SAND2005-1129C

Enabling Autonomic Scientific Applications within the Common Component Architecture *

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

Benjamin A. Allan, Sophia Lefantzi, and Jaideep Ray
Sandia National Lab, Livermore, USA
Email:{baallan, slefant, jairay}@ca.sandia.gov

Abstract

Emerging scientific applications are increasingly large, dynamic and complex, and require programming systems that enable the application to detect and dynamically respond to changing application state and execution context by adapting their behaviors and interactions. In this paper, we present an extension of the CCAFFEINE Common Component Architecture framework using Accord to enable such self-managing autonomic scientific applications. Accord supports the definition of autonomic components with programmable behaviors and interactions, and to enable runtime composition and management of these components using dynamically defined rules. The design, implementation, operation and evaluations of two self-managing simulations, the simulations of CH_4 ignition and shock hydrodynamics, are presented.

1 Introduction

Parallel/distributed simulations are playing an increasingly important role in science and engineering and are rapidly becoming critical research modalities. Emerging high performance parallel and distributed computing systems are enabling a new generation of simulations that are based on seamless, aggregation and interactions. For example, it is possible to conceive a new generation of scientific and engineering simulations that symbiotically and opportunistically combine computations, experiments, observations, and real-time data, and can provide important insights into complex phenomena.

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

However, the emerging computing systems introduce a new set of challenges due to their scale and complexity. Furthermore, the emerging simulations and the phenomena they model are similarly large, complex, multi-phased/multi-scale, dynamic, and heterogeneous (in time, space, and state). These simulations implement various numerical algorithms, physical constitutive models, domain discretizations, domain partitioners, communication/interaction models, and a variety of data structures. Codes are designed with parameterizations in mind, so that numerical experiments may be conducted by changing a small set of inputs. The choices of algorithms and models have performance implications which are not typically known a priori. Advanced adaptive solution techniques, such as variable step time integrators and adaptive mesh refinement, add a new dimension to the complexity - the application realization changes as the simulation proceeds. This dynamism poses a new set of application development and runtime management challenges. For example, component behaviors and their compositions can no longer be statically defined. Further, their performance characteristics can no longer be derived from a small synthetic run as they depend on the state of the simulations and the underlying system. Algorithms that worked well at the beginning of the simulation become suboptimal as the solution deviates from the space the algorithm was optimized for. For example, suboptimal, communication-heavy sections of the code become a bottleneck if the computational load drops sufficiently as an adaptive mesh simulation coarsens its mesh due to a lack of gradient.

Addressing the challenges outlined above requires a programming system that enables specification of applications which can detect and dynamically respond during execution to changes in both the execution environment and application states. This requirement suggests that: (1) The 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 composable. (3) The interface definitions of these components should be separated from their implementations to enable heterogeneous components to interact and to enable dynamic selection of components.

Component-based software architectures do address some of these requirements. Specifically, the Common Component Architecture (CCA) and its implementation CCAFFEINE framework address the requirements of high-performance parallel scientific applications and have been successfully used [13, 12, 11]. The CCA architecture supports application maintainability and extensibility. Further, the modularization achieved by componentization opens up the potential to change scientific computing in a fundamental way - components can be now be dynamically loadable and their behaviors modified based on current application state and requirements and the execution context. However, this requires extending CCA to enables components that can manage their behaviors and interactions in an autonomic manner.

In this paper, we present such an extension of the CCA CCAFFEINE framework using the Accord [15] programming system. Accord enables the definition of autonomic components with programmable behaviors and interactions, and to enable runtime composition and management of these components using dynamically defined rules. The design, implementation, operation and evaluation of two self-managing simulations, CH_4 ignition and shock hydrodynamics, are presented.

The rest of the paper is organized as the follows. Section 2 presents a conceptual overview of Accord programming framework. Section3 introduces CCAFFEINE and presents the design and implementation of its Accord-based autonomic extension. Section 4 presents the design, operation and evaluation of two autonomic scientific applications. Section 5 investigates related approaches and techniques. Sec-

tion 6 presents a conclusion.

2 The Accord Programming Framework

The Accord programming system [15] addresses the programming challenges outlined above, by extending existing programming systems to enable autonomic applications. Accord realizes three fundamental separations: (1) a separation of computations from coordination and interactions; (2) a separation of non-functional aspects (e.g. resource requirements, performance) from functional behaviors, and (3) a separation of policy and mechanism - policies in the form of rules are used to orchestrate a repertoire of mechanisms to achieve context-aware adaptive runtime computational behaviors and coordination and interaction relationships based on functional, performance, and QoS requirements. Accord is part of Project AutoMate [3], which is investigating autonomic solutions, based on the strategies used by biological systems, to deal with challenges of complexity, dynamism, heterogeneity and uncertainty. Its goal is to realize systems and applications that are capable of managing (i.e., configuring, adapting, optimizing, protecting, healing) themselves. The key components of Accord are described below.

Accord Programming Model: Accord extends existing distributed programming models, i.e., object, component and service based models, to support autonomic self-management capabilities. Specifically it extends the entities and composition rules defined by the underlying programming model to enable computational and composition/interaction behaviors to be defined at runtime using high-level rules. The resulting *autonomic elements* and their *autonomic composition* are described below. Note that other aspects of the programming model, i.e., operations, model of computation and rules for composition are inherited and maintained by Accord.

