BUILDING SELF-RECONFIGURING DISTRIBUTED VIRTUAL ENVIRONMENTS

Donald J. Welch
University of Maryland
College Park, Maryland 20741

ABSTRACT

A distributed virtual environment may be required to reconfigure itself to compensate for various conditions that can occur during execution. An example is the reentry of a virtual environment that was previously reconfigured out of the distributed virtual environment due to failure. If there is a human user of this virtual environment, care must be taken to insure that he is brought back into the distributed virtual environment in a way that makes sense. He cannot regain control of a tank that is out of ammunition while a computer-based simulation controls actively participating tanks.

The compensating reconfiguration function of a distributed virtual environment must detect conditions that dictate reconfiguration. It must determine the proper course of action and act on it, bringing the distributed virtual environment to a stable state as quickly as possible. Proper reconfiguration of a distributed virtual environment requires that the compensating reconfiguration software know the system configuration, the virtual state, and the mapping between them.

Building compensating reconfiguration software using traditional means is laborious and error prone. A rule-based tool that uses abstract views of the distributed virtual environment is a better way to produce compensating reconfiguration software. To show the viability of this approach I have developed a rule-based tool called Bullpen. This research compares Bullpen against manual coding in a case study that ranges over a wide array of requirements changes.

The results of this case study show that using Bullpen to build compensating reconfiguration components is superior to manually building the software in the kind of environments most commonly found in the military DVE domain. Using Bullpen takes less effort and is less complex than using manual programming techniques. The resulting component is less error prone and has acceptable reaction time.

Table of Contents

List of Tables

List of Figures

List of Abbreviations

AI	Abstract Interface
ALSP	Aggregate Level Simulation Protocol
API	Application Programming Interface
C	Conditions
CGF	Computer Generated Forces
CORBA	Common Object Request Broker Architecture
DIS	Distributed Interactive Simulation
DoD	U. S. Department of Defense
DVE	Distributed Virtual Environment
HITL	Human-in-the-loop Virtual Environment
HLA	High-Level Architecture
HTTP	Hyper-Text Transfer Protocol
ISTP	Interactive Sharing Transfer Protocol
RI	Reconfiguration Interface
RP	Reconfiguration Policy
RTI	Runtime Infrastructure (For HLA)
SC	System Configuration
VE	Virtual Environment
VC	Virtual Configuration
VRTP	Virtual Reality Transfer Protocol

1. INTRODUCTION

Users reconfigure distributed programs dynamically to serve many purposes, for example, to maintain a high level of service despite changing conditions in the execution environment. One domain wherein demand for dynamic reconfiguration is increasing is Distributed Virtual Environments (DVE). In a DVE, where individual Virtual Environments (VEs) interoperate to create a shared virtual world, loss of any of the components can result in unrealistic behavior in the virtual world. Other conditions, many of which dynamic reconfiguration can correct, can also lead to an inconsistent or unrealistic virtual world.

The objective of my research has been to discover ways that a developer could cause his DVE to maintain realistic behavior in spite of vagaries and change in the run time environment. I have sought to identify reliable and cost effective mechanisms to effect dynamic change in the application to compensate for external events in ways that make sense back in the virtual world. To the best extent possible, I have attempted to provide these mechanisms in such a way that developers could easily reason about their domain and the types of change their tools could face.

The base upon which my research builds is the space of DVEs either available or emerging. A number are currently in use. Many different technologies facilitate their implementation, including customized middleware and both broadcast and specialized network protocols. Each is best suited for a different set of uses, but all share the same Achilles' heel. They cannot automatically react to changed conditions to insure realistic behavior.

The most ambitious effort in DVEs is currently the United States Department of Defense (DoD) High-Level Architecture (HLA) program. Although it is not the only such program, it is a very useful domain for a number of reasons. One of the uses DoD has for HLA is to create a training environment by combining computer generated forces (CGF) and human-in-the-loop (HITL) simulators into a virtual world. Usually the systems are geographically dispersed – sometimes transcontinentally. The executions can run for extended periods and involve a significant use of non-computer resources making dependability important.

An interconnection infrastructure is the primary feature of HLA. [DMSO97b] Components do not pass messages to each other directly, but through the interconnection infrastructure. The interconnection infrastructure determines which actions are appropriate when it is called by the component. The HLA middleware is reified as the Runtime Infrastructure (RTI). It provides a number of services involving component participation in the DVE, the entities in the virtual world, the simulation time, and distribution of information. The HLA is application independent. VEs that are written to interact with the HLA RTI can be used without modification in any appropriate HLA based DVE.

Other DVEs do not use interconnection infrastructure, but use broadcast and/or multicast to disseminate messages to the rest of the DVE. A common approach is to use an interface layer or wrapper to provide services to the component VE, such as the Aggregate Level Simulation Protocol (ALSP). [Wea96] NPSNet and its descendent, Distributed Interactive Simulation (DIS) are also successful implementations of this approach.[Mac94][Shi95]

This fully distributed approach works well, but has two drawbacks that caused DoD to abandon it. First, it is not flexible enough. Changing the message format requires altering all the VEs, where as in HLA the interconnection infrastructure shields the component VEs from message format changes. The other drawback is scaling. Because each VE must process all messages it receives, this architecture cannot be used to build DVEs with large numbers of component VEs.

The other major approach to constructing DVEs is the use of specialized network protocols such as Virtual Reality Transfer Protocol (VRTP). [Bru97] VRTP provides enhanced message passing capability on top of the Hyper Text Transfer Protocol (HTTP) client-server model. It does not provide the level of service that the other approaches do, but because of its simpler design it promises a general-purpose infrastructure that allows the World-Wide-Web to host ad-hoc virtual worlds. The Interactive Sharing Transfer Protocol (ISTP) is also an enhanced network protocol designed for general purpose use, but much like ALSP it uses local virtual worlds for performance reasons. [Wat97]

Regardless of which protocol and architecture one starts with, the manual development of software to reconfigure a DVE in execution is difficult and error prone. Compensating for the changing conditions of a DVE through dynamic reconfiguration is complex. The mapping between the system and virtual states of the DVE is an important factor in determining the appropriate compensation. One of the premises motivating my work was that use of an appropriate tool could make this process easier, less complex and more accurate. I have kept the characteristics of these different approaches to DVE construction in mind when developing my compensating reconfiguration tool. Therefore, the results I will describe are not limited to a specific DVE architecture.

While developing a compensating reconfiguration tool, I have chosen to focus more on the HLA environment than the others for practical reasons. HLA has all the characteristics to make it an excellent candidate for compensating reconfiguration. In addition, my work can fill a shortfall in the current HLA implementation by providing automatic reconfiguration in response to failures. [Cal94] Finally, HLA provides an environment where it would be easiest to apply compensating reconfiguration to solve real problems.

1.1 Motivating Example

DVEs are gaining widespread use in the military for a number of reasons. In this era of diminishing defense funds, virtual training is less expensive than training with the actual weapons systems in the field. It is also safer to combine maneuver and marksmanship in a VE. Coordination mistakes in virtual reality can be useful learning experiences, while the same mistakes can be lethal on a live-fire range. Due to the increased speed of weapons systems and the extreme ranges at which engagements take place, soldiers require more space for maneuver. Each generation of new weapons systems compounds this problem. Meanwhile the encroachment of urban areas and the lack of a driving threat to national security reduce the training areas available. DVEs are an alternative that the military has declared it will rely on more heavily in the future. [USDAT95]

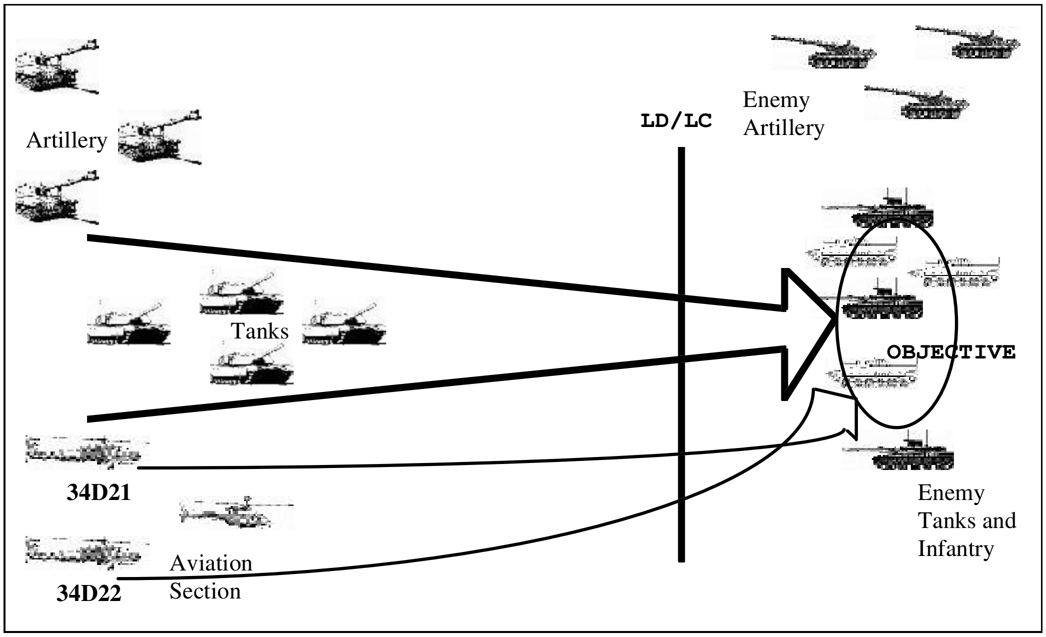

Figure 1.1: Virtual World View

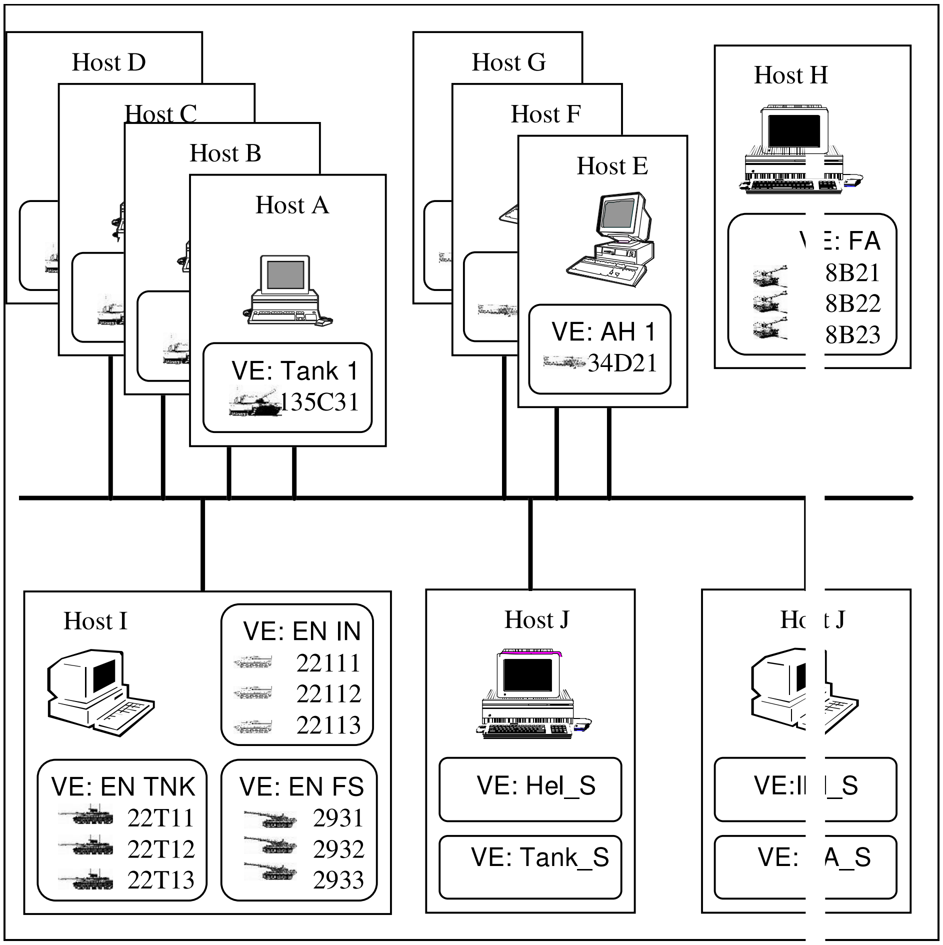

Figure 1.2: System Configuration View

Here I outline a very simple military DVE example to illustrate compensating reconfiguration. The purpose of this DVE is to train a tank platoon and attack helicopter section to work together to destroy defending enemy forces. This complex task requires split-second timing to maximize the massing of fire while minimizing exposure to enemy fire. The tank crews and helicopter crews interact with the virtual world through simulators that are just like their weapons systems, except that they interface only with the virtual world. Computer generated forces (CGF) – VEs with no human control–constitute the enemy and supporting friendly forces to complete the virtual world.

This DVE has two perspectives. The first is the virtual world as the users see it. This contains all the entities and their behaviors. The second is the system configuration as the engineers who build the DVE visualize it. It consists of the hosts, software and network connections that create the DVE. The entities link the two views. Each entity is owned or controlled by a component VE. Figures 1.1 and 1.2 show a simplified representation of each view. Each entity has a symbol in Figure 1.1, while Figure 1.2 shows a representation of the system configuration of the DVE. In the system view, the "bumper numbers" for the entities a VE owns are listed under that VE.

The example DVE begins execution and the tank platoon begins its attack on the enemy defending the hill on the right of Figure 1.1. At this time the link between the attack helicopter simulator *AH 1* and the rest of the simulation fails. Attack helicopter *34D21* is now an orphan, or not controlled by any VE. To the rest of the participants the helicopter *34D21* continues to fly at the speed and heading determined by its dead-reckoning algorithm.

Dead-reckoning is used in most DVEs to overcome the latency in the network and to reduce message traffic. [Mac94][Bah96] Each VE in the DVE has a dead-reckoning algorithm for each type entity in the virtual world. In between state updates, the algorithm is applied to the last known state of the entity to determine the estimated entity state. This smoothes the behavior of the entity and reduces frequency at which state updates must be sent.

The orphaned attack helicopter entity *34D21* will soon display unrealistic behavior. Dead-reckoning algorithms cannot maintain realistic behavior for very long, nor are they designed to. In the case of a tank at rest, dead-reckoning will be adequate for quite some time. For an attack helicopter in close combat with the enemy unrealistic behavior will appear quickly. Users of the DVE would expect the helicopter to maneuver sharply to avoid enemy fire. A consistent flight path will seem incongruous to the users.

To avoid this unrealistic behavior, an appropriate reconfiguration of this simulation is necessary. An appropriate compensating reconfiguration would be to transfer control of *34D21* to the spare VE *Hel_S* on Host *J*. This CGF will provide realistic enough behavior so that the rest of the participants can continue to train without distraction. The team work between the *34D21*'s attack helicopter crew and its spotters in the observation helicopter is a victim of the failure, but the other crews can still benefit from the DVE and the effects of the failure are limited.

As the battle continues, a surface-to-air missile hits *34D21*, the CGF owned attack helicopter. After that event, the network link is restored and the crewed simulator *AH 1* is once again ready to participate in the DVE. Giving the crew control of the destroyed helicopter would be futile. They are an expensive resource and just leaving them out of the exercise is not a good option either. If there was another CGF controlled attack helicopter in the DVE, giving the crew control of that entity would be a good solution. In this case, introducing a new attack helicopter entity to the DVE is the best solution according to user requirements. It allows the crew to train, while maintaining realism for the rest of the users.

1.2 Overview of Compensating Reconfiguration

This very simple example illustrates the types of dynamic reconfiguration necessary to maintain realism in a DVE. It also portrays how important virtual world information can be to the compensating reconfiguration decision making process. With this example, you can see how *compensating reconfiguration* is the application of dynamic reconfiguration to maintain a level of service in a DVE.

A set of virtual entities and their behaviors comprise a virtual world. Each entity has attributes that define its state within the virtual world. When a VE owns or controls an entity, it changes the attributes to keep the behavior of the entity realistic with respect to the virtual world. VEs that have been validated (the specifications are correct) and verified (meet the specifications), can be assumed to provide an entity with realistic behavior under normal circumstances. As long as a single functioning VE owns each entity, realistic behavior reigns in the virtual world. Should an entity not have an owner or should it have more than one owner, then realistic DVE behavior is not guaranteed.

Although the traditional approach of removing the entities of absent components permanently from the DVE will stop individual entities from behaving unrealistically, it does not provide the best solution. Additionally, as I illustrated in the motivating example, it is not desirable to leave human-in-the-loop VEs out of a DVE. Not only does the quality of the DVE suffer, but also the users are an expensive resource. In the military domain, one of the intended uses for DVEs is rehearsing complex missions. Since there might not be another opportunity for rehearsal, insuring that all the users get maximum benefit for every execution is desirable.

Basing compensating reconfiguration decisions on the state of the virtual world can be a very complex undertaking. Most users make use of abstractions to think about the virtual world. The appropriate fidelity of the abstraction will vary with the decisions, but some kind of abstraction is necessary. In most cases we reason about a car as a single entity, not the component bolts, etc. In some cases, say traffic flow, we reason about groups of cars. To keep the reconfiguration logic as simple as possible, it must deal with the virtual world using similar abstract groupings.

DVE builders can also use compensating reconfiguration to introduce additional functionality as the virtual world evolves. In a scenario where multiple low-probability mutually exclusive requirements exist, compensating reconfiguration software can insure that a necessary component is available. For example, the DVE would need naval fire support if the battle moves towards the shore, and allied guerrilla forces if the battle moves up into the mountains. The CGFs to simulate these two sets of entities could be available but not executing. Compensating reconfiguration software can detect the shift in the battle and start the proper components.

In a larger sense, the state of the virtual world is related to the load on the simulators. A ground unit that is in a holding area will not be generating many events, and will not unless the virtual world changes and it begins a different activity such as attacking. When deciding how to reallocate the objects to compensate for a failure, using the virtual state can result in a much better solution. The HLA RTI uses data distribution management techniques to reduce unnecessary message traffic. [Cal95] Using information of this ilk will contribute to better reconfigurations.

