Building Self-Reconfiguring Distributed Simulations Using Compensating Reconfiguration

Lt. Col. Don Welch, *U.S. Military Academy*
James Purtilo, *University of Maryland*

In distributed training simulations, simulators can lose their connections to the rest of the simulation. When this happens, the uncontrolled virtual entities exhibit unrealistic behavior. To avoid unrealistic behavior, the distributed simulation must reconfigure itself based on the state of the simulation software and the virtual world. 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 Department of Defense (DoD) distributed simulations.

The DoD distributed simulation domain encompasses a variety of uses, architectures, and techniques. DoD uses distributed simulation for test and evaluation, analysis, and training. Each of these categories brings with it different requirements for the distributed simulation architecture. Currently, DoD has simulations that use a totally distributed approach, as discussed in [1] and [2] but has mandated that all simulations use a middleware approach as defined by the high-level architecture (HLA) discussed in [3,4,5]. HLA is designed to support a family of simulations such as uses mentioned above and aggregate, disaggregate, and component levels of detail.

Failures in distributed training simulations can cause unrealistic behavior. Should a simulator crash or lose its link to the rest of the simulation, virtual objects the simulator owns will continue under the control of their *dead-reckoning* algorithms until they are removed from the simulation. There are ways to provide more realistic behavior. Starting a new copy of the simulation on a different host can re-establish sanity if the simulator requires little or no human involvement. For human-in-the-loop trainers, this approach is not practical because it is too expensive to maintain simulators with crews that do nothing but wait around for failures. To substitute a computer-controlled simulator for the absent trainer is more cost-effective and also can successfully maintain simulation realism.

In some cases, a manned simulator would only be lost temporarily. When the human-in-the-loop system returns, it cannot simply be left out of the exercise as would be a computer-controlled simulator. The crew represents a significant investment in resources and training opportunity. To give the manned simulator control of its original objects will not always be appropriate, such as returning control of the original helicopter to a user when it has already crashed on the virtual battlefield. To reintroduce a simulator back into a simulation is a complicated decision that requires knowledge of the virtual world as well as the simulation configuration.

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 inclusion in DoD distributed simulations. The compensating reconfiguration component created through this environment imposes an extremely small performance penalty on the simulation and is not an unreasonably complex burden for the simulation builders.

Related Work

In the DoD distributed simulation domain, there has been an abundance of work that defines the HLA [3,4,5]. The HLA addresses the late joining, early departure, and changing ownership of federates (simulator components). 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 [1].

The gluing together of disparate heterogeneous distributed systems forms the foundation of HLA. Understanding interconnection abstractions like Common Object Request Broker Architecture [6] and Polylith [7] 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 components. We feel that compensating reconfiguration is best built into the interconnection abstraction and provided as a service.

The end result of compensating reconfiguration is the dynamic reconfiguration of the application. There are two primary approaches to dynamic reconfiguration. The Conic approach moves the application to a quiescent state prior to reconfiguration [8]. This approach requires logic located in each component that will migrate a component to a quiescent state in finite time. This technique is more appropriate for simulations that do not run on wall-clock time. A virtual simulation cannot achieve a quiescent state and still maintain realistic behavior. C. Hofmeister's

approach is better suited for virtual simulations [9]. She requires that the components involved divulge their internal state, then loads this into the new component. Since simulators in a distributed simulation continuously divulge their internal state (which is most important to the rest of the simulation), the software is ready for dynamic reconfiguration without change.

N. Minsky has used laws to ensure consistency in the software architecture as it evolves. A set of invariant laws is enforced throughout the lifecycle of the software using independent monitoring [10]. We focus on keeping the behavior of the distributed system consistent throughout its execution.

Compensating Reconfiguration

Distributed simulations require dynamic reconfiguration to keep correct execution in the presence of external failures. The proper compensation for a failure is not always readily apparent. Making the correct compensation requires taking the current software and hardware configuration and status into account and can require the virtual world state and a mapping between the two. Current dynamic reconfiguration techniques provide only for considering system configuration and not the application state.