Autonomic Elements: An autonomic element extends programming elements (i.e., objects, components, services) to define a self-contained modular software unit with specified interfaces and explicit context dependencies. Additionally, an autonomic element encapsulates rules, constraints and mechanisms for self-management, and can dynamically interact with other elements and the system. An autonomic element is illustrated in Figure 1 and is defined by 3 ports:

The **functional port** (Γ) defines a set of functional behaviors γ provided and used by the element. $\gamma \in \Omega \times \Lambda$, where Ω is the set of inputs and Λ is the set of outputs of the element, and γ defines a valid input-output set.

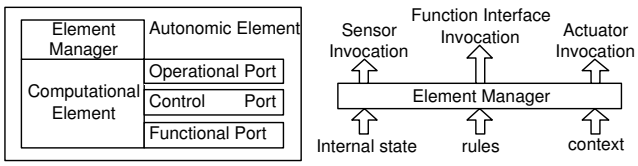

Figure 1. An autonomic element in Accord.

The **control port** ($\sum$) is the set of tuples (σ, ξ), where σ is a set of sensors and actuators exported by the element, and ξ is the constraint set that controls access to the sensors/actuators. Sensors are interfaces that provide information about the element while actuators are interfaces for modifying the state of the element. Constraints are based on state, context and/or high-level access polices.

The **operational port** (Θ) defines the interfaces to formulate, dynamically inject and manage rules that are used to manage the runtime behavior of the element, and the interactions between elements, between elements and their environments, and the coordination within an application.

Each autonomic element is associated with an element manager (possibly embedded) that is delegated to manage its execution. The element manager monitors the state of the element and its context, and controls the execution of rules. Note that element managers may cooperate with other element managers to fulfill application objectives.

Rules in Accord: Rules incorporate high-level guidance and practical human knowledge in the form of if-then expressions, i.e., IF *condition* THEN *actions*, similar to production rule, case-based reasoning and expert systems. *Condition* is a logical combination of element (and environment) sensors, function interfaces and events. *Actions* consist of a sequence of invocations of element and/or system sensors/ actuators, and other interfaces. A rule fires when its condition expression evaluates to be true and causes the corresponding actions to be executed. A priority based mechanism is used to resolve conflicts [14]. Two classes of rules are defined: (1) *Behavioral rules* that control the runtime functional behaviors of an autonomic element (e.g., the dynamic selection of algorithms, data representation, input/output format used by the element). (2) *Interaction rules* that control the interactions between elements, between elements and their environment, and the coordination within an autonomic application (e.g., communication mechanism, composition and coordination of the elements). Note that behaviors and interactions expressed by these rules are defined by the model of computation and the rules for composition of the underlying programming model.

Behavioral rules are executed by an element manager associated with a single element without affecting other elements. Interaction rules define interactions among elements. For each interaction pattern, a set of interaction rules are defined and dynamically injected into the interacting elements. The coordinated execution of these rules results in the desired interaction and coordination behaviors between the elements.

Autonomic composition in Accord: Dynamic composition enables relationships between elements to be established and modified at runtime. Operationally, dynamic composition consists of a composition plan or workflow generation and execution. Plans may be created at runtime, possibly based on dynamically defined objectives, policies, and applications and system context and content. Plan execution involves discovering elements, configuring them and defining interaction relationships and mechanisms. This may result in elements being added, replaced or removed or the interaction relationships between elements being changed.

In Accord, composition plans may be generated using the Accord Composition Engine (ACE) [4] and are expressed in XML. Element discovery uses the content-based middleware and discovery service, which are part of AutoMate [3]. Plan execution is achieved by a peer-to-peer control network of element managers and agents. A composition relationship between two elements is defined by the control structure (e.g., loop, branch) and/or the communication mechanism (e.g., RPC, shared-space) used. A Composition Manager translates this into a suite of interaction rules, which are then injected into corresponding element managers. Element managers execute the rules to establish control and communication relationships among these elements in a decentralized manner. Rules can be similarly used to add or delete elements. Note that the interaction rules must be based on the core primitives provided by the system. Accord defines a library of rule-sets for common control and communications relationships between elements. The decomposition procedure will guarantee that the local behaviors of individual elements will coordinate to achieve the application's objectives. Runtime negotiation protocols provided by Accord address runtime conflicts and conflicting decisions caused by a dynamic and uncertain

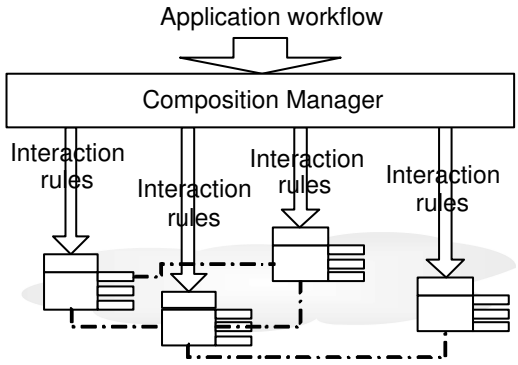

Figure 2. Autonomic application execution in Accord.

environment.

Accord decouples interaction and coordination from computation, and enables both these behaviors to be managed at runtime using rules. This enables autonomic elements to change their behaviors, and to dynamically establish/terminate/change interaction relationships with other elements. Deploying and executing rules does impact performance, however, it increases the robustness of the applications and their ability to manage dynamism. Further, our observations indicate that the runtime changes to interaction relationships are infrequent and their overheads are relatively small. As a result, the time spent to establish and modify interaction relationships is small as compared to typical computation times. A prototype implementation and evaluation of its performance overheads is presented in [14].