Compensating reconfiguration has applicability beyond distributed virtual environments. The domain must have the following characteristics to be a candidate for compensating reconfiguration. First, it must be a reconfigurable distributed environment. Through the reconfiguration primitives, The application must be able to be dynamically reconfigured. In most applications consistent state is necessary. That means that either the components involved in reconfiguration migrate to a quiescent state as Kramer and Magee [Kra85] advocate or the message traffic is frozen for the duration of the reconfiguration as Hofmeister proposes. [Hof91] Since much of the power of compensating reconfiguration comes from the mapping between the virtual world and system configuration, the essential parts of the application state must be exposed through the message traffic. Finally, compensating reconfiguration is powerful and requires resources. It is most practical when the reconfiguration decisions are not trivial.

Compensating reconfiguration is the most practical when the decisions involved are not simple. In DVEs the system configuration and virtual world are interrelated and compensating reconfiguration decisions are based on both. In the motivating example, the question of giving a HITL simulator control of an entity that is out of play is a simple example. From a systems perspective, the original entity is the proper choice. When including information from the virtual world the undesirability of that choice is clear.

1.3 Research Goals

Self-reconfiguring DVEs can be much more valuable than DVEs that are not. Compensating reconfiguration can be difficult to add to a DVE. In an environment where the DVE requirements change frequently, this can discourage DVE builders from including compensating reconfiguration. It may also dishearten the builders from experimenting with compensating reconfiguration to achieve better allocations of resources against possible conditions.

I am trying to provide the builders of distributed virtual environments a faster and more intuitive way to make DVEs self-reconfiguring. I have delineated some of the reasons why DVE builders would want to create self-reconfiguring DVEs. Once they make the decision to include compensating reconfiguration functionality in a DVE, the decision to use a tool should also follow.

Hypothesis

It is possible to employ tools with abstract interfaces to build distributed virtual environments that can dynamically reconfigure themselves to compensate for external conditions. Using a rule-based tool that abstracts the virtual state is superior to using traditional high-level language approaches. The tool-based approach is easier, less complex, and more accurate than hand-coding, while still being powerful enough to handle all realistic compensating reconfiguration requirements.

1.4 Summary and Overview of the Dissertation

The automatic restructuring of a distributed application in accordance with a user defined policy is compensating reconfiguration. I have developed a software engineering tool to support its inclusion in DVEs.

Using my tool is easier and less complex than hand-coding. It accomplishes this without an unreasonable performance penalty.

The remainder of this dissertation is divided up as follows. Chapter 2 is a discussion of previous work related to compensating reconfiguration. In Chapter 3, expands on my discussion of compensating reconfiguration and its characteristics. This includes definitions leading up to a definition for compensating reconfiguration. Chapter 4 explains the features and design of Bullpen, a compensating reconfiguration tool. In Chapter 5, I delineate the case study design that I used to prove my hypothesis. The analysis of the results occurs in Chapter 6. Chapter 7 is devoted to conclusions and future research directions for this work.

2. RELATED WORK

This research builds on a number of areas of computer science. It is related to the study of how to interconnect distributed systems. HLA uses a middleware architecture for interconnection and is the dominant architecture in current DVE research. This work focuses on putting an abstract layer on top of dynamic reconfiguration primitives. Thus, I have built on dynamic reconfiguration research that discovered how to reconfigure a distributed program in execution. Some of the concepts involved in insuring the consistency of a distributed program during development apply to keeping a program consistent during execution. I have applied some ideas from distributed fault-tolerance because there is some overlap between my work and that domain. Unifying these areas of research to build self-reconfiguring DVEs has not been addressed in the literature.

2.1 Interconnection

The distributed simulation domain is a specialized area of interconnected distributed systems. Thinking in distributed simulations architecture has evolved to embrace an interconnection abstraction exemplified by DoD's HLA. In fact, the Defense Modeling and Simulation Office has built prototypes of the HLA runtime infrastructure (RTI) on top of a CORBA compliant object management broker. [Sie96]

Early interconnection work such as HPC [LeB85] and V [Che88] addressed the support an abstraction should provide for building distributed systems. This includes communication, resource management, and process abstraction, all of which are present in HLA and other modern interconnection abstractions. CONIC combined the use of a specialized programming language with the gluing together of disparate systems. [Mag89] POLYLITH went a step further by completely divorcing the components from the type of interprocess communication used in the interconnected system. POLYLITH also supports a heterogeneous development environment. [Pur94]

2.2 Dynamic Reconfiguration

None of the early interconnection work provided dynamic reconfiguration services. More recent interconnection infrastructures do provide dynamic reconfiguration services as reconfiguration primitives. The goal of this research is to provide a powerful interface to dynamic reconfiguration making it easier to build self-reconfiguring distributed programs.

The CONIC system was the first environment to allow dynamic change to the configuration of a running distributed application. [Kra85] While doing this, Kramer and Magee introduced the groundwork that allows dynamic reconfiguration research to continue. They emphasized the importance of separating component programming and configuration. Systems without a clear separation are not candidates for dynamic reconfiguration; DVEs have this separation. They also delineated the dynamic reconfiguration requirements for programming languages, configuration and change specifications, validation, and configuration management. Their approach focuses on migrating components to a quiescent state prior to reconfiguration. They define a component in a quiescent state as one that is not involved in any transactions. This way they can insure state consistency across configurations. Components that participate in this type of reconfiguration must be able to migrate to a quiescent state on command. [Kra90b]

Because almost all DVEs transactions are independent, achieving a quiescent state is not daunting. The drawback for virtual environments that run on wall clock time is that it requires the surrounding nodes to move to a quiescent state too. In a DVE the more components that are not participating, the greater the likelihood of observed unrealistic behavior.

Hofmeister established a framework for how to modify a running distributed system. Structure, topography, and implementation are the three possibilities. Altering the structure involves adding and/or removing components. Changing topography comprises moving components to different hosts. Modifying the implementation involves substituting a different version of the same component. [Hof91]

Her concept for changing the implementation or topography requires capturing the internal application state prior to reconfiguration. If the component is moving, it must somehow save its state, so that it can reinitialize itself at its new location. If the reconfiguration is a substitution of a new version of a component, then that new component must be able to read and convert the old state information so that it can achieve consistency with the rest of the program.

In her framework changing the structure of the distributed program involves freezing the message traffic while making the change to insure consistency. VEs do not accept halts very gracefully, but they are very tolerant of small inconsistencies. So although all of her work does not apply, by relaxing some constraints it fits the needs of DVEs very well.

Hofmeister’s work focused on developing an application programming interface (API) for reconfiguration that facilitates all three types of dynamic reconfiguration. This work builds on Hofmeister's, providing an abstract interface to reconfiguration primitives. Using a programming languages metaphor, her work is like a high-level language. The interface is API calls to the POLYLITH software bus. This work is more akin to a forth generation language with the majority of the interface being rule-based. Where Hofmeister provided the developer with an abstract interface to perform dynamic reconfigurations, I am providing an abstract interface to decide both the appropriate reconfiguration and how to execute that reconfiguration.

Gupta and others have developed a formal framework for on-line program version change. [Gup96] Although on-line version change is different from this work, many of the concepts apply. Particularly, their formal definition of a valid reconfiguration is important to this work. To summarize, they deem a reconfiguration valid if after a finite time the program achieves a state reachable from the state prior to reconfiguration. Definitions 3.1 and 3.2 concerning consistency and correct reconfigurations build on Gupta’s definition.

2.3 Program Consistency

Keeping a distributed program consistent during its lifecycle is similar to keeping it consistent during its execution. During a program’s lifecycle, components are added, subtracted and changed to meet needs. After each change, the program must be consistent or it is worthless. Thinking of execution as a compressed lifecycle and dynamic reconfiguration as the process of modifying the program to meet needs, the parallels are clear.

Minsky has used laws to insure consistency in distributed software architectures as they evolve. He enforces a set of invariant laws throughout the lifecycle of the software by using independent monitoring. [Min96a] I focus on keeping the behavior of the distributed system consistent throughout its execution. My system trusts the component simulators by relying on the validation and verification process and does not require independent monitoring other than that enforced by the interconnection infrastructure.

Callahan has done work in the area of static rule-based configuration of distributed programs. He uses a rule-based interface to make the detailed process of assembling a heterogeneous distributed program more intuitive. [Cal91] I have adapted this concept to dynamic reconfiguration. I have also extended it by using internal program state information to make the reconfigurations more intelligent.

2.4 Distributed Fault-Tolerance

Fault-tolerance relates to compensating reconfiguration because some of the functions overlap. Compensating reconfiguration can provide a degree of fault-tolerance to a distributed simulation. The range of fault-tolerant software engineering provides protection from conditions that compensating reconfiguration does not address, and compensating reconfiguration handles conditions that fault-tolerant techniques do not consider.

Compensating reconfiguration can handle omission and timing failures by changing the configuration to provide the missing responses. [Cri91] One way it goes beyond fault-tolerance is in the case of a timing failure. The goal of fault-tolerant work is to provide the needed action, but it is not concerned about the fate of components involved in failure. Compensating reconfiguration considers the failed component's destiny because keeping human-in-the-loop VEs involved can be as important as making up for omission or timing failures.

Werner and others have developed a framework for providing fault-tolerance to distributed systems by transferring functionality between the components. [Wer96] This work addresses the simple case where all components can handle all the units of functionality. The heterogeneous case–when units of functionality must be matched with the appropriate components–is more complicated.

Russinovich has advocated putting fault-tolerance for off-the-shelf applications in the middleware layer. [Rus95] Off-the-shelf software is another way of looking at the legacy simulators involved in most distributed simulations. Although compensating reconfiguration is not strictly fault-tolerance it can be used to move in the direction that Russinovich has suggested.

2.5 Research Shortcomings

None of the areas mentioned above offers a complete solution to the problems presented in building self-reconfiguring DVEs. There have been calls for research to address this problem [Cal94], specifically in the HLA domain. Much of the work above solves constituent elements of the problem but there has been no unifying work.

Current fault-tolerance research results do not generalize enough to meet the requirements of distributed simulation. [Cri91] Fault-tolerance focuses on the loss of some required service, where distributed simulation requires handling other conditions such as the reemergence of a high-value simulator. The choice of compensating action is straightforward in fault-tolerant systems; in a distributed simulation, a combination of system state and application state may dictate divergent compensating actions.

Kramer and Magee have done some preliminary work on combining dynamic reconfiguration and fault-tolerance. They have focused on the loss of service but not the more general cases that arise in DVEs. [Kra90b] In their work they have also advocated canceling transactions in process when the failure occurs which is not appropriate for DVEs. Barbacci also has investigated merging the two areas and relies on the applications being quiescent as in CONIC. [Bar90] Neither does this work address the additional requirements of the DVE domain.

Compensating reconfiguration builds on research in interconnection, dynamic reconfiguration, static consistency, and distributed fault-tolerance. They all provide important pieces of my research. This work provides either ideas that have been integrated into my work or inspirations for modifications that more closely satisfy requirements in DVEs. Compensating reconfiguration collects and unifies ideas from these research areas to simplify the production of self-reconfiguring DVEs.

3. COMPENSATING RECONFIGURATION

This chapter discusses the software characteristics that differentiate compensating reconfiguration software from other types. The discussion starts with a set of definitions culminating in a definition of compensating reconfiguration. I continue with compensating reconfiguration's general requirements, then describe architectural features that make a DVE a candidate for compensating reconfiguration. Typical uses for compensating reconfiguration software complete the chapter.

3.1 Definitions

As with any distributed programs, DVEs can be reconfigured: adding or removing VEs will alter their structure with an appropriate redistribution of the entities involved. Substituting one VE for another alters their implementation, as would occur when replacing a human-in-the-loop VE with a CGF VE. The third type of reconfiguration entails altering geometry. In this case a VE would migrate from one host to another either to take advantage of load imbalances or to compensate for a failed host.

DVEs are distributed programs whose output is a virtual world. The quality of this output depends upon the integrated output of the component VEs. In DVEs, an entity's behavior can be thought of as an atomic unit of functionality. For a DVE to be fully functional, all the entities' behavior must be governed by a component VE. If an entity is not controlled by any VE then that part of the DVE's functionality is missing. Maintaining the required level of service means insuring that all entities are controlled by a VE.

Definition 3.1: A consistent DVE is a DVE in which every entity is controlled by one and only one VE.

In the HLA architecture the granularity is finer. Different VEs can simultaneously control different attributes of an entity. A consistent HLA DVE is one with every attribute controlled by one and only one VE. From Definition 3.1 the definition of a correct reconfiguration follows.

Definition 3.2: A correct reconfiguration is a dynamic reconfiguration where the DVE reaches a consistent state in a finite time.

This is similar to Gupta and other's definition of a correct dynamic version change of a program. They deem a reconfiguration valid if after a finite time, the program achieves a state reachable from the state prior to reconfiguration. [Gup96]

DVEs always operate under uncertainty due to lost messages and latency. The component VEs determine behavior based on the best available information. Because the DVE under normal operation tolerates this uncertainty, compensating also operates reconfiguration with this uncertainty. Therefore, the definition of a correct reconfiguration does not mention state consistency.

The next step is to define the goal of compensating reconfiguration: an appropriate reconfiguration. Appropriate reconfigurations are correct from the viewpoint of Definition 3.2. They go further, in that they consider the user requirements.

Definition 3.3 An appropriate reconfiguration is a dynamic reconfiguration that is correct, and is in accordance with the user requirements.

I continue towards a definition for compensating reconfiguration by clarifying the purpose of compensation. Compensating reconfiguration compensates for external conditions in the DVE.

Definition 3.4 An external condition is a condition that can easily be observed from outside the impacted VEs and affects the realism of the DVE.

The first part of this definition is subjective. A VE that appears to execute normally, but outputs unrealistic behavior for one of its entities would not be an external condition. This is because this condition cannot easily be observed from outside the VE. The logic for detecting this kind of behavior must be as sophisticated as is found in the VE software itself. In comparison, detecting the absence of messages from a non-functioning component VE is trivial.

It is tempting to say that external conditions are omission or timing failures. However, those definitions would limit the power of compensating reconfiguration. Events within the virtual world can be external conditions as well as events in the system configuration such as a heavily loaded processor. When an external condition affects a DVE, it may be possible to reconfigure the DVE during execution to ameliorate the condition and either restore realistic or avoid unrealistic behavior.

Definition 3.5 Compensating reconfiguration is the detection of any of a set of external conditions, deciding an appropriate response, and if necessary executing an appropriate dynamic reconfiguration that compensates for the external condition.

3.2 Requirements

I have distilled a minimum set of general requirements for compensating reconfiguration functionality in a distributed virtual environment. This functionality is necessary for any software that fulfills the compensating reconfiguration role.

Condition Detection

Condition detection is the identification of an external condition that negatively affects the behavior of a DVE. Not all external conditions must be handled, only those deemed important by the user to the realistic operation of a DVE.

Compensating reconfiguration differs from fault-tolerance and fault-resilience in that conditions are not limited to failures. Compensating reconfiguration can also reallocate resources in the face of evolving conditions in the system or virtual world. Compensating reconfiguration software must have the ability to detect all the external conditions for which the user wants to compensate. Without this ability, software cannot provide compensating reconfiguration.

A compensating reconfiguration component must be capable of detecting all conditions for which it must compensate. It can use active or passive means, but it must collect enough information so that it can identify and act upon the condition.

Compensation Decision

Once a condition is identified, the proper compensation is not always obvious nor is it static. As the system configuration and virtual world evolve during execution, the proper compensation will change. The compensation decision must take into account the system configuration, virtual world state, and the mapping between them. Not every decision must rely on all three, but all three must be available as input for every decision.

The software must be capable of determining an appropriate compensating reconfiguration as defined in Definition 3.5. Removing entities from the DVE would be a valid compensation if there were no other alternatives available. Removal is not desirable, but the invariant property that every entity is owned by one and only one VE still holds.

Virtual World

The virtual world is made up of a collection of attributes of the entities of the DVE. Not all attributes are important to the compensating reconfiguration software. The color of a tank might not be important to compensation decisions. However, whether or not the tank has been destroyed would be key to most implementations of compensating reconfiguration. Compensating reconfiguration builders must identify the significant entity attributes and track them.

The software must have access to an accurate version of the virtual world. Compensating reconfiguration code normally only needs a sub-set of the attributes that comprise the virtual world, but that sub-set must be representative of the VE's virtual world. If the DVE uses dead-reckoning, then the compensating reconfiguration software must also.

Users normally conceive of the entities of the virtual world as members of groups. Expressing requirements using these groupings is easier for the users, and DVE builders can more easily implement these requirements if the compensating reconfiguration code maintains these relationships.

System Configuration

The essential attributes of the system configuration are normally the DVE's hardware state and software state. These attributes can include characteristics of the components such as the hardware and the operating system that comprise the host. They can also include dynamic information such as the number of entities that the DVE currently owns.

To make compensation decisions, the component must have an up-to-date picture of the system configuration. It must know the status of the hosts and VEs in the system. In addition, it must know the current relationship between hosts and VEs.

Virtual World - System Configuration Mapping

The final set of information required for compensating reconfiguration decisions is the mapping between the system configuration and virtual world. This is the relationship between the component VEs and the entities. Compensating reconfiguration software must know not only which component owns which entities, but also which components *can* own which entities.