To compensate for an external condition can involve complex decisions. Straightforward, like-for-like substitutions are not always appropriate. To compensate reconfigurations involves heterogeneous changes to the simulation. By heterogeneous, we mean that a different type of simulator is substituted for the original. In the motivating example, it is impractical to keep a crewed simulator as the backup to the attack helicopter flight simulator. Another factor that adds to the complexity is that compensation decisions cannot rely solely on the current configuration of the distributed simulation. As shown in the motivating example, the internal state of the simulators must sometimes be taken into account. Since the system configuration and the simulator state are not static, the compensation logic must be dynamic, too.

The main concepts of compensating reconfiguration are first, mapping the virtual world state and system configuration; and second, using an abstract interface to build the decision logic. To maintain this mapping in software is complex. When requirements change, the more concentrated the code changes, the easier code changes are to make and the less likely they are to be in error. Using an abstract interface for the reconfiguration and compensation decisions allows the user to keep in mind the big picture and not become distracted by the dynamic reconfiguration implementation details.

We have built Bullpen, a tool to build compensating reconfiguration software in the distributed simulation domain. We named it Bullpen because as baseball managers must change their pitchers to meet the changing conditions of a game, our software must substitute simulators. Bullpen currently runs as an invisible support utility. It could just as easily be integrated into the run-time infrastructure if one is used. When it detects a condition of interest, Bullpen makes two decisions. The first decision determines the appropriate compensation for the condition. The second is how to dynamically change the structure of the distributed simulation to meet the desired configuration and maintain realistic simulation behavior.

Results

Our goal has been to produce compensating reconfiguration code with less effort that also is more accurate than using only a high-level programming language. In addition, we want to ensure that the compensating reconfiguration code can perform all the reconfigurations required by current applications. Finally, we want the execution speed penalty to be low enough to ensure that this is a practical approach.

We did a pilot study to determine whether our tool warrants further evaluation. In this pilot study, we used a number of different scenarios, all based either on military uses or military simulation exercises. For each scenario, we built the initial versions of the compensating reconfiguration software either in Java™ or with Bullpen. We then changed the requirements for the simulation and modified the software to match the new requirements. As we built and tested, we collected the metrics discussed below.

A lack of expressiveness in Bullpen's abstract interface would manifest itself in the worst case by our inability to perform one or more of the requirements changes. Since we were able to do all the changes from the scenario, Bullpen satisfies this provision. A less drastic lack of expressiveness would show itself through increased effort and complexity of the code needed to implement the changes. We did not face this situation, so we concluded that Bullpen is expressive enough as it stands.

We looked at six categories of requirements change. Changes to the reconfiguration interface—or the ways the reconfiguration code must interact with the distributed simulation infrastructure—are part of this category. An example is a new release of the run-time infrastructure with an updated application programming interface. Changes to the Reconfiguration Policy represent revisions to the choice of possible compensating dynamic reconfigurations. The Virtual Configuration category contains changes to the simulated world. This includes both the number and the associations between virtual entities. Likewise, changes to the System Configuration category include changes to the hosts or the software components that compose the distributed simulation. Changes to Virtual and System Configuration include requirements changes that involve the virtual world, the system configuration, and the mapping between them. The final category includes changes to the Conditions handled. An example of a condition is the return of a simulator. The Reconfiguration Policy and System Configuration are the changes the simulation builders are most likely to make during the prototyping phase. Changes to the Virtual Configuration are most likely to come from the users as they refine their concept for the simulation.

