly of components for the CCA-based implementation of the simulation. The simulation uses structured adaptive mesh refinement. In this implementation, the Runge-Kutta time integrator (**RK2**) with an **InviscidFlux** component supplies the right-hand-side of the equation on a patch-by-patch basis. This component uses a **ConstructLRStates** component to set up a Riemann problem at each cell interface, which is then passed to **GodunovFlux** for the Riemann solution. A **ConicalInterfaceIC** component sets up the problem

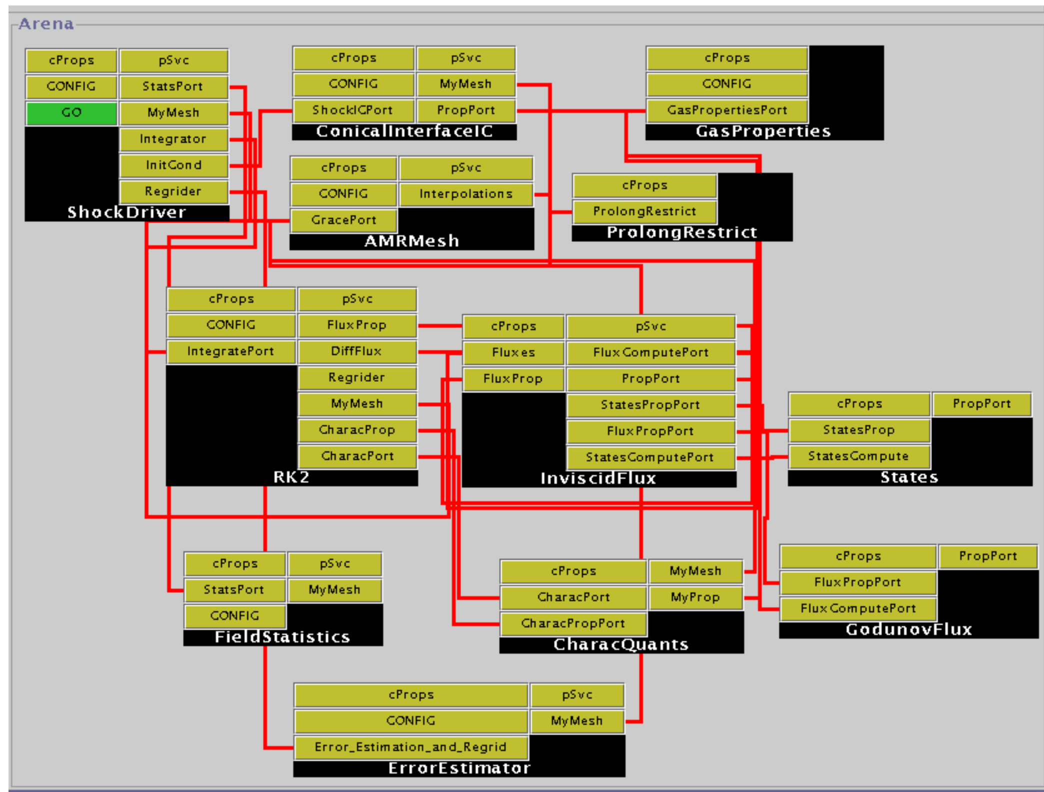


Figure 5.4: "Wiring" diagram of the shock-hydrodynamics simulation. A second-order Runge-Kutta (**RK**2) integrator drives **InviscidFlux** component – transformation into left and right (primitive) states is done by **States** and the Riemann problem solved by **GodunovFlux**. Sundry other components for determining characteristics' speeds (u + a, u - a, u), cell-centered interpolations etc. complete the code.

- a shock tube with Air and Freon (density ratio 3) separated by an oblique interface that is ruptured by a Mach 10.0 shock. The shock tube has reflecting boundary conditions above and below and outflow on the right. The **AMRMesh** and **GodunovFlux** are the significant components in this simulation from the performance point of view, and is used to illustrate self-managing behaviors in the discussion below.

Scenario 1: Self-optimization via component replacement

An EFM algorithm, which is based on a gas-kinetic scheme [56], may be used instead of the Godunov method with **RK2** in the implementation described above. **GodunovFlux** and **EFMFlux** demonstrate different performance behaviors and mean execution times as the size of the input array size increases, as shown in Figure 5.5. This difference in performance is primarily due to the difference in data locality and cache behaviors for the two implementations.

GodunovFlux is more expensive than **EFMFlux** for large input arrays.

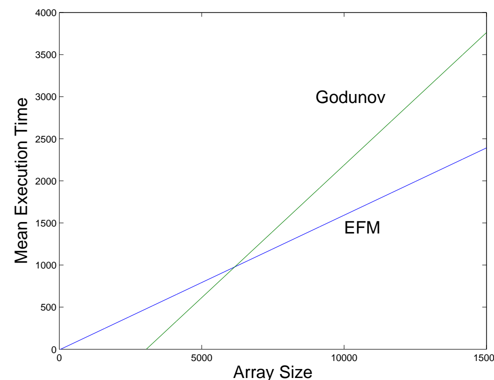


Figure 5.5: The average execution times for **EFMFlux** and **GodunovFlux** as functions of the array size (machine effects have be averaged out).

The appropriate choice of algorithm (Godunov or EFM) depends on simulation parameters, its runtime behaviors and the cache performance of the execution environment, and is not known a priori. In this scenario we use information about cache misses for **GodunovFlux** obtained using TAU/PCL/PAPI, to trigger self-optimization, so that when cache misses increase above a certain threshold, the corresponding instance of **GodunovFlux** is replaced with an instance of **EFMFlux**.

To enable the component replacement, one component manager is connected to the component **GodunovFlux** through the *RulePort* to collect performance data, evaluate rules, and perform runtime replacement. The component manager (1) locates and instantiates **EFMFlux** from the component repository, (2) detects all the provides and uses ports of **GodunovFlux**, as well as all the components connected to it, (3) disconnects **GodunovFlux** and delete all the rules related to **GodunovFlux**, (4) connects **EFMFlux** to related components and load in new rules, and finally (5) destroys **GodunovFlux**. The replacement is performed at a *quiet interval*. From the next calculation step, **EFMFlux** is used instead of **GodunovFlux**. However, other components in the application do not have to be aware of the replacement, since the abstract interfaces (ports) remain the same. After replacement, the cache behavior improves as seen in Figure 5.6.

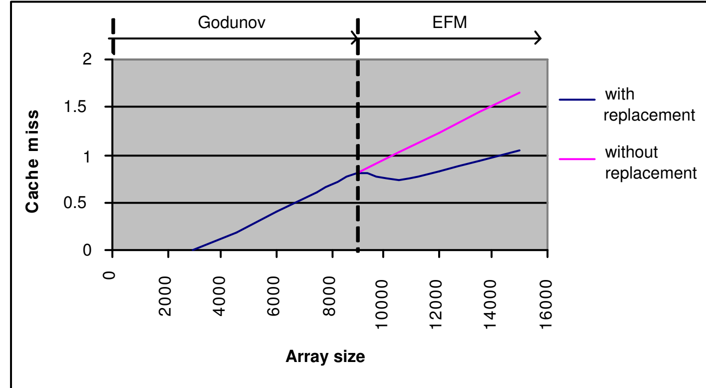


Figure 5.6: Replacement of **GodunovFlux** with **EFMFlux** to decrease cache misses.

Scenario 2: Self-optimization via component adaptation

The **AMRMesh** component supports structured adaptive mesh-refinement and provides two communication mechanisms. The first exchanges messages on a patch by patch basis and results in a large number of relatively small messages. The second packs messages from multiple patches to the same processor and sends them as a single message, resulting in a small number of much larger messages. Depending on the current latency and available bandwidth, the component can be dynamically adapted to switch the communication mechanism used.

