A Flexible Infrastructure for Distributed Deployment in Adaptive Sensor Webs

William R. Otte, John S. Kinnebrew, Douglas C. Schmidt, Gautam Biswas
Institute for Software Integrated Systems
Department of Electrical Engineering and Computer Science
Vanderbilt University, 2015 Terrace Place
Nashville, TN, 615-343-8197
wotte@dre.vanderbilt.edu, john.s.kinnebrew@vanderbilt.edu,
schmidt@dre.vanderbilt.edu, biswas@vuse.vanderbilt.edu

Abstract—Distributed sensor webs typically operate in dynamic environments where operating conditions, transient phenomena, availability of resources, and network connection quality change frequently and unpredictably. Often these changes can neither be completely anticipated nor accurately described during development or deployment. Our prior work has described how we developed agents and services that are capable of monitoring these changing conditions and adapting system parameters using the CORBA Component Model (CCM) deployment infrastructure as part of the *Multi-agent Architecture for Coordinated Responsive Observations* (MACRO) platform.

Our recent application of MACRO to the *South East Alaska MOnitoring Network for Science, Telecommunications, Education, and Research* (SEAMONSTER) project has identified new distributed deployment infrastructure challenges common to computationally constrained field environments in adaptive sensor webs. These challenges include standardized execution of low-level hardware-dependent actions and on-going data tasks, automated provisioning of agents for heterogeneous field hardware, and minimizing deployment infrastructure overhead. This paper describes how we extended MACRO to address these sensor web challenges by creating an action/effector framework standardizing the execution of lightweight actions and providing for automated provisioning of MACRO agents, in addition to footprint optimizations to the underlying CCM infrastructure.

TABLE OF CONTENTS

978-1-4244-2622-5/09/$25.00 2009 IEEE
IEEEAC Paper #1339, Version 2 Updated 02/11/2008.

1. INTRODUCTION

A variety of sensor webs [1] can now provide data in near real-time to help scientists study and predict weather, natural disasters, and climate change. Modern sensor webs enable information to be gathered from sensors around the globe and quickly transmitted to local or remote servers where significant computational resources are available for model building, data analysis, and prediction. With the appropriate infrastructure, these systems can facilitate the real-time collection and analysis of sensor data even under changing environmental conditions and multiple concurrent science objectives.

Sensor webs are large-scale, networked systems often made up of heterogeneous computing platforms including both commodity servers and distributed real-time embedded (DRE) systems. Unfortunately, the configuration and operation of individual sensor webs are often performed in an *ad hoc* manner, which impedes adding new sensors, updating/-modifying their software, and reconfiguring them to accommodate evolving conditions and changing science needs.

Like other DRE systems, such as shipboard computing [2] and fractionated spacecraft [3], the field subsystems of sensor webs can benefit from recent advances in middleware infrastructures. The use of *quality-of-service (QoS)-enabled component middleware* helps automate remoting, lifecycle management, system resource management, deployment, and configuration in DRE systems. QoS-enabled component middleware supports explicit configuration of QoS aspects (*e.g.*, priority and threading models), and provides many desirable real-time features (*e.g.*, priority propagation, scheduling services, and explicit binding of network connections). In integrated, adaptive sensor webs, QoS-enabled component middleware helps address the large, heterogeneous set of sensor assets and computational resources that must be coordinated and managed to address weather, climate change, and disaster prediction/management problems.

Sensor web hardware and sensors are also increasingly configurable and must operate in *open* environments where operating conditions, workload, resource availability, and connectivity cannot be accurately characterized *a priori*. Our previous work described the design of the *Multi-agent Architecture for Coordinated Responsive Observations* (MACRO) [4], which provides a QoS-enabled component middleware platform that automates many system configuration and management tasks for sensor web applications, including dynamic system management and autonomous operation of configurable sensor webs in open DRE system environments. This paper addresses new distributed deployment challenges resulting from applying the MACRO platform to the *South East Alaska MOnitoring Network for Science, Telecommunications, Education, and Research* (SEAMONSTER) [5], which is a representative sensor web for monitoring glacial change and watershed effects.

The remainder of this paper is organized as follows: Section 2 summarizes adaptive sensor web challenges in SEAMONSTER, including standardized execution of low-level hardware-dependent actions and on-going data tasks, automated provisioning of agents for heterogeneous field hardware, and minimizing deployment infrastructure overhead; Section 3 describes how we addressed these challenges by extending MACRO to include an action/effector framework that standardizes the execution of lightweight actions, automates the provisioning of MACRO agents, and optimizes the footprint of the underlying QoS-enabled component middleware; Section 4 empirically evaluates how these extensions address deployment challenges; Section 5 compares MACRO with related work; and Section 6 presents concluding remarks.

2. Motivation

Overview of SEAMONSTER

SEAMONSTER is a glacier and watershed sensor web at the University of Alaska Southeast (UAS) in Alaska [5]. This sensor web monitors and collects data regarding glacier dynamics and mass balance, watershed hydrology, coastal marine ecology, and human impact/hazards in and around the Lemon Creek watershed and Lemon Glacier. The collected data is used to study the correlation between glacier velocity, glacial lake formation and drainage, watershed hydrology, and temperature variation.

The SEAMONSTER sensor web includes sensors and weatherized computer platforms that are deployed on the glacier and throughout the watershed to collect data of scientific interest, as shown in Figure 1. The data collected by the sensors is relayed via wireless networks to a cluster of servers that filter, correlate, and analyze the data. These data collection and processing applications are being transition to run atop a QoS-enabled component middleware platform consisting of the *Component-Integrated ACE ORB* (CIAO) [6], which is open-source, QoS-enabled, component middleware that implements the OMG Lightweight CORBA Component Model (CCM) [7] and Deployment and Configuration [8] specifications.

Distributed Deployment and Adaptation Challenges in SEAMONSTER

Effective deployment of data collection and filtering applications on SEAMONSTER field hardware and dynamic adaptation to changing environmental conditions and resource availability present significant software challenges for efficient operation of SEAMONSTER. While SEAMONSTER servers provide significant computational resources, the field hardware is computationally constrained. The server-based MACRO agents perform extensive planning and scheduling to provide direction and coordination of tasks performed by the computationally limited field resources. In the field, the limited computational resources require software solutions with a small footprint and low computational complexity.

