Proceedings of the 1997 Winter Simulation Conference
ed. S. Andradóttir, K. J. Healy, D. H. Withers, and B. L. Nelson

USING COMPENSATING RECONFIGURATION TO MAINTAIN MILITARY DISTRIBUTED SIMULATIONS

Donald J. Welch

Department of Computer Science
University of Maryland
College Park, MD 20742, U.S.A.

James M. Purtilo

Institute for Advanced Computer Studies
and Department of Computer Science
University of Maryland
College Park, MD 20742, U.S.A.

ABSTRACT

In Department of Defense (DoD) distributed training simulations, events occur that can cause unrealistic behavior. These events can be system events, such as the loss of a host or a network link to that host. They can also be events that happen only in the simulated world, such as an activity that migrates, bringing increased activity to a different federate. As a result of these events, the distributed simulation may have to restructure itself to maintain realistic behavior. The restructuring must take place in accordance with a set of rules mandated by both the domain and specific application. Restructuring decisions must rely on information from both the simulated world and the system configuration. Reconfigurations must be made quickly to minimize unrealistic behavior. Automatic reconfiguration is not only faster than manual, but automatic restructuring brings the added benefit of fewer support staff. We call the automatic restructuring of a distributed application with respect to a set of rules *Compensating Reconfiguration* and we have developed a software engineering environment that could be used to support its inclusion in DoD distributed simulations.

1 INTRODUCTION

The Department of Defense (DoD) distributed simulation domain encompasses a wide variety of uses, architectures, and techniques. DoD uses distributed simulation for test and evaluation, analysis, and training. Each of these categories place different requirements on the distributed architecture. Currently DoD has simulations that use a totally distributed approach, as discussed in DIS (1994), and Weatherly (1996) but has mandated that all simulations use a middleware approach as defined by the high level architecture (HLA) discussed in DMSO (1997). HLA is designed to support a family of simulations that include aggregate, disaggregate, and component levels of detail.

System anomalies in distributed training simulations can cause unrealistic behavior. Should a component lose its link to the rest of the simulation, those virtual entities will continue under the control of their *dead-reckoning* algorithms until they are removed from the simulation. This crude attempt at avoiding unrealistic behavior is not very effective. Starting a new copy of the component on a different host can reestablish sanity if the component was a computer generated force (CGF). For human-in-the-loop trainers this approach is not practical, because it is too expensive to keep simulators with crews just waiting for failure. Substituting a CGF for the absent trainer is more cost-effective and can successfully maintain simulation realism.

In some cases a manned simulation would only be lost temporarily. When the manned simulation returns, the distributed simulation cannot leave it out of the exercise because it represents a significant investment in resources and training opportunity. Giving the simulator control of its original entities will not always be appropriate. For example, a user regaining control of his original helicopter that has already crashed on the virtual battlefield is futile. As you can see the reintroduction of a simulator back into a simulation is a complicated decision that requires knowledge of the simulation state as well as the system configuration state.

To maintain realistic behavior the simulation must sometimes restructure itself as it is executing. This can include the starting, stopping, migrating, and initializing of individual simulations. The exercise controllers can manually reconfigure the simulation. Since the longer a reconfiguration takes the longer unrealistic behavior exists, faster is better. In addition, human controllers are costly. Automatically restructuring the simulation is a better solution.

We call the automatic restructuring of a distributed application in accordance with a set of rules *Compensating Reconfiguration*. We have developed a software engineering environment that could support its inclu-

sion in Department of Defense (DoD) distributed simulations. The Compensating Reconfiguration component created through this environment imposes a very small performance penalty on the simulation, and is not an unreasonably complex burden for the simulation builders.

1.1- Related Work

In the DoD distributed simulation domain there has been an abundance of work in defining the HLA as shown in DMSO Interface Specification(1997) DMSO Object Model Template (1997) and DMSO Rules (1996) The HLA addresses the late joining, early departure and changing ownership of federates. However, fault-tolerance does not seem to have been adequately addressed, and certainly it has not been addressed within the context of demands such as fewer support staff and human-in-the-loop simulations as pointed out by Calvin (1994).

The gluing together of disparate heterogeneous distributed systems forms the foundation of HLA. Understanding interconnection abstractions like CORBA and POLYLITH Purtilo (1994) is critical to understanding HLA. Using standard interconnection abstractions makes the development of a software engineering environment practical. These abstractions make it possible for our framework to work with existing systems without resorting to changing any of the simulations themselves.