In this scenario, we use the current system communication performance to adapt the communication mechanism used. As PAPI [3], PCL [4], and TAU [5] do not directly measure network latency and bandwidth, this is indirectly computed using communication times and message sizes. **AMRMesh** exposes communication time and message size as sensors, which are used by the component manager to get the current bandwidth as follows:

$$bandwidth = \frac{commTime_1 - commTime_2}{msgSize_1 - msgSize_2} \tag{5.3}$$

Here, '$commTime_1$' and '$commTime_2$' represent the communication times for messages with sizes '$msgSize_1$' and '$msgSize_2$' respectively. When the bandwidth falls below a threshold, the communication mechanism switches to patch by patch messaging (i.e., algorithm 1). This is illustrated in Figure 5.7. The algorithm switching happens at iteration 9 when channel congestion is detected, and results in comparatively smaller communication times in the following iterations.

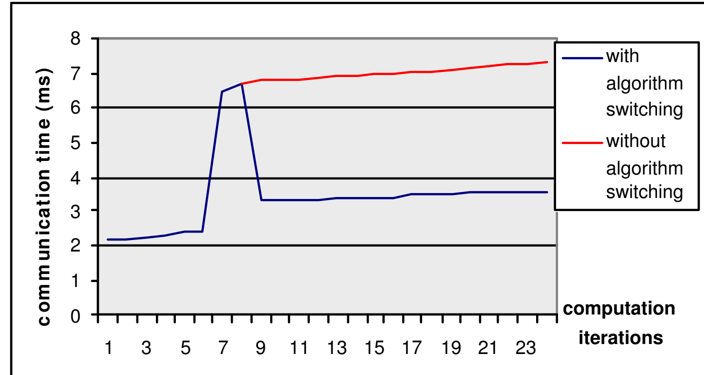


Figure 5.7: Dynamically switch algorithms in **AMRMesh**.

Scenario 3: Self-healing via component replacement

While Godunov methods with **RK2** tend to be more accurate, they become unstable for stronger shocks and larger density ratios. One solution is to replace **GodunovFlux** in these cases with **EFMFlux**. The appropriate choice of algorithm (Godunov or EFMFlux) depends on the Mach number and the density ratio, and is once again not known a priori. In the best of cases, an algorithm will operate for some time before failing to converge and indicating an error; at other times, it will work "reliably" and produce wrong (even qualitatively wrong) results. In the case where an error can be identified, we have the option of dynamically replacing one algorithm by another by simply replacing the component implementing the algorithm. Of course, the same change has to be performed on all the processors. While dynamically changing components does raise some fundamental issues (e.g. in this case, the simulation is neither purely EFM-based nor Godunov-based, and is not mathematically consistent either), it is expected that the results will be at least qualitatively correct. Since such simulations often require substantial computational resources, obtaining qualitative answers may be preferable to simply exiting with an error.

In this scenario we investigate the dynamic replacement of **GodunovFlux** with **EFMFlux** so that it continues to provide qualitatively correct results. The adaptation is triggered when **GodunovFlux** fails to converge, i.e., its iteration count increases above a certain threshold, and causes the instance of component **GodunovFlux** to be replaced by an instance of component **EFMFlux**. The replacement process is the same as that described in scenario 1 above.

5.3.2 A Self-Managing CH_4 Ignition Simulation

This section focuses on the overall performance improvement of the CH_4 ignition simulation. The ignition process is represented by a set of chemical reactions, which appear and disappear when the fuel and oxidizer react and give rise to the various intermediate chemical species. In the simulation application, the chemical reactions are modeled as repeatedly solving the ChemicalRates equation (G) [2] with different initial conditions and parameters using one of a set of algorithms called backward difference formula or BDFs. The algorithms are numbered from 1 to 5, indicating the order of accuracy of the algorithm. BDF_5 is the highest order method, and is most accurate and robust. It may, however, not always be the quickest. As a result, the algorithm used for solving the equation G has to be selected based on current condition and parameters. In this application, the bulk of the time is spent in evaluating the equation G. Therefore, reducing the number of G evaluations is a sufficient indication of speed independent of the experimentation environment.

As shown in Figure 5.8, the rule-based execution decreases the number of invocation to equation G, and the percentage decrease is annotated for each temperature value. It results in an average 11.33% computational saving. As the problem becomes more complex (the computational cost of G increase), the computational saving will be more significant.

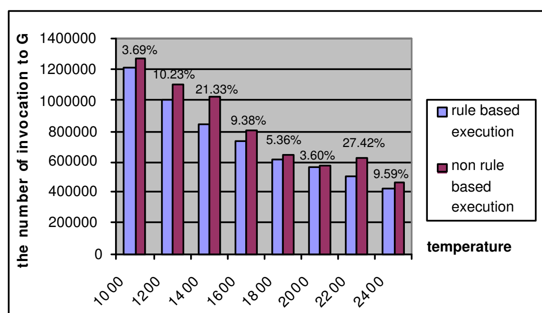


Figure 5.8: Comparison of rule based and non rule based execution of CH_4 ignition.

5.3.3 Experimental Evaluation

The prototype was evaluated on a Beowulf cluster. The cluster contains 64 Linux-based computers connected by 100 Mbps full-duplex switches. Each node has an Intel(R) Pentium-4

1.70GHz CPU with 512MB RAM and is running Linux 2.4.20-8 (kernel version). In this prototype, computational components were enhanced with sensors and actuators, and manager components were introduced into the application. The overheads associated with initialization of computational components and managers and the runtime execution of component and composition rules were evaluated.

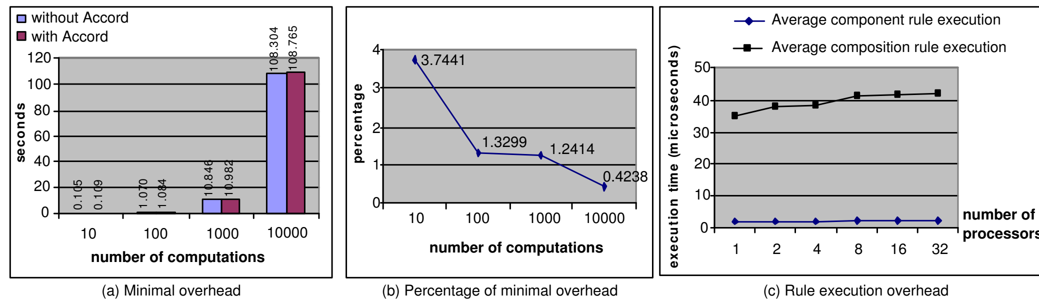


Figure 5.9: Experimental evaluation of Ccaffeine-based Accord prototype.

Experiment 1 (Figure 5.9 (a) and (b)): This experiment measures the runtime overhead introduced by the Accord runtime management framework in a minimal rule execution mode, i.e., the manager components load rules and query sensors but rule execution is disabled during the experiment. The application execution time with and without Accord are plotted in Figure 5.9 (a) and the percentage overhead is plotted in Figure 5.9 (b). The major overhead in this case is due to the loading and parsing of rules. It can be seen from the plots that this overhead is very small compared to the application execution time.

Experiment 2 (Figure 5.9 (c)): This experiment evaluates the average execution time of component rules and composition rules. The figure shows that, as the number of processors increases, the average execution time of both the component rules and composition rules increase but only slightly. This slight increase is primarily due to the time for reconciliation among manager instances, which depends on the number of nodes involved. Once reconciliation is completed, component manager instances perform the replacement in parallel. As seen from the figure, the average execution time of a composition rule is much larger than that of a component rule. This is because, in order to replace a component, the manager has to instantiate a new component, connect it to other components, and load new rules. However, the execution of component rules only involves invoking the component's actuators.

Note that while the framework does introduce overheads, the benefits of self-management would outweigh these overheads. Further, the overheads are not significant when compared to the typical execution time of scientific applications, which can be in hours, days, and even weeks.

5.4 Summary and Conclusion

This chapter presented a component based prototype of Accord programming system that enables self-managing component-based scientific applications capable of detecting and dynamically responding to changing requirements, state and execution context. The programming system extends the common component architecture (CCA) and the Ccaffeine framework. It enables the behaviors and interaction of components and applications to be defined using high level rules and provides a runtime framework for the correct and efficient execution of these rules. Mechanisms for detecting and resolving rule conflicts are provided. The operation of the programming system was illustrated using a self-managing hydrodynamics shock simulation and a self-managing CH_4 ignition simulation. A performance evaluation was presented.