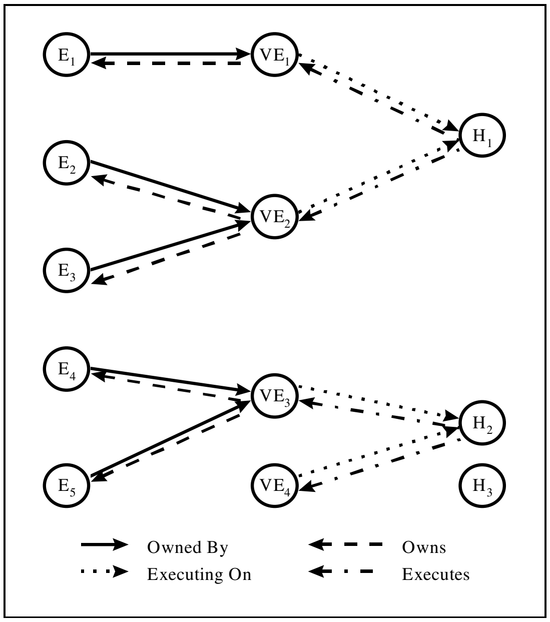

Figure 3.1: Current Mapping Functions

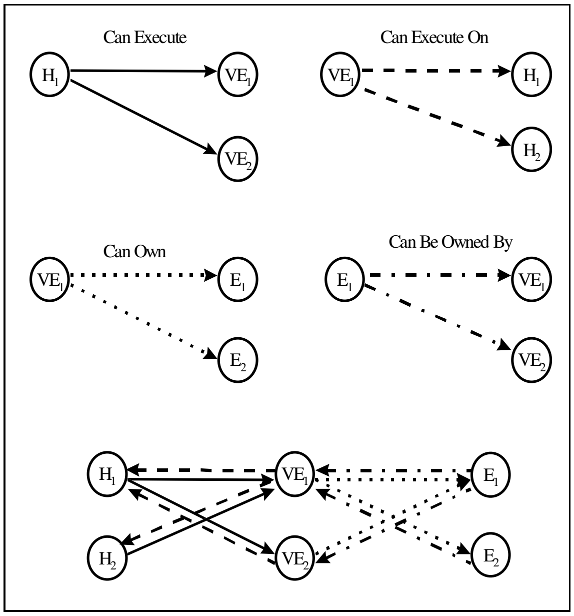

Figure 3.2: Potential Mapping Functions

Each host has a set of VEs that it can execute. This set is normally a subset of all VEs in the system due to software compatibility and the software that is loaded on the host. From that set of VEs, a sub-set (possibly empty) makes up the VEs that are executing on it. A similar relationship exists between the VEs and entities. Each VE in the system can own a sub-set of the entities, and does own a sub-set of the entities that it can own. The hosts, VEs and entities that comprise the DVE may change during execution and the mapping functions must keep pace.

Figure 3.1 illustrates the mapping functions between entities, VEs and hosts. Each entity is *Owned* by one and only one VE. This is a bijective function. The complementary function *Owns* is surjective. Every entity must be owned, but VEs may own zero or more entities. The same relationship exists between VEs and hosts. *Executing On* is a bijective function and *Executes* is surjective. A VE does not need to own any entities, nor is a host required to be running a VE. Spare VEs and hosts would not own any entities or be running any VEs.

The other mapping that the compensation decisions are based on determines the constraints on the relationships between entities and VEs and hosts. These functions do not have the individual item in their domain and codomain, but item types. There may be fifty M1A2 tanks in the DVE, but constraints for the entity type M1A2 tank are sufficient for determining which entities that VEs can own.

Grouping the items in the DVE into types is straightforward. Virtual characteristics determine entity types. An M1A2 tank is an entity type. VEs that can simulate M1A2 tanks do not care which specific tank it is; they can model the behavior for all M1A2 tanks. Determining the VE types can be more complex. The software normally defines the VE type. In some cases, VEs running the same software would not be the same type. The builders might divide the DVE because of its scale. In that case, identical software running in two different partitions would be considered two different VE types. Host type definition can also go beyond the hardware and operating system. Host types can be grouped by the software that is installed on the machine or other characteristics that limit which VEs the hosts can execute.

Figure 3.2 illustrates the potential mapping functions. These functions are all surjective, because more than one item can map to an item in the codomain of the function. The allowable relationships are maintained for all item types in the DVE. These are static, they are a characteristic of VE software and will not change unless the VE software is redesigned. Which type an item falls into may change during execution. For example, consider a VE that can own a maximum of four entities. When it takes control of a fourth entity it must now be considered a different VE type.

The compensating reconfiguration component must maintain a mapping between the virtual world and the system configuration if its full power is to be available to the compensation decisions. The virtual world can provide indicators about future VE load that can affect compensation decisions. It must also provide the mapping between the host, VE, and entity types as well as the individual members of those types.

Implement Reconfigurations

Dynamic reconfiguration of the DVE is the way compensating reconfiguration code exerts its influence on the DVE. Not every condition necessitates dynamic reconfiguration, but compensating reconfiguration code must be able to dynamically reconfigure the DVE.

Compensating reconfiguration components make a DVE self-reconfiguring. They handle external conditions, compensating for those conditions through dynamic reconfiguration of the DVE. In determining how to reconfigure the DVE, compensating reconfiguration components rely on the current mapping between the virtual world and system state as well as the allowable relationships functions.

Once the compensating reconfiguration component decides on a configuration, it must then dynamically reconfigure the DVE to that configuration. The component uses the reconfiguration primitives from the DVE, and any other mechanisms needed to perform the reconfiguration. Depending on the architecture and the reconfiguration itself this usually includes any number of operations, responses from the affected VEs, and changes to its own internal state all of which it must execute in the appropriate order.

Requirements Summary

These requirements are the minimum set of requirements for compensating reconfiguration software. To build specific software for a DVE, additional requirements are necessary to define the reconfiguration policy that is the heart of compensating reconfiguration. However, every compensating reconfiguration component must meet these general requirements at a minimum.

3.3 Target Architecture Characteristics

To apply compensating reconfiguration to a DVE, the architecture must have two main characteristics. It must have external conditions that are detectable, and that dynamic reconfiguration can be correct. In addition, the components of the DVE must have reconfiguration designed into them.

External Conditions

The external conditions that cause incorrect behavior for a DVE must be detectable. (See definition 3.4.) In many DVE architectures VEs are required to send periodic messages even if there is no state change. The lack of this *heartbeat* message would indicate a problem. In the HLA design just the message traffic reveals most of the information that compensating reconfiguration requires. Compensating reconfiguration code can also use active techniques such as requesting a status report or distributing agents through out the DVE to detect conditions. If a VE tries to reenter after being reconfigured out of the DVE there must be some way for the compensating reconfiguration code to know. In a DIS or ALSP environment, messages from the VE would indicate this condition has occurred. The interconnection infrastructure might stop the attempt right at the host. An agent at the host could detect this condition and inform the compensating reconfiguration code. The DVE builder has many approaches for implementing condition detection, but however he does it, the code must be able to detect external conditions.

The other key characteristic of external conditions is that they must be correctable by reconfiguration. In the most basic sense, if the DVE has no spare resources that can be allocated to compensate for a failure, then compensating reconfiguration is futile. It is not necessary to be able to compensate for all possible conditions for compensating reconfiguration to be useful. In fact, it is highly unlikely that the resources would be available to do so. However, compensating for high-probability or high-cost conditions can be extremely useful from an engineering standpoint.

Reconfigurable DVE

The architecture must support dynamic reconfiguration of the DVE. Most architectures address the late joining and early departure of component VEs. This is critical. A less ubiquitous feature is allowance of the reconfiguration of VEs that are not participating. In the current implementation of HLA, the transfer of object attributes' (virtual entities) ownership requires the participation of the current owner. If the current owner has failed, ownership of those entities cannot be transferred. So HLA currently does not support compensating reconfiguration. There are environments that support this kind of functionality such as Polylith [Hof93], so although the burden is not great, the capability must still be present.

The component VEs themselves must be capable of being reconfigured. If it is not possible for a VE to realistically take control of an entity during execution, then using it for compensating reconfiguration is inutile.

Most VEs work with a main simulation loop and no critical stack information is stored from iteration to iteration. In addition, the salient information is constantly being passed between the VEs, therefore the compensating reconfiguration software can obtain the critical state information in a DVE just by listening. This critical state information can then be used to initialize a VE after a reconfiguration.

Most VEs run on wall clock time. Therefore, there is no need to keep an event history for rollback. If a DVE uses virtual time, then the reconfigurable component VEs must have a mechanism for receiving the event history list so that if necessary, virtual time can be rolled back. Not all component VEs must be able to participate in compensating reconfiguration. Compensating reconfiguration can be implemented on a sub-set of the DVE and still be useful.

If a DVE architecture does not have these characteristics, then it cannot be made self-reconfigurable through compensating reconfiguration. Conversely, if it does have these characteristics then compensating reconfiguration can be integrated into the architecture to make it self-reconfiguring.

3.4 Typical Use

The DVE domain is an excellent candidate for compensating reconfiguration. Though I make no claim as to the completeness of this list, I describe some typical uses for compensating reconfiguration below based upon my personal experience with military DVEs, the Advanced Concepts Technology Demonstration (ACTD) for HLA, and the DoD vision for distributed simulation. [Bea97, USDAT95]

In DVEs the most common use for compensating reconfiguration would be to handle *orphaned* entities. Entities become orphans when their component VE for some reason stops participating in the DVE. Putting the *orphans* back under the control of an appropriate VE is a primary mission of compensating reconfiguration.

In most architectures, transferring control of an *absentee's* entities to other VEs will effectively reconfigure the VE out of the DVE. Depending on the nature of the failure, the absentee VE may return unexpectedly and still believe that it has control of its original entities. This condition must be dealt with because state updates coming from two different VEs will most likely be different and produce anomalous behavior.

Even if the failure was such that the VE is aware that it must rejoin the DVE this may require a reconfiguration. If the objects were distributed to other VEs, then it might be desirable for load-balancing purposes to move them back to the original VE. Another case would be where the entities were moved to a less capable VE and moving them back would be good policy. Finally, if the original VE had human users, then reintegrating them into the simulation would be crucial.

In a DVE with limited resources, the builders might want to reconfigure VEs into the simulation only as they are needed. Reconnaissance flights are only flown over the battlefield periodically. However, simulating a reconnaissance flight can put a great strain on the DVE. Large DVEs use some form of data distribution management to reduce unneeded message traffic. A reconnaissance flight may need messages from many or even all of the regions of the simulation, foiling the attempt to localize message traffic. Having the reconnaissance flight VE in the DVE only when it is needed can minimize this problem.

Another example of a condition that may be handled through compensating reconfiguration is load balancing. The nature of military scenarios is that only a fraction of the entities will be active at any given time. Inactive entities will generate far fewer events than active ones. Processing events generated by other VEs is another sink for processor time. VEs that have entities that are involved in the action will have a heavy processor load, while VEs that are separated from the action by the data distribution management scheme will have much lower processor requirements. The transfer of entities to relieve heavily used processors may be useful in a DVE that has little excess capacity.

3.5 Summary

Compensating reconfiguration is the dynamic reconfiguration of a distributed system to compensate for external conditions. The dynamic reconfigurations must be correct, leaving the DVE in a consistent state, and they must be in accordance with the user’s needs. Compensating reconfiguration software has three main concepts: condition detection, compensation decisions, and dynamic reconfiguration. Software without these capabilities is not compensating reconfiguration software.

Compensating reconfiguration can enhance distributed programs, especially DVEs. However, compensating reconfiguration must have support from architecture. Candidate architectures must have detectable conditions that are correctable by dynamic reconfiguration. In addition, they must be dynamically reconfigurable which although not difficult to implement, is less frequently found in current implementations. Compensating reconfiguration will most commonly be used to provide fault-resilience. However, it is much more capable than that. When fully exploited it can greatly enhance the flexibility and reliability of a DVE.

4. BULLPEN: A COMPENSATING RECONFIGURATION TOOL

I have built a tool called Bullpen to facilitate adding compensating reconfiguration functionality to DVEs. I call it Bullpen because of the similarities between compensating reconfiguration and the function of a baseball manager with respect to his pitching staff. A manager must make decisions concerning his bullpen once a game is underway, i.e. what to do once his starting pitcher gets tired. He must decide on the proper compensation, such as substituting a right-handed pitcher. This is like compensating reconfiguration; conditions occur during play which affect the outcome of the game. In making compensation decisions, the manager must consider the pitchers he has available (system configuration), the hitters that are scheduled to come to bat (virtual world) and how the hitters match up to the pitchers (mapping).

My goal is to make it easier for DVE builders to make compensating reconfiguration a part of DVEs. For this to be the case, it must be at least as easy to initially create a DVE that includes compensating using Bullpen as it is by manual programming. It must also be easier to maintain the DVE as its requirements evolve. In the military DVE domain, the users tend to change the requirements frequently throughout the life of the project. Because reconfiguring a DVE to compensate for conditions is a resource allocation problem, the DVE builders should be able to experiment and improve their solution. A tool that is no more difficult to set up initially, but is much easier to work with as the requirements evolve, will be of great benefit to DVE builders.

To include compensating reconfiguration function easily into a DVE you must minimize both effort and complexity. These two factors generally complement each other, but in the military DVE domain complexity has added importance. Military DVE builders generally are not programmers. The primary job of a DVE builder is to integrate existing components to create a DVE, not to program the components of the DVE. Therefore, the less complex the tool interface and the more it resembles the way the users think of the DVE, the easier it will be to use.

The compensating reconfiguration code produced by Bullpen must be high quality. The important quality issues include the reliability, the reaction time, and the expressiveness of the interface. Decreased expressiveness and execution speed usually offsets the advantages of tools. An effective tool must balance these tradeoffs.

I designed Bullpen so that it would require less effort both in the initial construction and in the implementation of requirements changes likely in the military DVE domain. I also minimized the complexity of the interface, making it conform more directly to the way the non-technical users think about the DVE. While doing so, I kept the interface expressive enough to handle military DVE domain requirements.

4.1 Bullpen in DVE Architectures

There are two primary approaches to DVE architecture. The fully distributed approach is characterized by independent components passing messages over a network either through general or specialized protocols. Each component is responsible for providing any services it requires. Interconnection infrastructure architectures provide a number of services to include message passing between the components.

Bullpen can be used in either architecture. In both architectures, it can execute as an independent component of the DVE. It interacts with the DVE just as other VEs would, by receiving messages and occasionally sending messages to compensate for a condition. Figure 4.1 illustrates this scheme in a fully distributed architecture. Bullpen can also be part of the interconnection infrastructure, providing compensating reconfiguration services to the DVE as depicted in Figure 4..2.

This case study includes scenarios using both the fully distributed and interconnection infrastructure architectures; conceptually they are very close. Bullpen only inserts itself into the DVE execution when required. Most of the time it simply monitors the message traffic of the DVE. In each scenario of the case study the DVE uses multicast for message passing. Consequently, there was no significant overhead to the DVE during normal operation. In an architecture that uses unicast then there would be a slight overhead associated with the additional message recipient.

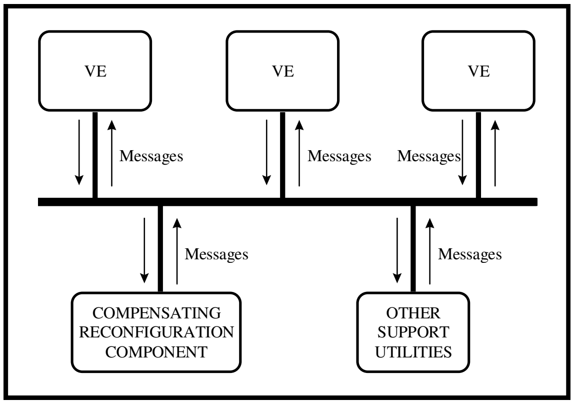

Figure 4.1: Fully Distributed DVE Architecture

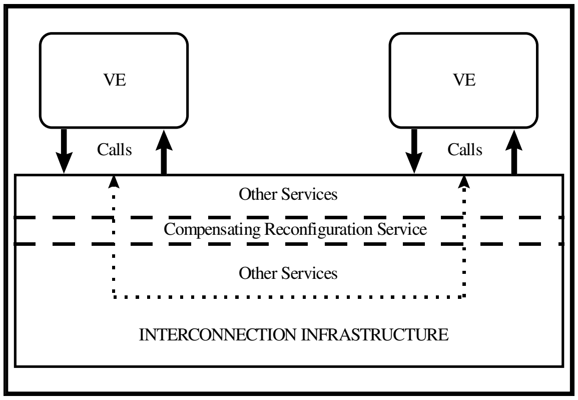

Figure 4.2: Interconnection Infrastructure Architecture

4.2 Bullpen Architecture

When it detects a condition of interest, Bullpen makes two decisions. The first decision determines what action the DVE should take to compensate for this condition. If reconfiguration is necessary, then it makes the second decision, i.e., how to dynamically change the DVE to meet the desired configuration. When making these decisions, it relies on its own abstract version of the system configuration, virtual world, and virtual world - system configuration mapping. Bullpen has three independently executing sub-components that perform these functions.

Scout is the sub-component that detects conditions of interest and reports them along with any preliminary information to Coach. Coach contains the decision-making logic for the compensation and reconfiguration decisions. Coach executes the reconfigurations by using the DVE reconfiguration protocol. Scoreboard is responsible for furnishing the abstract virtual world, system configuration, and mapping information to Coach as it makes its decisions.

Scout

Scout reifies the condition detection function. All information entering the compensating reconfiguration software passes through Scout. Scout determines what Bullpen should do with each message that it receives. In all three scenarios that I have used for this study, Scout detects conditions passively by monitoring message traffic. Scout can also query hosts or use other active means to detect conditions but there would be a small performance penalty.

In HLA and DIS each VE is required to send a periodic heartbeat message, if it has not sent any other messages a specified duration. This provides a way to detect *absentee* VEs. If Bullpen has not heard from a VE before the wait time has expired then it assumes the VE is down. Consequently, Scout reports the VE to Coach as an *absentee*.

Another condition that Scout has been used to detect is the return of a VE that Bullpen has previously reconfigured out of the DVE. When Scout receives a message from one of these VEs it reports the condition to Coach. In addition, I have used Scout to detect high processor utilization and the movement of virtual entities into a geographic area of the virtual world.