3 An Autonomic Component Framework using CCAFFEINE and Accord

CCAFFEINE [6], a Sandia National Laboratories framework implementation compliant with the CCA core specification, provides the fast and lightweight glue to integrate external and portable component peers into a SCMD (Single Component Multiple Data) style parallel application. Fast means that the CCAFFEINE glue does not get between components in a way that slows down their interactions. Lightweight means that CCAFFEINE only provides the functionality necessary to link components together and bring them into an executable state.

In order to enable the runtime self-managing scientific applications, we develop an autonomic component framework using Accord and CCAFFEINE. The autonomic framework allows CCAFFEINE components to instantiate and expose control ports composed of sensors and actuators. It also introduces two specialized types of components: (1) **Component Manager** that monitors and manages the computational behaviors of individual components, e.g., selecting the optimal algorithms and modifying internal states, and (2) **Composition Manager** that manages, adapts and optimizes the execution of an application at runtime. These manager components encapsulate the Accord operational port.

Both, Component Manager and Composition Manager components 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 manager components are not part of the CCAFFEINE framework, and consequently provide the programmers the flexibly to integrate them into their applications only as needed. For example, assuming there are 3 components ‘A’, ‘B’, and ‘C’ in one application, programmers can integrate two Component Managers components, ‘CMA’ and ‘CMB’, to manage component ‘A’ and ‘B’ separately by making ‘A’ use the *RulePort* provided by ‘CMA’ and ‘B’ use the *RulePort* provided by ‘CMB’. Programmers could also integrate only one Component Manager ‘CM’ to manage both ‘A’ and ‘B’ by making ‘A’ and ‘B’ use the *RulePort* provided by ‘CM’. As we can see from the example, component ‘C’ does not use the *RulePort*, and therefore it will not be controlled by any Component Managers. Similarly, programmers can choose to use the Composition Manager or not. The architectures of the two manager components are described in the following sections.

Our design of the Component Manager and Composition Manager is 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, we encapsulate the manager functionalities into two component types and provide programmers with the flexibility of integrating them with other components in the applications.

- The manager functionalities are provided by components instead of being integrated with the CCAFFEINE framework. This prevents the framework from being 'overweight' and thus avoids the consequent performance and maintenance implications.
- By encapsulating the manager functionality into these components and providing abstract interfaces for invoking this functionality, we can modify and improve the manager functionality without affecting other components and the framework. We can either add additional functionality into the manager components, or create other components that deal with specific functions and integrate them with the manager components via the 'uses/provides design pattern' [1].

3.1 Component Manager

The Component Manager provides a port named *RulePort*, as shown in Figure 3. Instances of the Component Manager are instantiated after the other applications components are instantiated and their ports are connected within the CCAFFEINE framework. This is done in two steps: (1) Managed component instances need to expose their sensors and actuators to the Component Manager instances by invoking the 'addSensor' and 'addActuator' functions, and (2) Rules to manage the components should be loaded into the Component Manager instances, possibly from a disk file, by invoking the 'loadRules' function. The initialization of Component Manager instances is a one-time operation.

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

Figure 3. The *RulePort* provided by Component Manager and Composition Manager

Scientific application often are executed as a series of computation phases. Between two successive phases, computation inside components and communication between components are paused, and components are reconfigured for the next mathematical calculation. This is called the quiet state. Management functionalities need to be performed during these quiet states. The managed components invoke the 'fire' function to inform the Component Managers that they have entered into a quiet state. These managed components must be programmed by users to invoke the 'fire' function, for example, at the beginning/end of each phase or once every several phases, to establish the optimization/adaptation frequency.

In most cases, no state needs to be carried between two successive computation phases. This means that the components are 'stateless', and often they are reset with a new set of parameters at the beginning of different phases. In our prototype implementation, Component Managers will execute rules to change the configuration parameters only at quiet states. Since these parameters are applied during the next computation phase, the changing of parameters is tantamount to changing the computational behaviors of the managed components.

The CCAFFEINE framework employs a SCMD model, which says that all the components in one application are copied to all the involved nodes, as illustrated in Figure 4. Component Manager instances on each node independently evaluate and execute the rules to manage and possibly change the computational behaviors of the managed components. The result of this independence is that at the same time step, the managed components on different nodes will demonstrate different computational behaviors,

since they work on different data and in different execution environments. For example, the managed component instance 'A1' on node 'X' is asked to use algorithm 'm', while at the same time step, the component instance 'A2' on node 'Y' uses algorithm 'n'. The different computational behaviors demonstrated by individual instances of the same component are made transparent to other component instances by the inherent encapsulation characteristic of components, since these instances ('A1' and 'A2' in our example) still implement the same abstract interface. Therefore, consistent computation across all the nodes can be guaranteed.