Field nodes in a sensor web often have a large number of observable phenomena in their area of interest. The type, duration, and frequency of observation of these phenomena may change over time, based on changes in the environment, occurrence of transient events in the environment, and changing goals and objectives in the science mission of the sensor web. Moreover, limited power, processing capability, storage, and network bandwidth constrain the ability of these nodes to continually perform observations at the desired frequency and fidelity. Dynamic changes in environmental conditions coupled with limited resource availability requires individual nodes of the sensor web to rapidly revise current operations and future plans to make the best use of their resources.

To handle dynamic changes effectively, sensor web nodes must be capable of goal-driven, functional adaptation. Moreover, they must be able to adapt the local system in light of resource constraints and fluctuations throughout the sensor web to maintain efficient and correct operation of the overall system. Prior work [9] describes how MACRO addresses these challenges by combining the planning and resource management services of its server agents with the template plan schemas of its field agents. This paper extends our prior work by focusing on the following unexplored challenges associated with providing a flexible deployment infrastructure to support system management and dynamic adaptation of the SEAMONSTER field nodes.

Challenge 1: Standardized Execution of Planned Low-Level Actions and Data Tasks

Most tasks performed by MACRO agents on the SEAMONSTER server cluster involve on-going data processing and analysis that are implemented by components selected and configured during planning/scheduling. A scheduled plan for the deployment and operation of these configured components is passed to a resource management service, which allocates them to individual server nodes and adjusts configuration settings and operating system parameters to handle fluctuations in resource usage and availability. The resource management service employs the deployment infrastructure to coordinate the deployment, configuration, connection, and

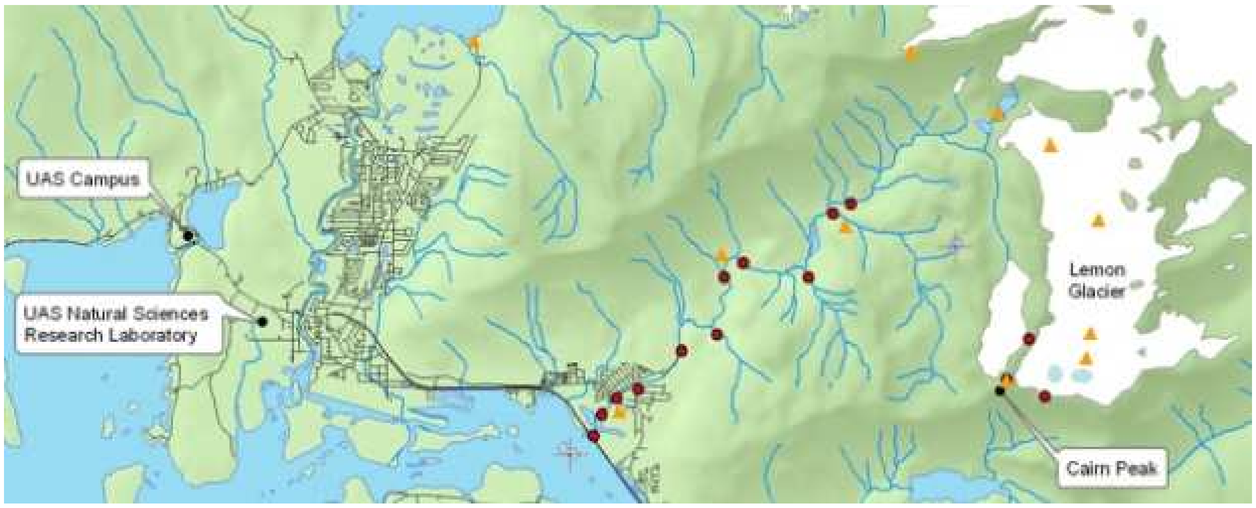


Figure 1. SEAMONSTER field sensors and UAS servers

execution of the specified components. This provides a standardized, flexible system for implementing tasks as configured components.

Data collection and transmission tasks on field nodes are implemented as components for the same reasons as data processing tasks on the servers. However, many of the other activities that MACRO agents plan and perform on field nodes consist of low-level, hardware-dependent actions that execute only briefly to configure sensors or the power management hardware subsystem. Implementing these short-lived "actions" as components would require proportionally much greater overhead for their deployment and execution than for data processing "tasks" that execute over a long period of time and must transmit data streams to other components. Given the limited computational resources available on field nodes, the overhead for implementing brief, low-level actions as components is unacceptable.

An additional, smaller level of granularity for action implementation is therefore necessary for efficient execution of many planned activities on field nodes. While the agents could implement these actions directly, it would require hard-coding of hardware-dependent actions into each field agent. Alternatively, grouping these actions into larger pre-planned sets of actions executing as a component would proportionally reduce the overhead. However this would negatively impact maintainability through duplication of action code segments and constrain the available options for planning. Instead, a standardized deployment and execution framework, such as that provided by the middleware for components, but with lower overhead, would greatly enhance the maintainability of the system and simplify initial system development. Section 3 describes how such a framework has been designed and incorporated in MACRO to address this challenge.

Challenge 2: Automated Agent Provisioning for a Variety of Field Hardware

Field nodes in a sensor web may have a large number of possible configurations, due to a variety of sensors, software, and situations that they may be tasked to observe and react to. Consequently, the agents that manage these nodes must be as flexible as possible. Hard-coding available tasks into agent code requires that new versions of each agent be created as nodes add new responsibilities or hardware. The solution developed to address the challenge described in Section 2 should include integration with the deployment infrastructure to download and load at run-time appropriate action implementations. Section 3 describes how the deployment infrastructure may be leveraged to dynamically provision agents with available, context-specific actions at deployment time.

Challenge 3: Minimizing Deployment Infrastructure Overhead

The SEAMONSTER sensor web, described in Section 2, includes many field nodes operating with extremely limited computational resources. SEAMONSTER includes two types of computational platforms for field nodes [10]:

• **Primary Microservers.** These units are weatherized single board computers (SBC) that are designed to have very limited power consumption and precise control over the power consumption of the SBC and attached devices. The SBC is a commercial off-the-shelf (COTS) product that has a 200 MHz low-power ARM processor with 64 MB of built-in RAM.

• **Adjunct Microservers.** These units are repurposed COTS Linksys NSLU-2 network attached storage devices that are essentially inexpensive SBCs. These computers consist of a 133 Mhz (with simple hardware modifications possible to reach 266 MHz) ARM processor with 32 MB of built in RAM. These units provide a low-cost alternative to using Primary Microservers for some field nodes, however they lack powercontrol capabilities and have even more limited computational power primarily due to the minimal amount of RAM.