Scout has no abstract interface. Condition detection is very architecture and implementation specific. It also has the most impact on performance in the DVE during normal operation. An abstract interface to condition detection would have a much greater impact on performance than the abstract interfaces in use in other parts of Bullpen. In addition, requirements for condition detection generally do not change as frequently as other categories. Once the conditions of interest are determined for an architecture and domain, they are not as likely to change as are other inputs to compensating reconfiguration.

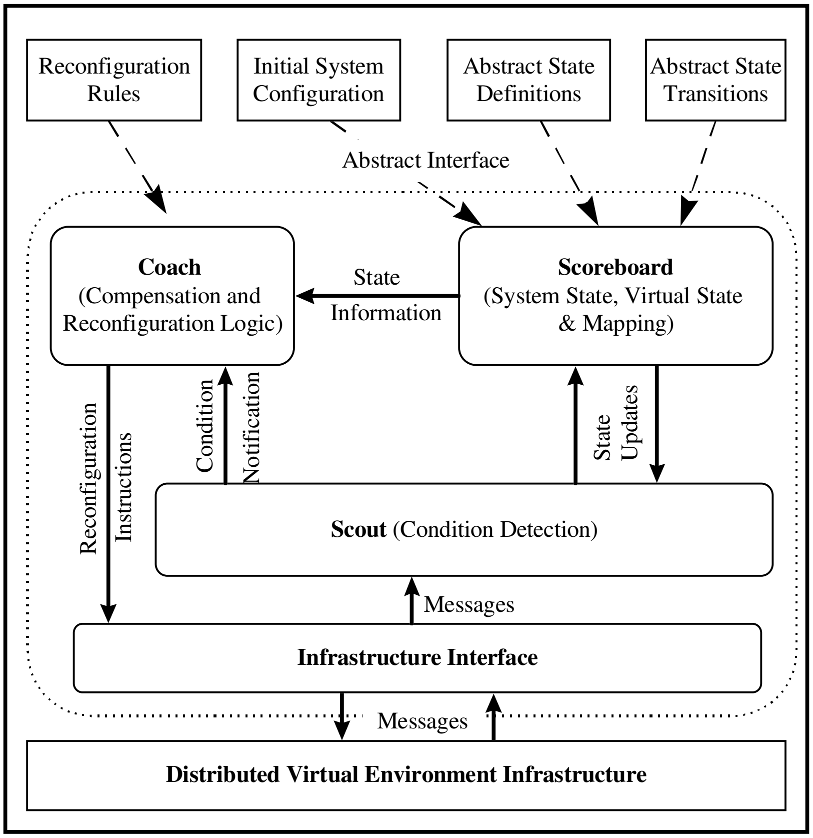

Figure 4.3: Bullpen Architecture

Coach

Coach encapsulates the decision logic of Bullpen. At its heart is the Java Expert System Shell (JESS) developed by the Johnson Space Center. [Jes97] JESS, as a part of Coach, can query Scoreboard and send reconfiguration instructions to the DVE. JESS is a forward-chaining expert system and handles both the compensation and reconfiguration decisions.

The abstract interface of Coach is the CLIPS expert system programming language. [STB97, Gia93] It is an "if-then" type rule-based system. In addition to the rules, the expert system has functions that the DVE builder can be write in high-level source code. These functions are how the expert system queries the database and sends reconfiguration instructions to the DVE.

Coach serializes each condition to reduce the complexity of the rules by eliminating the interaction possibilities of simultaneous conditions. Although this can slow reaction time, the advantage of reduced complexity far outweighs the disadvantage.

Coach Walk-through Example

To better illuminate how Coach works I will use the rules in Figure 4.4 to step through a simple example. First Coach receives a message from Scout indicating an absentee condition. Coach asserts information about the absent VE and the entities that it owns as facts in the expert system. Coach handles each of the orphaned entities in this example singly.

```
(defrule is_ground_vehicle
   ?orp <- (orphan (handled no) (obj-type M1A2 | M109 | T80 |  BMP | SP2))
=>
   (modify ?orp (handled sw) (software VehCGF)))

(defrule search_for_host
   (orphan (name ?name) (software ~none))
   ?orp <- (orphan (name ?name)(handled sw) (software ?software))
=>
   (modify ?orp (host (F_FindHost ?software spare)) (handled host)))

(defrule start_ve
   (orphan (name ?name) (handled host) (host ~none) (software ~none))
   ?orp <- (orphan (name ?name) (host ?host) (software ?software))
=>
   (modify ?orp (new-fed (F_StartVE ?host ?software)) (handled ve)))

(defrule transfer_orphan
   (orphan (name ?name) (handled ve) (new-fed ~none))
   ?orp <- (orphan (name ?name) (new-fed ?new-fed))
=>
   (F_TransferObject object ?name ?new-fed)
   (F_AttributeUpdate ?name)
   (retract ?orp))

(defrule VE_failed
   ?orp <- (orphan (name ?name) (handled fed) (new-fed none))
=>
   (modify ?orp (handled sw)))

(defrule remove_object
   ?orp <- (orphan (name ?name) (handled host) (host none))
=>
   (F_RemoveEntity entity ?name)
   (retract ?orp))
```

Figure 4.4: Example Reconfiguration Rules

The first rule to act on the orphans is the *is_ground_vehicle* rule. This rule identifies software that can own the orphaned entity. The rule adds this choice to the orphan template for further use.

The next rule to fire is *search_for_host*. This rule will only fire when the previous rule has identified appropriate software. In the case where there is no appropriate software, this chain of logic terminates and it purses another. Finding a host that can accommodate the designated software to create a replacement VE requires querying the database. The rule does this through the function *F_FindHost*. It returns a suitable host name if one is available which is stored in the orphan template.

When an orphan template has identified replacement software and a host the *start_VE* rule will fire. The function *F_StartVE* performs this operation by sending reconfiguration messages to the DVE. The function starts by creating a unique name for the new VE. Invoking the UNIX operating system *rsh* call to start the VE on the designated host, Bullpen waits for a message from the new component to insure that the operation was successful. This rule returns the name of the new VE and it is stored in the orphan template.

The *transfer_orphans* rule performs the final step for a successful replacement. It fires when an orphan template indicates that it has an appropriate replacement VE running and waiting for it. This rule will use the function *F_TransferObject* to update Scoreboard and invoke the reconfiguration primitives to transfer the entity to the new VE. In this case the VE, which started from scratch, does not have state information for its new entity. Consequently, the function *F_UpdateAttributes* sends messages to the new VE with the virtual state information from Scoreboard. The successful reconfiguration is complete and Coach retracts the *orphan* template.

As Coach follows the above path, there are numerous places for failure. The final two rules deal with this. If the operation to start the new VE was unsuccessful, then the rule *VE_failed* will fire. All this rule does is reset a fact in the orphan template so that the expert system will start over for this orphan searching for a suitable host. This cycle will continue for all orphans until either Coach has assigned them to new VEs, or the possible hosts have been exhausted. If Coach cannot find a suitable host then the rule *remove_entity* fires. This is the last resort for Bullpen. Bullpen removes the orphan entity from the DVE. The behavior is not ideal, but without more resources, there is no choice. It is a correct reconfiguration because each entity in the DVE is still controlled by one and only one VE.

Coach Summary

Coach encapsulates the decision-making logic. It presents an abstract interface to the user that is easy to reason about. The CLIPS language is powerful and expressive. Using functions it can query Scoreboard and reconfigure the DVE by applying the reconfiguration primitives of the DVE architecture. The functions are a higher level abstraction of the reconfiguration primitives and queries. They are not normally modified while responding to changing requirements or prototyping. Incorporating new conditions or changes to the DVE architecture reconfiguration primitives require changes to the functions used by JESS which means recompiling Bullpen.

Scoreboard

Scoreboard is Bullpen's internal database. The purpose of Scoreboard is to provide coach with system and virtual world information. Scoreboard keeps salient information about the system and virtual state. More importantly, it keeps an abstract hierarchical representation of the virtual state. The users of military DVEs think of the virtual world in abstract terms. Scoreboard organizes virtual entities into hierarchies of units, each of which has its own state. Reasoning about the compensation and reconfiguration rules in this abstract manner, rather than reasoning about the attributes of the entities, not only makes it easier, but also helps to minimize the effects of changes to the virtual world on the reconfiguration rules.

Scoreboard maintains the mapping between the virtual world and system configuration. There is a natural relationship between the virtual world and the system world that Bullpen must be able to access to fully exploit its capability. DVE builders can make rough predictions about the future system usage based on the current abstract state. A unit that is *ineffective* will be generating fewer events because a large percent of its objects are *dead* and only sending minimal messages. In addition, ineffective units are rarely used in battle. Therefore the entities that are part of that ineffective unit that are still *alive* will also be less likely to generate events because they will not be maneuvering and shooting at the enemy. They will most likely be held in reserve or in a rear area while being brought back up to strength. A VE that owns an *ineffective* unit would be a good candidate to take over the entities from an absent VE, if no other spares are available.

Entity Grouping and Abstract Entities

Normally it is easier to reason about entities as groups. It is easier to conceive of a car, rather than the chassis, the people in the car, and the fuel in the gas tank. This is certainly the case in the military domain. The military organizes its units in hierarchies. Military users who will define the requirements think in these abstract terms. They think in terms of battalions rather than fifty-four tanks, thirty-two armored personal carriers and seventy-six trucks.

Scoreboard keeps this hierarchical grouping as *Abstract Entities* and allows Coach to query based upon it. Figure 4.5 is an illustration of the hierarchy. A tank company consists of platoons. Each platoon contains individual vehicles that are the actual entities in the DVE. Depending on the mission, the mix of platoons will vary. A tank company may have all tank platoons, or may have a mix of infantry and tanks. Reasoning about the abstract entities higher up in the hierarchy as opposed to the actual entities protects the logic from changes in the make-up of the virtual world. If a rule is based on the state of the Tank Company in Figure 4.5 and that company gets an extra tank platoon, then the rule can accommodate the additional platoon without modification.

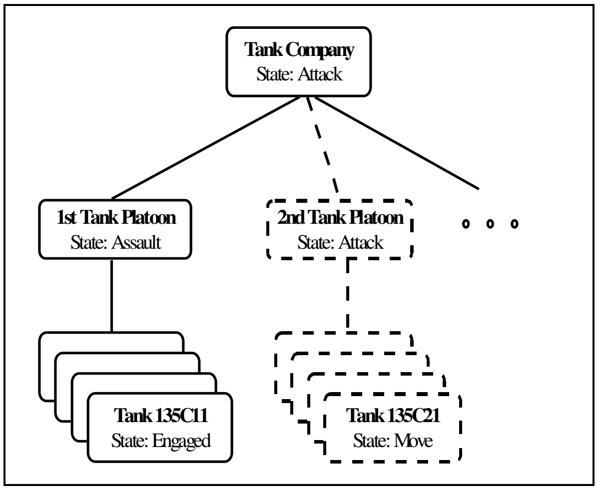

Figure 4.5: Abstract Entity Hierarchy

```
LEVEL "task_force"
  ABSTRACT "TF1-14" "task_force" ILK "maneuver" "halt"
    LEVEL "team"
      ABSTRACT "A/1-14IN" "TF1-14" ILK "maneuver" "halt"
        LEVEL "platoon"
          ABSTRACT "1/A/1-14IN" "A/1-14IN" ILK "maneuver" "halt"
            ENTITIES "114A11", "114A12", "114A13", "114A14"
          ABSTRACT "2/A/1-14IN" "A/1-14IN" ILK "maneuver" "halt"
            ENTITIES "114A21", "114A22", "114A23", "114A24"
          ABSTRACT "3/C/1-35AR" "A/1-14IN" ILK "maneuver" "halt"
            ENTITIES "135C31", "135C32", "135C33", "135C34"
      ABSTRACT "B/1-35AR" "TF1-14" ILK "ground maneuver" "halt"
        LEVEL "platoon"
          ABSTRACT "1/C/1-35AR" "A/1-14IN" ILK "maneuver" "halt"
            ENTITIES "135C11", "135C12", "135C13", "135C14"
        END LEVEL
    END LEVEL
END LEVEL
```

Figure 4.6: Abstract Entity Grouping Example

The DVE builder groups entities into hierarchies through an abstract interface in Scoreboard. He defines the abstract entities for each level of the hierarchy. Each abstract entity may consist of additional abstract entities through the definition of another level. At the bottom of the hierarchy, instead of another level definition there is a list of entities for that abstract state. Each abstract state has a name, the level it is defined in, its ilk and initial state. Its ilk is useful for defining a category of abstract states that will all use the same state transitions. Figure 4.6 shows an example of the entity groupings.

Abstract Entity State Transitions

Figure 4.7 illustrates the state transitions for different levels of the abstract entity hierarchy. At the top level, the Tank Company changes state based on the state of the units one level lower in the hierarchy. There are only two transitions: to the *Consolidate* and *Ineffective* states. If at any time half of its platoons are *Ineffective*, then the Company is *Ineffective*. Once all of the platoons that are not *Ineffective* are *Consolidating*, the Company is *Consolidating*.

The Tank Platoon is similar. It changes state based on the state of the individual vehicles assigned to it. The Tank Platoon begins in the *Attack* state. Once it starts to engage the enemy, it moves into the *Assault* state. When all its vehicles that are still functioning are *Consolidating*, the Platoon is *Consolidating*. If at any time it looses more than half its vehicles, the Platoon moves to the *Ineffective* state.

At the bottom level the individual tanks will change state based on their own actions. Each VE determines the state of the entities that it controls. In this example, the Tank starts out preparing for the operation in an assembly area. When it begins to move towards the enemy it changes state to *Move*. When it comes under fire or starts to fire at the enemy it transitions to the *Engaged* state. When it reaches the geographic area of the objective it is *Consolidating*. Of course if at any time it is destroyed, breaks down, or runs out of ammunition or fuel it becomes *not mission capable*.

Reasoning about the states of the individual vehicles can be very complex. Without keeping a representation of the relationships between the entities it might be impossible to match the thinking of the users with the implementation of compensating reconfiguration. Regardless it is less complex to keep the implementation aligned with the users thinking when the relationship is kept in the database.

Scoreboard must move the abstract entities along their state transitions based on the behavior of the actual virtual entities of the DVE (vehicles). Scoreboard provides an abstract interface to these state transitions. Figure 4.8 shows example state transitions for the abstract entities. DVE builders can define transitions for individual abstract entities or for types of abstract entities. The example in Figure 4.8 shows the transitions for two types of units: tank platoons and tank companies.

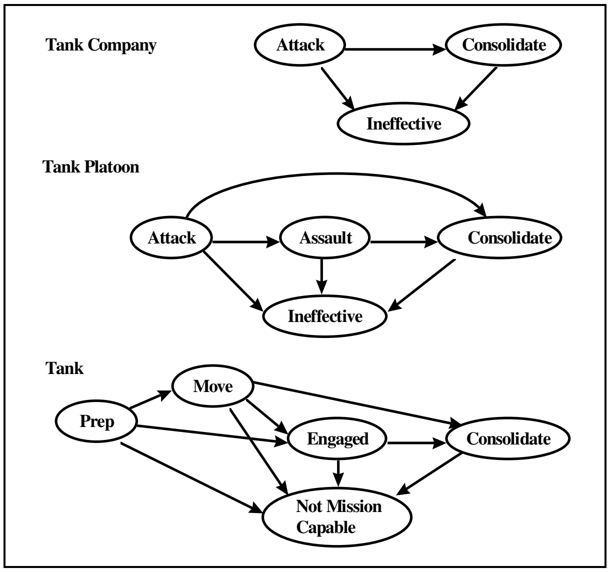

Figure 4.7: Example Abstract State Transitions

```
LEVEL unit tank_company
    TRANSITIONS
        IF (state = "ineffective" >= 50% )
            -> "ineffective";
        FOR {state != "ineffective" }
            IF (state = "consolidate"  = 100%)
                -> "consolidate";
END

LEVEL unit tank_platoon
    TRANSITIONS
        IF (state = "not mission capable") >= 40%
            -> "ineffective";
        IF (state = "engaged" > 0%) OR ((x > 150) > 0%)
            -> "assault";
        FOR { state != "not mission capable"}
            IF (x > 400 = 100%)
                -> "consolidate";
END
```

Figure 4.8: Example Abstract Entity State Transition Definitions

The abstract entity state transitions are based on the same concept as "triggers" used in relational database management systems. When an entity's attribute changes, either through a message or a dead-reckoning update, the trigger fires. Scoreboard then checks to see if the abstract state one level above also changes. If so, it recursively follows the hierarchy until the abstract state either does not change or it reaches the top of the hierarchy.

The first transition for the tank company says that if more than fifty percent of its subordinates in the hierarchy are *not mission capable*, then the company transitions to *not mission capable*. The second transition says that for all the subordinate units that are still operational (not *not mission capable*) if all of them are *consolidating* then the company transitions to *consolidating*.

The tank platoon has three transitions: to *not mission capable*, to *assault*, and to *consolidate*. The first is exactly like the Tank Company. The second transition is based on the state of the individual tanks as reported by their controlling VEs. If any tanks are *engaged*, then the entire platoon transitions to *assault*. The final transition is based on the location of the tanks. When all the operational tanks cross a geographic line, the platoon transitions to the *consolidate* state. This scheme takes advantage of the hierarchical groupings and common transitions for similar units to simplify the specification of transitions for abstract states.

Scoreboard Summary

Scoreboard stores information about the virtual world and system configuration for Coach to use when making compensation and reconfiguration decisions. It uses three abstract interfaces to simplify its initialization. The DVE builders make the declaration of the items stored in Scoreboard and their relationships separately from the logic to manipulate these items as the DVE evolves.

4.3 Summary

Bullpen is designed to provide compensating reconfiguration functionality to a DVE at a bargain. Using abstract interfaces, the compensating reconfiguration process is easier to reason about. The most common areas of requirements changes are dealt with by abstract interfaces. Encapsulation of the different operations performed by Bullpen helps to isolate changes, and minimize the locations affected by a requirements change. Abstract states keep the implementation more closely aligned with the user requirements, and provide a powerful way of exploiting the relationship between the virtual world and system state.