Chapter 6

Accord-WS: Autonomic Service-based Accord

This chapter discusses the prototype of Accord based on the WS-Resource specifications [31] and the Web service specifications [10, 7, 8, 26]. Accord utilizes human knowledge to guide the behaviors and compositions of services in response to changing requirements and execution context. In the autonomic service-based Accord, this is achieved by adapting the service behaviors and their interactions using dynamically defined rules. Key components of the prototype include: (1) the formulation of autonomic services that extend WS-Resources with specifications and mechanisms for self-management and (2) a distributed runtime infrastructure to enable decentralized and dynamic compositions of these services.

6.1 Autonomic Services

An autonomic service (shown in Figure 6.1) consists of (1) a WS-Resource [31] providing functionalities and stateful information, (2) a coordination agent sending and receiving interaction messages for the associated WS-Resource, and (3) a service manager that manages the runtime behaviors of the WS-Resource and its interactions with other autonomic services. Applications can be developed as compositions (possibly dynamic and opportunistic) of these autonomic services.

Each managed WS-Resource is extended with a control port specified as a WSDL [26] document consisting of sensors and actuators for external monitoring and control of its internal state. The control port can be exposed as part of the service port or as a separate document to the service manager. An example of the control port is shown in Figure 6.7.

The coordination agent acts as a programmable notification broker [8] for the associated WS-Resource. As shown in Figure 6.2, a coordination agent consists of 4 modules that work in parallel: (1) a **listener** module that listens to the incoming messages from other autonomic services, (2) **message handlers** that process the messages using functions defined in the **message**

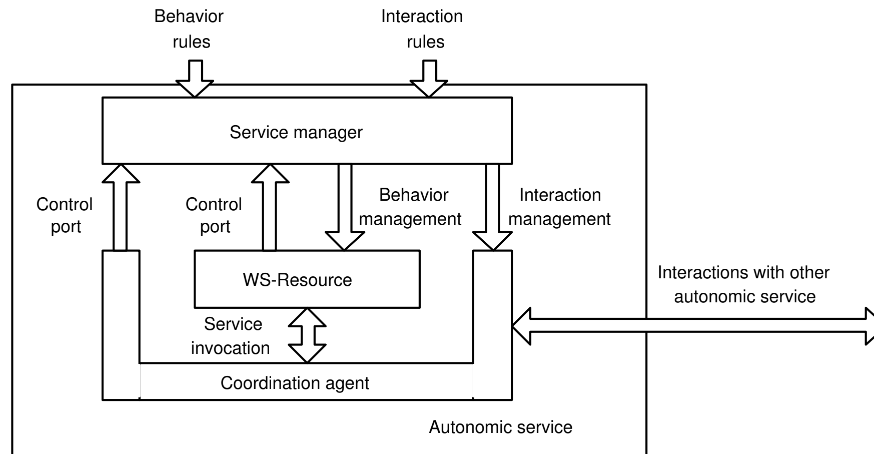


Figure 6.1: An autonomic service.

function table, (3) **libraries** that provides functions for processing messages (e.g., translating message formats and combining messages), and invoking the associated WS-Resource and getting response messages, and (4) a **publisher** that sends the response messages to the subscribers. The coordination agent exposes sensors and actuators to the service manager that allows the manager to query and modify its **message function table** and **message subscriber table**. The service manager can dynamically reconfigure the coordination agent by changing the **message function table** to select functions to process messages, and by changing the **message subscriber table** to add and delete subscribers.

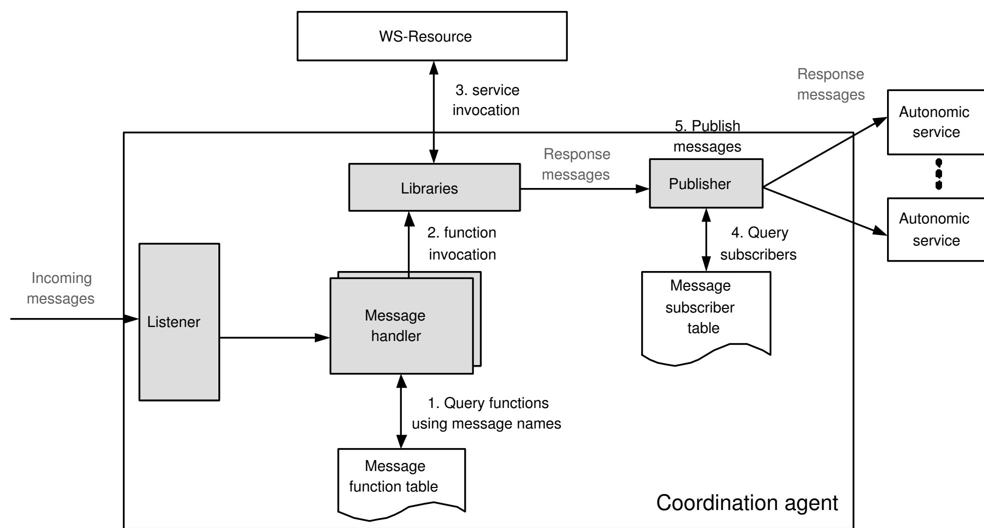


Figure 6.2: Message processing in a coordination agent.

The service manager performs (1) functional management using sensors and actuators exposed by the associated WS-Resource based on behavior rules defined by users or derived from application requirements and objectives, and (2) interaction management using sensors and actuators exposed by the coordination agent based on interaction rules derived from application workflows.

6.2 The Runtime Infrastructure

The runtime infrastructure consists of the Accord portal/composition manager, peer service managers, and other supporting services as shown in Figure 6.3.

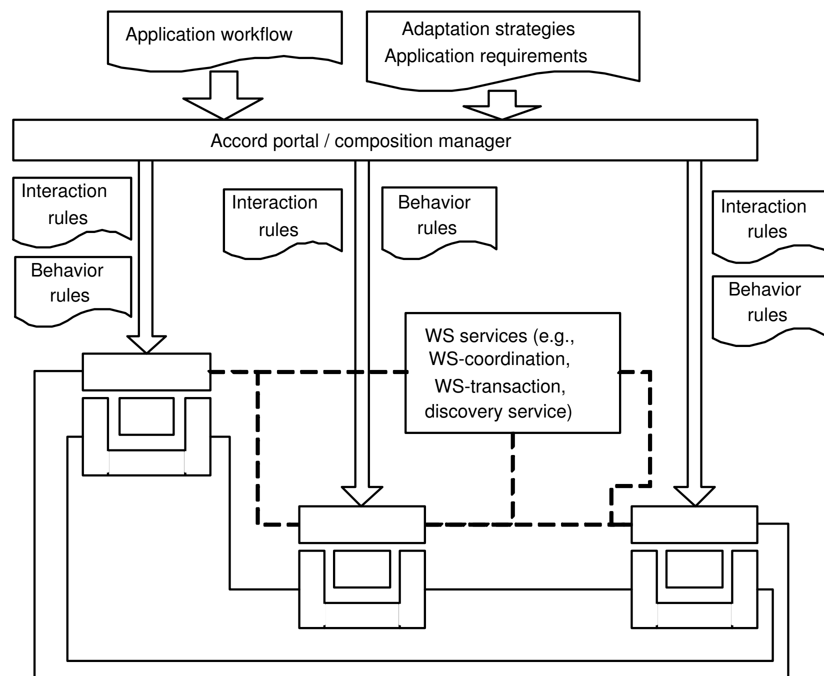


Figure 6.3: The runtime framework. The dashed lines represent the interactions among managers. The solid lines represent the interactions among WS-Resources.

6.2.1 Workflow Execution

To execute an application workflow that may be defined by a user or generated by an automated workflow generation engine such as [38], [24], [12], [55], [43], or [32], the composition manager first discovers and locates the relevant WS-Resources, instantiates a coordination agent for each of the WS-Resources, and further instantiates a service manager for each WS-Resource

and coordination agent pair to enable service behavior and interaction adaptations. Coordination agents interact with their associated WS-Resources using SOAP messages. Service managers are located within the same memory space with their associated coordination agents, and they interact with each other through pointers. The communications among service managers are based on sockets.