Each platform presents an environment where the resident footprint of the middleware infrastructure and component implementations is critically important. Excessive footprint will at best cause excessive memory swapping to occur, significantly degrading performance and shortening the life of attached flash drives, and at worst cause deployment failure due to exhaustion of memory, as happened occasionally during initial trials of MACRO software in the SEAMONSTER testbed. Section 3 describes initial efforts to reduce the footprint of the middleware and Section 6 describes our planned approach to further reduce middleware overhead.

3. Minimizing Infrastructure Overhead in MACRO

This section explains how MACRO addresses the challenges described in Section 2. We begin with an overview of the agent-based system developed in our previous work, along with a description of its middleware infrastructure. We then outline the new MACRO Action/Effector framework and explain how it addresses the deployment infrastructure challenges encountered in the SEAMONSTER project.

Overview of MACRO

The *Multi-agent Architecture for Coordinated, Responsive Observations* (MACRO) platform provides a powerful computational infrastructure for enabling the deployment, configuration, and operation of large-scale sensor webs that are composed of many constituent sensor webs. Figure 2 shows how MACRO supports intelligent autonomy via agents at the following two levels of abstraction:

• **Mission level**, where agents interact with users to allocate high-level science tasks to sensor webs and coordinate scheduled plans to achieve these goals, and

• **Resource level**, where local server and field agents achieve mission goals through functional adaptation of a sensor web in light of current environmental conditions and resource availability.

The work presented in this paper focuses on the resource level of MACRO, which is applicable to individual sensor webs, such as SEAMONSTER.

System adaptation for current conditions and science goals, described as a set of desired data products and results, is directed by MACRO server-based agents with functional knowledge of the sensor web system and available software components and actions. MACRO server-based agents employ novel services, such as the *Spreading Activation Partial Order Planner* (SA-POP) [11] and the *Resource Allocation and Control Engine* (RACE) [12]. These agents use the SA-POP service to (1) decompose goals into subgoals that are achieved at the server or by individual field nodes and (2) plan/schedule for their achievement.

With information from field agents about current conditions and local activities, SA-POP produces scheduled, high expected utility plans to achieve an optimized set of current goals. These scheduled plans are also broken into subplans by SA-POP. These subplans describe (1) the selection/-configuration of server-based software components, which are allocated and managed by the RACE service on the servers, and (2) hardware-dependent actions on individual field nodes, as well as additional component deployments.

Although the sub-plans generated by SA-POP on the servers can provide an important starting point for deployments and actions on the field nodes, changing local conditions may invalidate those plans or require modification to them for effective, rapid reaction to environmental phenomena and changing resource availability. Since local field agents have limited computational resources, extensive planning and scheduling, such as that provided by SA-POP, is not possible for rapid reaction to local changes. Instead, field agents use a set of template plan schemas that cover a range of conditions and local subgoals to which they are applicable.

Server-based agents provide the field agents with the current set of local subgoals to pursue and suggested schema instantiations corresponding to the sub-plans produced by SA-POP. The task of the field agent is therefore the simpler choice of an appropriate set of schemas to instantiate as local conditions evolve. The extensive planning/scheduling performed by MACRO server agents using SA-POP—together with the choice of plan schemas to instantiate by MACRO field agents—provide effective system adaptation to achieve science goals in light of changing environmental conditions and resource availability.

The implementation of agents in MACRO is based on the CIAO [6] QoS-enabled component middleware (described in Section 3 to ensure interoperability across heterogeneous computing platforms, reduce development costs, and improve overall robustness and scalability. The agents operate on the

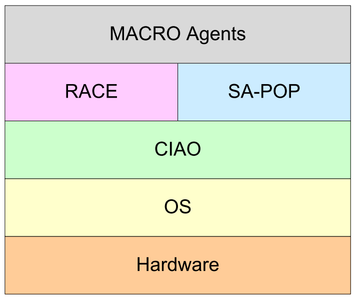


Figure 3. The MACRO Architecture

CIAO middleware to ensure that a diverse set of science objectives can be met, as shown in Figure 3. This architecture helps facilitate real-time, adaptive data acquisition, analysis,

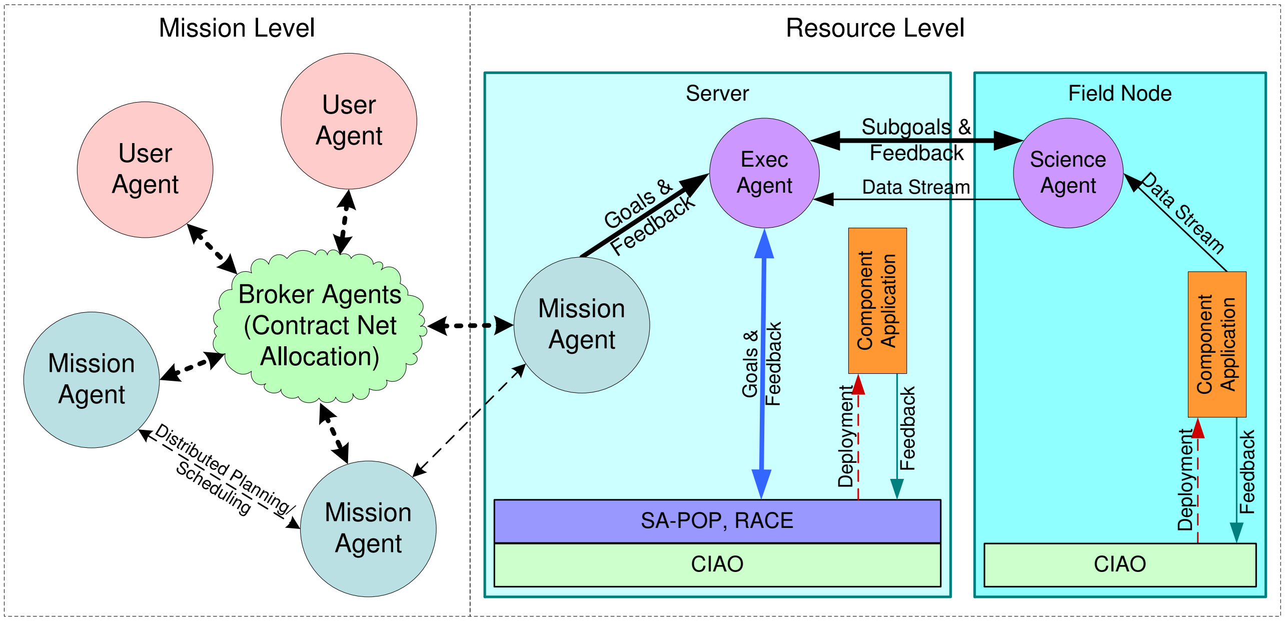


Figure 2. MACRO Agent Architecture

fusion, and distribution.