People may argue that hard-encoding these control information within the components can provide the similar capabilities. However, this assumes that all possible states of the application and execution environment and all required adaptations are known a prior and are coded into the components, which is not always possible. Separating the management and control behaviors from computation and making them programmable at runtime provides the following benefits:

- The programming complexity is decreased. We separate the control logic from other computation and specify the logic as (possibly dynamically defined) rule. This reduces the size of the components and allows them to specialize in the task that they are meant to perform.
- The components' reusability is increased. Users need only to modify the rules instead of modifying the internal implementation of the components to make them suitable for other applications.
- Often, the nature of the adaptations are not known a priori and may depend on the current execution state and context of the application itself. The separation makes dynamic specification of adaptation rules possible.

3.2 Composition Manager

The Composition Manager also provides a *RulePort*, shown in Figure 3. The instances of the Composition Manger are initialized by receiving the sensors and actuators exposed by the managed components using the 'addSensor' and 'addActuator', and loading in the rules (possibly from a disk file) using the 'loadRules' function. Managed components will notify the Composition Manager instances of the quiet state by invoking the 'fire' function. The Composition Manager instances then evaluate the rules based on the current context and state, and if their conditions evaluate to true, execute the associated actions. Actions may change the behavior of a managed component and/or replace managed component instances that have failed or are not operating suitably.

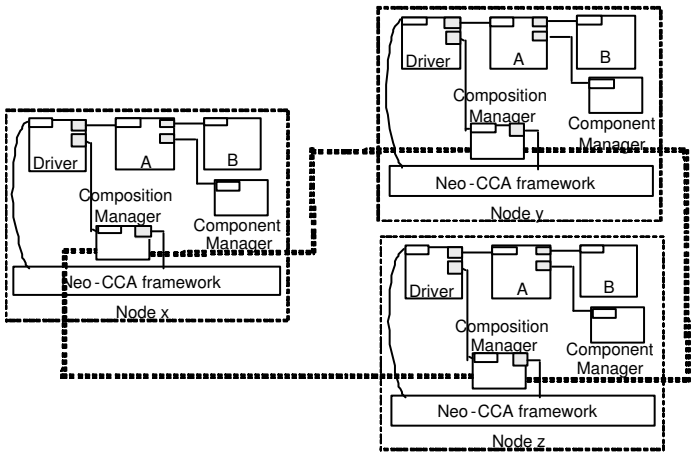

Figure 4. The architecture of an autonomic component applications using Accord and the CCAFFEINE framework

When replacing a managed component instances, the new component does not need to provide and use the exactly same ports as the old one. However, the new component must at least provide all the

active ports (which are used by other components in the application) of the old component. If the new component uses some ports that cannot be provided by the existing components in the application, replacement with the new component may require instantiating new components.

The Composition Manager instances on different nodes may independently generate different replacement plans based on their local execution contexts. There are two cases (see Figure 4): (1) The Composition Manager on node x proposes to replace component instance 'A' with 'A1', and the Composition Manager on node y wants to replace component instance 'B' with 'B1'. In this case, either both the plans are propagated to all the nodes or they are declined. (2) The Composition Manager on node x proposes to replace 'A' with 'A1', while the Composition Manager on node y proposes to replace 'A' with 'A2'. In this case, either both nodes should replace 'A' with 'A1' or with 'A2' on all nodes to conform to the SCMD model. Therefore, negotiation is needed among all the Composition Managers to choose one replacement plan that is acceptable to all the managers.

In our current prototype, replacement plans are assigned one of two different priorities. A high priority means that the replacement is necessary, for example, the old components cannot work correctly or have failed. The low priority means that the replacement is optional, for example, the new components have better performance than the old ones - the old components however still work correctly. In case of a conflict, replacement plans with higher priority are propagated to and accepted by all the managers. If there are multiple high priority plans, a runtime error is generated and reported to the users. For plans with a low priority, a cost model is used to approximate the performance gains of each plan and the plan with the best overall gain is selected and applied by all managers.

As mentioned in the previous sections, many runtime situations cannot be predicted at development time. Therefore, the Composition Manager provides an channel for user interaction [16]. The user can use this channel to monitor and control the application (e.g., pause the execution of the application) and to inject new rules into the application.

4 Illustrative Applications and Experimental Evaluations

In this section we will present examples of how the CCAFFEINE-based Accord framework was used to optimize scientific simulations by dynamically changing algorithms in response to simulation parameters and to increase the stability by replacing failed components at runtime. In both cases, the Accord framework executed a set of rules to determine the self-optimizing and self-healing strategies.

4.1 Self-optimization: CH_4 Ignition

4.1.1 Problem Description

Realistic simulations of igniting systems (even simple ones like a stoichiometric mixture of methane and air) present many of the characteristics that bedevil scientific simulations. The simulation processes are represented by a set of chemical reactions, which do not appear simultaneously - rather, they appear (and disappear) when the fuel (methane) and oxidiser (oxygen) react and give rise to the various intermediate chemical species. The rates at which these processes operate vary over orders of magnitude; further, with the liberation of heat, there are states where various processes negate each other leading to conditions where various "intermediate" species might exist at almost constant levels (such states are called equilibrium states). A mixture of CH_4 and air at 300K is considered to be at equilibrium; igniting it by abruptly raising its temperature to 1800K constitutes a non-equilibrium state, at which point a large number (but not all) the chemical processes are activated.

Thereafter, one could proceed in one of two ways. One could evolve in increments of time (timesteps) small enough that the fastest process is well resolved in time; alternatively, one could exploit the smooth temporal evolution with high order algorithms to take larger timesteps, at the cost of increased storage and some inter- and extrapolations. High order algorithms are more robust and will provide an answer; choosing an optimal algorithm provides robustness and time savings. The choice of the algorithm can, to a first approximation, be decided by the "degree of non-equilibrium" i.e. in our case, the starting temperature.

