Domain-Driven Reconfiguration in Collaborative Virtual Environments*

Donald J. Welch Jr.
dwelch@cs.umd.edu
Department of Computer Science
University of Maryland

James Purtilo
purtilo@cs.umd.edu
Institute for Advanced Computer Studies
and Department of Computer Science
University of Maryland

March 27, 1997

Abstract

When virtual environments (VE) collaborate to create a shared virtual world, events occur that can have catastrophic effects on that virtual world. 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 virtual world, for example, a virtual activity that migrates, bringing increased activity to a different VE. To maintain acceptable or realistic behavior can require the restructuring of the collaborative virtual environment (CVE) during execution. The restructuring must take place in accordance with a set of rules mandated by the domain and specific application. The reconfiguration must occur quickly, to maintain realism for the users. Automatic restructuring brings the added benefit of fewer support staff. We call the automatic restructuring of a distributed application with respect to these rules *Domain-Driven Reconfiguration* and we have developed a software engineering environment to support its inclusion in CVEs.

Keywords: dynamic reconfiguration, virtual environments

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

INTRODUCTION

When building Collaborative Virtual Environments (CVE) the developers must present a realistic view of a virtual world to the users. One element is dealing with system anomalies. System anomalies in distributed simulations can cause unrealistic behavior. Should a component lose its link to the rest of the simulation, its virtual entities will either continue under the control of their *dead-reckoning* algorithms or disappear from the simulated world. In either case, unrealistic behavior eventually results. Starting a new copy of the component on a different host can reestablish sanity if the component is under the autonomous control of a computer. Human-in-the-loop virtual environments (VE) do not make good *hot spares*. It is too expensive to keep VEs with users just waiting for failure. Substituting an application to control the entities with no human-in-the-loop can successfully maintain realism.

In some cases a VE would only be lost temporarily. When the VE returns, the CVE cannot leave it out of the system because that user represents a significant investment in resources and training opportunity. Giving the VE control of its original entity will not always be appropriate. For example, a user regaining control of his helicopter that has already crashed on

the virtual battlefield is inutile. The reintroduction of a component back into a simulation is a complicated decision that requires knowledge of the system state as well as the virtual state.

System anomalies are not the only types of events that can cause unrealistic behavior in CVEs; virtual events can too. Suppose the CVE builders set up a virtual world with the environmental entities (bridges, buildings, etc.) controlled on low-power hosts. Unexpectedly, many forces from both sides move into an area controlled by just one application. At this point there is not an increase in the use of system resources, but as the battle unfolds, the environmental entities will begin to change at an increased rate. If the environmental applications cannot keep up, realism will suffer. Migrating the application to a more powerful host can maintain realism, but it must be done before unrealistic behavior begins.

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 VEs. The controllers can manually reconfigure the CVE, however a human controller may not be fast enough. In all cases human controllers are costly. Automatically restructuring the simulation is a better solution.

We call the automatic restructuring of a distributed application *Domain-Driven Reconfiguration* and we have developed a software engineering environment to support its inclusion in CVEs. The Domain-Driven Reconfiguration component created through this environment imposes a very small performance penalty on the simulation, and is not an unreasonably complex burden for the CVE builders.

Related Work

The gluing together of disparate heterogeneous distributed systems forms the foundation on which we built this work. Using standard interconnection abstractions makes the development of a software engineering environment practical. Our solution supports interconnection abstractions like POLYLITH,[10] HLA,[9] and CORBA.[11] These abstractions make it easy for our framework to tap into existing systems without changing any of the component VEs.

The end-result of Domain-Driven Reconfiguration is the dynamic reconfiguration of the CVE. A running distributed application might change itself three ways: structure, topography, or implementation. Changing the structure involves adding or removing components. Changing topography requires the migration of a component to a different host. Changing the implementation means substituting a different version of a component. Domain-Driven Reconfiguration requires and supports all three types of reconfiguration.

There are two primary approaches to dynamic reconfiguration. The CONIC approach moves the application to a quiescent state prior to reconfiguration.[5] This approach requires logic located in each component that will migrate a component to a quiescent state in finite time. A quiescent state is not realistically achievable for CVEs. Hofmeister's approach is better for CVEs.[4] She requires the components involved to divulge their internal state, then loads this into the new component. Since VEs in a CVE 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.