Overview of MACRO's QoS-enabled Component Middleware

The MACRO middleware infrastructure is based on the CORBA Component Model (CCM) [13], which is an extension to the Common Request Broker Architecture (CORBA) [14] that supports Component Based Software Engineering. CCM enhances re-usability by allowing developers to focus only on application business logic, abstracting away the details of communication and configuration. Components interact with one another only through well-defined ports, which include *facets* (provided interfaces), *receptacles* (required interfaces), and *event sources and sinks* (asynchronous publish/subscribe transport).

The CCM middleware used in MACRO is the *Component Integrated ACE ORB* (CIAO) [15]. CIAO is a QoS-enabled implementation of the Lightweight CCM (LWCCM) [16] specification built on top of *The ACE ORB* (TAO). CIAO provides a clear separation of concerns between *configuration logic*, specified at deployment time via XML-based meta-data, and *business logic*.

CIAO's deployment and configuration capabilities are provided by the *Deployment and Configuration of Component Based Systems* (DnC) [17] specification, which was created by the OMG in response to the need for generic and standard mechanisms for deploying component-based applications. The DnC standard includes both a *data model* (*i.e.*, descriptions of components, component compositions, target domains, and associated configuration meta-data) and a *run-time model* (*i.e.*, a set of interfaces used to manage application life-cycles).

The DnC *run-time model* in CIAO is implemented by the *Deployment And Configuration Engine* (DAnCE) [18]. DAnCE is a set of daemons executing in the *domain*, which is the collection of nodes and communication methods that comprise the target environment. Important elements of the run-time model are shown in Figure 4 and include:

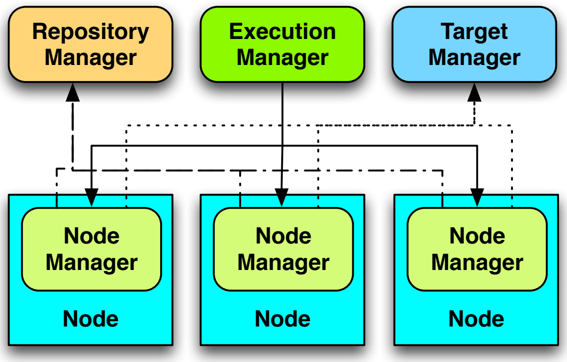


Figure 4. DAnCE Daemons

• **Node Manager**, which is a daemon that runs on all nodes in the domain and is responsible for deploying, configuring, and managing all components deployed to that node. This daemon also supports monitors necessary to report resource status on the node to the MACRO agents. Each node in the sensor web will have a running Node Manager.

• **Execution Manager**, which is a daemon that coordinates the activities of all *Node Managers* in a given domain. This daemon is the primary point of control for the life-cycle of all component applications. Primary microservers with direct connections to the SEAMONSTER server cluster will have Execution Managers.

• **Target Manager**, which is a daemon that collates and reports resource availability in a given domain. Information is collected from resource monitors installed in individual *Node Managers*. Like the *Execution Manager*, this daemon will run on primary microservers with direct connections to the servers.

• **Repository Manager**, which is a daemon that maintains a collection of component meta-data and binary implementations. Individual *Node Managers* may contact nearby repositories to download binaries for components they are tasked to deploy, and MACRO agents may query the repository for information about components available for deployment. An instance of the *Repository Manager* will run on the primary server for use by the MACRO server agents and server deployments. Another instance will reside on primary microservers with direct connections to SEAMONSTER server cluster for use by nodes in the field.

Applying MACRO to Address SEAMONSTER Challenges

The remainder of this section explains how MACRO applies and enhances the CIAO and DAnCE middleware described above addresses the sensor web challenges identified in Section 2.

Addressing Challenge 1: MACRO's Action/Effector Framework—MACRO's Action/Effector framework has been developed to provide a standardized mechanism that has two primary benefits for implementing short-lived, lightweight "actions," as opposed to on-going "tasks" implemented as components. First, it allows the MACRO agents with their SA-POP planning service and plan schemas to use a common vocabulary for describing preconditions, dependencies, and effects of individual actions, as well as resource requirements of the associated action implementations. Second, it provides a clear separation of concerns between invoking the action and the business logic of the action, similar to that of components, *i.e.*, it provides a mechanism that agents can use to execute a set of actions without knowledge at compile or link time of the implementation of those actions.

Action meta-data. Listing 1 describes the `Action_Info` data structure which allows an action to provide meta-data about itself to the system/agents. This meta-data describes properties (*e.g.*, a unique identifier, argument identifiers and types, return value identifier and type) and requirements (*e.g.* CPU and memory requirements, hardware/sensor resources, and component or object references). This data structure is implemented as a CORBA valuetype, which will leave open the possibility for derivation though inheritance should additional fields need to be added later without breaking backwards compatibility with the interfaces described below.

Action interface. Listing 2 describes the interface for the Action itself. This interface provides a vehicle for provision of meta-data, and operations to manage the full lifecycle of an Action. To provide lightweight actions with minimal overhead, this interface is specified as a *local* interface, which instructs the CORBA IDL compiler to omit generation of code that allows for remote invocation of the object, creating a locality constrained object. This design substantially reduces overhead, as shown in Section 4. While this locality constraint prevents MACRO agents from directly accessing Action objects, the framework provides a mechanism which does not constrain their use by those agents. This framework allows MACRO agents to access and execute actions while hiding the complexities of action deployment and execution through the Effector interface described in Section 3.

```
struct Property {
  string name;
  any value;
};
typedef sequence<Property> Properties;
valuetype Action_Info {
  public string id;
  public Properties resource_requirements;
  public Properties init_arguments;
  public Properties exec_arguments;
  public Properties reference_requirements;
}
```

Listing 1. Action Meta-Data IDL

```
local interface Action {
  readonly attribute Action_Info info;
  void initialize (in ObjSeq references);
  void execute (in any arguments,
                out any result);
  void release ();
};
```

Listing 2. Action Interface

The Action attribute `info` allows the Action implementation to self-describe its meta-data, ultimately providing information to the agents about its requirements and capabilities. This information is also used by an implementation of the Effector interface to determine which object references and arguments are to be passed to the operations contained in this Action interface.