The end-result of Compensating Reconfiguration is the dynamic reconfiguration of the application. A running distributed application might change itself three ways: structure, topography, or implementation. Changing the structure involves adding or removing components. HLA supports the dynamic joining and departing of federates from the federation. Currently HLA does not support Comensating Reconfiguration, because federates cannot reconfigure other federates as they can in other middleware such as POLYLITH.

There are two primary approaches to dynamic reconfiguration. The CONIC (Kramer 1990) approach moves the application to a quiescent state prior to reconfiguration. This approach requires logic located in each component that will migrate a component to a quiescent state in finite time. This technique may be appropriate for analysis, aggregate, and test and evaluation simulations. Quiescence does not properly describe training simulations.- Hofmeister's (1991) approach is better for training simulations. She requires the components involved to divulge their internal state, then loads this into the new component. Since federates continuously divulge their internal state (the part that the rest of the simulation cares about anyway), the software is ready for dynamic reconfiguration without change.

Some fault-tolerant techniques apply to some instances of distributed simulation as pointed out by Cristian (1991). We are working to provide a framework through which we can apply traditional fault-tolerance as well as going beyond to the rich environment of Compensating Reconfiguration. There has been some work in merging fault-tolerance and dynamic reconfiguration, in Kramer (1990) but it does not cover the complexities of the distributed simulation training domain.

2- SIMPLE MOTIVATING SCENARIO

To help illustrate the functionality required by for compensating reconfiguration we will go through a simple scenario. Our example is a military simulation that integrates manned simulators and computer generated forces (CGF) to create a rich training environment. An aviation section (one observation helicopter, two attack helicopters) supports a platoon (four tanks) during an attack. The tanks and helicopters are crewed simulators, while the rest of the friendly and enemy forces are CGFs. A local area network connects all the simulators and backups.

The goal of this exercise is to train the users. Therefore a goal of the simulation builders is to insure that a system failure will not result in the rest of the participants noticing unrealistic behavior. Additionally, the simulation must bring a manned system that becomes available again back into the exercise as quickly and seamlessly as possible. The simulation builders write the rules to achieve these goals.

The exercise starts and all participants are taking part in the war game. A attack helicopter simulator program crashes. The uncontrolled virtual helicopter continues to behave according to its dead-reckoning algorithm as implemented in all the other simulators. This is only acceptable for a situation-dependent period of time.

In this scenario, a semi-autonomous force (SAF)—which is a simulator that mostly controls its entities automatically, relying on a human operator for only "strategic" direction—is available as a spare. By making the attack helicopter a computer generated force (CGF), the exercise can continue for the rest of the users.

A second attack helicopter simulator fails because there is a break in the network connection. Since the SAF spare is not available, the proper compensation would be to start another simulator. In this case, the secondary spare is a very simple simulator that is completely automated and will only gracefully remove the entity from the simulated battle. It takes

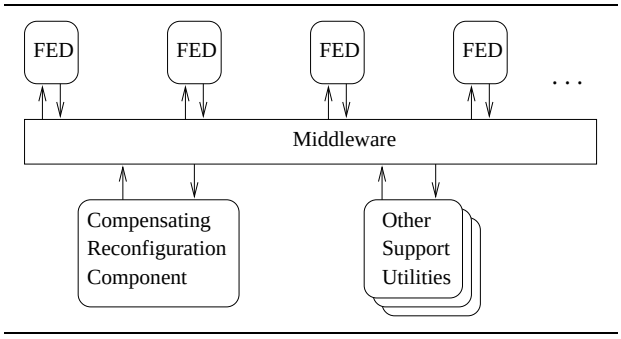

Figure 1: Distributed Simulation Architecture with Compensating Reconfiguration Component

control of the attack helicopter and tells the other participants that it has a mechanical problem and lands the aircraft immediately.

The network problem clears, and the attack helicopter crew cannot just be left out of the simulation. They must be reincorporated, to get full value from the simulation and the time invested in the exercise. Just giving the crew control of its original attack helicopter entity would be futile, since that entity can no longer realistically participate in the simulation. A better solution would be for the crew to control the other attack helicopter which is still involved in the attack.