Using Bullpen does not guarantee appropriate or correct reconfigurations. Just like any tool, the user can improperly apply it with incorrect results. Using Bullpen does make it easier for the user to focus on the reconfiguration actions. In manual programming the compensation decisions are imbedded in the high-level source code, which obfuscates the logic. Bullpen’s abstract interfaces isolate the compensation decisions from the rest of the code and make them easier to reason about.

5. CASE STUDY DESIGN

I conducted a case study to understand the advantages of using Bullpen as compared to traditional manual programming techniques when building compensating reconfiguration software for DVEs. I wanted to understand the effort expended and the complexity of implementing requirements changes with Bullpen as compared manual programming. I wanted to compare the performance of the components created by both methods so as expose any performance penalty associated with Bullpen, whether in terms of reaction time, expressiveness, or reliability of the compensating reconfiguration software.

This case study has three similar but slightly different scenarios, performed sequentially. The major goal of the first scenario was to validate a design for Bullpen, one adequate to carry through the subsequent experimentation. For the second scenario, I compared Bullpen with a hand-coded version in a realistic environment. In the final scenario, I further investigated compensating reconfiguration, especially in the areas where Bullpen does not do as well. Requirements changes that could not be made using only the abstract interfaces of Bullpen and small changes are weaknesses of Bullpen.

5.1 Methodology

I employed generally the same methodology for all scenarios tested. First I developed an environment in which to test the approach, including definition of the virtual world. I then developed the components to support the environment: the message passing infrastructure, the simulated VEs, and the support utilities and instrumentation that allowed me to observe the DVE.

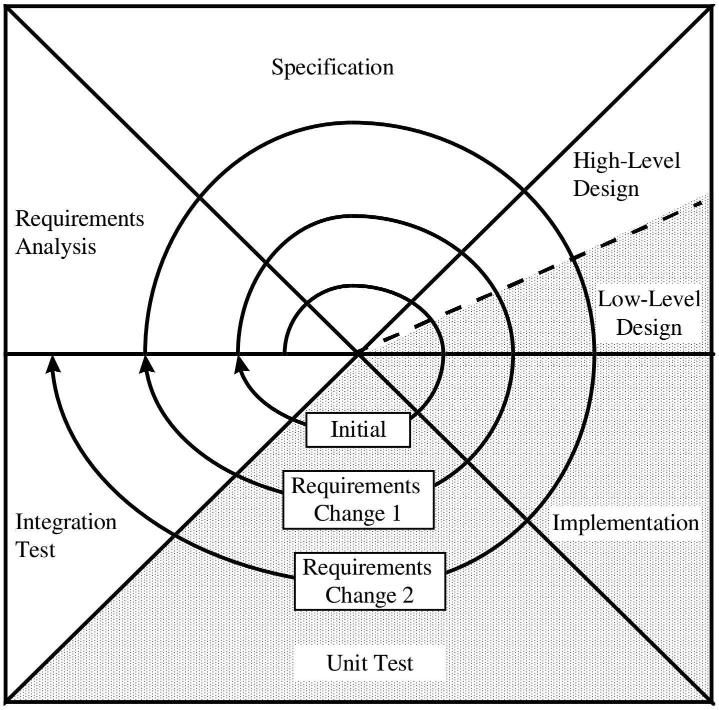

Figure 5.1: Scenario Lifecycle

The HLA scenario was a little different in that the virtual world was the same as was used in the Advanced Concepts Technology Demonstration (ACTD) for the HLA RTI. [Bea97] The requirements changes in this scenario were also provided by ACTD project team.

After developing the environment, I began iterating through the requirements changes. I followed a spiral lifecycle process as depicted in Figure 5.1. The initial DVE construction with compensating reconfiguration functionality included is the first step. Each successive requirements change is shown as an additional traverse of the spiral.

The initial DVE construction differed slightly between the versions. For manual programming, I built the baseline compensating reconfiguration component in accordance with the initial requirements using traditional object-oriented techniques. Initial construction with Bullpen started with the Bullpen framework. The framework was then adapted to the DVE architecture. Once that was completed, I then added the compensating reconfiguration functionality to conform to the initial requirements.

The requirements change requests were made from the current version in a cumulative manner. For each requirements change request, I conducted the requirements analysis, specification and high-level design for all methods simultaneously because the process was identical for all compensating reconfiguration versions. Then for each version being evaluated, I then did detailed design, implementation and unit test. After completing those phases I then conducted integration test.

In Figure 5.1 the non-shaded portion indicates the parts of the lifecycle that are common to all versions. Requirements Analysis, Specification, and High-level Design produced products that were shared by all versions. I conducted Integration Test on the separate versions at the same time. All versions would pass a task before moving on to the next. The shaded portion indicates the portions that I performed and evaluated independently for each version. These phases all produced different artifacts. The effort and one of the complexity metrics that I collected were produced during these phases.

5.2 Independent Variables

The independent variables consisted of the compensating reconfiguration version and the type of requirements change. During the first scenario, I used three versions of Bullpen to verify the design of Bullpen. For all the scenarios, I built a hand-coded version to compare against Bullpen.

Requirements Change Taxonomy

It is important to identify the types of requirements changes that Bullpen handles well along with the types it does not. To help make this evaluation I created a taxonomy of requirements changes to compensating reconfiguration. In addition to these categories, I also tracked the results from the initial creation of compensating reconfiguration for that scenario. A summary of the categories is in Table 5.1.

Virtual Configuration changes involve changes to the virtual world and its members. Introducing a new military unit with its member entities to the DVE is a change to the virtual world. Changes in this category would normally come from the users of the DVE. They are not normally among the most difficult or complex changes. However, in a large DVE such as the Advanced Technology Concept Demonstration they are frequent and so must be implemented quickly.

Changes that are part of the System Configuration category involve the hardware or software that make up the DVE. The addition or subtraction of host computers is an example, as is the introduction of new VE software. Some of these changes will come from external sources, such as the availability of additional resources or the unexpected unavailability of expected resources. Some changes come from the DVE builders themselves as they use prototyping to learn more about how to allocate the available resources.

Table 5.1: Requirements Change Categories

Category (Abbreviation)	Short Description
Virtual Configuration (VC)	Changes to the virtual world only. Members, Organization.
System Configuration (SC)	Changes to the hosts, or software components involved in the DVE.
Reconfiguration Policy (RP)	The choice of reconfiguration actions in response to conditions. The reconfiguration rules.
Reconfiguration Interface (RI)	The ways the reconfiguration code must interact with the DVE architecture. Changes to the reconfiguration primitives.
Conditions (C)	Changes to the conditions handled.
Abstract Interface (AI)	Changes that are implemented solely through Bullpen's abstract interface. No recompiling is required for Bullpen.
Initial	The initial implementation of compensating reconfiguration. Includes all types of requirements.

Reconfiguration Policy changes alter the appropriate response to a condition. If the current policy says that a failure of a logistical VE is handled by removing those entities from the DVE, then a Reconfiguration Policy change would be a policy that transferred the orphan entities to other available logistical VEs. These changes will come from both the users and DVE builders.

The Reconfiguration Interface is the reconfiguration primitives defined by the DVE architecture. So requirements changes in the Reconfiguration Interface are driven by changes to these primitives. For example, suppose the architecture currently requires only one message to transfer entity ownership from one VE to another. If the DVE reconfiguration interface were to change so that all involved VEs must now give their approval before transferring an entity, the resulting requirements change would be a Reconfiguration Interface change. These types of changes are infrequent and externally driven. They normally occur when upgrading the DVE architecture or interconnection interface.

The least frequently encountered changes are Conditions requirements changes. A major change would be the handling of a new condition such as high cpu utilization on a host. Adding the capability to load-balance the hosts is a Conditions category change. A minor change might be to increase the time interval used to denote an absent VE. These changes might come from the users or the DVE builders, but would not be common during the development of a DVE project.

Many requirements changes fall into more than one category. For example, the addition of a new type of entity to the DVE would be included in Virtual Configuration, because the virtual world has changed. Should a new VE control the new entities, then the requirements change is also part of the System Configuration category. If the entity type were radically different, requiring a new definition of an appropriate reconfiguration, the change would also be part of the Reconfiguration Policy category.

Because many of these changes are not mutually exclusive, I have added an aggregate category called Abstract Interface. Requirements changes that were made exclusively through Bullpen’s abstract interfaces make up this category. Most requirements changes that are in the Virtual Configuration, System Configuration, and Reconfiguration Policy categories are also in the Abstract Interface category. I excluded members of those categories that also fell into the Reconfiguration Interface or Condition change categories because they required changes to Bullpen’s source code. This category is representative of the types of requirements changes found in military DVE projects. This category is also representative of how Bullpen performs as part of an interconnection infrastructure.

The Initial category is for the first-time construction of the compensating reconfiguration components for a scenario. This information is important in determining how practical Bullpen is in normal use. The original implementation for every DVE constitutes the baseline from which I can measure requirements changes. First-time construction is a collection of requirements that fall into all of the categories.

The most important categories of change are the Virtual Configuration, System Configuration, Reconfiguration Policy, and the Abstract Interface aggregate category. Requirements changes in these categories are the most frequent. The users and the DVE builders drive them. The users will change the virtual world to meet the goals of the DVE use. The DVE builders will change the system composition and reconfiguration rules to provide the best DVE for the users. [Shi97]

5.3 Dependent Variables

In the dependent variables, I wanted to capture the effort and complexity of using Bullpen compared to developing compensating reconfiguration by hand. In addition, I wanted to determine how the resulting compensating reconfiguration components performed.

Source Lines of Code

SLOC is a common measure of effort. Many of the drawbacks to using SLOC are mitigated in this case study. Since I did all the coding, there was no difference in style and the domain was constant. All the high-level language is the same programming language, Java ™, although there are a number of abstract interfaces used in Bullpen.

In my SLOC count I did not include comments, white space, or debugging code. It does include code that was added, modified or deleted from a module. I did not count the removal of dead-code (that is, code that became unreachable due to modifications in the program.)

Implementation Time

I measured time in person-hours. I did not count time that was shared among all the methods. Shared time includes requirements analysis, specification and high-level design. All the time that I spent in detailed design, coding and unit test is included in the implementation time (the shaded portion of Figure 5.1). I included this metric as a balance to SLOC. SLOC is more objective, but because of the differences between languages used, implementation time is an important factor in effort.

Locations

The more dispersed the modifications to a program, the more complicated those modifications. Therefore, I tracked the locations that were touched while implementing a change. I defined a location as a method in high-level source code. In CLIPS, the rule-based language, a location is a group of rules that fire on the same set of conditions. In other words, rules that fire on the same facts (different values) are a group. In CLIPS, I delineated the rules in a single location by comments. In the other languages used in Bullpen, I determined locations more subjectively; my rule of thumb was that changes that fell into one screen were in the same location.

Defects

To judge the complexity and reliability of the compensating reconfiguration code I used the defects identified during integration testing. Once I had tested the modifications made in individual components in isolation, I tested the compensating reconfiguration software in the DVE of that scenario and tracked the identified defects. This metric includes all defects regardless of severity; incorrect reconfigurations is a subset of defects.

Repair Time

I used the defect repair time to contribute to my complexity evaluation. I reason that the more difficult the repair, the more complex the code. In repair time I include the time I took to identify, repair, and test the repair. I did not include the time required for the full regression test.

Incorrect Reconfigurations

To supplement the defect metric I also tracked the most serious categories of defects; incorrect reconfigurations. Incorrect reconfigurations are defects that result in an inconsistent DVE as defined in Definition 3.2. All defects that I identified that left the DVE with entities that are not owned by a functioning VE, or entities that are controlled by more than one VE make up incorrect reconfigurations. This metric contributes to my determination of reliability. I reason that the more incorrect reconfigurations identified during integration test, the more likely production code will cause an inconsistent DVE in operation. Since I was not able to evaluate the components in a production DVE, I use incorrect reconfigurations as a predictor for errors in use.

Reaction Time

Reaction time is part of the analysis of Bullpen's performance. Reaction time is partitioned into three pieces: the detection time, decision time, and reconfiguration time. The following example explains the three parts of reaction time. VE_X sends out an update and then immediately crashes at time 0. Due to network latency, its next heartbeat will be overdue at 5010 at the compensating reconfiguration component. At 5010 the compensating reconfiguration software determines a condition has occurred and begins to decide how to compensate. That 5010 is detection time and will not vary between a high-level language and Bullpen implementation. The compensating reconfiguration code determines that it must transfer the entities controlled by the absent VE to spare VE_Y. At time 5075 it sends the first of the messages that must be sent to reconfigure the DVE. I attribute 65 to the decision time. This is the where the differences between the two versions will be the greatest. At 15000 all requests have been answered, and notifications sent so that the DVE is now properly reconfigured. This 9075 is reconfiguration time. This time will also vary between the two approaches, but the differences will be proportionately much smaller. This is because during reconfiguration, the software still makes some decisions, but the majority of this time is spent waiting for responses from the VEs involved in the reconfiguration.

5.4 Threats to Validity

My goal is to prove the viability of a tool-based approach. Because it is my tool and I am designing and conducting the case studies, there is a potential for bias to enter my work. I have tried to address this as much as possible within the context of my playing the role of advocate and evaluator.

Internal

The first time that I address a given requirements change I gain experience that helps during with the other versions. Generally, I can work a little faster and in some cases know errors to avoid. The first version I modify should have longer implementation time and may have more defects. To ameliorate this effect, I alternated the order of evaluation for each requirements change.

I chose two metrics to reflect effort: source lines of code and time in person-hours. Do these accurately reflect the effort required? Detailed design, implementation and unit test only take up a part of the total life-cycle. Because this case study deals with requirements changes it uses a compressed life-cycle without maintenance. I am assuming that requirements analysis, specification, high-level design and integration test effort is equal between all methods used. Based on my experience in production projects and with this research I feel this is a good assumption. If I am wrong, this will affect my results.

I use source lines of code (SLOC) to measure effort. Not all SLOC are created equal even within the same language. In addition, I equate SLOC from different languages. To make this use more valid I use both SLOC and implementation time as effort metrics. The correlation is close over the entire case study, but this division of effort reveals more about using Bullpen in different types of requirements change.

My complexity metrics are the locations where changes are made to implement the requirements changes, the defects identified during integration test, and repair time for those defects. I reason that the more spread out the changes are the more difficult they are to conceptualize. I assume that more complex code results in more defects that are harder to identify and repair. Should this assumption be invalid, then my judgment of complexity may be incorrect. More conventional measures of complexity such as function points do not meet my needs. Many conventional complexity measures are implementation independent which defeats the purpose of this research.

I have been involved in building tools for compensating reconfiguration for a significant period of time. Consequently, I have reached conclusions about the architecture and implementation of compensating reconfiguration code. It is impossible for me to ignore this point of view when constructing the hand-coded versions. There may be better hand-coded architectures that yield better results than what I have implemented. To ameliorate this condition I have tried to design an architecture during the initial construction that is fundamentally different from Bullpen's architecture. When implementing the requirements changes, this forced me to think about the implementation in a different way. By forcing myself to look at the problem a little differently, I feel it increased my chances of seeing higher quality solutions, rather than having a solution quickly appear to me because I had worked a similar problem in Bullpen.

External

In my role as advocate and evaluator, I chose over half of the requirements changes used in this case study. To counter this I have incorporated a scenario from an actual DVE project. I have also tried to choose changes in which my approach out performs hand-coding as well as ones in which hand-coding is superior.

This case study was conducted with a simulated DVE environment. I did not have the resources–human, hardware and software–to test in a real environment. To simulate the environment I abstracted away much of the virtual world and focused on the systems aspect and those features salient to my work. In addition, I also simplified the DVE interconnection architecture focusing on the dynamic reconfiguration services and neglecting such services as data distribution management.

There is a risk of abstracting away an important aspect of the environment in doing this. However, the dilemma I faced is “the chicken or the egg.” I believe that HLA would benefit greatly from compensating reconfiguration by integrating Bullpen into the run-time infrastructure. However, HLA does not support dynamic reconfiguration as currently implemented. If this type of service is to be included in future versions of HLA, then research must be done to show that it will work. Therefore, by abstracting, I am able to do the preliminary work that may lead to its eventual incorporation into the HLA.

I have chosen the military domain as my model for DVEs. Should this be a poor choice, this work would have little practical use beyond military DVEs. Compensating reconfiguration is best at dealing with complex DVEs that must be reliable. Currently, HLA represents the largest use of DVEs of this type. HLA has interest worldwide and will be a significant player for the near future of DVEs. Therefore, providing compensating reconfiguration to HLA DVEs will have a significant effect on the DVE domain.

5.5 Scenarios

I used three scenarios in this case study. The scenarios were similar, but each involved a different environment and each had a slightly different goal.

Fly-off Scenario

The main goal for this scenario was to select the best design for Bullpen. It was essentially a head-to-head competition to determine which design would prevail, hence the name Fly-off scenario. While determining the final design for Bullpen, I also wanted to gather data that compared the eventual Bullpen design to a hand-coded implementation of compensating reconfiguration.

My original plan was to use four versions of compensating reconfiguration (hand-coding and three versions of Bullpen) for all the case studies in this scenario. The three Bullpen versions were called Forward-Chaining, Backward-Chaining and Procedural. Forward-Chaining Bullpen used one forward-chaining inference engine (JESS) for both the compensation logic and the reconfiguration logic. It is the version that I eventually selected for Bullpen's design that is described in Chapter 4. The Backward-Chaining Bullpen has two inference engines: a backward-chaining engine for the compensation logic, and the forward-chaining engine for the reconfiguration logic. This version had strict encapsulation of the two types of decisions. The third version, Procedural Bullpen, used a procedural scripting language to implement the compensation logic, and parameterized scripts to implement the reconfiguration logic. Its purpose was to take advantage of abstract states and the virtual system mapping in Scoreboard, but not bear the overhead of the expert systems. It also had strict separation of the two decisions. It soon became obvious that Procedural Bullpen was inferior to all other versions, so I did not use it for all the requirements changes in the scenario.