The composition manager decomposes the application workflow into interactions rules and injects them into corresponding service managers, which then configure associated coordination agents to dynamically establish publication/subscription relationships and manipulate interaction messages. Specifically, a service manager configures the **message function table** by associating the messages that this autonomic service subscribes to with functions for processing them. Similarly, it also configures the **message subscriber table** by associating the messages that this autonomic service produces with a list of subscribers. These operations are performed by the service manager by invoking the actuators provided by the coordination agent. Further, service managers configures the associated WS-Resource based on the behavior rules defined by users or generated from application requirements.

The advantages of workflow decomposition are illustrated using an itinerary application. This application consists of an **AirlineService**, **HotelService**, and **CarService**, and is used by travellers to reserve airline tickets and hotel rooms, and rent cars for the journey.

- Decreasing communication overhead: The decentralized composition enabled by workflow decomposition is shown in Figure 6.4 (b). Compared to the centralized composition specified using BPEL4WS [15] (shown in Figure 6.4 (a)), decentralized composition enables direct interactions among involved services, and therefore avoids unnecessary messages and relieves the bottleneck caused by the centralized unit.

- Exploring parallelism: After the rules are deployed, autonomic services without data and control dependencies can proceed in parallel, otherwise they are forced to wait until the required data is received. For example in the itinerary application, the parallel execution of **CarService** and **HotelService** can be explicitly defined or automatically discovered, since the two services have no data dependencies and the workflow does not enforce any execution construct on them. Further, the two elements do not have to wait until

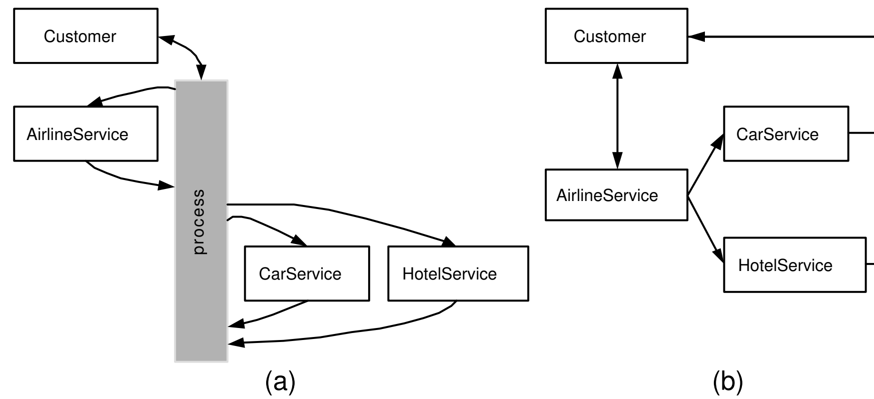


Figure 6.4: The itinerary workflow specified using (a) BPEL4WS and (b) Accord interaction rules.

the **AirlineService** finishes. They can start as soon as the required information (the destination airport and time) is available. As a result, the implicit parallelism can be discovered and exploited as the workflow is decomposed and executed in a decentralized manner.

- Facilitating dynamic composition: Dynamic composition involves addition, deletion and replacement of services, and changes in their interactions at runtime. These changes can be achieved by adding, deleting, or modifying related interaction rules accordingly, as discussed in the next section.

6.2.2 Dynamic Composition

Application workflows need to be changed accordingly when business logic or user requirements change. In most cases, these changes only affect a part of the workflow. Workflow decomposition discussed above can benefit the dynamic composition of autonomic services by constraining the modification to the associated part of the workflow without affecting the rest of the application.

In Accord, dynamic composition is enabled by adding, deleting, or modifying interaction rules in service managers, which automatically reconfigures the associated coordination agents accordingly. For example, a new service **ParkService** is added into the itinerary application, shown in Figure 6.5. First, the **CompositionManager** creates a service manager and a coordination agent for the **ParkService**, and then inserts interaction rules into the **ParkService** and

AirlineService. The service managers of the two involved services will configure the message function tables and message subscriber tables at the associated coordination agents based on these rules. As a result, **ParkService** registers as a notification subscriber to the **AirlineService** and the **CompositionManager** collects reservation information from the **ParkService** before it generates the final itinerary for the users. Since the **ParkService** only interacts with the **AirlineService** and **CompositionManager**, only these two services need injection or modification of interaction rules. **CarService** and **HotelService** are not affected.

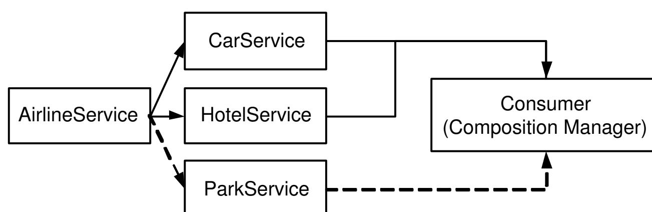


Figure 6.5: A new service **ParkService** is added to the itinerary workflow. The dashed lines denote the new interaction relationships created due to the addition of the new service.

6.3 An Illustrative Application: The Autonomic Data Streaming Application

This section illustrates the self-managing behaviors enabled by the autonomic service-based Accord using an autonomic data streaming application shown in Figure 6.6. The applica-

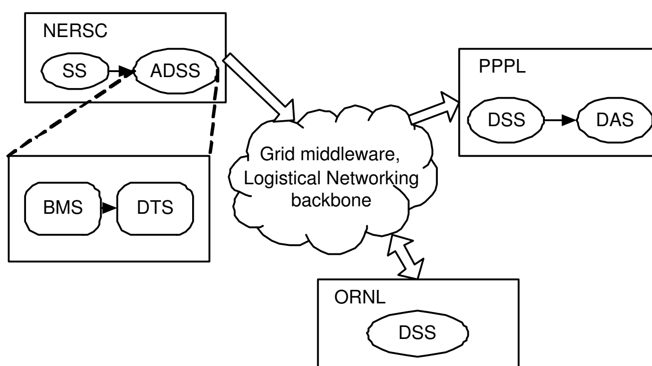


Figure 6.6: The autonomic data streaming application based on Accord-WS.

tion consists of the G.T.C. fusion simulation that runs for days on a parallel supercomputer at NERSC (CA) and generates multi-terabytes of data. The data are analyzed and visualized live at PPPL (NJ), while the simulation is running at NERSC (CA). The data also have to be archived either at PPPL (NJ) or ORNL (TN). Data streaming techniques from a large number

of processors have been shown to be more beneficial for such a runtime analysis than writing data to the disk [20]. The goal of the autonomic data steaming is to stream data from the live simulation to support remote runtime analysis and visualization at PPPL while minimizing overheads on the simulation, adapting to network conditions, and eliminating loss of data. The application workflow consists of the following five core services:

- The Simulation Service (**SS**) executes in parallel on 6K processors of the Seaborg IBM SP at NERSC and generates data at regular intervals that has to be transferred at runtime for analysis and visualization at PPPL, and archived in data stores at PPPL or ORNL.
- The Data Analysis Service (**DAS**) runs on a 32 node cluster located at PPPL. The service analyzes and visualizes the steaming data.
- The Data Storage Service (**DSS**) archives the streamed data using the Logistical Networking backbone [54], which builds a Data Grid of storage services located at ORNL and PPPL.
- The Autonomic Data Streaming Service (**ADSS**) is constructed using the Accord autonomic service architecture and manages the streaming of data from the SS to the DAS (at PPPL) and DSS (at PPPL/ORNL). It is a composite service composed of two services:
  - The Buffer Manager Service (**BMS**) manages the buffers allocated by the service based on the rate and volume of data generated by the simulation and determines the granularity of blocks used for data transfer.
  - Data Transfer Service (**DTS**) manages the transfer of blocks of data from the buffers to remote services for analysis and visualization at PPPL, and archiving at PPPL or ORNL.

Two self-managing scenarios for **ADSS** are described below.