Since our work results from our investigation of fault-tolerance, we have examined these techniques and found them too specialized for Domain-Driven Re-

configuration problems. [3] Because of the time constraints imposed by virtual reality, we designed our Domain-Driven Reconfiguration framework to trade-off between consistent state and reaction time. Fault-tolerance acts on the loss of some required service, where Domain-Driven Reconfiguration handles losses as well as gains. The choice of compensating action is straight forward in fault-tolerant systems; while in a Domain-Driven Reconfiguration system, a combination of system state and application state may dictate divergent compensating actions.

We are using the CVE domain to develop our ideas on Domain-Driven Reconfiguration. The largest implementation of CVEs is the Department of Defense (DoD) distributed simulation program. This program involves a number of architectures and an attempt to develop an architecture that meets all DoD requirements as well as serving other industries such as entertainment and education. [7][2] [12] Understanding these architectures and insuring that we can service them gives us a ready domain to prove our ideas and the opportunity to solve a practical problem.

DOMAIN-DRIVEN RECONFIGURATION REQUIREMENTS

To make the concept of Domain-Driven Reconfiguration more understandable we have delineated the requirements for a Domain-Driven Reconfiguration system in the CVE domain. Figure 1 shows how a Domain-Driven Reconfiguration component fits into the architecture of a CVE.

Functional Requirements

A necessary, but not sufficient condition for CVEs to maintain realism is for each virtual entity to be controlled by one and only one VE at all times. This

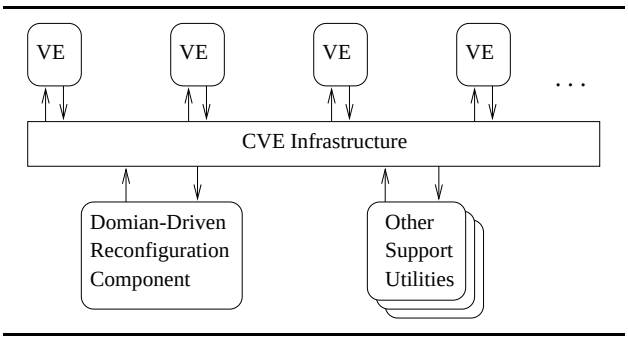

Figure 1: Collaborative Virtual Environment with a Domain-Driven Reconfiguration Component

condition is the driving principle for Domain-Driven Reconfiguration.

A Domain-Driven Reconfiguration component must react to multiple event types, and those may be either system or application events. It must not only detect those events, but also gather all the preliminary information available.

Ideally the Domain-Driven Reconfiguration component can detect system events by listening to the application message traffic. This is low-cost and unobtrusive. It is common [9][8] [2] to require each component VE to update its state periodically, and failure to do so indicates a problem. A more expensive but faster technique is to query the VEs regularly. Another technique is to use either mobile or stationary agents that report information or events. This approach generates less message traffic but requires the VE hosts to run additional software. All three methods must be supported in Domain-Driven Reconfiguration.

A change to the application state may also trigger an event of interest to the Domain-Driven Reconfiguration system. Detection of events that occur only within the application is also necessary.

A Domain-Driven Reconfiguration implementation must have its own copy of the CVE state. By keep-

ing its own copy, it does not have to add to the message count by requesting state information, and avoids the network latency penalty while making a decision. Without application state it can not identify application events. More importantly, it must rely on the VEs to divulge their internal state prior to reconfiguration, and that is not practical for failures.

CVEs, especially DoD distributed simulations, may be very large. DoD policy makers envision that distributed simulations will involve thousands of virtual entities.[7] The state space for such a system will be astronomical. Making application state based decisions with the raw state of each virtual entity may be very difficult. Queries about the application state should be simple and non-ambiguous.

CVEs commonly deal with the latency problem by using techniques called *dead-reckoning* and *forward reckoning*.[1] Both techniques estimate the current state based on the last reported state, forces acting on the object and the current time. These techniques require that all components of the CVE employ the same algorithms, to insure that all have a consistent picture. When VEs use techniques such as these, the Domain-Driven Reconfiguration component must also use them.

When the Domain-Driven 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. 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. CVEs can create events simultaneously; consequently the Domain-Driven Reconfiguration system must be capable of concurrently handling multiple decisions. As the application executes, to include reconfiguring itself, the decision-making logic must take the changes into account.