Table 5.2: Compensating Reconfiguration Designs

Design	Condition Detection	Compensation Logic	Reconfiguration Logic	Abstract State / Virtual System Mapping
Hand-Coded	High-Level Source Code			
Forward-Chaining Bullpen	High-Level Source Code	Forward-Chaining Expert System		Scoreboard Abstract Interface
Backward-Chaining Bullpen	High-Level Source Code	Backward-Chaining Expert System	Forward-Chaining Expert System	Scoreboard Abstract Interface
Procedural Bullpen	High-Level Source Code	Procedural Scripting Language	Parameterized Scripts	Scoreboard Abstract Interface

The environment for this scenario was relatively small and involved a heterogeneous mix of VEs. At times there were up to nineteen hosts involved. The maximum number of VEs was seventeen and up to twenty-five entities made up the virtual world. This was a very heterogeneous mix, with nine different types of VEs involved and seven different types of entities. There were fifteen requirements changes plus the initial implementation in this scenario. All categories of requirements changes were represented in this scenario.

The DVE architecture was based on DoD's DIS. [Shi95] This is a fully distributed architecture. Each VE is standalone and communication is by multicast. Reconfiguration actions are executed at the component level, rather then the entity or attribute level. The reconfiguration interface is relatively simple. Few responses or acknowledgments are necessary to execute a reconfiguration. The requirements changes in this scenario in general are about as complicated as the Capstone scenario and more complicated than the HLA scenario.

HLA Scenario

The goal of this scenario was to determine if there was an advantage to using Bullpen under the same environment as an actual DVE project. I also wanted to investigate Bullpen in a larger environment. The DVE project that I based this scenario on was the Advanced Concepts Technology Demonstration (ACTD) for the HLA [Bea97] which was used in exercise Unified Endeavor 98.

The purpose of the ACTD was to prove that HLA would work in real use. HLA does not now provide compensating reconfiguration services; the DVE operators manually provide these services. Therefore, the reconfiguration policy is simpler than in the other two series of case studies. The HLA interface was set prior to the start of work on this DVE project and it did not change during construction of the DVE. So there are no Reconfiguration Interface requirements changes in this scenario. The use of conditions was kept simple, absent VEs and reentering VEs. Consequently, there are also no Conditions requirements changes.

This scenario was larger, but less complex than the Fly-off scenario. It involved up to forty-three hosts and three different types of VEs. There were over five hundred entities of forty-six different types of entities in the virtual world. A single VE type could own most of the entities and since in most cases a spare of this type was available, the compensation decisions were usually simple.

Capstone Scenario

The major goal of this scenario was to answer any unanswered questions from the earlier scenarios. Therefore, this scenario explores the types of changes that Bullpen does not handle well. These include requirements changes that involve changes to the high-level source code of Bullpen and small changes to the virtual world and DVE components.

This scenario is more like the first scenario in that the number of elements in the DVE is smaller, but the complexity of the decisions is much greater. The environment included up to twenty hosts of three different types, with up to twenty VEs of twelve VE types and forty-five entities comprised of nine types of entities.

5.6 Summary

This case study includes a variety of requirements changes in an attempt to characterize the difference in effort, complexity, and performance between hand-coding and using Bullpen to produce compensating reconfiguration components. Three scenarios make up the case study. Two of the scenarios are synthetic, and designed to explore the capabilities of Bullpen; the other was based on an actual DVE project to determine how well Bullpen would work in a production environment. The results are characterized within the context of the requirements change taxonomy.

Table 5.3 shows a summary of the mix of hosts, VEs, and entities representative of each scenario. The more types of hosts and VEs, the more complex the compensating reconfiguration logic.

Table 5.3: Typical DVE Configuration by Scenario

Scenario	Hosts	Host Types	VEs	VE Types	Entities	Entity Types
Fly-off	19	2	17	9	25	7
HLA	43	3	43	3	546	46
Capstone	20	3	20	12	45	9

Table 5.4 breaks down the requirements changes evaluated by category for each scenario. The rows list the three scenarios and a total for that scenario. The All column is the number of requirements changes including initial implementation in that scenario regardless of category. Requirements changes can count in more than one category so the All category is not the sum of the other columns.

Table 5.4: Requirements Changes by Category

	VC	SC	RP	RI	C	Initial	AI	All
Fly-off	3	7	9	3	1	1	10	16
HLA	6	7	11	0	0	1	14	21
Capstone	2	3	8	3	2	1	7	13
Total	11	17	28	6	3	3	31	50

Table 5.5 shows the average size (SLOC) of the changes in each scenario by requirements change category. System Configuration changes were the smallest and simplest. The other categories were equivalent in size.

Table 5.5: Average Change Size

	VC	SC	RP	RI	C	AI	All
Fly-Off	56	61	164	179	79	99	114
HLA	166	83	117	–	–	122	122
Capstone	27	96	98	106	213	68	92
Total	119	72	127	152	145	102	113

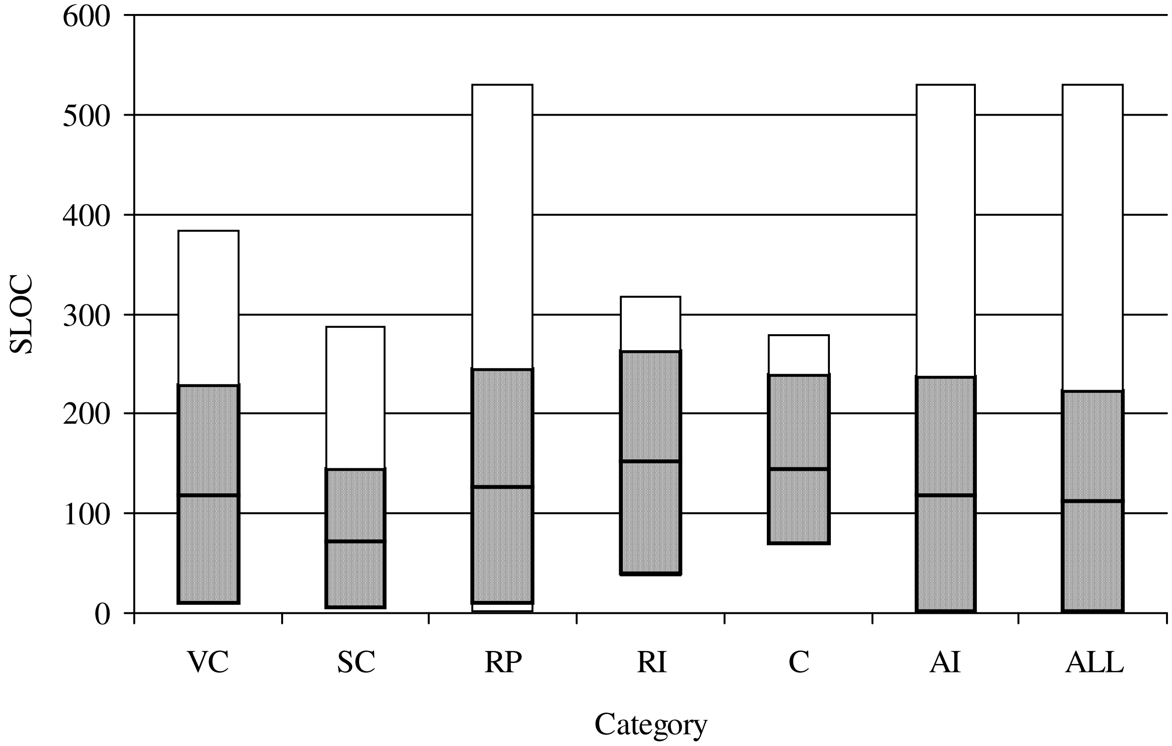

Figure 5.2: Requirements Categories Size Range

Figure 5.2 graphically displays the range for each change category. The center line is the average and the shaded boxes represent one standard deviation above and below the mean. The white boxes show the largest and smallest requirements changes in each category. Reconfiguration Policy case studies had the widest range of size. However, this chart shows that all case study categories represent a wide range of size.

6. RESULTS AND ANALYSIS

The immediate goals of this research have been to show that using Bullpen requires less effort, is less complex, and the resulting code performs well enough for practical use. In demonstrating these benefits, I gain support for my broader hypothesis concerning the benefits of automating any compensating reconfiguration in DVEs.

This chapter is divided into sections for each of these evaluation criteria, namely, effort, complexity, and performance. Within each section I present my results by scenario, starting with a table of dependent variables. The table columns are for the requirements change categories, plus a summary column for all requirements changes in that scenario. Because a requirements change can fall into more than one category, the All column will not equal the sum of the category columns.

To present a better visualization of the case study results, there is a graphical representation comparing the two approaches. The graphs are also organized by requirements category. For all the metrics, lower numbers are better. The bars represent the percentage of Bullpen's results as compared to hand-coding. In those cases where the results for hand-coding are less than those for Bullpen the bar is truncated to one-hundred percent.

6.1 Effort

There are two metrics used in effort: source lines of code (SLOC) and implementation time. SLOC is a common measure of effort. Because not all the code is written in the same language this assessment also includes the implementation time, which embodies the time it took to do detailed design, make the code modifications and conduct unit test on the code. Both these metrics are useful in presenting a picture of the effort expended. Time is the most relevant, but SLOC is more objective.

Fly-off Scenario

I have left out of this presentation the results from the Backward-chaining Bullpen and Procedural Bullpen. By the third requirements change I determined that Procedural Bullpen was not expressive enough. The implementation size of one modification was three times that of hand-coding. It was obvious that this method would not prove to be the best so I stopped further evaluation of it early in my study.

Backward-Chaining Bullpen required about ten percent more effort than Forward-Chaining Bullpen throughout this series of case studies. This was due to the overhead involved with transferring the compensation decision to the reconfiguration logic. The results were not interesting beyond that conclusion. Because Backward-chaining was not part of the other scenarios I omit the results to make the data presentation more consistent.

Table 6.1 and Figure 6.1 show the effort results for the Fly-off scenario. In all categories, Bullpen required less effort than hand-coding. In the Conditions and Initial categories the difference was small.

The categories in which Bullpen required the least effort in comparison to hand-coding were Virtual Configuration and System Configuration. Some of the virtual configuration requirements changes involved virtual entity grouping. In these cases, Bullpen took much less effort than hand-coding. The system configuration requirements changes tended to be more difficult to handle with Bullpen's abstract interfaces and during this scenario I was the least skilled with CLIPS programming. Therefore, the time per SLOC was higher in this category than the others. Overall, Bullpen handled changes to the two configurations very well.

Table 6.1: Fly-off Scenario Effort

Category		VC	SC	RP	RI	C	Initial	AI	All
Number		3	7	9	3	1	1	10	16
Size	Bullpen	72	164	937	365	71	1163	561	2243
	H-C	262	698	2012	436	86	1354	1424	3690
Time	Bullpen	1.1	5.4	17.5	9.6	2.0	27.5	8.9	71.0
	H-C	4.7	12.0	39.9	2.1	2.3	33.1	20.1	127.0

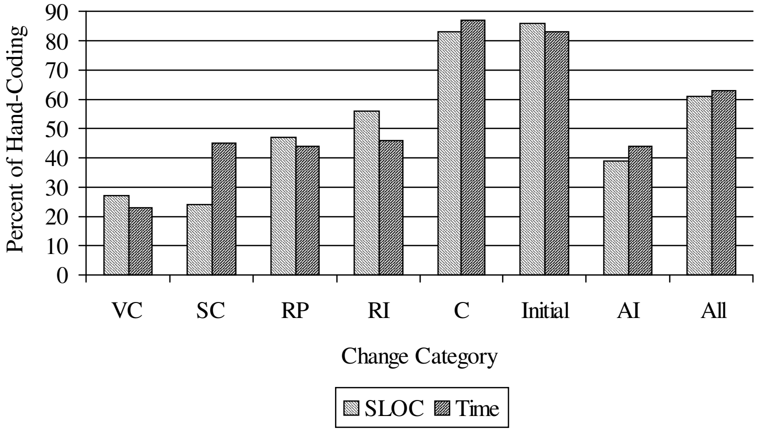

Figure 6.1: Fly-off Scenario Effort Graph

Handling reconfiguration policy changes with Bullpen required less than half the time of hand-coding. Requirements changes that involved the conditions and reconfiguration interface also changed the reconfiguration policy. Since Bullpen took more effort to implement the former two types of changes, the Reconfiguration Policy category effort is a little higher than it would be with more pure requirements changes.

The Abstract Interface category collects the results from all the most important categories. The requirements changes that contribute to the Abstract Interface category are those that are driven by the users, and by prototyping. The effort expended using Bullpen was well under half of that spent using hand-coding.

The effort required to make code modifications in response to changes in the reconfiguration interface was lower than I expected. I expected to see results much closer to the condition category results. Two factors contributed to this outcome. First, Bullpen's architecture isolated the code changes very well. The second is that the changes in the Reconfiguration Interface category also required reconfiguration policy changes that were made through Bullpen's abstract interfaces.

The Initial category changes included adapting the Bullpen framework to a new reconfiguration interface. The effort required to modify Bullpen to accommodate changes in the reconfiguration interface is high in proportion to the other categories, hence Bullpen's effort was closer to hand-coding in initial implementation. The advantages of Bullpen's abstract interfaces were still enough to make the initial implementation easier in Bullpen.

The results from this scenario showed that Bullpen was effective in reducing the effort necessary to implement the types of requirements changes driven by user demands and prototyping. It is not as effective in those requirements that are produced by changes to the DVE architecture, or a redirection of the purpose of the compensating reconfiguration.

HLA Scenario

Table 6.2 and Figure 6.2 show the results for effort in the HLA scenario. The HLA scenario is based on a real DVE project that did not have any changes in the Conditions or Reconfiguration Interface categories. The reconfiguration policy was simpler than either of the other scenarios, but the number of entities was much higher.

Table 6.2: HLA Scenario Effort

Category Number		VC 6	SC 7	RP 11	Initial 1	AI 14	All 21
Size	Bullpen	986	516	676	1108	905	2873
	H-C	1006	646	1890	2130	2259	5256
Time	Bullpen	3.9	2.6	15.2	19.9	14.3	39.5
	H-C	9.0	8.5	32.8	30.1	34.9	78.3

The most interesting result was the large code modification necessary using Bullpen to make requirements changes in the Virtual Configuration and System Configuration categories. Also note that the implementation time per SLOC is significantly less in those two categories. The reason is that in Bullpen many of the changes to the virtual world and system configuration are made in files that describe the relationship between objects. Because this scenario had large numbers of entities and VEs, most of the modifications in these requirements changes was “cut and paste” work. Cut and paste work with declarative files is significantly simpler and faster than working with source code.

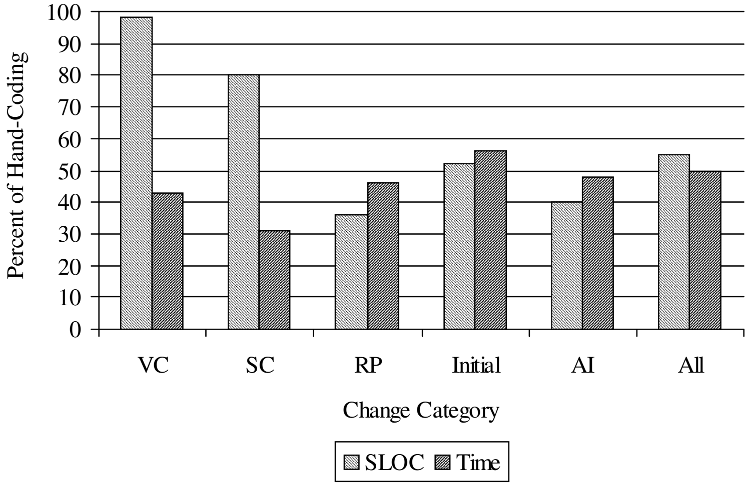

Figure 6.2: HLA Scenario Effort Graph

The other interesting result from this scenario was the effort expended in the initial creation of the compensating reconfiguration. Based on the first scenario, I improved the Bullpen framework from which initial implementation starts. Using the improved Bullpen framework to create an initial compensating reconfiguration component was significantly easier than from the old framework or hand-coding.

Overall, in a scenario with simpler changes, but on a much larger scale Bullpen took about one half the effort of manual programming. The abstract interface for Bullpen's internal database paid large effort dividends in this scenario. Keeping the logic concerning virtual entities and the mapping between the entities and the VEs separated from the storage of the entity and VE characteristics made implementing changes much easier. Modifying a listing of entities is easy and fast when it is not embedded in the logic applied to manipulate them.

Capstone Scenario

The Capstone scenario includes all categories of requirements changes. In these requirements changes I wanted to further explore the areas where Bullpen had required more effort in relation to hand-coding. I did not address those areas were I knew Bullpen was superior to hand-coding from previous scenarios. Therefore, Bullpen does not compare as well to hand-coding in this scenarios as in the other two scenarios.