Change Category	SLOC	Time
Virtual Configuration (User Driven)	23%	20%
System Configuration (Prototyping)	35%	85%
Virtual Configuration and System Configuration	11%	17%
Reconfiguration Policy (Prototyping)	28%	37%
Reconfiguration Interface	68%	41%
Conditions	83%	87%

Table 1. *Effort compared to high-level language implementation.*

Effort

We focused on the effort required to implement requirements changes. Our experience with the military domain shows that the requirements will change so many times that the effort spent to modify the compensating reconfiguration code will overshadow the initial construction effort. In addition, the ease of implementing changes makes for an effective prototyping tool. The effort metrics we used were source lines of code (SLOC)[1] and time. The initial effort using Bullpen averaged 84 percent of the effort required to implement the same functional system using only high-level code (Table 1).

The effort required to change the functionality of the compensating reconfiguration software was much less with Bullpen than with a high-level source code approach. Bullpen did not perform as well when the changes were to the System Configuration as it did in the other categories, but these were the simplest changes to implement in both systems.

Complexity

We also examined the complexity of the modifications as a result of the requirements change. We reasoned that less complex code is easier to build and less error prone. We used three metrics to determine complexity: number of locations in the code modified, number of defects found during integration test, and repair time for those defects (which includes all types of defects, regardless of their cause). Our reasoning was that more complex code will tend to produce more defects, and those defects will be more difficult to repair. The complexity of the code used to build the initial systems using Bullpen was only 68 percent of the complexity of the high-level code version.

In the most common categories of requirements change, Bullpen showed the best performance (Table 2). As the composition of the simulation evolved, Bullpen was far less complex to deal with than high-level language. The changes to the conditions category were the worst performers again. In the areas of change most commonly encountered in military simulations, Bullpen far out-performed the conventional approach.

Correct Reconfigurations

We also looked at Bullpen's tendency to produce correct reconfigurations. We define a correct reconfiguration as one that results in all objects in the virtual world being controlled by only one executing simulator. We assume that all the simulators in the distributed simulation have been validated and verified. Therefore, objects under the control of a validated and verified simulator will behave realistically. An object not under control of a functioning simulator is bound to eventually behave unrealistically. Should an object be under the simultaneous control of two simulators, it also is not guaranteed to behave in a realistic manner. We assume that a compensating reconfiguration component that makes more incorrect reconfigurations in integration test is more likely to make incorrect reconfigurations in actual use. Bullpen showed that is it equal to or better than high-level language in all categories (Table 3).

Response Time

Finally, we looked at response time. Abstract interfaces generally impose a performance penalty as a cost of an easier-to-understand interface. For Bullpen to be a worthwhile tool, the performance penalty must be within acceptable limits. Since 100 milliseconds is perceived as instantaneous by humans, we were willing to accept a penalty of about 100 milliseconds. Bullpen-generated code took an average of 102 milliseconds longer to determine the correct reconfiguration. In neither case was the decision time significant with respect to the total response time, which demonstrates that the Bullpen approach is fast enough to be practical.

Conclusions

Through our work in the distributed simulation domain, we believe that it is possible to build self-reconfiguring distributed systems using an abstract interface. The advantages of developing distributed systems this way include less effort, less complicated code, and fewer errors. A rule-based abstract interface is powerful enough to handle the reconfiguration requirements found in the military distributed simulation domain. In addition, the response time is fast enough for virtual simulations.

With Bullpen, we can build compensating reconfiguration components with less initial effort, but more important, the code is less complex and easier to modify in response to changing requirements. We found that changes to requirements that involve the virtual world, both the virtual world and system configuration, the reconfiguration policy, and the reconfiguration interface showed the greatest gains

Table 2. *Categories of change compared to high-level language implementation.*

Change Category	Locations	Defects	Repair Time
Virtual Configuration (User Driven)	21%	17%	7%
System Configuration (Prototyping)	63%	100%	100%
Virtual Configuration and System Configuration	25%	100%	100%
Reconfiguration Policy (Prototyping)	19%	14%	8%
Reconfiguration Interface	32%	14%	13%
Conditions	125%	100%	100%

Change Category	Average Incorrect Reconfigurations
Virtual Configuration (User Driven)	25%
System Configuration (Prototyping)	100%
Virtual Configuration and System Configuration	100%
Reconfiguration Policy (Prototyping)	11%
Reconfiguration Interface	27%
Conditions	100%

Table 3. *Categories of change and performance compared to high-level language implementation.*

using Bullpen. These categories also represent the most common types of requirements changes that occur in the military distributed system domain.