6.3.1 Service Adaptation

BMS selects the appropriate blocking technique, orders blocks in the buffer and optimizes the size of the buffer(s) used to ensure low latency high performance steaming and minimize

the impact on the simulation execution. The adaptations are based on the current state of the simulation, more specifically the following three runtime parameters: (1) The data generation rate, which is the amount of data generated per iteration divided by the time required for the iteration, and can vary from 1 to 400 Mbps depending on the domain decomposition and the type of analysis to be performed. (2) The network connectivity and the network transfer rate. The latter is limited by the 100 Mbps link between NERC and PPPL. (3) The nature of data being generated in the simulation, e.g., parameters, 2D surface data or 3D volume data. **BMS** provides the following three algorithms:

- Uniform Buffer Management: This algorithm divides the data into blocks of fixed sizes, which are then transmitted by the **DTS**. This static algorithm is more suited for the simulations generating data at a small or medium rate (50Mbps). Using smaller block sizes have significant advantages at the receiving end as less time is required for decoding the data and processing blocks for analysis and visualization.

- Aggregate Buffer Management: This algorithm aggregates blocks across iterations and the **DTS** transmits these aggregated blocks. This algorithm is suited for high data generation rates, i.e., between 60-400 Mbps.

- Priority Buffer Management: This algorithms orders data blocks in the buffer based on the nature of the data. For example, 2D data blocks containing visualization or simulation parameters are given higher priority as compared to 3D raw volume data.

To enable the adaptation, the **BMS** exports two sensors, “DataGenerationRate” and “DataType”, and one actuator, “BlockingAlgorithm” as part of its control port shown in Figure 6.7.

The self-optimization behavior of BMS is governed by the rule shown in Figure 6.8, which states that if the data generation rate is greater than the peak network transfer rate (i.e., 100 Mps), the aggregate buffer management is used, otherwise the uniform buffer management algorithm is used. The resulting behavior of this rule is plotted in Figure 6.9. The figure show that BMS switches to aggregate buffer management at simulation time intervals between 75 sec to 150 sec and 175 sec and 250 sec as the simulation data generation rate peaks to 100Mbps and 120 Mbps during these intervals. The aggregation is an average of 7 blocks. Once the data

generation rate falls to 50Mbps, BTS switches back to the uniform buffer management scheme, by constantly sending 3 blocks of data on the network.

Figure 6.9 (b) plots the percentage overhead on simulation execution without and with autonomic management. Overhead is computed as the absolute difference between the time required to generate data without data streaming and the time required to stream the data using ADSS. The plots show that BTS switches from uniform buffer management to aggregate buffer management at data generation rates of around 80-90 Mbps. This increases the overhead slightly, however the overheads remains less than 5%. Without autonomic management, the overheads increase about 10% for higher data rates as BTS continues to use uniform buffer management.

6.3.2 Application Adaptation

This scenario addresses data loss in the cases of extreme network congestion or network failures. These cases cannot be addressed using simple buffer management or replication. One option in these cases to avoid data loss is to write data locally at NERSC rather than streaming. However, the data will not be available for analysis and visualization until the simulation complete, which could be days. Writing data to the disk also causes significant overheads to the simulation [20].

ADSS address these cases by temporarily or permanently switching the streaming to the DSS at ORNL instead of PPPL. NERSC and ORNL are connected by a 400 Mbps link which has a lower probability of being saturated. The data can be later transmitted from ORNL to PPPL. Congestion is detected by observing the buffer - when the buffer is filled to a predefined capacity, the ADSS switches subsequent streaming to ORNL, and when the buffer is no longer saturated, switches the steaming back to PPPL. Note that the data that is already queued continues to be concurrently steamed to PPPL. If the service observes that buffer is being saturated continuously, it infers that there is a network failure and permanently switches the streaming to ORNL. In this case, the blocks already in the PPPL buffer are transferred to the ORNL queue. The rule specifying this self-management behavior is listed in Figure 6.10.

The resulting self-healing behavior is plotted in Figure 6.11. The figure shows that as the ADSS buffer(s) get saturated, the data streaming switches to the DSS at ORNL, and when the

buffer occupancy falls below 20% it switches back to PPPL. Note, that while the data blocks are written to ORNL, data blocks already queued for transmission to PPPL continue to be streamed. The figure also shows that, at simulation time 1500 (X axis), the PPPL buffers once again get saturated and the streaming switches to ORNL. If this persists, the steaming would be permanently switched to ORNL.

6.4 Summary

This chapter presented the autonomic service-based Accord for self-managing Grid applications. It enables the development of autonomic services and the formulation of autonomic applications as the dynamic composition of autonomic services, where the runtime computational behavior of the services as well as their compositions and interactions can be managed at runtime using dynamically injected rules. As a result, applications are capable of adapting their runtime behaviors to deal with the dynamism and uncertain of Grids and Grid applications. An autonomic data streaming application is used to illustrate the self-managing behaviors enabled by Accord.

```
<controlPort name=``BMS_controlPort`` service=``BufferManagerService``>
  <types>
    <sensor name=``DataGenerationRate``>
      <element name=``DataGenerationRateReq`` type=``string``/>
      <element name=``DataGenerationRateResp`` type=``double``/>
    </sensor>
    <sensor name=``DataType``>
      <element name=``DataTypeReq`` type=``string``/>
      <element name=``DataTypeResp`` type=``string``/>
    </sensor>
    <actuator = name=``BlockingAlgorithm``>
      <element name=``BlockingAlgorithmReq`` type=``string``/>
    </actuator>
  </types>

  <message name=``GetDataGenerationRateIn``>
    <part name=``body`` element=``DataGenerationRateReq``/>
  </message>
  <message name=``GetDataGenerationRateOut``>
    <part name=``body`` element=``DataGenerationRateResp``/>
  </message>
  <message name=``GetDataTypeIn``>
    <part name=``body`` element=``DataTypeReq``/>
  </message>
  <message name=``GetDataTypeOut``>
    <part name=``body`` element=``DataTypeResp``/>
  </message>
  <message name=``SetBlockingAlgorithm``>
    <part name=``body`` element=``BlockingAlgorithmReq``/>
  </message>

  <portType name=``BMSControlPortType``>
    <operation name=``SensorDataGenerationRate``>
      <input message=``tns:GetDataGenerationRateIn``/>
      <output message=``tns:GetDataGenerationRateOut``/>
    </operation>
    <operation name=``SensorDataType``>
      <input message=``tns:GetDataTypeIn``/>
      <output message=``tns:GetDataTypeOut``/>
    </operation>
    <operation name=``ActuatorBlockingAlgorithm``>
      <input message=``tns:SetBlockingAlgorithm``/>
    </operation>
  </portType>
</controlPort>
```

Figure 6.7: The control port of **BMS**.

```
<rule name=``BlockingRule`` attribute=``active``>
  <trigger name=``DGR`` sensor=``DataGenerationRate`` op=``GT``
    value=peakRate type=``float``/>

  <when>
    <operand trigger=``DGR``/>
  </when>
  <do>
    <action actuator=``BlockingAlgorithm``>
      <input value=``aggregation`` type=``string``/>
    </action>
  </do>
  <else>
    <action actuator=``BlockingAlgorithm``>
      <input value=``uniform`` type=``string``/>
    </action>
  </else>
</rule>
```

Figure 6.8: The behavior rule for **BMS**.