CVEs will have protocols and in many cases provide infrastructure to support components that join and depart the CVE during execution. [9] [12] [2] Whichever dynamic reconfiguration technique is apropos for the CVE architecture, the Domain-Driven Reconfiguration component must be able to exploit it. Not only must the CVE builder have access to the full power of the dynamic reconfiguration support, but he must have a powerful way of employing that technique too. He must have access to state information, as well as apply loops, conditionals, and waits.

The Domain-Driven Reconfiguration component must insure that regardless of the frequency, order, or timing of reconfigurations the CVE will always end up with a coherent structure and geometry, if possible. Reconfigurations must be atomic. The reconfiguration must be finished, or leave the CVE in its original form.

Non-functional Requirements

There are a number of different architectural approaches to interconnecting virtual environments. The two most popular are middleware, for example DoD's High Level Architecture HLA, [9] and the completely distributed approach typified by NPSNET [6] and DIS [2]. The framework must support both these architectural approaches, and should be as architecturally neutral as possible.

The perception of human users is the performance driver. Because VEs usually operate with very low performance margins, the Domain-Driven Reconfiguration component must impose little overhead on the CVE. Decisions must be made quickly enough to allow reconfigurations to be transparent to the users. The techniques available for reconfiguration are limited by human perception, for example rollback would usually result in unrealistic behavior. The driving principle is

continuously presenting a realistic virtual environment to the users.

The engineers assembling a CVE must deal with a wide array of topics. This software engineering environment must provide a tool that allows the CVE builders to reason about Domain-Driven Reconfiguration at an abstract level. They should not have to deal with the implementation details.

The Domain-Driven Reconfiguration system must be able to handle any size distributed application. This might require that it run on a host that is as powerful as the component VEs, but the reconfiguration software cannot be the limiting factor. This includes multiple Domain-Driven Reconfiguration components cooperating in a partitioned CVE. It might also include splitting the Domain-Driven Reconfiguration component over multiple machines.

BULLPEN: A DOMAIN-DRIVEN 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 CVE.

We have exercised the Bullpen prototype on an CVE architecture that we based on DoD's High Level Architecture (HLA). This is an excellent domain to prove our ideas because HLA represents current thinking in CVE/Simulation interconnection and it is the largest use of CVEs in the world.

Bullpen runs as an invisible support utility, it only inserts itself into the CVE when required. Most of the time it will monitor all the message traffic of the

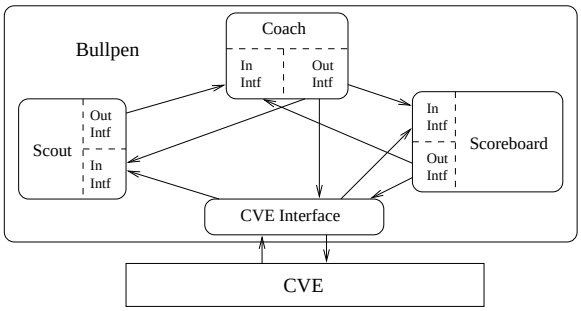

Figure 2: Bullpen Structure and Data Flow

CVE and do nothing. If the CVE is using multicast or broadcast then there is no overhead, with unicast there is the slight penalty of the additional message to Bullpen. When it detects an event of interest, Bullpen then makes two decisions. The first decision determines what the CVE should do to compensate for this event. If reconfiguration is called for then it makes the second decision, i.e., how to dynamically change the structure of the CVE to meet the desired configuration.

Bullpen has three independently executing subcomponents. Scout is the sub-component whose mission is to detect events of interest and report them along with any preliminary information to Coach. Coach contains the decision-making logic for both the end-state and how-to reconfiguration decisions. Coach directly executes the reconfigurations by using the CVE protocol. Scoreboard is responsible for keeping a valid copy of the application and system state. In our prototype this includes implementation of the dead-reckoning algorithm. Scoreboard answers queries from Coach about both types of states.

Bullpen has five major threads running whenever it is executing. Scout continues to handle incoming messages and test for absent component VEs regardless of what Bullpen is doing. This way Bullpen can identify an event as soon as it happens. Coach spawns a new

thread for each event it receives, therefore enabling it to handle simultaneous events. Scoreboard handles 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.