The species and the reactions they participate in are described by the *chemical mechanism*. We use the GRI 1.2 [2] CH_4–Air mechanism with 32 species and 177 reversible reactions, specified in the CHEMKIN [10] format. Thermo-chemical data is read in from files, processed, and used to compute the chemical source terms. We assume that the volume of the gas expands uniformly to keep pressure uniform in space and time. While this is physically unrealistic, it neither mitigates the mathematical severity of the problem nor does it have a bearing on the optimization process.

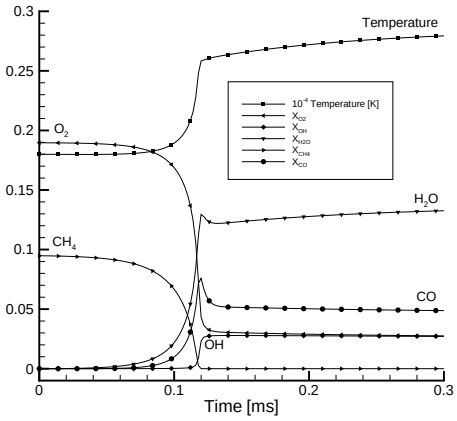

Figure 5. Evolution of temperature T and the mole fractions of O_2, OH, H_2O, CH_4 and CO as a function of time when a stoichiometric mixture of methane and air is ignited at 1800K.

Fig. 7 (top) shows the original code for 0D ignition. The **ThermoChemistry** component embodies the chemical interactions - it provides the source terms for temperature and species due to chemistry. **ThermoChemistry** is a thin C++ wrapper around Fortran 77 subroutines abstracted from pre-existing codes for chemically reacting flow [18]. **Initializer** imposes the initial condition – a vector of double precision numbers specifying the (stoichiometric) mass fractions for CH_4 and O_2 and zero for the rest, except N_2 which rounds up the sum of mass fractions to 1. The initial temperature is 1800 K, and the initial pressure is 1 atm. **Cvode** is an implicit stiff/non-stiff integrator that time-advances the system as it ignites (Fig. 5). This is a thin wrapper around the Cvode [8] integrator library.

Cvode contains a set of algorithms (called backward difference formula or BDF) 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. In the process of evolving the simulation in time, the equation G in **Ref** is evaluated repeatedly. The bulk of the time is spent in evaluating G; thus reducing the number of G evaluation is a sufficient indication of speed. As the chemistry becomes more complex, G evaluations are expected to be the only parameter of any consequence.

In the next two subsections, we describe the process of generating the rules (to specify the optimal algorithm based on the temperature), discuss the execution of the CH_4 ignition simulation based on these rules (using CCAFFEINE-Accord framework), and compare the rule-based simulation with a non-rule-based one to demonstrate the improvement in performance.

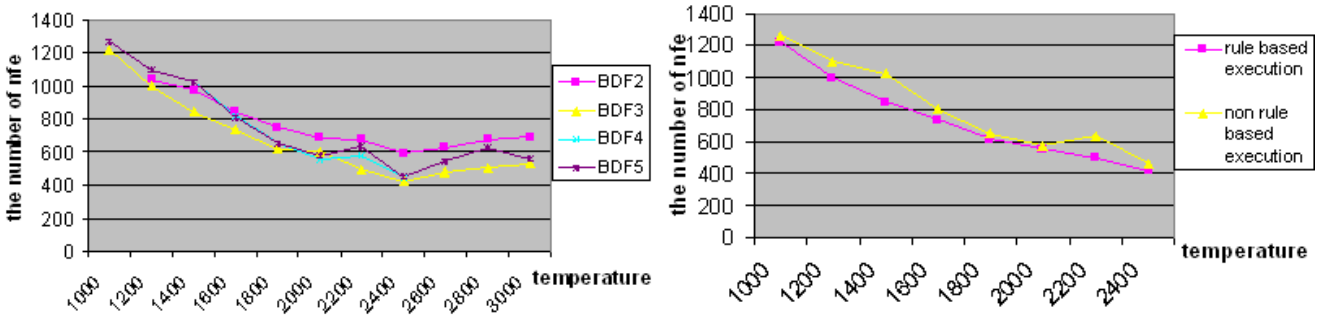

Figure 6. Left: Comparison of BDF_i at different temperature. Right: The comparison of the performance of the Accord based CH_4 ignition simulation and the original one.

4.1.2 Rule Generation

As shown in Figure 6 (Left), the performance of BDF_i varies at different temperatures. Therefore, to achieve the best performance, the simulation needs to dynamically select the optimal algorithm according to current ignition temperature.