The effort and complexity saving shown by this approach supports prototyping. Experimenting with different mixes of spare resources and reconfiguration policy should allow the builders to achieve more effective self-reconfiguring code. Our approach lowers the cost of this experimentation.

We found that our system was more than powerful enough to handle the requirements of military distributed simulations; therefore, we believe that this approach will generalize to other distributed systems that must reconfigure themselves during execution in response to changing conditions. Even though our initial work has been with the compensating reconfiguration function as a component of the distributed program, we believe the proper place for compensating reconfiguration is in the middleware. ◆

About the Authors

Lt. Col. Don Welch is an associate professor of computer science at the U.S. Military Academy (USMA). He teaches software engineering and has experience as an Army software engineer. His military assignments include infantry and special missions units. His current research interests include dynamic reconfiguration, software engineering, managing the risk from year 2000 failures, and distributed simulation. He has a bachelor's degree from USMA, a master's degree in computer science from California Polytechnic State University, and a doctorate from the University of Maryland.

Department of Electrical Engineering and Computer Science
United States Military Academy
West Point, NY 10996
E-mail: Donald-Welch@usma.edu

James Purtilo is an associate professor of computer science at the University of Maryland, where he also holds an appointment in the Institute for Advanced Computer Studies. He is a senior member of the Institute of Electrical and Electronics Engineers, having previously received a doctorate from the University of Illinois at Urbana. His research is in software engineering, with a special focus on software interconnection. Most recently, he served as general chairman for the Fourth International Conference on Configurable Distributed Systems.

Department of Computer Science
University of Maryland
College Park, MD 20741
E-mail: purtilo@cs.umd.edu

References

1. Calvin, J. and D. Van Hook, "Agents: An Architectural Construct to Support Distributed Simulation," *Proceedings of the 11th Distributed Interactive Simulation Standards Workshop*, September 1994.
2. Weatherly, R., A. Wilson, B. Canova, E. Page, A. Zabek, and M. Fischer, "Advanced Distributed Simulation Through the Aggregate-Level Simulation Protocol," *29th International Conference on System Sciences*, Wailea, Hawaii, Jan. 3-6, 1996, pp. 407-415.
3. Defense Modeling and Simulation Office, *High-Level Architecture Rules*, Version 1.2, August 1997.
4. Defense Modeling and Simulation Office, *High-Level Architecture Interface Specification*, Version 1.2, August 1997.
5. Defense Modeling and Simulation Office, *High-Level Architecture Object Model Template*, Version 1.1, February 1997.
6. Siegel, J., *CORBA Fundamentals and Programming*, Wiley Computer Publishing Group, New York, 1996.
7. Purtilo, J., "The Polylith Software Bus," *ACM Transactions on Programming Languages*, Vol. 16, January 1994, pp. 151-174.
8. Kramer, J. and J. Magee, "The Evolving Philosopher's Problem: Dynamic Change Management," *IEEE Transactions on Software Engineering*, Vol. 16, No. 11, November 1990, pp. 1293-1306.
9. Hofmeister, C. and J. Purtilo, "Dynamic Reconfiguration of Distributed Programs," *Proceedings of the 11th International Conference on Distributed Computing Systems*, 1991, pp. 560-571.
10. Minsky, N., "Independent On-Line Monitoring of Evolving Systems," *Proceedings of the 18th International Conference on Software Engineering*," March 1996.

Note

1. Size includes the number of SLOC added, modified, and removed. If code was made "dead" or nonreachable by other modifications, it was removed and not counted.