These operations allow the Effector to manage the lifecycle of Actions. The `initialize` operation is invoked upon creation of the Action, providing it with object references to deployed components and objects that the business logic may

```
extern ''C'' {
  Action_ptr create_action (void);
}
```

Listing 3. Action Factory

```
interface Effector {
  Action_Info load_action (in string library_name,
                           in string factory_name);
  void unload_action (in string id);
  Action_Info query_action (in string id);
  StringSeq list_actions ();
  void execute_action (in string id,
                       in any arguments,
                       out any result);
};
```

Listing 4. Effector Interface

need in order to successfully execute. The `execute` operation implements the business logic of the Action. This operation accepts two parameters, both of type CORBA Any, which is a generic container which may contain any valid CORBA data type, allowing the Actions to accept arguments or provide results in a flexible, but standardized, manner. Finally, the `release` operation informs the Action that it is about to be deallocated so that it may release any resources that it holds.

Each Action implementation provides a factory method (an example of which is found in Listing 3) that is used by the Effector to construct instances of the action at run-time. Similar to the method used by the DnC specification [17] to construct component instances, this factory method is declared as `extern ''C''`, which will allow the Effector interface to load actions at run-time using methods similar to `dlopen` and `dlsym`.

Effector interface. Listing 4 describes the Effector interface, which is used by the MACRO agents to load and execute actions. This interface is provided as either a facet or a supported interface on a component. It is used by MACRO agents to execute plans/schemas and interact with the components providing abstractions of the available hardware, as shown in Figure 5. For example, the `load_action` method may be used by an agent or other Effector client to load a new action from a named shared library that contains a provided factory symbol. The operations on the Effector interface allow MACRO agents to (1) manage the lifecycle of Actions installed in the Effector, (2) determine which Actions have been loaded and query their meta-data, and (3) instruct the

```
component EffectorProvider {
  provides Effector effect;
  attribute Action_Factories factories;
};
```

Listing 5. Example Component with Effector

Effector to execute an Action.

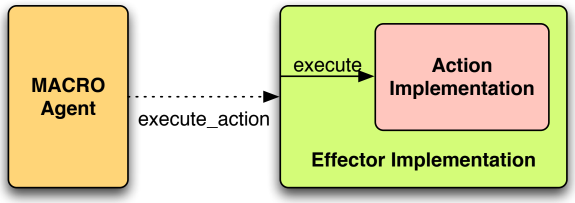


Figure 5. The Action/Effector Framework

Addressing Challenge 2: Providing Flexible Agent Provisioning—The Action/Effector framework described in Section 3 provides a mechanism through which MACRO agent implementations may be dynamically provisioned at deployment time with Action objects apropos to the particular hardware configuration, including its suite of available sensors, on which the agent resides. Component interface descriptions, similar to standard CORBA object descriptions, may have attributes of arbitrary types. As seen in Listing 5, the example component has an attribute of type `Action_Factories`, which is a sequence of structures containing a pair of string member variables indicating a library name and factory symbol name.

Component deployments are described via XML files that capture information about component configuration, topology, and connections. These XML descriptors may be used to populate the value of this attribute with desired library name/-factory name pairs at deployment time. Moreover, through the mechanism used to describe the implementation dependencies of components (*i.e.*, shared libraries implementing a component), it is possible to indicate to the NodeManager that shared libraries implementing Actions also be downloaded from the RepositoryManager, as described in Section 3. This approach allows the component providing the Effector interface to invoke the `load_action` operation for each library/entrypoint pair provided during activation.

Addressing Challenge 3: Reducing Middleware Footprint—Initial efforts to run MACRO (and the associated middleware infrastructure) presented difficulties and, in some cases, failures due to the large footprint of the default configuration of CIAO and the limited memory capacity of the SEAMONSTER nodes. To reduce memory footprint, the initial application of the deployment infrastructure to SEAMONSTER field hardware included two straightforward modifications:

• **Leverage compiler optimizations.** Most compilers have the ability to provide space-saving optimizations to most code, which an experienced programmer can easily leverage to provide footprint reduction.

• **Leverage mechanisms present in underlying middleware.** The build system of the middleware underpinning of CIAO provides configuration settings that allow one to strip unneeded features from compiled binaries, which can provide also provide substantial footprint savings in resource-constrained environments.

While these steps are relatively straightforward and not particularly novel, Section 4 shows that they were sufficient to reduce the static footprint of the middleware stack to a level that allowed successful use of the MACRO platform on SEAMONSTER hardware. Section 6 discusses the approach we are undertaking to further reduce middleware footprint/-overhead.

4. Experimental Results

This section presents the results of experiments that evaluate (1) the effectiveness of MACRO's Action/Effector framework for lightweight, hardware-dependent actions and (2) the reduction of middleware footprint described in Section 3. These results show that the efforts described reduced the total static footprint of MACRO and its underlying middleware stack. They also show the reduction in overhead achieved by implementing short-lived actions in the Action/Effector framework discussed in Section 3, rather than using heavier-weight components.

Hardware/Software Testbed and Experiment Methodology

The static footprint results were obtained via a cross-compiler toolchain used to build software for the SEAMONSTER hardware. This toolchain consists of `g++` 4.1.2 and `ld` 2.17, which are hosted on Debian Linux 4.0 and target `arm-linux-gnu`. The CIAO middleware platform was version 0.6.6.

For the initial baseline results, this platform was compiled using default options, with debugging symbols disabled and the compiler optimization level at `O3`, which instructs the `g++` compiler to optimize for speed. For the results based on our optimization efforts, the middleware was compiled using built-in methods for reducing footprint and the compiler was instructed to optimize using `Os`, which instructs the `g++` compiler to optimize for space. In all cases, we used the GNU `strip` utility to remove any debugging symbols from the compiled binaries to ensure the footprint metrics just measured the size of the executables.

Executable footprint sizes were determined by statically linking all required symbols from the underlying middleware into the final binary, ensuring that all necessary symbols from the underlying middleware are present, while not including any unnecessary symbols. For the purposes of calculating the size of a component, we assume that any symbols necessary from the underlying middleware were already present in the component server, and thus the calculation of the component footprint sizes was obtained by summing the size of the *shared* libraries that implement the component. This size includes CORBA stubs and skeletons, the servant (the component specific portions of the container), and the executor (business logic) implementation.