To generate the rules, an ignition problem was specified with a starting temperature and a given (stoichimetric) proportion of the fuel and air. During the execution, we increased the temperature gradually, executed the application, and generated rules as follows:

```
IF 1000 <= temperature < 2000 THEN BDF 3
IF 2000 <= temperature < 2200 THEN BDF 4
IF 2200 <= temperature        THEN BDF 3
```

4.1.3 Accord Enabled CH_4 Ignition Simulation

The rules obtained from a simplified problem are used for the CH_4 ignition simulation (shown in Figure 7). In the original application (shown in Figure 7 on the top), **Initializer** always invokes the **Cvode** component using the same BDF algorithm without regard to the changing temperature. In the Accord enabled application, we added two components, **ComponentManager** to read in rules from a disk file and notify **RuleExecutor** of the optimal BDF algorithm. This is done by evaluating the rules based on the current temperature. The **RuleExecutor** then sets the BDF algorithm on **Cvode** before invoking it. We go through the **RuleExecutor** to optimize **Cvode** in order to keep **Cvode** unchanged; a direct connection between **ComponentManager** and **Cvode** would have required changes to **Cvode** to accept and

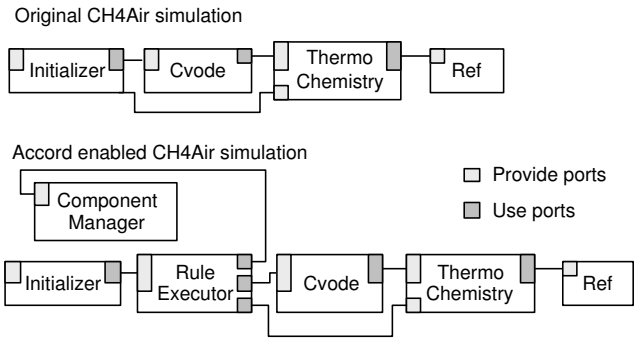

Figure 7. Component "wiring" schematic for the CH_4 ignition problem, without Accord (on the top) and with Accord (below).

implement the dictates of the **ComponentManager**. **Cvode** thus remains a pristinely scientific component.

The **RuleExecutor** sends the temperature and the variable used by **Cvode** as the BDF parameter to the **ComponentManager** where they are interpreted as a sensor and an actuator respectively. Based on rules, the **ComponentManager** identifies an optimal BDF parameter and returns it to the **RuleExecutor**, which then proceeds to set it on **Cvode** preparatory to invoking it to solve a problem. This process is repeated before every **Cvode** invocation.

As shown in Figure 6 (Right), the rule-based execution decreases the number of invocation to equation G in **Ref** component. Since the bulk of the time is spent in this, we see a clear computational saving. As the problem becomes more complex (the computational cost of G increase), the savings in G evaluations translate to a proportionately larger savings in runtime. While it is not difficult to believe that the use of an optimal strategy to solve a problem is beneficial, we have shown *how* this might be achieved in a general manner, by exploiting a generic rule-interpreter and an orchestrator to manage a purely scientific package, which, further, needs no changes.

The same set of components can be used to simulate another ignition problem with a mixture of H_2 and air. We load **Cvode**, **ThermoChemistry**, and **Ref** components with different initialization parameters, and provide **ComponentManager** with new rules as follows:

```
IF 1000 <= temperature < 1200 THEN BDF 2
IF 1200 <= temperature < 1800 THEN BDF 4
IF 1800 <= temperature < 2400 THEN BDF 3
IF 2400 <= temperature THEN BDF 4
```

Since we separate the controllable logic of selecting the optimal algorithm from other implementation logic and express them in rules, all the involved components can be reused for the H_2 ignition simulation without modifying their implementation and re-compiling them. As we mentioned in Section 3.1, the reusability of components is increased.

4.2 Self-healing: Shock Hydrodynamics Problem

4.2.1 Problem Description

In this example we show how runtime replacement of components may affect the robustness of simulation codes. We simulate 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 in [20, 21, 22]. The governing equations (the compressible Euler equations) in conservative form are:

$$\mathbf{U}_t + \mathcal{F}(\mathbf{U})_x + \mathcal{G}(\mathbf{U})_y = 0 \tag{1}$$

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 [17] 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

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

The Godunov method is used to determine $\mathcal{F}_{i+1/2}^{n+1/2}$ 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 [23] is setup, which is solved (iteratively) to obtain the fluxes $\mathcal{F}_{i+1/2}^{n+1/2}$. 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 [19]).

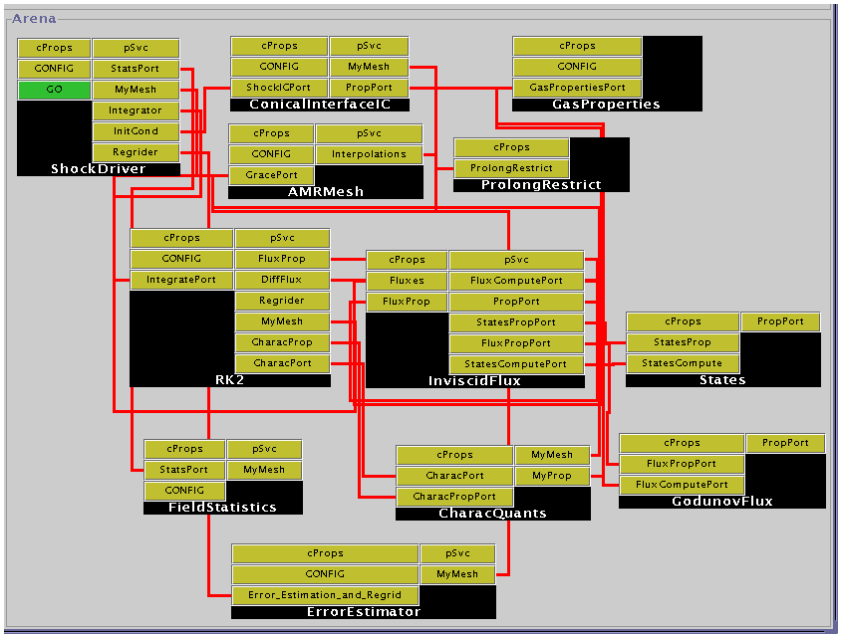

Figure 8. "Wiring" diagram of the shock-hydrodynamics simulation. A second-order Runge-Kutta (*RK2*)integrator drives *InviscidFlux* component – transformation into left and right (primitive) states is done by *States* and the Riemann problem solved by *GodunovFlux*. Sundry other components for determining characteristics' speeds (u + a, u - a, u), cell-centered interpolations etc. complete the code.