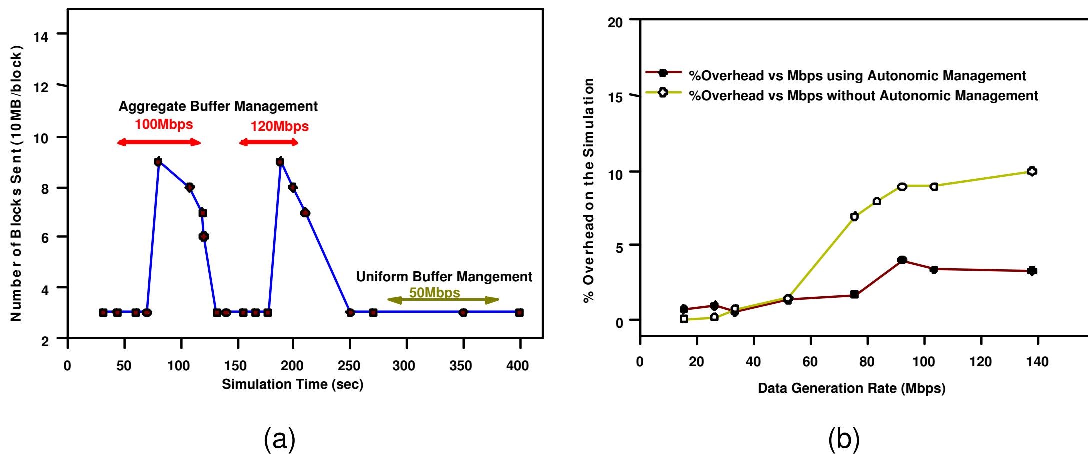


Figure 6.9: (a) Self-optimization behaviors of the Buffer Management Service - BTS switches between uniform blocking and aggregate blocking algorithms based on application data generation rates and network transfer rates and the nature of data generated. (b) Percentage overhead on simulation execution simulation with and without autonomic management.

```
<rule name=``TransferRule`` attribute=``active``>
  <trigger name=``transferFailed`` sensor=``DataTransfer``
             op=``EQ`` value=``0`` type=``integer``/>
  <trigger name=``transferSwitch`` sensor=``NumOfSwitches``
             op=``LT`` value=switchThreshold type=``integer``/>

<when>
  <and>
    <operand trigger=``transferFailed``/>
    <operand trigger=``transferSwitch``/>
  </and>
</when>

<do>
  <action actuator=``TransferAlgorithm``>
    <input value=``local`` type=``string``/>
  </action>
</do>

<when>
  <not>
    <operand trigger=``transferSwitch``/>
  </not>
<do>
  <action actuator=``TransferAlgorithm``>
    <input value=``local`` type=``string``/>
  </action>
  <action actuator=``Accord:SetRuleAttribute``>
    <input value=``TransferRule?type=``string``/>
    <input value=``inactive`` type=``string``/>
  </action>
</do>

<else>
  <action actuator=``TransferAlgorithm``>
    <input value=``remote`` type=``string />
  </action>
</else>
</rule>
```

Figure 6.10: The interaction rule for ADSS.

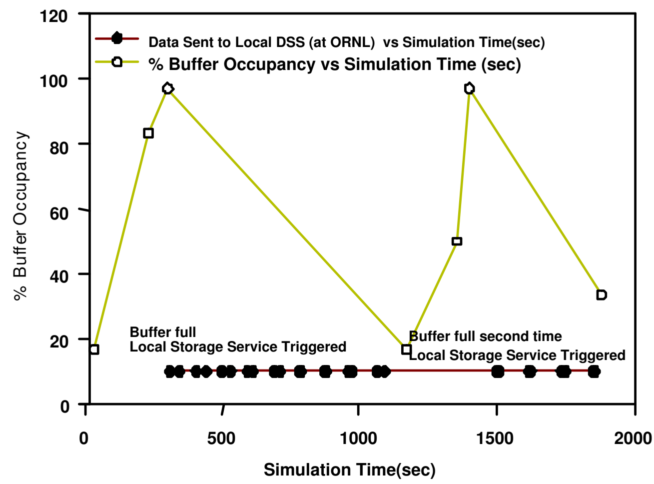


Figure 6.11: Effect of switching from the DSS at PPPL to the DSS ORNL in response to network congestion and/or failure.

Chapter 7

Summary, Conclusion, and Future Work

The primary objective of the research presented in this thesis is to investigate a programming system that addresses the programming requirements of pervasive Grid applications and environments. Specifically, it enables the development and execution of autonomic self-managing applications that can dynamically adapt themselves to address changing requirements and execution context.

7.1 Summary

The thesis presented the Accord programming system for autonomic self-managing applications. Accord builds on existing programming systems and extends them to (1) enable the definition of autonomic elements that encapsulates functional and non-functional specifications, rules, and mechanisms for self-management, (2) enable the formulation of self-managing applications as dynamic composition of autonomic elements, and (3) provide a runtime infrastructure that enables the correct and efficient runtime rule execution to enforce adaptation behaviors.

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

A component based prototype of Accord extends the Common Component Architecture to enable self-management of component-based scientific applications. This prototype supports both function and performance oriented adaptation, enables dynamic composition by replacing components at runtime, and provides consistent and efficient rule execution for intra- and inter-component adaptation behaviors. Two scientific simulations, the self-managing hydrodynamics shock simulation and the self-managing CH_4 ignition simulation, are used to illustrate the operations of the system and the self-managing behaviors.

A service based prototype of Accord extends the Axis framework to support self-managing service-based applications and enables runtime adaptation of service and service interactions, and dynamic service composition. The itinerary reservation application is used to illustrate the operations of this prototype.

Accord is part of the AutoMate project [1]. Project AutoMate 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.

7.2 Conclusion

The characteristics of pervasive and Grid environments impose unique requirements for the programming systems, that the programming systems must be able to support applications that can detect and dynamically respond during execution to changes in both, the state of execution environment and the state and requirements of the application.

Dominant programming systems for parallel and distributed computing are limited in their ability to address these requirements primarily due to their inherent assumptions about the underlying environment, for example they assume reliable environment and static interactions. They do however provide some core mechanisms that can be used to enable required adaptation behaviors. For example, CORBA [1] supports late-binding and dynamic invocation of object instances, which can be used to enable dynamic selection of appropriate object instances

[1] http://automate.rutgers.edu/

possibly based on current execution context. CORBA further provides interceptors that can be used to manipulate the messages in the ORB and to introduce new behaviors at runtime into application execution. Component based programming systems also provide similar capabilities. The specification of CCA [14] embraces the idea of dynamic replacement of components. This feature can be used to enable dynamic selection of components that implement the same ports based on current context. The web service architecture and WSRF proposed in recent years support runtime customization of services, for example, dynamic binding of communication protocols.

The Accord programming system extends these programming paradigms to meet the requirements. This is done by separating context-sensitive concerns and enabling element behaviors and interactions to be defined at runtime. Specifically, this is achieved by extending computational elements to autonomic elements with the specifications of high-level rules and mechanisms for self-management, and providing a distributed runtime infrastructure that consistently and efficiently enforces these rules to enable autonomic self-managing functional, interaction, and composition behaviors.

Further, a new generation of scientific and business applications are enabled by Accord as demonstrated in this thesis.

7.3 Directions For Future Work

We envision the following key directions for future extension of the research presented in this thesis:

- Adaptation across layers: The research presented in this thesis mainly focuses on the application and programming system layer. However, some features may span multiple layers. To fully exploit the dynamism in environments and requirements, adaptation should be enabled in multiple layers, from application and programming system layer to middleware layer and further to the "virtual organization" [52] layer. Corresponding adaptation capabilities and models should be defined for each layer. Further, interaction protocols between layers and interfaces should be formalized and standardized.

- Autonomic generation of rules and workflows: Interaction rules can be generated from

application workflows. Similarly behavior rules can be generated from application requirements and objectives, instead of being defined by users. This involves investigating workflow patterns, categorizing requirements and objectives, and designing corresponding rule templates. A runtime rule generator will be investigated to dynamically analyze workflows and requirements and translate them into corresponding rules.

- Knowledge-based rule execution and conflict resolution: Scientific and business application present different requirements for rule execution and conflict resolution. This thesis focuses on scientific applications. Business models and policies will be investigated to enable rule execution and conflict resolution for business applications, and integrated with the Accord programming system.

- Negotiation between managers: Currently element managers collaborate with each other and resolve conflicts based on rules. Element managers will be provided with negotiation capability to dynamically achieve consensus during conflicts or disagreements. Negotiation protocols and mechanism used by element managers will be investigated. They can be built on the negotiation and consensus research projects being actively investigated in both academia and industry.

References

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

[2] GRI-Mech. http://www.me.berkeley.edu/gri_mech/.

[3] PAPI: Performance Application Programming Interface. http://icl.cs.utk.edu/projects/papi.

[4] PCL - The Performance Counter Library. http://www.fz-juelich.de/zam/PCL.

[5] TAU: Tuning and Analysis Utilities. http://www.cs.uoregon.edu/research/paracomp/tau/tautools/.

[6] The Message Passing Interface (MPI) standard. http://www-unix.mcs.anl.gov/mpi/.

[7] WS-BaseNotification 1.0 specification. ftp://www6.software.ibm.com/software/developer/library/ws-notification/WS-BaseN.pdf.