Runtime results were obtained using a primary microserver described in Section 2. This microserver consists of a 266 Mhz ARM processor with 64 MB of built-in RAM. The operating system is a derivative of the Debian Sarge running GNU/Linux kernel 2.4.26, which was provided by the manufacturer of the microserver (Technologic Systems).

Initial Footprint Reduction

The results of the efforts described in Section 3 are summarized in Table 4. The `ExecutionManager` and

Entity	Default	Optimized	Savings
ExecutionManager	12,203 KB	11,136 KB	1,067 KB
NodeManager	13,865 KB	12,623 KB	1,242 KB
NodeApplication	12,710 KB	11,460 KB	1,250 KB
Null Component	670 KB	605 KB	65 KB

Table 1. Results of Initial Footprint Optimization

`NodeManager` (which were described in Section 3) and the `NodeApplication` (which is a component server spawned during the deployment process) each experienced a reduction in footprint of ∼1 megabyte. The combined savings reduced the footprint of node-local infrastructure (*i.e.*, the `NodeManager` and `NodeApplication`) from 26.5 MB to 24 MB.

Although this reduction allowed us to deploy and operate a prototype MACRO-based application on the SEAMONSTER hardware, this deployment consumed nearly all available physical memory on the primary microservers, and resulted in frequent thrashing on the memory-constrained adjunct microservers. As discussed in Section 6, additional work is needed to further reduce the footprint of the infrastructure and component implementations.

Impact of Action/Effector Framework on MACRO Execution Overhead

MACRO's Action/Effector framework (described in 3) substantially reduces footprint overhead compared to using CIAO's complete component implementations to encapsulate SEAMONSTER tasks and actions. Table 4 summarizes the differences in footprint size between these two approaches.

When implemented as a component, the action has a footprint of over half a megabyte, substantially limiting the number of action implementations that could simultaneously be deployed to a single resource-limited field node.

Implementation Type	Size
Component	623 KB
Action Implementation	23 KB
Effector	123 KB

Table 2. Action/Effector Footprint

When the action was implemented in MACRO's new Action/Effector framework, however, its footprint was only 23 KB, which is a fraction of the memory required by an executing component. Moreover, an implementation of the Effector framework as a component facet adds only 123 KB to the footprint of an existing MACRO agent component, one of which is required per node.

Deployment Latency	Average Time (Seconds)
Component	218.96
Action/Effector	3.23

Table 3. Action/Effector Footprint

A more important result, moreover, is the deployment latency experienced by a component compared against the latency experienced by an Action implementation. In this case, deployment latency refers to the amount of time from the moment deployment is started (*e.g.* `load_action` in the Action/Effector framework) until deployment is completed and the component or Action is ready for invocation. As shown in Table 4, which documents the average of twenty runs of each, the difference in deployment latency is dramatic, with component deployment requiring over three minutes while an Action is deployed in only three seconds. These results do not include the time required to download the component implementation from the `RepositoryManager`, which could be substantial over a bandwidth-limited wireless connection, but is only required the first time a component is used on the microserver.

5. Related Work

This section compares our work on MACRO with related work.

Resource-Constrained Component Models. Programming in the Many (PitM) [19] is an *architectural style* aimed at the domain of distributed, highly mobile, severely resource constrained embedded systems. While this component model meets the stringent footprint requirements of SEAMONSTER, it lacks the interoperability and rich ecosystem of services offered by CORBA and CCM. PitM also limits communication between components to message-passing, lacking the rich interface-based communication possible with CIAO. The SOFtware Architectures (SOFA) component model [20] based on *Architecture Definition Languages*, which view applications as hierarchies of connected components. This component model provides capability for run-time modifications that may be lighter weight than CIAO components, but which must be described at *design time*[21], thereby limiting flexibility compared with MACRO.

Decision-theoretic planning and scheduling. The planning service used by MACRO server-based agents – SA-POP – is a decision-theoretic planner allowing uncertainty both in environmental conditions and action outcome, like C-SHOP [22] that does so with hierarchical planning and Drips [23] that produces conditional plans. However, to enable planning with resource constraints, such as those of sensor webs, many have chosen to separate the planning and scheduling/resource aspects of the problem (*e.g.*, [24] and [25]). This approach works well when the resource/time constraints are relatively loose or there are relatively few alternatives in the planning process that could use fewer or different resources. However, with tight resource constraints, as are often present in sensor webs, others have chosen to integrate planning and scheduling as SA-POP does. For example, IxTeT [26] uses partial-order planning like SA-POP and allows interleaving resource conflict resolution with the planning process, but does not perform decision-theoretic planning and incorporates scheduling/timing information directly into the action representation.

Plan schemas for resource-constrained planning and scheduling. The MACRO field agents use plan schemas (also called template plans or skeletal plans) [27], which have also been used in other situations where complete planning was too time consuming for appropriate responses. MACRO's plan schemas have been enhanced with scheduling information, such as in [28], and generated through partial order planning techniques, like [29]. The combination of MACRO server-based agents using the SA-POP planning/scheduling service with generated schemas used by MACRO field agents provides a uniquely flexible solution for autonomy in sensor webs with a server cluster connected to DRE field systems.

6. Concluding Remarks

The SEAMONSTER project exhibits distributed deployment infrastructure challenges common to computationally constrained field environments in adaptive sensor webs, including standardized execution of low-level hardware-dependent actions and on-going data tasks, automated provisioning of agents for heterogeneous field hardware, and minimizing deployment infrastructure overhead in general. This paper presents the results of applying the MACRO platform with extensions designed to address these deployment challenges in SEAMONSTER. In particular, the Action/Effector framework addressed key deployment challenges as follows:

• Separate concerns by creating a unified mechanism for deploying and executing brief, low-level actions.

• Substantially reduce footprint of individual actions versus implementation as a component.

• Improve deployment latency by two orders of magnitude over a similar component implementation.

The remainder of this section summarizes lessons learned from applying MACRO to SEAMONSTER and outlines our future work optimizing MACRO's QoS-enabled component middleware infrastructure.

Lessons Learned

The lessons learned from our extensions to the MACRO distributed deployment infrastructure include:

• **Feasible integration of non-component entities.** The Effector/Action framework has demonstrated the feasibility of integrating non-component entities into component assemblies where footprint, latency, or lifetime rules out the use of a full component.