In Fig. 8 we see the assembly of components. We see a Runge-Kutta time integrator (**RK2**) with an **InviscidFlux** component supplying the right-hand-side of the equation, patch-by-patch. 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 - a shock tube with Air and Freon (density ratio 3) separated by an oblique interface which is ruptured by a Mach 3.5 shock. The shock tube has reflecting boundary conditions above and below and outflow on the right. Godunov methods with **RK2** become unstable for stronger shocks and larger density ratio. One solution is to replace **RK2** with a 3^{rd}order Runge-Kutta scheme, which includes a part of the imaginary axis in its stability region; a cheap solution (but better for an illustration of the flexibility afforded by components) is to replace **GodunovFlux** with **EFMFlux**, based on a gas-kinetic scheme [19].

Whether a certain algorithm (Godunov, in this case) will work for a given set of simulation parameters (the Mach number and the density ratio in this case) is 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 needs to be performed across all the processors. While the dynamic changing of 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), and 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 example we will demonstrate this dynamic replacement of **GodunovFlux** with **EFMFlux** (triggered by a **GodunovFlux** error) and provide qualitatively correct results.

4.2.2 Accord-enabled Shock-Hydrodynamics Problem

To enable the dynamic replacement of the **GodunovFlux** component with the **EFMFlux** component, a **CompositionManager** component is added to the shock-hydrodynamics simulation. The **CompositionManager** provides a *RulePort*, which is used by **ShockDriver** and **GodunovFlux**.

During initialization, **GodunovFlux** exposes its internal state as a sensor to **CompositionManager** via invoking the 'addSensor' function, and **CompositionManager** reads in the rules from a disk file. Dynamical replacement of components can only be performed at quiet states, which is determined and explicitly programmed in **ShockDriver** by invoking the 'fire' function to notify **CompositionManager**. The **CompositionManager** then inquires the internal state of **GodunovFlux** to check the rule condition and determine the dynamic replacement plan.

CompositionManager instances on different nodes independently generate the replacement plans, which may differ. Only one plan is selected and propagated to all the nodes. Since our problem involves stability and correctness - i.e. the entire simulation fails to proceed if even one processor reports the unsuitability of **GodunovFlux** - a plan for *replacement* is heeded by all processors. To perform the replacement, **CompositionManager** instances will (1) locate and instantiate **EFMFlux** from the component repository, (2) detect all the provides and uses ports of **GodunovFlux**, as well as all the components connected to it (in our case, **InviscidFlux** using *DiffPort* and *PropertiesPort* provided by **GodunovFlux**, and **GasProperties** providing *PropPort* used by **GodunovFlux**), (3) disconnect **GodunovFlux**, (4) connect **EFMFlux** to **InviscidFlux** and **GasProperties**, and finally (5) destroy **GodunovFlux** instances. From the next calculation step, **EFMFlux** is used instead of **GodunovFlux**. However, other components in the application will not notice the replacement, since only the abstract interfaces (ports) are visible to them while the implementations are hidden behind.

The key requirement is that the dynamic replacement must be completed at the same time, so that the new component will be used by all the nodes from the next calculation iteration. This is achieved by the *blocking* function 'fire'. **CompositionManager** instances will be synchronized before the 'fire' function returns. The 'fire' function is designed as an atomic operation, which guarantees that either the replacement is completed successfully or not performed at all.

5 Related Work

Related research efforts in systems supporting dynamically adaptive applications can be classified based on the nature of the adaptations they support. In systems supporting *statically-defined adaptations*, the adaptation codes must be coded into the application code and be defined at compile time. Systems that enable adaptations by extending an existing programming languages, for example [7], or by defining new adaptation languages, for example [9], fall into this category.

With statically-defined adaptation enabling applications to dynamically customize/adapt their behaviors at runtime, the possible adaptation must be known a priori and must be coded into the application. If new adaptations are required or applications requirements change, the application code has to be modified and the applications re-compiled.

In systems supporting *dynamically-defined adaptation*, adaptations (in the form of code, scripts or rules) can be added, removed and modified at runtime. Accord and [25] fall into this category. These systems separate adaptation as an aspect and express it in terms of rules (conditions and actions) that can be dynamically managed. In [25], adaptations are only performed on pre-defined method invocations, similar to 'injectors' and 'filters' [5]. Adaptation behaviors across multiple invocation are not supported. In Accord, rules are systematically composed of pre-defined sensors and actuators to provide more comprehensive adaptation behaviors. The adaptations can occur at any quiet state rather than at pre-defined method invocations.