Just from this very simple example, you can see the complexities involved in determining the proper compensation for an event. Decisions will have to be based on both the configuration state and the simulation state. Since both are dynamic, the logic must be adaptive.

3- COMPENSATING RECONFIGURATION REQUIREMENTS

To make the concept more understandable we have delineated the requirements for a Compensating Reconfiguration component in the DoD HLA domain. Figure 1 shows how a Compensating Reconfiguration component fits into a middleware architecture. It joins the federation as another federate and listens to all messages passed within the federation. When it compensates for an event, it does so through the middleware primitives, taking advantage of the middleware's services. Guidelines for simulation developers to take better advantage of Compensating Reconfiguration are beyond the scope of this paper.

Coherence. A necessary, but not sufficient condition for a distributed simulation to maintain realism is for each virtual entity to be controlled by one and only one simulation at all times. This condition we call coherence and is the driving principle for Compensating Reconfiguration.

Event-Detection. A Compensating Reconfiguration component must react to multiple system and application event types. Ideally the Compensating Reconfiguration component can detect system events by listening to the application message traffic. This is low-cost and unobtrusive. Although not preferred, active measures can be used with higher, but still low, cost. Additionally, it must be able to detect events that are defined only in the application state.

Simulation/Configuration Mapping. Because decision making requires access to both the state of the simulated world as well as the configuration state, the software must be able to map between the two, using information from both to make a decisions.

State Monitoring. To base decisions on the simulation state the component must have access to that state. Because distributed simulations may include many entities each with a host of attributes, access to the simulation state must be in abstract form to keep the decision logic from getting overly complicated.

Decision-Making. When the Compensating Reconfiguration component identifies an event, it must decide which compensating action is appropriate. It must do this for all the event types it handles, and it must decide based on its defining rules and the current state of the simulation and configuration. The rules must be expressive enough to handle complicated conditions while allowing the simulation builder to think in terms of end-results rather than implementations. Distributed simulations can create events simultaneously; consequently the Compensating Reconfiguration system must be capable of concurrently handling multiple decisions. As the application executes, to include reconfiguring itself, the decision-making logic must take those changes into account.

Reconfiguration. The HLA interface specification provides the means for components to join and depart the federation during execution. (DMSO Interface Specifcation 1997) Most other middleware allows this process to be managed by components that are not participating in the reconfiguration. Not only must the simulation builder have access to the full power of the API, but he must have a powerful way of employing that API. He must have access to the state information, as well as be able to apply loops, conditionals, and waits. In addition, Compensating Reconfiguration components must insure that regardless of the frequency, order, or timing of reconfigurations the simulation will always end up with a coherent structure.

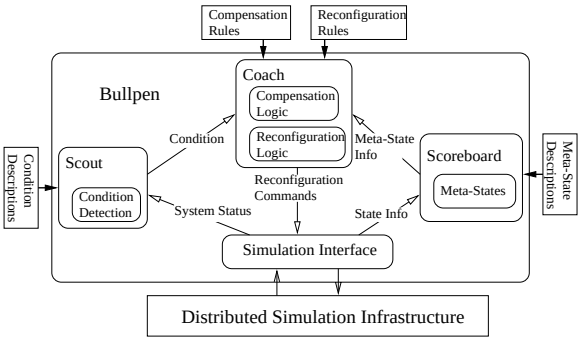

Figure 2: Bullpen Structure and Data Flow

4- BULLPEN: A COMPENSATING RECONFIGURATION FRAMEWORK

Our approach to solving this problem is a software engineering environment called Bullpen. A key part is the Bullpen framework, which includes three components. Figure 2 shows the components of the Bullpen Framework and the data flow between the components and the simulation. Bullpen gets its policies at runtime through the use of various files.

We have developed the Bullpen prototype on a simulation architecture based on DoD's High Level Architecture (HLA). Bullpen runs as a support utility; it only inserts itself into the simulation when required. Most of the time it will monitor all the message traffic of the simulation and do nothing. If the simulation is using multicast or broadcast then there is no real overhead. With unicast there is the slight penalty of an additional network message per simulation message. When it detects an event of interest, Bullpen then makes two decisions. The first decision determines a compensating reconfiguration, while the second decides how to implement that reconfiguration.