Table 6.3: Capstone Scenario Effort Table

Category		VC	SC	RP	RI	C	Initial	AI	All
Number		2	3	8	3	2	1	7	13
Size	Bullpen	40	126	781	335	579	699	188	1851
	H-C	53	296	1134	307	402	1528	659	2896
Time	Bullpen	0.5	2.6	16.4	4.5	9.7	14.0	5.7	33.9
	H-C	0.8	5.1	23.9	6.3	6.9	29.6	14.3	57.1

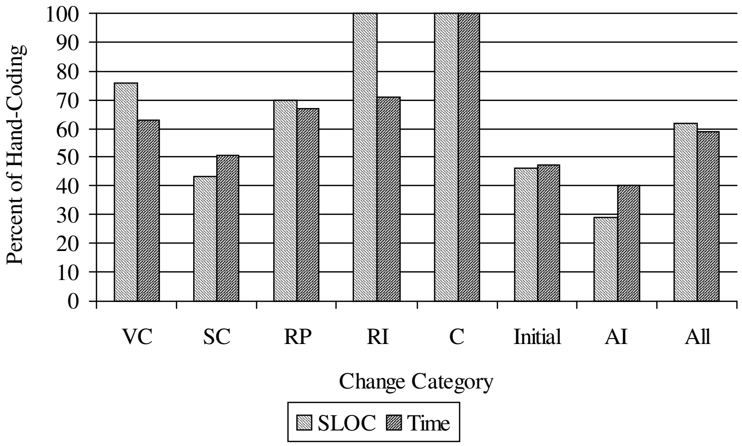

Figure 6.3: Capstone Scenario Effort Graph

Both Virtual Configuration requirements changes were very simple. For Bullpen to differentiate itself from manual programming requires larger and more complex changes. Bullpen still provided a slight advantage, but it is not as pronounced as the two previous scenarios.

The effort expended to implement changes to the reconfiguration interface in Bullpen was more than I found in the Fly-off scenario case studies. These requirements changes better isolated the modifications in the reconfiguration interface from alterations in the reconfiguration policy, confirming my understanding of the Fly-off results.

The Reconfiguration Policy category requirements changes showed higher effort than the two previous scenarios. In both the case studies that changed the conditions, the reconfiguration policy was also necessarily changed. Since Bullpen requires much more effort to implement condition changes than hand-coding the reconfiguration policy results were higher than they would be otherwise.

The Initial category effort continued to decrease in proportion to hand-coding. Again, I attribute this to Bullpen framework enhancements. These enhancements were much less extensive than after the fly-off scenario, but still realized a slight improvement.

In the Abstract Interface category requirements changes, Bullpen performed the best of all three scenarios. Because of the nature of the requirements changes in this scenario I did not expect this, and attribute these results to my improvement as rule-based programmer. In the Abstract Interface requirements changes most of the modifications were made in the CLIPS interface to Coach. By this scenario I was writing CLIPS rules more efficiently and faster. Although the requirements changes are more complex, the results were better using Bullpen.

Effort Summary

Overall many more requirements changes dealt with the type generated by users and during prototyping. I did explore the types of changes that would occur less frequently in DVEs, i.e., reconfiguration interface, conditions handled changes, and initial implementation but not to the extent of the others.

Table 6.4 and Figures 6.4 and 6.5 show the effort results for the entire case study. This case study includes a wide range of requirements changes in type and size. The effort necessary to use Bullpen decreased through out the scenarios as I improved it and became more skilled at its use. The only category where the change was drastic was the Initial category, which would show better overall results if the improvements had been in place from the start of the case study.

Table 6.4: Case Study Effort

Category		VC	SC	RP	RI	C	Initial	AI	All
Number		11	17	28	6	3	3	31	50
Size	Bullpen	1098	806	2404	730	650	2970	1654	6917
	H-C	1321	1640	5036	1013	488	5012	4342	11842
Time	Bullpen	5.5	10.6	50.9	14.1	11.7	61.2	28.9	122.8
	H-C	14.5	25.6	96.6	27.3	9.2	97.6	64.5	214.0

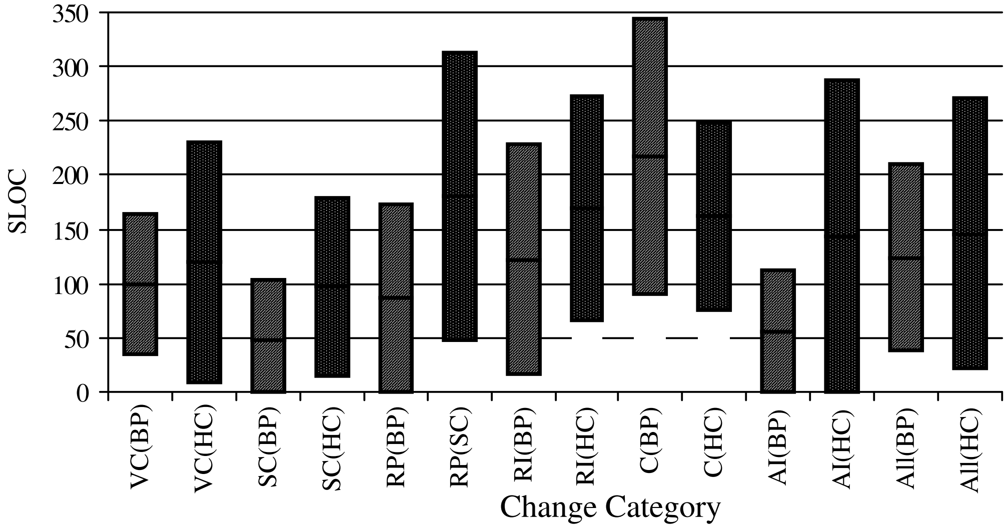

Figure 6.4: Case Study Change Size (Average and Standard Deviation Range)

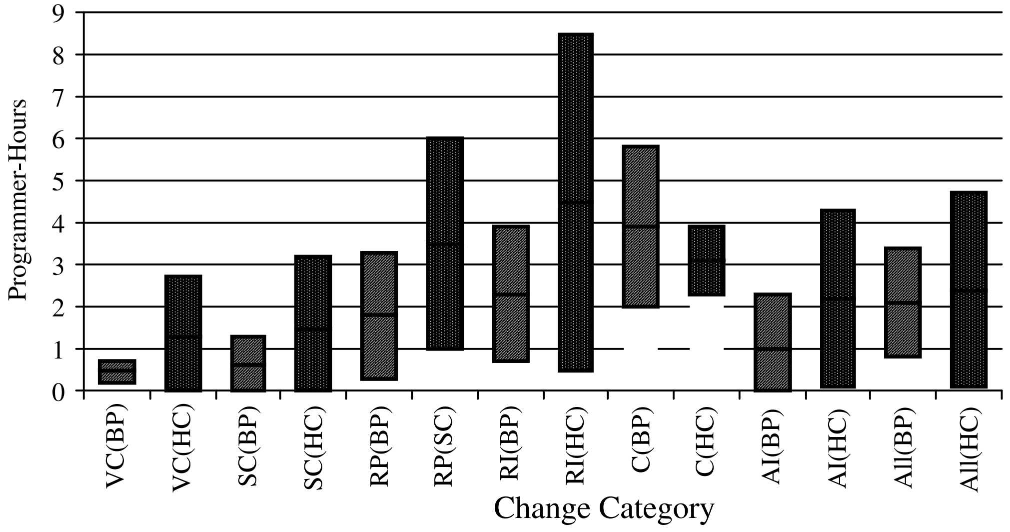

Figure 6.5: Case Study Implementation Time (Average and Standard Deviation Range)

Implementation time and SLOC correlate very well with each other. Over the course of all fifty requirements changes the proportion of time spent implementing the change in Bullpen matches the size of the change. Dividing the requirements changes by category differentiates the time per SLOC between Bullpen and hand-coding. This effect is most pronounced in the Virtual Configuration category. Most of the code changes involving the virtual world were very simple. Reconfiguration Policy changes took longer per line of code. CLIPS was used to implement most Reconfiguration Policy changes and this shows that although CLIPS is more difficult per SLOC it is also much more powerful in this domain.

Using Bullpen to implement changes to the Virtual Configuration and System Configuration, and Reconfiguration Policy required less than half the effort of hand-coding. This is due to the abstract interfaces of Bullpen which more closely resemble the thinking of the users. The Abstract Interface category focuses on changes made through Bullpen's abstract interfaces. Accordingly, these requirements changes are where Bullpen required the least effort when compared to manual programming. The Abstract Interface category is also a summary of the types of changes most likely to be encountered in the military DVE domain, and therefore is its best predictor of practical usefulness.

Initial implementation of compensating reconfiguration using Bullpen does not result in an effort penalty. In fact, it takes less effort to get compensating reconfiguration up and running using Bullpen. This contributes to the overall effort advantage of using Bullpen over hand-coding compensating reconfiguration components.

Using Bullpen to implement Reconfiguration Interface category changes still results in effort savings, but it is not as significant as in the other categories. Some of this effect derives from Bullpen's advantage in implementing the reconfiguration rules changes that naturally accompany changes to the reconfiguration interface.

Changes to the conditions handled are drastic alterations the compensating reconfiguration function. Making these changes using Bullpen requires more effort than hand-coding. This is because the supporting code for Bullpen's abstract interfaces must be altered to handle the new condition. The interfaces that are so helpful when handling other categories of change are a hindrance in these cases.

6.2 Complexity

To compare the complexity of using Bullpen compared with manual programming I used three metrics. The three metrics are the count of locations modified, the number of defects, and the defect repair time. Location count is a direct measure of complexity. In a general sense, the more dispersed the modifications in the code, the more difficult it is to understand the modification and its impact on the existing code. The other two dependent variables are an indirect measure of complexity. The more complex a task is, the higher the likelihood of errors, and thus defects. The greater the complexity of the code, the more difficult it will be to identify errors and repair defects. Over all fifty requirements changes, the correlation between the three metrics was close, and was representative of my intuition.

Fly-off Scenario

The Fly-off scenario requirements changes are more involved than the HLA scenario but less so than the Capstone scenario. In many of these requirements the compensating reconfiguration logic must use all the different types of information available.

Table 6.5: Fly-off Scenario Complexity

Category Number		VC 3	SC 7	RP 9	RI 3	C 1	Initial 1	AI 10	All 16
Locations	Bullpen	8	16	42	19	10	33	26	94
	H-C	21	87	156	78	8	59	86	245
Defects	Bullpen	2	4	9	4	1	15	7	28
	H-C	6	14	32	13	0	25	23	63
Repair Time	Bullpen	0.2	1.2	3.0	2.4	0.2	2.8	2.0	8.1
	H-C	1.4	3.4	8.4	3.7	0	3.7	5.5	13.6

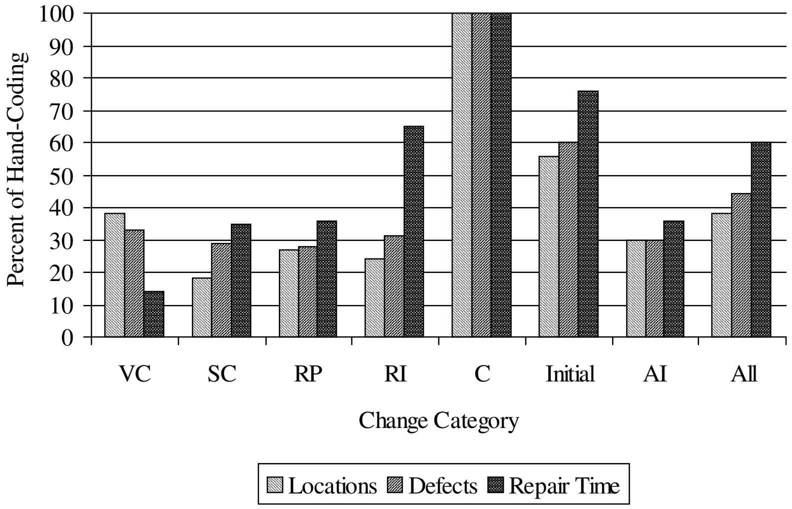

Figure 6.6: Fly-off Scenario Complexity Graph

The complexity results resemble the effort results. Changes that were categorized as changes to the Virtual Configuration, System Configuration, and Reconfiguration Policy were the least complex to realize with Bullpen when compared to hand-coding. Condition changes were more complex in Bullpen than hand-coding and the complexity reduction from using Bullpen was not as great in Initial implementation as it was in the other categories.

Code modifications that were stimulated by reconfiguration interface changes were less complex than hand-coding but with an important caveat. Bullpen produced fewer defects. However, those defects were more difficult to repair. The code modifications were more closely grouped, but had more far reaching side-effects. This lead to some very difficult defect repair, which is not a desirable characteristic.

The complexity comparison in the Abstract Interface category showed that using Bullpen was superior in all the dependent variables. Using Bullpen resulted in modifying less than one-third of the locations that manual programming did. In addition, using Bullpen resulted in less than one-third the defects, and the repair time for those defects was significantly less than hand-coding. However, repair time per defect was higher for Bullpen than manual coding. This is the only scenario in which this occurred. I attribute this to my lack of familiarity with using Bullpen. By the later scenarios I was more familiar with Bullpen and apparently better able to repair defects.

HLA Scenario

The goal of the HLA scenario was to exercise Bullpen in a realistic environment. This environment is less elaborate than the other two scenarios, so the full power of Bullpen was not fully exploited during this scenario. There were no requirements changes in the Reconfiguration Interface or Conditions categories; categories where Bullpen has not shown an decided advantage.

As in the Fly-off scenario, Bullpen proved to be less complex than using hand-coding. The repair time per defect for Bullpen when compared to hand-coding was significantly less in this series of requirements changes. Many of the modifications in Bullpen were “cut and paste” and many the errors were clerical in nature. This made them relatively easy to identify and fix.

Table 6.6: HLA Scenario Complexity

Categories Number		VC 6	SC 7	RP 11	Initial 1	AI 14	ALL 21
Locations	Bullpen	14	21	32	43	32	95
	HLL	42	64	87	182	103	334
Defects	Bullpen	3	6	11	3	11	20
	HLL	13	16	29	8	24	53
Repair	Bullpen	0.8	1.9	2.5	1.2	2.7	5.3
Time	HLL	4.4	3.0	7.9	13.0	5.8	25.8

The only category where this was not the case was in System Configuration changes. This is because system configuration changes were among the simplest changes made during this scenario. One defect that had an abnormally high repair time has skewed the data. That defect should have been found during the initial creation of the component. As is many times the case, it was not and it was more difficult to diagnose and repair further along in the product lifecycle.

Proportionally the complexity of implementing compensating reconfiguration using Bullpen remained the same as the Fly-off scenario. This shows that Bullpen presents a simpler interface for solving simple problems and that as the complexity of the problem increases, Bullpen's interface also gets more complex, but only in direct relation to the problem being solved.

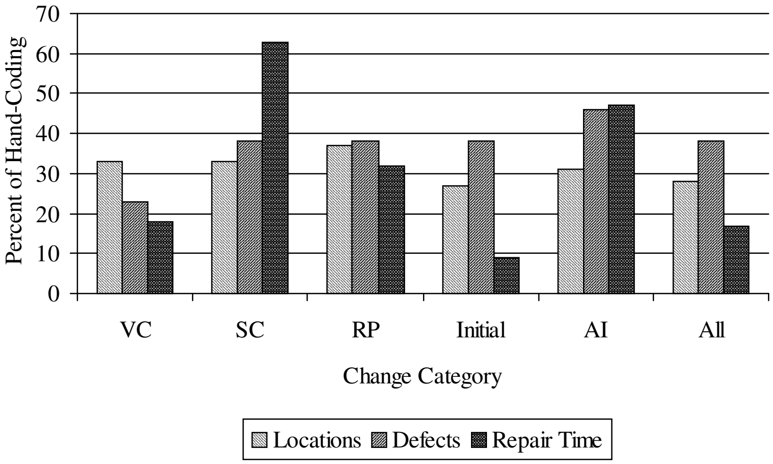

Figure 6.7: HLA Scenario Complexity Graph

Capstone Scenario

The capstone scenario involves many requirements changes that test Bullpen specifically where it does not do well. Consequently, the complexity results are the highest of the three scenarios, especially when the initial implementation is removed from the comparison.

Virtual Configuration changes resulted in code modifications that were almost as spread out as in hand-coding. Because these modifications were mostly to lists of entities and their relationships, the modifications were not complex according to the other measures. In fact the repair time was the least of all the categories. However, the implementation time was significantly less, which along with the low defect rate and repair time lends credence to the assertion that the code modifications were not as complex.

Table 6.7: Capstone Scenario Complexity

Category Number		VC 2	SC 3	RP 8	RI 3	C 1	Initial 1	AI 7	All 13
Locations	Bullpen	8	13	68	32	46	58	27	163
	H-C	13	53	114	30	40	189	73	341
Defects	Bullpen	1	3	11	3	4	1	7	15
	H-C	3	6	13	4	2	2	14	22
Repair Time	Bullpen	0.4	0.9	4.5	1.2	1.8	0.4	2.8	6.2
	H-C	3.4	4.5	6.0	3.1	0.5	1.7	8.9	14.2

The Reconfiguration Policy has higher complexity results, when compared to hand-coding in this scenario than in any others. The reconfiguration policy in many of these requirements changes is quite involved, with higher defect rates for both methods. In addition, both Condition category requirements changes also involved changes to the reconfiguration policy. Just as with the Capstone effort results, the Reconfiguration Policy complexity is higher than it would be if it the conditions handling modifications were excluded from that category. Since most of these requirements changes were aimed at Bullpen's weaknesses, its defects and repair time increased more for this scenario than did hand-coding.

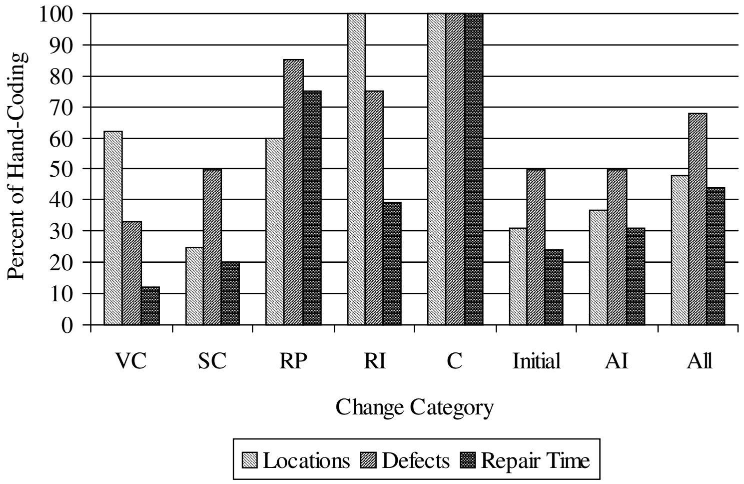

Figure 6.8: Capstone Scenario Complexity Graph

The Abstract Interface category had generally low complexity figures for Bullpen. This shows that the proportional increase in complexity was in the changes that were made without the abstract interfaces. This is confirmed by looking at the Reconfiguration Interface and Conditions category, which shows that Bullpen was as or more complex to use for changes in these categories.