[8] WS-BrokeredNotification 1.0 specification. ftp://www6.software.ibm.com/software/developer/library/ws-notification/WS-BrokeredN.pdf.

[9] OWL Web Ontology Language Overview. http://www.w3.org/TR/owl-features/, 2004.

[10] Publish-Subscribe Notification for Web services. http://www-106.ibm.com/developerworks/library/ws-pubsub/WS-PubSub.pdf, 2004.

[11] A. Abrahams, D. Eyers, and J. Bacon. An Asynchronous Rule-Based Approach for Business Process Automation Using Obligations. In *Third ACM SIGPLAN Workshop on Rule-Based Programming (RULE'02)*, pages 323–345, Pittsburgh, PA, 2002. ACM.

[12] M. Agarwal and M. Parashar. Enabling autonomic compositions in grid environments. In *the 4th International Workshop on Grid Computing*, Phoenix, AZ, 2003.

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

[14] B. A. Allan, R. C. Armstrong, A. P. Wolfe, J. Ray, D. E. Bernholdt, and J. A. Kohl. The CCA core specification in a distributed memory SPMD framework. *Concurrency Computation*, 14(5):323–345, 2002.

[15] T. Andrews, F. Curbera, H. Dholakia, Y. Goland, J. Klein, F. Leymann, K. Liu, D. Roller, D. Smith, S. Thatte, I. Trickovic, and S. Weerawarana. Business Process Execution Language for Web Services version 1.1. ftp://www6.software.ibm.com/software/developer/library/ws-bpel.pdf, 2003.

[16] Apache. WebServices - Axis. http://ws.apache.org/axis/, 2005.

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

[18] D. Beazley and P. Lomdahl. Controlling the data glut in large-scale molecular-dynamics simulations. *Computers in Physics*, 11(3), 1997.

[19] F. Berman, R. Wolski, H. Casanova, W. Cirne, H. Dail, M. Faerman, S. Figueira, J. Hayes, G. Obertelli, J. Schopf, G. Shao, S. Smallen, N. Spring, A. Su, and D. Zagorodnov. Adaptive Computing on the Grid Using AppLeS. *IEEE transactions on parallel and distributed systems*, 14(4):369–382, 2003.

[20] V. Bhat, S. Klasky, S. Atchley, M. Beck, D. McCune, and M. Parashar. High Performance Threaded Data Streaming for Large Scale Simulations. In *GRID 2004*, number 243-250, 2004.

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

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

[23] L. Capra, W. Emmerich, and C. Mascolo. A Micro-Economic Approach to Conflict Resolution in Mobile Computing. In *Workshop on Self-healing Systems (SIGSOFT'02)*, pages 31–40, Charleston, SC, USA., 2002. ACM.

[24] A. J. S. Cardoso. *Quality of Service and Semantic Composition of Workflows*. PhD thesis, University of Georgia, 2002.

[25] K. Channabasavaiah, K. Holley, and E. M. Tuggle Jr. Migrating to a service-oriented architecture. http://www-106.ibm.com/developerworks/webservices/library/ws-migratesoa/, 2003.

[26] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. Web Services Description Language (WSDL) 1.1. http://www.w3.org/TR/wsdl, 2001.

[27] Y. Diao, J. L. Hellerstein, S. Parekh, and J. P. Bigus. Managing Web server performance with AutoTune agents. http://www.research.ibm.com/journal/sj/421/diao.html.

[28] F. Bergenti DII and A. R. DEIS. Three Approaches to the Coordination of Multiagent Systems. In *the 2002 ACM symposium on Applied computing*, Madrid, Spain, 2002.

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

[30] S. Fischmeister. Mobile code paradigms. http://www.softwareresearch.net/site/teaching/WS0203/PDFdocs.DS/mobile_agents.pdf, 2002.

[31] I. Foster, J. Frey, S. Graham, S. Tuecke, K. Czajkowski, D. Ferguson, F. Leymann, M. Nally, I. Sedukhin, D. Snelling, T. Storey, W. Vambenepe, and S. Weerawarana. Modeling Stateful Resources with Web Services. http://www-128.ibm.com/developerworks/library/ws-resource/ws-modelingresources.pdf, 2004.

[32] N. Furmento, A. Mayer, S. McGough, S. Newhouse, T. Field, and J. Darlington. ICENI: optimization of component applications within a grid environment. *Parallel computing*, 2002.

[33] G. A. Geist, J. A. Kohl, and P. M. Papadopoulos. CUMULVS: Providing fault-tolerance, visualization and steering of parallel applications. In *the Environment and Tools for Parallel Scientific Computing Workshop*, Lyon, France, 1996.

[34] N. Karonis, B. Toonen, and I. Foster. MPICH-G2: A Grid-Enabled Implementation of the Message Passing Interface. *Journal of Parallel and Distributed Computing*, 2003.

[35] J. Kenny, S. Benson, Y. Alexeev, J. Sarich, C. Janssen, L. McInnes, M. Krishnan, J. Nieplocha, E. Jurrus, C. Fahlstrom, and T. Windus. Component-Based Integration of Chemistry and Optimization Software. *Journal of Computational Chemistry*, 25(14):1717–1725, 2004.

[36] B. Khargharia, S. Hariri, M. Parashar, L. Ntaimo, and B. U. Kim. vGrid: A framework for building autonomic applications. In *the 1st International Workshop on Heterogeneous and Adaptive Computing-CLADE 2003*, Seattle, WA, USA, 2003.

[37] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Videira Lopes, J. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *the European Conference on Object-Oriented Programming (ECOOP)*, 1997.

[38] J. Woo Kim and R. Jain. Web Services Composition with Traceability Centered on Dependency. In *the 38th Hawaii International Conference on System Sciences*, 2005.

[39] B. Kohn, E. Kraemer, D. Hart, and D. Miller. An agent-based approach to dynamic monitoring and steering of distributed computations. In *International Association of Science and Technology for Development (IASTED)*, Las Vegas, Nevada, 2000.

[40] S. Lefantzi, J. Ray, C.A. Kennedy, and H.N. Najm. A Component-based Toolkit for Reacting Flows with High Order Spatial Discretizations on Structured Adaptively Refined Meshes. *Progress in Computational Fluid Dynamics*, 2004. In press.

[41] S. Lefantzi, J. Ray, and H. N. Najm. Using the Common Component Architecture to Design High Performance Scientific Simulation Codes. In *the International Parallel and Distributed Processing Symposium*, Nice, France, 2003.

[42] E. C. Lupu and M. Sloman. Conflicts in policy-based distributed systems management. *IEEE transactions on software engineering*, 25(6):852–869, 1999.

[43] S. Majithia, I. Taylor, M. Shields, and I. Wang. Triana as a graphical web services composition toolkit. In *the UK e-Science Programme All Hands Meeting*, Nottingham, UK, 2003.

[44] V. Mann, V. Matossian, R. Muralidhar, and M. Parashar. DISCOVER: An environment for web-based interaction and steering of high-performance scientific applications. *Concurrency and Computation: Practice and Experience*, 13(8-9), 2001.

[45] V. Matossian and M. Parashar. Autonomic Optimization of an Oil Reservoir using Decentralized Services. In *the 1st International Workshop on Heterogeneous and Adaptive Computing– Challenges for Large Applications in Distributed Environments (CLADE 2003)*, Seattle, WA, USA, 2003.

[46] Microsoft. Service Orientation and Its Role in Your Connected Systems Strategy. http://msdn.microsoft.com/architecture/soa/default.aspx?pull=/library/en-us/dnbda/html/srorientwp.asp, 2004.

[47] J. D. Mulder. *Computational steering with parametrized geometric objects*. PhD thesis, Universiteit van Amsterdam, 1998.

[48] R. Muralidhar and M. Parashar. A distributed object infrastructure for interaction and steering. *Concurrency and Computation: Practice and Experience*, 2003.

[49] H. Nakada, S. Matsuoka, K. Seymour, J. Dongarra, C. Lee, and H. Casanova. GridRPC: A Remote Procedure Call API for Grid Computing. http://www.eece.unm.edu/ apm/docs/APM_GridRPC_0702.pdf, 2002.