Bullpen is designed to be used in a wide array of heterogeneous environments. Consequently we wrote Bullpen in Java^{TM} to make it portable across operating systems. As a framework we designed it to function in the HLA environment without change. The only coding required is the implementation of the reconfiguration strategies. However, transferring it to a different architecture, or use under very different conditions may require modifications of some of the methods. Therefore we have included a number of hingepoints that are discussed with their sub-components below.

Bullpen itself may run on one, two or three separate hosts to support scalablity. To support this we encapsulated inter-component communication to allow easy migration to and from a distributed architecture.

Scout

Scout reifies the event detection function. It keeps its own small lists, but otherwise relies on Scoreboard for application state information. Scout has two major objects. Figure 3 shows the structure of Scout and the data flow between the objects. The Message Handler processes messages received from the CVE and Coach. Anytime Scout observes a message from a component VE it updates its heartbeat record. Scout is also party to reconfigurations made by Coach so that it understands the current structure of the CVE. The Message Handler detects events by checking for messages from VEs that Bullpen has reconfigured out of the CVE, while it updates heartbeats. Monitor, the other major object, constantly checks the active list for VEs whose heartbeats have expired.

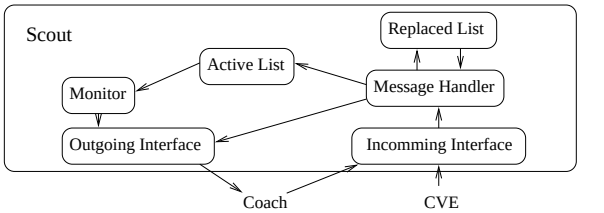

Figure 3: Scout Structure and Data Flow

Scout uses two threads during its execution. The Message thread includes the Incoming Interface, the Message Handler and the Replaced List. The Checker thread includes the Monitor and Outgoing Interface. Both threads share the active list.

In its current implementation Scout relies totally on message traffic to discover events of interest. Using intelligent agents to send reports back to Scout requires no changes. However, the CVE builder may want to use active techniques to determine system events. CVE builders can easily adapt Scout to use network or infrastructure tools to actively determine whether events of interest have occurred. Scout can also detect events that are defined solely by the application state with trivial changes.

Coach

Coach has one type of Decision-Maker for each type of event that it handles. Each event that Coach receives causes a Decision-Maker instantiation that lives until it compensates for the event. Multiple instances of the same type of Decision-Maker can coexist without a problem. The end result of a Decision-Maker invocation is the execution of a Strategy. A Strategy reconfigures the CVE to achieve the end-state structure chosen by the Decision-Maker. Figure 4 shows Coach's structure. Note that each Decision-Maker will have numerous Strategies to choose from.

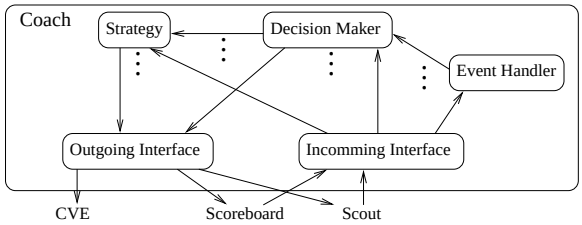

Figure 4: Coach Structure and Data Flow

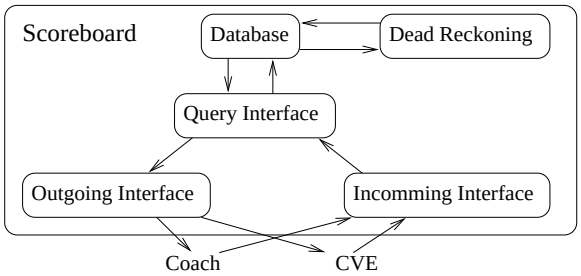

Figure 5: Scoreboard Structure and Data Flow

A set of rules determines the actions of the Decision-Maker. Currently the decision is a n-ary tree of sub-decisions. Each interior node represents a sub-decision based on information about the system or application state. Each sub-decision has a default branch to insure that it handles the event, even if handling the event requires no action. The leaf nodes are Strategies. Currently Strategies are Java^{TM} classes with the full expressiveness of a complete programming language. Strategies can query Scoreboard, make system calls, and access the CVE infrastructure to perform the restructuring.