• **Unitary Effector may limit framework flexibility.** A unitary Effector (*i.e.*, one which is incapable of operating in a hierarchical manner with other effectors) may limit flexibility in dynamic sensor web environments. Extending the Effector interface to support hierarchical and peer behavior with other Effectors deployed to the same node(s) potentially has two advantages: (1) it allows Effectors to expand their vocabulary as nearby nodes and devices power up/down in response to changing power availability and (2) it allows the creation of "meta-Actions," which are ordered compositions of one or more concrete actions across one or more Effectors.

• **A synchronous Effector interface may cause unacceptable delays.** If an Action hangs or takes longer to complete than expected, the present synchronous interface will also cause the agent plan execution code to hang. This behavior is undesirable, however, since it may cause the agent to miss other important deadlines in its current plan of execution. Asynchronous Effector and Action interfaces can alleviate this concern.

• **CIAO footprint is too large for resource constrained systems.** The stringent resource constraints (*i.e.*, 32-64 MB RAM and processors operating at 266 MHz or less) of SEAMONSTER field hardware were a significant hurdle due to the overhead (especially memory footprint) of CIAO components and deployment infrastructure. Previous CIAO developments focused on environments with significantly greater resources, *e.g.*, more than a gigabyte of RAM and processors faster than two gigahertz. While CIAO is operational on the SEAMONSTER hardware, as indicated in Section 4, further work is needed to make the middleware efficient under tight resource constraints.

• **DaNCE footprint and deployment latency is too high for resource constrained systems.** As shown in Section 4, the largest consumers of memory in the middleware stack are the DAnCE daemons, in particular the `ExecutionManager` and `NodeManager`. The footprint of the newer deployment and configuration aspects of the middleware has been largely overlooked until now and needs to be addressed. Perhaps more importantly is the latency experienced during deployment, which has been observed to take as long as several minutes on SEAMONSTER hardware.

Future Work

To further reduce the overhead of CIAO components and the DAnCE deployment infrastructure, we are working on multiple approaches, including context-aware generative techniques to prune unnecessary code/features:

• **Generative component specialization.** The CCM specification includes several features and capabilities in the component definition that may not be necessary in all situations, such as generic navigation, introspection, and security features, which contribute to footprint bloat. Generative techniques could be used to prune these features on a case-by-case basis.

• **Generative container specialization.** The CIAO container is intended to be a generic solution providing a large feature set to satisfy user needs in most situations. As such, it contains features and services that may not be necessary in specific deployments, and could be pruned by generating scenario-specific container implementations.

• **Improve separation of concerns in DAnCE.** The current DAnCE implementation tangles concerns of deployment and configuration with the run-time elements of the component server in the `NodeApplication`. This entanglement increases footprint by replicating large swathes of deployment logic in each component server. Careful analysis and refactoring is therefore needed to substantially decrease footprint and deployment latency.

ACE, TAO, CIAO, DAnCE, RACE, and SA-POP are open-source software that can be downloaded from `download.dre.vanderbilt.edu`.

References

[1] K. Delin and S. Jackson, "Sensor Web for In Situ Exploration of Gaseous Biosignatures," 2000.

[2] P. Lardieri, J. Balasubramanian, D. C. Schmidt, G. Thaker, A. Gokhale, and T. Damiano, "A Multi-layered Resource Management Framework for Dynamic Resource Management in Enterprise DRE Systems," *Journal of Systems and Software: Special Issue on Dynamic Resource Management in Distributed Real-time Systems*, vol. 80, no. 7, pp. 984–996, July 2007.

[3] O. Brown and P. Eremenko, “Fractionated Space Architectures: A Vision for Responsive Space,” in *Proceedings of the 4th Responsive Space Conference*. Los Angeles, CA: American Institute of Aeronautics & Astronautics, Apr. 2006.

[4] D. Suri, A. Howell, D. C. Schmidt, G. Biswas, J. Kinnebrew, W. Otte, and N. Shankaran, “A Multi-agent Architecture for Smart Sensing in the NASA Sensor Web,” in *Proceedings of the 2007 IEEE Aerospace Conference*, Big Sky, Montana, Mar. 2007.

[5] D. R. Fatland, M. J. Heavner, E. Hood, and C. Connor, “The SEAMONSTER Sensor Web: Lessons and Opportunities after One Year,” *AGU Fall Meeting Abstracts*, pp. A3+, Dec. 2007.

[6] N. Wang, D. C. Schmidt, A. Gokhale, C. Rodrigues, B. Natarajan, J. P. Loyall, R. E. Schantz, and C. D. Gill, “QoS-enabled Middleware,” in *Middleware for Communications*, Q. Mahmoud, Ed. New York: Wiley and Sons, 2004, pp. 131–162.

[7] *Light Weight CORBA Component Model Revised Submission*, OMG Document realtime/03-05-05 ed., Object Management Group, May 2003.

[8] *Deployment and Configuration Adopted Submission*, OMG Document mars/03-05-08 ed., Object Management Group, July 2003.

[9] J. S. Kinnebrew, W. R. Otte, N. Shankaran, G. Biswas, and D. C. Schmidt, “Intelligent Resource Management and Dynamic Adaptation in a Distributed Real-time and Embedded Sensor Web System,” Vanderbilt University, Tech. Rep. ISIS-08-906, 2008.

[10] W. R. Otte, J. S. Kinnebrew, D. C. Schmidt, G. Biswas, and D. Suri, “Application of Middleware and Agent Technologies to a Representative Sensor Network,” in *Proceedings of the Eighth Annual NASA Earth Science Technology Conference*, University of Maryland, June 2008.

[11] J. S. Kinnebrew, A. Gupta, N. Shankaran, G. Biswas, and D. C. Schmidt, “Decision-Theoretic Planner with Dynamic Component Reconfiguration for Distributed Real-time Applications,” in *The 8th International Symposium on Autonomous Decentralized Systems (ISADS 2007)*, Sedona, Arizona, Mar. 2007.

[12] N. Shankaran, D. C. Schmidt, Y. Chen, X. Koutsoukous, and C. Lu, “The Design and Performance of Configurable Component Middleware for End-to-End Adaptation of Distributed Real-time Embedded Systems,” in *Proc. of the 10th IEEE International Symposium on Object/Component/Service-oriented Real-time Distributed Computing (ISORC 2007)*, Santorini Island, Greece, May 2007.