Bullpen has three independently executing components. It is Scout's mission to detect events of interest and report them along with any preliminary information to Coach. Coach contains the decision-making logic for both the compensation and reconfiguration decisions. Coach directly executes the reconfigurations by using the services provided by the distributed simulation infrastructure. Scoreboard is responsible for keeping a valid copy of both the application and system state.- The application state includes *meta-states*. Meta-states keep the compensating reconfiguration logic simple while minimizing changes when requirements change. In our prototype this includes implementation of the dead-reckoning algorithm.

Bullpen has at least five major threads running whenever it is executing. Scout continues to handle incoming messages and to test for absent simulators regardless of what else Bullpen is doing. This way Bullpen can identify an event when it happens. Coach manages simultaneous events by spawning a new thread for each event it handles. Scoreboard deals with queries and updates in one thread while updating the database through dead-reckoning in another. This way we insure that the database always represents an accurate picture of the state.

Scout. Scout reifies the event detection function. It keeps its own small lists, but otherwise relies on Scoreboard for application state information. Anytime Scout observes a message from a federate it updates that federate's heartbeat. Scout is also party to reconfigurations made by Coach so that it understands the current structure of the simulation. It detects events two ways: by discovering messages from federates that Bullpen has reconfigured out of the simulation, and by detecting an expired heartbeat.

Coach. Coach uses a separate type of Decision-Maker for each type of event that it handles. Each Decision-Maker uses the rules defined by the simulation developers and the current state to choose a compensating action. Compensating actions are implemented as a Strategy, which can be a Java^{TM} method or a *Partial-Order-Planner*. When the Strategy is a method, it can use conditionals, loops, and waits while it executes the reconfiguration. It also has the ability to query for system and/or application state. This is very powerful, but forces the developer to deal with the reconfiguration protocol. A planner works off a set of rules and is easier to reason about, but may be slower. Coach also uses a locking protocol to ensure that reconfiguration actions are atomic and leave the federation structure consistent.

Scoreboard. Scoreboard's purpose is to keep an accurate version of the system and application state and provide an interface for queries. Scoreboard monitors message traffic updating itself, to include using *triggers*—The same concept found in Relational Databases, a change in the data causes a *trigger* to fire. That trigger executes an algorithm that may modify the database and recursively cause other *triggers* to fire. to keep accurate meta-states. The federation's dead-reckoning algorithms are part of Scoreboard.

Bullpen needs access to the application meta-state while making a decision. In large simulations making decisions using the state of entities can be very complex. When stating requirements, simulation users will tend to think in terms of aggregate states of virtual object groups. These we call meta-states.

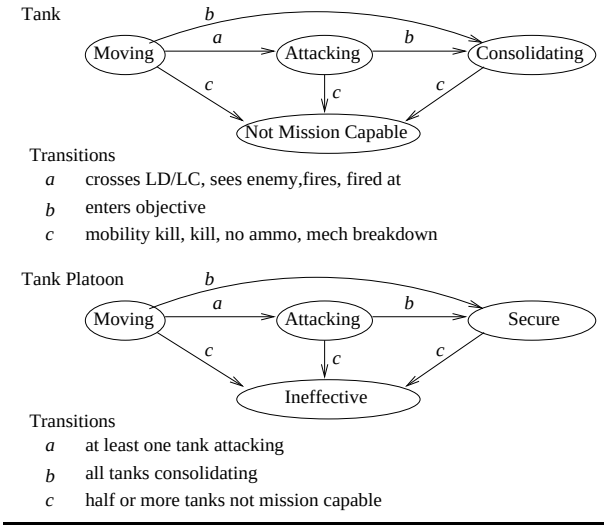

Figure 3: State Transition Diagram for an Individual Tank and a Grouping of Tanks

To better explain meta-states look at figure 3. The top diagram shows the transitions and rules for a single virtual object—a tank. The bottom diagram shows the state transitions for a unit—a tank platoon—containing four tanks. The individual tanks change state based on their location, status, or the actions of other virtual objects. The tank platoon changes state when all or some of the member tanks change state. If we were to follow this hierarchy to division level (∼5000 vehicles) you can see that the complexity of the conditionals required to use the state would be enormous. This makes using the application state unrealistic. However, by encoding state changes and relationships in the database through the use of a trigger, the use of meta-states of the levels in the hierarchy becomes very convenient.