Coach implements a locking protocol that insures that reconfigurations are atomic. Once Bullpen starts a reconfiguration, it completes the reconfiguration. If Coach determines the reconfiguration is unneeded partly through, only after it finishes, does it undo the entire reconfiguration.

Coach runs in its own thread. When it receives notification of an event from Scout, it creates a new thread to handle the decision. This way Bullpen can handle as many simultaneous events as its host has resources.

Coach was designed with two hinge points in mind. We encapsulated the decision-making algorithms used in Coach so that changing them will not change any other parts of Bullpen. This is, in fact, critical to our research to find the most effective techniques for this domain. We also designed the locking protocol so that we can easily change it to use better performing methods as we discover them.

Scoreboard

Scoreboard's purpose is to keep an accurate version of the system and application state and provide an interface to Coach for queries. Figure 5 shows the structure and the data flow between the objects. Scoreboard monitors message traffic updating itself, to include using *triggers* to keep accurate meta-states. Dead-reckoning is required for our prototype domain, and therefore is currently part of Scoreboard.

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

To better explain meta-states look at figures 6 and 7. Figure 6 shows the transitions and rules for a single virtual object—a tank. Figure 7 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 ($\sim$ 5000 vehicles)

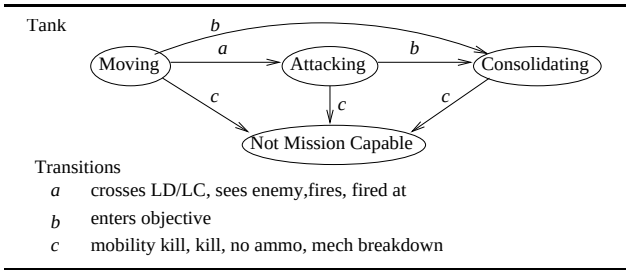

Figure 6: State Transition Diagram for an Individual Tank

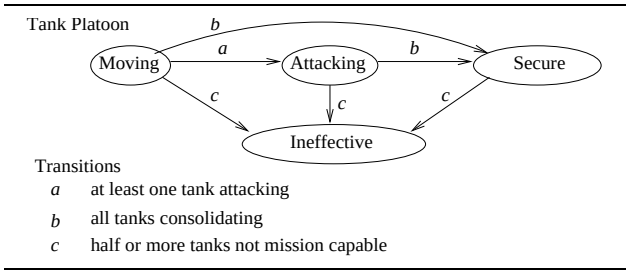

Figure 7: State Transition Diagram for a Grouping of Tanks

you can see that the complexity of the conditionals required to use the state would be enormous. Using the raw attributes of the entities to determine application state is unrealistic. However, by encoding state changes and relationships in the database through the use of a trigger, using meta-states becomes very convenient.

In this prototype, the application state is not based only on messages from the controlling VEs. In between updates from the controlling VE, each component of the CVE must also calculate the estimated state for every virtual object in the CVE. Scoreboard uses the same dead-reckoning algorithms used by the rest of the CVE to insure that its version of the application state is up to date.

Scoreboard runs two separate threads. Scoreboard's Query Interface runs it its own thread, handling queries in a first-come first-served manner. We feel that it is just as important to keep the database accurate as it is to handle queries from Coach. In very large CVEs we may find a priority scheme or a separate thread for updates more appropriate. The Dead Reckoning object also runs in its own thread. The two threads have synchronized access to the Database.

Scoreboard's hinge point is its database, so that changing the way that we store the state information will have no effect on the rest of Bullpen. It is relatively easy for us to substitute other relational databases or objectbases if they are more appropriate to the domain. The only requirement is for *trigger-like* functions that can update the meta-state.

Performance

Assessing the performance of a Domain-Driven Reconfiguration component is difficult. Its primary goal is to avoid unrealistic behavior. Unrealistic behavior springs from a virtual entity not being controlled by one and only one VE. However, with dead reckoning unrealistic behavior will not manifest itself until some time after the actual event. Because the user's judgment determines unrealistic behavior, it is difficult to determine exactly when it occurs. Additionally, unrealistic behavior is application and situation dependent. Without control, virtual aircraft in close combat will misbehave sooner than trucks on a highway.

We are satisfied that Bullpen meets performance requirements. It will never force unrealistic behavior on the CVE, even if it cannot guarantee the unrealistic behavior will not occur. A coherent configuration has every virtual entity controlled by one and only one VE. Under Bullpen, the CVE will always reach a coherent configuration—if one is possible. Although brief interludes of unrealistic behavior are undesirable, users can tolerate them and still have a successful CVE experience.