[13] *CORBA Components v4.0*, OMG Document formal/2006-04-01 ed., Object Management Group, Apr. 2006.

[14] *The Common Object Request Broker: Architecture and Specification*, 3.0.2 ed., Object Management Group, Dec. 2002.

[15] N. Wang, K. Balasubramanian, and C. Gill, “Towards a real-time corba component model,” in *OMG Workshop On Embedded & Real-time Distributed Object Systems*. Washington, D.C.: Object Management Group, July 2002.

[16] *Lightweight CCM FTF Convenience Document*, ptc/04-06-10 ed., Object Management Group, June 2004.

[17] *Deployment and Configuration of Component-based Distributed Applications, v4.0*, Document formal/2006-04-02 ed., OMG, Apr. 2006.

[18] G. Deng, J. Balasubramanian, W. Otte, D. C. Schmidt, and A. Gokhale, “DAnCE: A QoS-enabled Component Deployment and Configuration Engine,” in *Proceedings of the 3rd Working Conference on Component Deployment (CD 2005)*, Grenoble, France, Nov. 2005, pp. 67–82.

[19] M. Mikic-Rakic and N. Medvidovic, “Architecture-level support for software component deployment in resource constrained environments,” in *CD ’02: Proceedings of the IFIP/ACM Working Conference on Component Deployment*. London, UK: Springer-Verlag, 2002, pp. 31–50.

[20] T. Kalibera and P. Tuma, “Distributed component system based on architecture description: The sofa experience,” in *On the Move to Meaningful Internet Systems, 2002 - DOA/CoopIS/ODBASE 2002 Confederated International Conferences DOA, CoopIS and ODBASE 2002*. London, UK: Springer-Verlag, 2002, pp. 981–994.

[21] P. Hnetynka and F. Plasil, “Dynamic reconfiguration and access to services in hierarchical component models,” in *CBSE*, 2006, pp. 352–359.

[22] A. Bouguerra and L. Karlsson, “Hierarchical Task Planning Under Uncertainty,” *3rd Italian Workshop on Planning and Scheduling (AI* IA 2004). Perugia, Italy*, 2004.

[23] P. Haddawy, A. Doan, and R. Goodwin, “Efficient Decision-Theoretic Planning: Techniques and Empirical Analysis,” *Proceedings of the Eleventh Conference on Uncertainty in Artificial Intelligence*, 1995.

[24] B. Srivastava and S. Kambhampati, “Scaling up Planning by Teasing Out Resource Scheduling,” *Proc. European Conference on Planning*, 1999.

[25] A. El-Kholy and B. Richards, “Temporal and Resource Reasoning in Planning: The parcPLAN Approach,” *Proceedings of the 12th European Conference on Artificial Intelligence (ECAI-96)*, pp. 614–618, 1996.

[26] P. Laborie and M. Ghallab, “Planning with Sharable

Resource Constraints," *Proc. 14th Int. Joint Conf. on AI*, pp. 1643–1649, 1995.

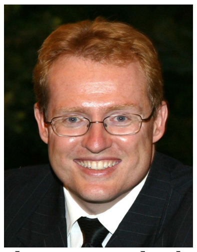
[27] P. Friedland and Y. Iwasaki, "The concept and implementation of skeletal plans," *Journal of Automated Reasoning*, vol. 1, no. 2, pp. 161–208, 1985.

[28] S. Miksch, Y. Shahar, and P. Johnson, "Asbru: A Task-Specific, Intention-Based, and Time-Oriented Language for Representing Skeletal Plans," in *Proceedings of the 7th Workshop on Knowledge Engineering: Methods & Languages (KEML-97)*, 1997, pp. 9–19.

[29] L. Ihrig and S. Kambhampati, "Design and Implementation of a Replay Framework Based on a Partial Order Planner," in *Proceedings of the National Conference on Artificial Intelligence*, 1996, pp. 849–854.

William R. Otte is a Ph.D. student in the Department of Electrical Engineering and Computer Science (EECS) at Vanderbilt University. His research focuses on middleware for distributed real-time and embedded (DRE) systems. He is currently involved in several aspects of developing a Deployment and Configuration Engine (DAnCE) for CORBA Components. This work involves investigation of techniques for run- time planning and adaptation for component based applications as well as specification and enforcement of application quality of service and fault tolerance requirements. William graduated with a B.S. in Computer Science from Vanderbilt University, Nashville TN in 2005.

John S. Kinnebrew is a PhD student in computer science at Vanderbilt University. He received his B.A. in computer science from Harvard University in 2001. His current research focuses on artificial intelligence planning and scheduling techniques in autonomous systems.



Douglas C. Schmidt is a Professor of Computer Science at the Institute for Software Integrated Systems (ISIS) at Vanderbilt University. He is a widely cited expert on patterns, frameworks, and middleware for distributed real-time and embedded (DRE) systems. Dr. Schmidt has published over 400 technical papers and 9 books and has given over 400 invited talks and tutorials that cover a range of research topics, including patterns, optimization techniques, and empirical analyses of software frameworks and domain-specific modeling environments that facilitate the development of DRE middleware and applications running over high-speed networks and embedded system interconnects. Dr. Schmidt has served as a Deputy Office Director and a Program Manager at DARPAs Information Technology Office (ITO) and Information eXploitation Office (IXO), where he led the national RandD effort on middleware and model-driven development technologies for enterprise DRE systems. In addition to his academic research and government service, Dr. Schmidt has over fifteen years of experience researching and developing ACE, TAO, CIAO, and CoSMIC, which are widely used, open-source middleware frameworks and model-driven development tools that contain a rich set of components and domain-specific languages that implement patterns and product-line architectures for mission-critical enterprise DRE systems.

Gautam Biswas is a Professor of Computer Science and Computer Engineering as well as a Senior Research Scientist at the Institute for Software Integrated Systems (ISIS) at Vanderbilt University. His primary areas of research is in modeling, simulation, monitoring, and control of complex systems as well as planning, scheduling, and resource allocation algorithms in manufacturing systems and distributed real-time environments. Dr. Biswas has published over 300 technical papers and is well-recognized for his projects in diagnosis, fault-adaptive control, and learning environments. These projects have been supported with funding from DARPA, NASA, NSF, AFOSR, and the Department of Education. He has served as an associate editor for a number of journals and has been program chair or co-chair for a number of conferences.