4.1- Bullpen Advantages

Using Bullpen yields sufficient compensating reconfiguration functionality for less effort than a traditionally programmed approach.- Bullpen combines rule-based interfaces and a *software kit* to provide the performance, expressiveness, and capability necessary to support the distributed training simulation domain. As a result, less effort if required to build the functionality from scratch, but as the requirements evolve, it is easier make the appropriate changes. A side benefit of this is that prototyping requires less effort, and is therefore more likely to be included in a project. Since the purpose of compensating reconfiguration is to implement engineering decisions about the allocation of resources against the risks, the more

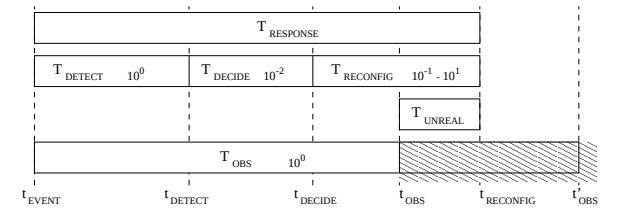

Figure 4: Reconfiguration Timeline (not to scale)

knowledge the designers have the more likely they will develop a good allocation design.

While providing these benefits, it is crucial that the resulting software meet the performance requirements for compensating reconfiguration. Performance can be broken down into three broad areas: expressiveness, response-time, and correctness.

When using an abstraction to simplify a task, we trade full control and knowledge of the process for simplicity. By using a rule-based approach we allow the developers to reason about compensating reconfiguration while hiding the details. The risk is that we have hidden details that are critical to a correct solution; out abstraction is not expressive enough to meet the domain's requirements. To guard against this, we are experimenting with both synthetic scenarios and actual DoD scenarios to find requirements that we cannot satisfy due to our abstractions being overly general. So far we have not found this to be a shortcoming of our design.

Normally, the implementation of an abstraction will include some performance penalty over a less abstract approach. We feel that some response-time degradation is acceptable, as long as it does degrade the usefulness of the application. Almost any automated solution will be faster then using human controllers to reconfigure the system, so comparing to that standard is not very useful. A well optimized hand- crafted component can normally outperform an abstract implementation. However, if faster response-time is not the only consideration, some relaxation of this standard is acceptable.

We look at response-time by it down into three components and comparing the abstraction penalty to the other parts of the total response-time. Figure 4 shows the timeline for a reconfiguration. These times are all very hardware dependent so we limit ourselves to an order of magnitude analysis. t_{EVENT} is the time the event occurs. t_{detect} is when Bullpen detects that event. With heartbeats, this will average half the heartbeat expiration, measured in seconds.(DIS 94) t_{DECIDE} marks the completion of Bullpen's decision

process. $t_{RECONFIG}$ is the time that the reconfiguration of the simulation is complete. The elapsed time T_{DECIDE} is the only part that is attributable to the logic which has an abstract interface. Usual values are in hundredths of a second. In the best case where the simulation builders dedicated a *hot spare* to this event $T_{RECONFIG}$ would have a duration measured in tenths of a second. At the other end of the spectrum, loading and initializing a simulator would give $T_{RECONFIG}$ a value measured in tens of seconds.

t_{OBS} is the time where a user of the distributed simulation will observe unrealistic behavior. It will occur at some non-deterministic interval (usually seconds) after t_{EVENT}. If it is also after $t_{RECONFIG}$, say at t'_{OBS} then no unrealistic behavior occurs. If it is before $t_{RECONFIG}$ then the simulation will display unrealistic behavior during the interval T_{UNREAL}. The duration of T_{UNREAL} is dependent on $T_{RESPONSE}$. $T_{DETECT}(10^0)$ dominates T_{REACT} in the best case, and $T_{RECONFIG}(10^1)$ dominates in the worst case. T_{DECIDE} is never the dominant term. Therefore the penalty imposed by our abstraction is not a critical part of the response time.

The final part is correctness. The created component must be able to provide correct reconfigurations, if it is designed to meet the requirements. We assume that all participating simulations have passed validation and verification, so that given a correct view of the simulation state they will produce realistic behavior. We deem a compensation correct, if afterwards, each entity is controlled by one and only one component that has all the information and authorization that it needs to manage that entity.

5 CONCLUSIONS

We have implemented a number of scenarios using our approach to compensating reconfiguration as well as a more traditional high-level language approach. We have found that our approach provides sufficient performance for the scenarios that we have investigated. Our rule-based interface is well-suited to expressing both the compensation and reconfiguration logic. We have not found any requirements that are excessively difficult to implement using our system, and in fact we have found it much simpler to reason about reconfigurations as sets of rules rather than procedural algorithms. Although there is a performance penalty, we have found it not be perceptually significant. Our properly built implementations, have resulted in correct reconfigurations.