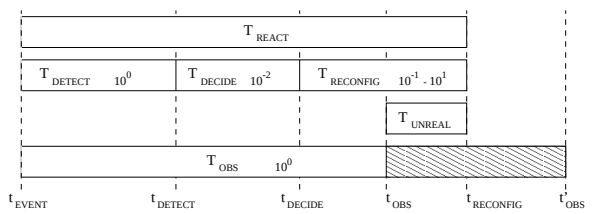

Figure 8: Reconfiguration Timeline (not to scale)

Figure 8 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.[2] t_{decide} marks the completion of Bullpen's decision process. The elapsed time T_{decide} is the only part that is attributable to Bullpen. Usual values are in hundredths of a second. $t_{reconfig}$ indicates the completion of the reconfiguration. In the best case where the CVE 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 VE would give $T_{reconfig}$ a value measured in tens of seconds.

t_{obs} 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 CVE will display unrealistic behavior during the interval T_{unreal}. The duration of T_{unreal} is dependent on T_{react}. $T_{detect}(10^0)$ dominates T_{react} in the best case, and $T_{reconfig}(10^1)$ dominates in the worst case. T_{decide} is never the dominant term.

Example Scenario

To help illustrate how a Domain-Driven Reconfiguration component generated by Bullpen works we will go through a simple scenario. Our example is a military CVE, 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) in an attack. The tanks are crewed simulators as are the helicopters. The rest of the friendly forces as well as the enemy are CGFs.

```
SWITCH (FEDERATE TYPE ==) {
CASE  "M1"
  SWITCH ("federate" ID APP ==) {
  CASE "tank_sim"
    SWITCH ("Host" "host_11" "M1_CGF" ==) {
    CASE "hot" "CS_StartM1HotSpare"
    ELSE "CS_StartSimpM1CGF"  }
  CASE "m1_cgf"
    SWITCH ("Unit" "tank_plt" ==) {
    CASE "aslt"
      SWITCH ("Host" "host_11" "M1_CGF" ==) {
      CASE "hot" "CS_StartM1HotSpare"
      ELSE "CS_StartSimpM1CGF"  }
    ELSE "CS_StartSimpM1CGF"  }
  CASE "simple_m1_cgf" "CS_StartSimpM1CGF"
  ELSE "CS_StartSimpM1CGF"
}
:
```

Figure 9: Replacement Rules

The goal of this exercise is to train the users. So the CVE builders want to insure that a system anomaly will not result in the other users noticing unrealistic behavior. Additionally, the CVE must bring a manned system that becomes available again back into the exercise as quickly and seamlessly as possible.

The CVE builder writes the rules with this goal in mind. A set of rules is described for each event that Bullpen must handle. Figure 9 shows a subset of the rules for replacing a VE that fails to communicate with the CVE.

The exercise starts and all participants are taking part in the war game. A tank simulator drops its network connection. The virtual tank continues to behave according to its dead-reckoning algorithm. This is acceptable for a non-deterministic period of time. If the tank was in the middle of a fire-fight and maneuvering abruptly, the participants will notice unrealistic behavior sooner than if the platoon was road marching down a straight road.

Scout detects that the virtual tank's heartbeat has expired. Scout places the virtual tank on the replaced list and sends the information to Coach. Coach spawns a process to decide what to do. The Decision-Maker follows the sub-tree for a M1 tank. It finds out from Scoreboard that the manned tank simulator controlled the virtual M1. In this scenario, the exercise developers allocated the resources to support a hot spare for the tanks, so Scoreboard tells Coach that the hot spare is still available. It then sends commands to the CVE to change ownership of the virtual tank to the hot spare software/host. Scoreboard and Scout update their state to reflect the reconfiguration.

Should a second M1 also be lost, the hot spare would not be available, and Bullpen would restructure the CVE differently. Bullpen is adaptive in this way because it has a dynamic picture of the system and application state.