[50] E. Ort. Service-Oriented Architecture and Web Services: Concepts, Technologies, and Tools. http://java.sun.com/developer/technicalArticles/WebServices/soa2/, 2005.

[51] W. Mulderand S. Osher and J. A. Sethan. Computing Interface Motion in Compressible Gas Dynamics. *Journal of Computational Physics*, 100(2):209–228, 1992.

[52] M. Parashar and J.C. Browne. Conceptual and Implementation Models for the Grid. In *IEEE, Special Issue on Grid Computing*, volume 93, 2005.

[53] S. Parker and C. Johnson. An integrated problem solving environment: The scirun computational steering environment. In *HICCS-31*, 1998.

[54] J.S. Plank and M. Beck. The Logistical Computing Stack – A Design For Wide-Area, Scalable, Uninterruptible Computing. In *DNS: 2002 Dependable Systems and Networks, Workshop on Scalable, Uninterruptible Computing*, Bethesda, Maryland, USA, 2002.

[55] S. R. Ponnekanti and A. Fox. Sword: A developer toolkit for building composite web services. In *the 11th International World Wide Web Conference*, 2002.

[56] D. I. Pullin. Direct Simulation Methods for Compressible Ideal Gas Flow. *Journal of Computational Physics*, 34:231–244, 1980.

[57] S. Rathmayer and M. Lenke. A tool for on-line visualization and interactive steering of parallel hpc applications. In *the 11th International Parallel Processing Symposium (IPPS'97)*, Geneva, Switzerland, 1997.

[58] J. Ray, R. Samtaney, and N.J. Zabusky. Shock Interactions with Heavy Gaseous Elliptic Cylinders : Two Leeward-Side Shock Competition Models and a Heuristic Model for Interfacial Circulation Deposition at Early Times. *Physics of Fluids*, 12(3):707–716, 2000.

[59] J. Ray, N. Trebon, R. C. Armstrong, S. Shende, and A. Malony. Performance Measurement and Modeling of Component Applications in a High Performance Computing Environment: A Case Study. In *the 18th International Parallel and Distributed Processing Symposium (IPDPS04)*, Santa Fe, NM, USA, 2004.

[60] L. Renambot, H. E. BAL, D. Germans, and H.J.W. Spoelder. CAVEStudy: an Infrastructure for Computational Steering in Virtual Reality Environments. In *the 9th IEEE International Symposium on High Performance Distributed Computing*, pages 57–61, Pittsburgh, PA, 2000.

[61] R. L. Ribler, J. S. Vetter, H. Simitci, and D. A. Reed. Autopilot: adaptive control of distributed applications. In *the High Performance Distributed Compuring Conference*, pages 172–179, 1998.

[62] S. M. Sadjadi and P. K. McKinley. Transparent self-optimization in existing corba applications. In *the 1st international conference on autonomic computing*, NYC, NY, USA, 2004.

[63] R. Samtaney, J. Ray, and Norman J. Zabusky. Baroclinic Circulation Generation on Shock Accelerated Slow/Fast Gas Interfaces. *Physics Fluids*, 10(5):1217–1230, 1998.

[64] R. Samtaney and N.J. Zabusky. Circulation Deposition on Shock-Accelerated Planar and Curved Density Stratified Interfaces : Models and Scaling laws. *Journal of Fluid Mech.*, 269:45–85, 1994.

[65] J. Smoller. *Shock Waves and Reaction-Diffusion Equations, Series of Comprehensive Studies in Mathematics*. Springer-Verlag, 1982.

[66] B. Srivastava and J. Koehler. Web Service Composition - Current Solutions and Open Problems. In *ICAPS 2003 Workshop on Planning for Web Services*, pages 28–35, 2003.

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

[68] N. Trebon, J. Ray, S. Shende, R. C. Armstrong, and A. Malony. An approximate method for optimizing HPC component applications in the presence of multiple component implementations. Suffix SAND2003-8760C, Sandia National Laboratories, 2003.

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

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

[71] G. Valetto and G. Kaiser. Using process technology to control and coordinate software adaptation. In *the 25th international conference on Software engineering*, 2003.

[72] W.M.P. van der Aalst, A.H.M. ter Hofstede, B. Kiepuszewski, and A.P. Barros. Workflow Patterns. *distributed and parallel databases*, 14(3), 2003.

Curriculum Vitae

Hua Liu

2005 Ph.D. in Computer Engineering; Rutgers University, NJ, USA.

2001 MS in Computer Engineering; Beijing University of Posts & Telecoms, Beijing, China.

1998 BS in Computer Science; Beijing University of Posts & Telecoms, Beijing, China.

2001-2005 Graduate Assistant, Center for Advance Information Processing, Rutgers University, NJ, USA.

2001 Software Engineering, Bell-labs, Lucent Technologies, Beijing, China.

1998-2001 Research Assistant, BNR lab jointly managed by Nortel Networks and Beijing University of Posts & Telecoms, Beijing, China.

Publications

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

Accord: A Programming Framework for Autonomic Applications . H. Liu and M. Parashar. IEEE transaction on Systems, Man, and Cybernetics, special issue on Engineering Autonomic Systems, Editors: R. Sterritt and T. Bapty, IEEE Press, 2005.

Rule-based Visualization in the Discover Computational Steering Collaboratory. H. Liu, L. Jiang, M. Parashar and D. Silver. Journal of Future Generation Computer System, Special Issue on Engineering Autonomic Systems, Elsevier Science, volume 21, issue 1, page 53 - 59, Jan 2005.

AutoMate: Enabling Autonomic Grid Applications. M. Parashar, H. Liu, Z. Li, V. Matossian, C. Schmidt, G. Zhang and S. Hariri. Cluster Computing: The Journal of Networks, Software Tools, and Applications, Special Issue on Autonomic Computing, Kluwer Academic Publishers.

Enabling Autonomic Grid Applications: Requirements, Models and Infrastructure. M. Parashar, Z. Li, H. Liu, C. Schmidt, V. Matossian and N. Jiang, Hot Topics, Lecture Notes in Computer Science, Springer Verlag, 2005.

Enabling Self-management of Component-based High-Performance Scientific Applications. H. Liu and M. Parashar, Proceedings of the 14th IEEE International Symposium on High Performance Distributed Computing (HPDC-14), Research Triangle Park, NC, July 2005.

A Framework for Rule-Based Autonomic Management of Parallel Scientific Applications. H. Liu, and M. Parashar, Proceedings of the 2nd IEEE International Conference on Autonomic Computing (ICAC-05), Seattle, Washington, USA, June 2005.

A Component-based Programming Framework for Autonomic Applications. H. Liu, M. Parashar, and S. Hariri, Proceedings of the 1st IEEE International Conference on Autonomic Computing (ICAC-04), IEEE Computer Society Press, New York, NY, USA, pp. 10 - 17, May 2004.

Rule-based Visualization in a Computational Steering Collaboratory. L. Jiang, H. Liu, M. Parashar and D. Silver, Proceedings of the International Workshop on Programming Paradigms for Grid and Metacomputing Systems, International Conference on Computational Science 2004 (ICCS 2004), Krakow, Poland, June 2004.

Enabling Autonomic, Self-managing Grid Applications. Z. Li, H. Liu and M. Parashar, Proceedings of SELF-STAR: International Workshop on Self-Properties in Complex Information Systems, Springer Verlag, Bertinoro, Italy, May-June, 2004.

DIOS++: A Framework for Rule-Based Autonomic Management of Distributed Scientific Applications. H. Liu and M. Parashar, Proceedings of the 9th International Euro-Par Conference (Euro-Par 2003), Lecture Notes in Computer Science, Editors: H. Kosch, L. Boszormenyi, H. Hellwagner, Springer-Verlag, Klagenfurt, Austria, Vol. 2790, pp 66 73, August 2003.

AutoMate: Enabling Autonomic Applications on the Grid. M. Agarwal, V. Bhat, H, Liu, V. Matossian, V. Putty, C. Schmidt, G. Zhang, L. Zhen, M. Parashar, B. Khargharia and S. Hariri, Proceedings of the Autonomic Computing Workshop, 5th Annual International Active Middleware Services Workshop (AMS2003), Seattle, WA, USA, IEEE Computer Society Press, pp 48-57, June 2003.