ALua [24] is probably most closely related to Accord. Both these systems separate configuration from computation and perform interaction/coordination and adaptation in an interpretive manner. And they both support the execution of dynamically defined adaptation specification (code, scripts, rules) in an even-driven manner to adapt application behaviors. However, Accord allows more control to guarantee the correctness and consistency of programs during and after adaptation by using elements (objects, components, and services) as the adaptation units. The adaptation of individual elements, such as setting the value of a variable or selecting an algorithm, are encapsulated within these elements and access to them is controlled by the sensors/actuators constraints (specified b y their control ports). The addition/deletion/replacement of elements is restricted by their functional signatures (specified by their functional ports) and system requirements (specified by their operational ports).

6 Conclusion and Future Work

In this paper, we presented Accord, which builds on existing programming frameworks to enable autonomic applications via extending the elements (objects, components, and services) and composition rules defined by the underlying frameworks to enable computational and composition/interaction behaviors to be defined at runtime using high-level rules. We implemented a prototype based on CCA CCAFFEINE framework to enable the self-optimization and self-healing features of scientific simulations. An experimental evaluation of the performance was also presented.

In CCA CCAFFEINE framework, the communication mechanism between components is restricted to functional calls and their coordination relationships are pre-defined and unable to be changed at runtime for the purpose of high-performance. Therefore, our current work discussed in this paper mainly focus on adapting individual components and replacing components at runtime. We will implement the other aspects of dynamic composition proposed in Accord using other frameworks, for example, OGSA/WSRF, to demonstrate the dynamic changing of communication paradigms (RPC, messaging,

et.c) and coordination models according to the changing environment. Future research includes the dynamic and opportunistic composition of autonomic elements, rules, execution and management of autonomic applications.

References

[1] CCA forum. http://www.cca-forum.org/.

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

[3] M. Agarwal, V. Bhat, Z. Li, H. Liu, V. Matossian, V. Putty, C. Schmidt, G. Zhang, S. Hariri, and M. Parashar. AutoMate: Enabling autonomic applications on the grid. In *Proceedings of the Autonomic Computing Workshop*, Seattle, WA, 2003.

[4] M. Agarwal and M. Parashar. Enabling autonomic compositions in Grid environments. In *Proceedings of the 4th International Workshop on Grid Computing*, pages 34–41, Phoenix, AZ, 2003. IEEE Computer Society Press.

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

[6] 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.

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

[8] S. D. Cohen and A. C. Hindmarsh. Cvode, a stiff/nonstiff ode solver in c. *Computers in Physics*, 10(2):138–143, 1996.

[9] G. Duzan, J. Loyall, and R. Schantz. Building adaptive distributed applications with middleware and aspects. In *Proceedings of the 3rd international conference on Aspect-oriented software development*, pages 66–73, Lancaster, UK, 2004. ACM.

[10] R. Kee, F. Rupley, and J. Miller. Chemkin-ii: A fortran chemical kinetics package for the analysis of gas phase chemical kinetics. Sandia Report SAND89-8009B, Sandia National Labs., Livermore, CA, August 1993.

[11] J. P. Kenny, S. B. Benson, Y. Alexeev, J. Sarich, C. L. Janssen, L. C. McInness, M. Krishnan, J. Nieplocha, E. Jurrus, C. Fahlstrom, and T. L. Windus. Ccomponent-Based Integration of Chemistry and Optimization Software. *J. Comput Chem*, 25:1717–1725, 2004.

[12] 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.

[13] S. Lefantzi, J. Ray, and H. N. Najm. Using the common component architecture to design high performance scientific simulation codes. In *Proceedings of the International Parallel and Distributed Processing Symposium*, Nice, France, 2003.

[14] H. Liu and M. Parashar. DIOS++: A framework for rule-based autonomic management of distributed scientific applications. In *Proceedings of the 9th International Euro-Par Conference (Euro-Par 2003), Lecture Notes in Computer Science*, pages 66–73, Klagenfurt, Austria, 2003. Springer-Verlag.

[15] H. Liu, M. Parashar, and S. Hariri. A component-based programming framework for autonomic applications. In *Proceedings of the 1st IEEE International Conference on Autonomic Computing (ICAC-04), IEEE Computer Society Press*, pages 278 – 279, New York, NY, 2004.

[16] V. Mann, V. Matossian, R. Muralidhar, and M. Parashar. DISCOVER: An environment for web-based interaction and steering of high-performance scientific applications. *Concurrency and Computation: Practice and Experience*, 13(8-9):737 754, 2001.

[17] W. Mulder, S. Osher, and J. Sethian. Computing Interface Motion in Compressible Gas Dynamics. *J. Comp. Phys.*, 100:p209, 1992.

[18] H. Najm and P. Wyckoff. Premixed flame response to unsteady strain-rate and curvature. *Combustion and Flame*, 110(1-2):92–112, 1997.

[19] D. I. Pullin. Direct Simulation Methods for Compressible Ideal Gas Flow. *J. Comp. Phys.*, 34:231–244, 1980.

[20] 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. *Phys. Fluids*, 12(3):707–716, March 2000.

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

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

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

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

[25] Z. Yang, B. H. C. Cheng, R. E. K. Stirewalt, J. Sowell, S. M. Sadjadi, and P. K. McKinley. An aspect oriented approach to dynamic adaptation. In *Proceedings of the first workshop on Self-healing systems*, pages 85–92, Charleston, South Carolina, 2002. ACM.