In terms of the effort required to produce compensating reconfiguration functionality, we have found great advantages to our approach. We have found that on the average the initial implementation takes less than half the time of our hand-coding approach. Where we really found a large advantage was when we slightly changed the requirements as would occur when prototyping. Because Bullpen encapsulates the different logic involved, changes where isolated to specific parts of bullpen and very easy to make.

We have shown Compensating Reconfiguration as a viable technique for use in distributed simulation. It can maintain realism in the presence of failures. We have shown it is also useful to automatically keep the system optimally configured as the simulation evolves. Compensating Reconfiguration can be critical to maintaining realism in training simulations, and it can be reduce the support staff.

We have identified a weakness in current distributed simulations system research. The unexpected loss, or reentry of a federate can result in unrealistic behavior. Reacting to these anomalies requires more complex logic than traditional fault-tolerance provides and faster reactions than current dynamic reconfiguration systems. We have identified a solution to this and in the process developed a software engineering environment to create components that allow distributed simulations to perform compensating reconfiguration in response to any type of event. Simulation builders can use this component to allocate resources and avoid unrealistic behavior. It performs within the time constraints of training simulations and shields the simulation builder from the details involved.

ACKNOWLEDGMENTS

This research was funded by Office of Naval Research contract number N000149410320.

REFERENCES

Calvin, J. O., and D. J. Van Hook. 1994. AGENTS: An Architectural Construct to Support Distributed Simulation. In *11th Distributed Interactive Simulation Standards Workshop.*

Cristian, F. 1991. Understanding Fault-Tolerant Distributed Systems. *Communications of the ACM*, 34 (2): 57–78.

Defense Modeling and Simulation Office. 1996. High Level Architecture Rules, Version 1.0.

Defense Modeling and Simulation Office. 1997. High Level Architecture Interface Specification, Version 1.1.

Defense Modeling and Simulation Office. 1997. High Level Architecture Object Model Template, Version 1.1.

DIS Steering Committee. 1994. The DIS Vision, A Map to the Future of Distributed Simulation, Ver-

sion 1.

Gibson, T. J. 1992. Developing a Command and Control System in War. *IEEE Communications.*

Hofmeister, C. R. and J. M. Purtilo. 1991. Dynamic Reconfiguration of Distributed Programs. In *Proceedings of the 11th International Conference on Distributed Computing Systems*, 560–571.

Kramer, J. and J. Magee. 1990. The Evolving Philosophers Problem: Dynamic Change Management. *IEEE Transactions on Software Engineering*, 16 (11): 1293-1306.

Kramer, J., J. Magee and A. Young. 1990. Towards Unifying Fault and Change Management. In *Proceedings of the IEEE International Workshop on Distributed Computing Systems in the 90's*, 57–63.

Purtilo, J. M. 1994. The POLYLITH Software Bus. *ACM Transactions on Programming Languages*, 16: 151-174.

Weatherly, R. M., A. L. Wilson, B. S. Canova, E. H. Page, A. A. Zabek and M. C. Fischer. 1996. Advanced Distributed Simulation through the Aggregate Level Simulation Protocol. In *29th International Conference on System Sciences* 407–415.

AUTHOR BIOGRAPHIES

DONALD J. WELCH is a Lieutenant Colonel in the United States Army and a Ph.D. candidate at the University of Maryland College Park. Upon completion of his study he will become a member of the Department of Electrical Engineering and Computer Science, United States Military Academy, West Point. He received the B.S. degree from West Point and the M.S. degree in computer science from California Polytechnic State University at San Luis Obispo. His research interests are in the dynamic reconfiguration and other systems and software engineering aspects of distributed simulations.

JAMES M. PURTILO is an Associate Professor in the Department of Computer Science at the University of Maryland College Park and also holds an appointment in the University of Maryland Institute for Advanced Computer Studies. He received the B.A degree in mathematics from Hiram College, Hiram, OH, the M.A. degree in mathematics from Kent State University, Kent, OH, and the Ph.D. degree in computer science from the University of Illinois at Urbana in 1986. His research focuses on interoperability issues that arise when software modules are composed into a large system.