The exercise continues with the virtual tank controlled as a CGF rather than by a crew. The network problem clears and the tank simulator reenters the CVE, trying to control its original virtual tank. Scout detects this and notifies Coach which then spawns another Decision-Maker. The Decision-Maker traverses the sub-tree for reentering M1 tanks. It determines from Scoreboard that the disputed virtual tank is currently controlled by a CGF, and the returning VE has human users. Therefore it replaces the CGF with the crewed tank simulator. This includes not only changing ownership of the virtual tank, but stopping the CGF, and updating the tank simulator with the latest state information from Scoreboard.

This scenario did not use meta-states, nor did it show canceling events. It only used two basic events, an absentee, and a reentry. Bullpen is capable of reacting to more complicated situations, but this example presents a flavor for what it can do and how it does it.

CONCLUSIONS

Although CVE is an arena where Domain-Driven Reconfiguration can be particularly effective, it is not the only domain where it is useful. We are also applying this technology to the distributed simulation domain that sometimes relaxes performance constraints, but tightens state consistency requirements. Domain-Driven Reconfiguration can be used to provide basic fault-tolerance, and load-balancing in any distributed application that requires complex, time-sensitive reconfiguration decisions.

Based on our experiences, we have identified some weaknesses in our implementation. We plan to address these in the near future. Chief among these are:

1. The rules for the decision logic are much more procedural than we would like. Currently our tree-structured decision logic performs well and provides adaptability, but we would like to make the interface more abstract. We are going to experiment with a backward-chaining system to try to improve the ease of use without degrading performance or power.

2. Currently, the Java^{TM} based strategies give us power and performance. We are going to experiment with an AI planning approach. Our aim in this research is to allow one set of planning rules

to be written for an architecture and used across CVE implementations.

3. Currently once Bullpen starts a reconfiguration, it must complete it. Then if necessary Bullpen must undo the reconfiguration. This can be very inefficient and we plan to investigate ways to handle this more efficiently.

We have identified a weakness in current CVE system research. The unexpected loss, or reentry of a VE 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. It requires application as well as system state and application state must be abstract. We have identified a solution to this and in the process developed a software engineering environment to create components that allow CVEs to perform compensating reconfiguration in response to any type of event. Using this component CVE builders can allocate resources to avoid unrealistic behavior. It performs within the time constraints of CVEs and shields the CVE builder from the details involved. It works with the two predominant CVE architectures and as a result shows promise for use in domains other than CVEs.

References

[1] Hubert Bahr and Ronald F. DeMara. A Concurrent Model Approach to Scaleable Distributed Interactive Simulation. In *15th Distributed Interactive Simulation Standards Workshop*, Orlando, FL, September 1996.

[2] DIS Steering Committee. The DIS Vision, A Map to the Future of Distributed Simulation, May 1994. Version 1.

[3] Flaviu Cristian. Understanding Fault-Tolerant Distributed Systems. *Communications of the ACM*, 34(2):57–78, February 1991.

[4] Christine R. Hofmeister and James M. Purtilo. Dynamic Reconfigureation of Distributed Programs. In *Proceedings of the 11th International Conference on Distrubuted Computing Systems*, pages 560–571, 1991.

[5] Jeff Kramer and Jeff Magee. The Evolving Philosophers Problem: Dynamic Change Management. *IEEE Transactions on Software Engineering*, 16(11):1293–1306, November 1990.

[6] Micheal Macedonia, Michael Zyda, David Pratt, Paul Barham, and Steven Zeswitz. NPSNET: A Network Software Architecture for Large Scale Virtual Evnironments. *Presence*, 3(4), Fall 1994.

[7] Under Secretary of Defense (Acquisition and Technology). Modeling and Simulation Master Plan, January 1995. Draft.

[8] Defense Modeling and Simulation Office. High Level Architecture Rules, August 1996. Version 1.0.

[9] Defense Modeling and Simulation Office. High Level Architecture Interface Specification , February 1997. Version 1.1.

[10] James M. Purtilo. The POLYLITH Software Bus. *ACM Transactions on Programming Languages*, 16:151–174, January 1994.

[11] Jon Siegel. *CORBA Fudamentals and Programming.* Wiley Computer Publishing Group, New York, 1996.

[12] Richard M. Weatherly, Annette L. Wilson, Bradford S. Canova, Ernest H. Page, Anita A. Zabek, and Mary C. Fischer. Advanced Distributed Simulation through the Aggregate Level Simulation Protocol . In *29th International Conference on*

System Sciences, pages 407–415, Wailea, Hawaii,
3-6 January 1996.