On the average, using Bullpen still proved to be less complex than hand-coding; notwithstanding the focus of this scenario on the weak points of Bullpen. However, there is a complexity penalty that the Bullpen user pays when dealing with changes in the Reconfiguration Interface or Conditions categories.

Complexity Summary

Combining all fifty requirements changes presents a good picture of the complexity of Bullpen over a wide range of uses. The requirements changes ranged from large to small and covered all types of change in the taxonomy. There are fewer requirements changes in the Reconfiguration Interface and Conditions categories. In normal use, these two categories would change much less frequently than the others. The Abstract Interface category is the bellwether category, since it represents the most common use for Bullpen in the military DVE domain.

Table 6.8: Case Study Complexity

Category		VC	SC	RP	RI	C	Initial	AI	All
Number		11	17	28	6	3	3	31	50
Locations	Bullpen	30	50	142	51	56	134	85	352
	HLL	76	204	357	108	48	430	262	920
Defects	Bullpen	10	13	31	7	5	19	25	63
	HLL	22	36	74	17	2	35	61	138
Repair Time	Bullpen	1.4	4.0	10.0	3.6	2.0	4.4	7.5	19.6
	HLL	9.2	10.9	22.3	6.8	0.5	18.4	20.2	53.6

Bullpen is less than half as complex to use as hand-coding according to these results. In all common categories of requirements change the dispersion of the code modifications, the number of defects and repair time is about forty percent of hand-coding. The initial implementation of a compensating reconfiguration component is also half as complex as hand-coding. During the normal lifecycle of a DVE the users can expect their work to be substantially less complex when using Bullpen.

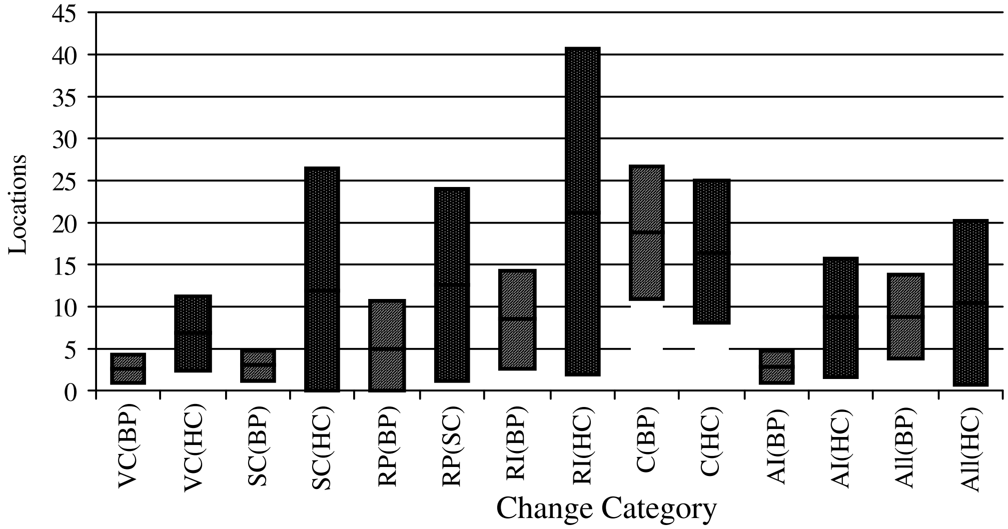

Figure 6.9: Case Study Locations (Average and Standard Deviation Range)

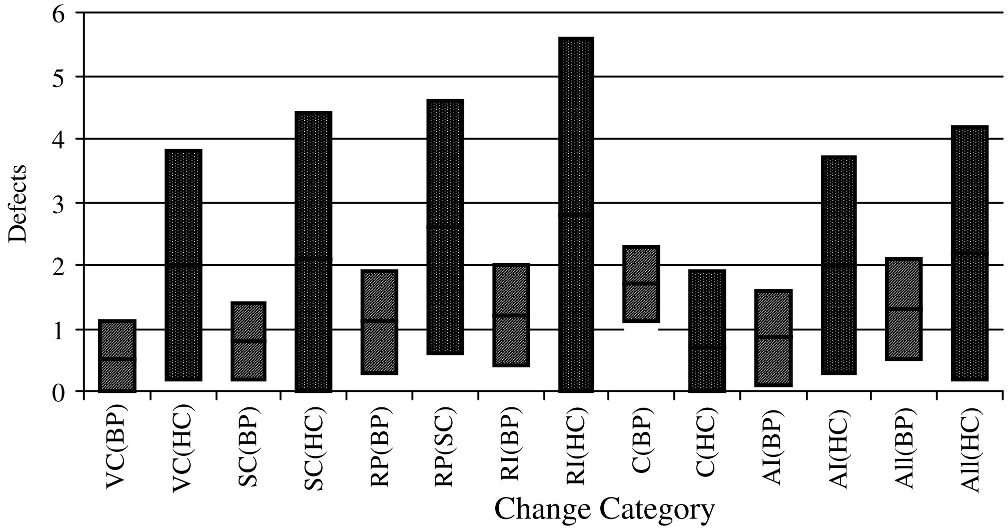

Figure 6.10: Case Study Defects (Average and Standard Deviation Range)

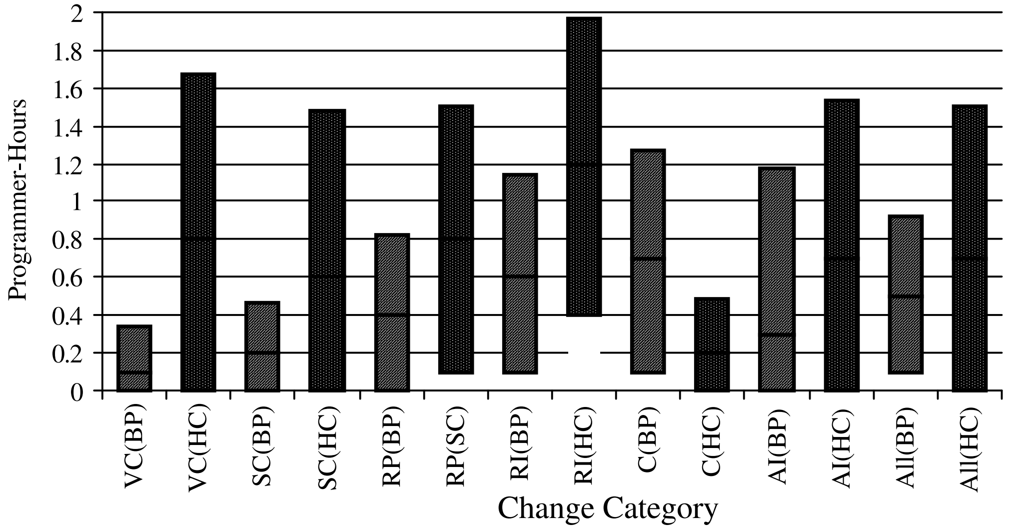

Figure 6.11: Case Study Defect Repair Time (Average and Standard Deviation Range)

Reconfiguration Interface category changes were simpler using Bullpen, and since much of the code modification does not exploit the Bullpen's abstract interfaces this is a very encouraging result. Changes to the reconfiguration interface required some amount of modification of the CLIPS rules that govern reconfiguration. The advantage of CLIPS rules over high-level source code contributes to this result.

One final way that I look at the complexity Bullpen is by analyzing the defect reports from the entire case study. Table 6.9 shows a simple statistical analysis of the defect reports from all the case studies. In general, defects in the hand-coded versions required larger repair that was spread over more locations. These repairs also took longer. Additionally, hand-coding produced the most difficult defects to handle as the interaction between different parts of the compensating reconfiguration program was less well controlled.

Table 6.9: Defect Analysis

		Repair Size	Repair Time	Repair Locations
Bullpen	Average	3.516	0.290	1.322
	Median	2	0.2	1
	Std Dev	4.202	0.213	0.747
	Max	23	1.0	4
Hand-Coded	Average	6.923	0.425	1.758
	Median	3	0.2	1
	Std Dev	11.458	1.086	1.734
	Max	84	10.4	12

6.3 Performance

Performance in this research has three components: reliability, expressiveness, and reaction time. Because I could not test any of the software in actual operation, I use the defects from integration test to predict the reliability of the software. Expressiveness is manifested in the effort and complexity results. Reaction is that time which can be attributed to the implementation version. For the advantages Bullpen shows in effort and complexity, the performance penalty is not large enough to render it impractical.

Reliability

Using a tool should not result in a reliability penalty, and should lead to products that are more reliable. To determine the reliability of Bullpen I examined two dependent variables, defects and incorrect reconfigurations. Defects include all defects discovered during integration testing. Incorrect reconfigurations are the most serious of those defects from the user's point of view. Incorrect reconfigurations are defects that leave the DVE in an inconsistent state by Definition 3.1.

Fly-off Scenario

When dealing with defects, especially incorrect reconfigurations, the numbers get smaller. This can skew the proportions somewhat. For example, in the conditions category Bullpen had one hundred percent of the incorrect reconfigurations that hand-coding produced. However, Bullpen produced code did not result in any incorrect reconfigurations, but neither did manually produced code. Table 6.10 and Figure 6.12 show the number of defects and count of the most serious defects identified during integration test.

Table 6.10: Fly-off Scenario Reliability

Category		VC	SC	RP	RI	C	Initial	AI	All
Number		3	7	9	3	1	1	10	16
Defects	Bullpen	2	4	9	2	3	15	7	28
	H-C	6	14	32	13	0	25	23	63
Incorrect Reconfigurations	Bullpen	1	1	1	1	0	3	1	5
	H-C	2	5	13	5	0	7	10	23

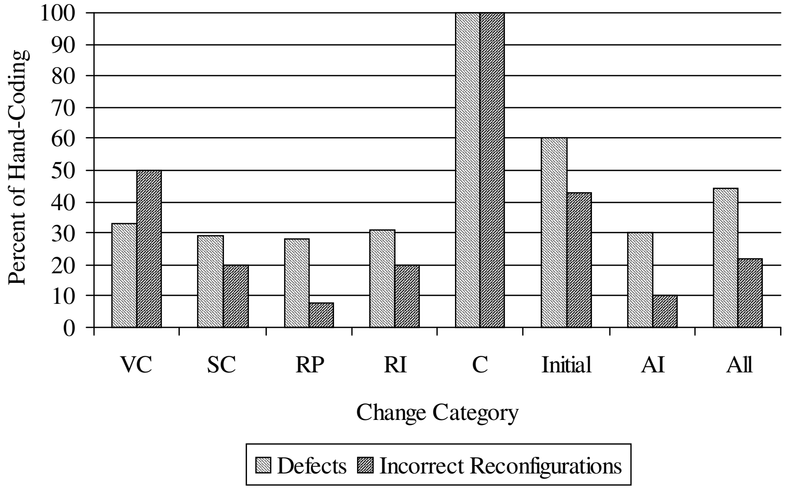

Figure 6.12: Fly-off Scenario Reliability Graph

Overall Bullpen followed the pattern set by effort and complexity. The categories were Bullpen code is significantly more reliable than hand-coding include Virtual Configuration and System Configuration and Reconfiguration Policy. Changes in the Reconfiguration Interface category made with Bullpen were more reliable than hand-coding in the same proportions as the other categories. The reasons are the same for reliability as complexity discussed in Section 6.2.

The defects that Bullpen produced included proportionally fewer of the most serious defects. Bullpen's abstract interface makes it easier to reason about the reconfigurations, leading to fewer incorrect reconfigurations, however, changes to the virtual configuration did not follow this pattern. There were proportionally more incorrect reconfigurations produced by Bullpen in these requirements changes. This is an anomaly, over all scenarios virtual configuration case studies had almost equal ratios for defects and incorrect reconfigurations.

HLA Scenario

The HLA scenario because it was based on a real DVE project was less involved than the others. As a result the requirements changes tend to be simpler than in the other scenarios. All the requirements changes were of the type that were user-driven or prototyping driven.

Table 6.11: HLA Scenario Reliability

Categories Number		VC 6	SC 7	RP 11	Initial 1	AI 14	All 21
Defects	Bullpen	3	6	11	3	11	20
	H-C	13	16	29	8	24	53
Incorrect Reconfigurations	Bullpen	2	2	1	0	2	4
	H-C	8	10	11	2	11	23

The proportion of all defects discovered in the Bullpen code was less than half of the hand-coded version. The percentage of the most severe defects–incorrect reconfigurations–was less than one quarter of the defects found in the hand-coding. I attribute this to two causes. First, Bullpen presents an interface that makes it easier to reason about reconfiguration actions. The second is that many times a program abort due to a null pointer or another coding error of that ilk would cause an incorrect reconfiguration. The compensating reconfiguration code would be partially through a reconfiguration when it would terminate, leaving the DVE in an inconsistent state. Because bullpen is a more mature tool, it tended to have significantly fewer of these types of defects.

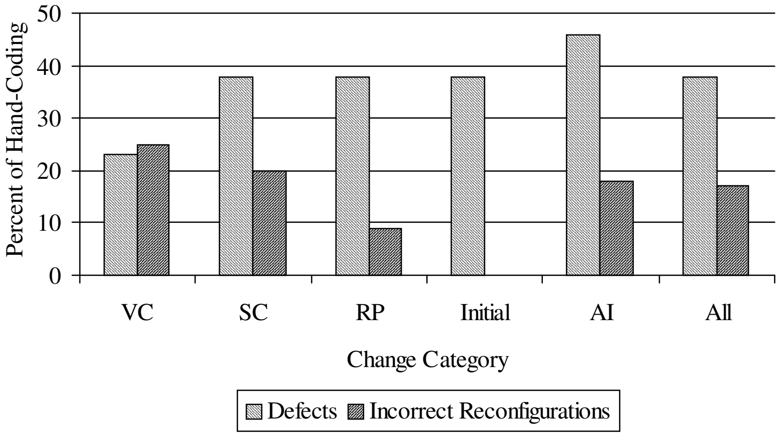

Figure 6.13: HLA Scenario Reliability Graph

Capstone Scenario

The capstone scenario requirements changes fall into all categories. Because the requirements changes were aimed more at Bullpen's weaknesses the overall ratio of defects in Bullpen produced components is higher than the other scenarios.

For reliability and effort the initial implementation results improved with each scenario as Bullpen improved as a tool. This did not appear to happen in the Capstone scenario. The ratio is misleading. The reality is that by this point Bullpen had significantly improved. There were fifteen defects found during the initial implementation of the Fly-off scenario and three in the HLA scenario. There was only one found during initial implementation in this scenario. As I gained experience with creating hand-coded compensating reconfiguration components the defect rate also dropped even more dramatically. The number of defects dropped from twenty-five to eight to two. Each method produced one incorrect reconfiguration.

Table 6.12: Capstone Scenario Reliability

Category Number		VC 2	SC 3	RP 8	RI 3	C 2	Initial 1	AI 7	All 13
Defects	Bullpen	1	3	11	3	4	1	7	15
	H-C	3	6	13	4	2	2	14	22
Incorrect Reconfigurations	Bullpen	0	2	5	2	1	1	3	7
	H-C	2	2	6	2	1	1	7	11

The most interesting aspect of the Capstone scenario is that overall a smaller proportion of Bullpen's defects were incorrect reconfigurations. However, in this scenario, the proportion of the most serious kind was almost equal to all defects. At this point few defects are not incorrect reconfigurations. Most of the annoying little problems have been removed from both sets of code, and my understanding of side-effects and the interactions of the components in the DVE is better. Since these requirements changes are more difficult to implement in Bullpen than the other scenarios, there tends to be more incorrect reconfigurations.

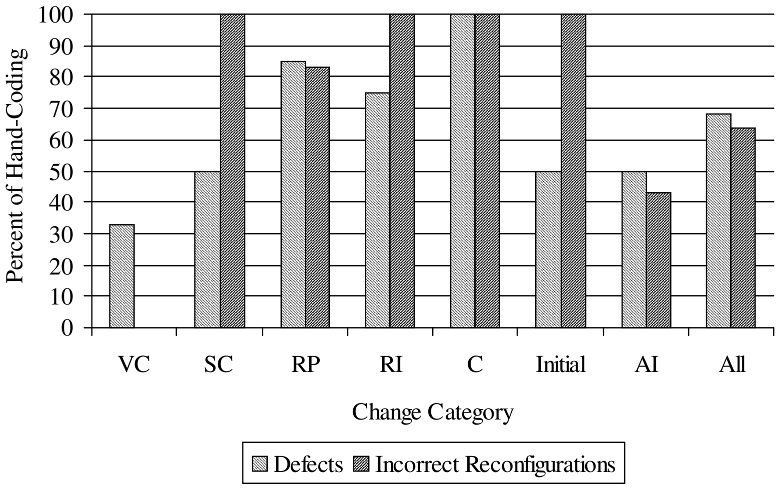

Figure 6.14: Capstone Scenario Reliability Graph

Reliability Summary

When looking at reliability from the perspective of all three scenarios, Bullpen's strengths are the same as they were for effort and complexity. In general Bullpen produced code had less than half the defects and about one quarter of the incorrect reconfigurations of hand-coded components.

Table 6.13: Case Study Reliability

Category Number		VC 11	SC 17	RP 28	RI 6	C 3	Initial 3	AI 31	All 50
Defects	Bullpen	10	13	31	7	5	19	25	63
	H-C	22	36	74	17	2	35	61	138
Incorrect Reconfigurations	Bullpen	3	5	7	3	1	4	6	16
	H-C	12	17	30	7	1	10	28	57

The exception is in changes to conditions handled. Bullpen is probably more prone to errors for Conditions category changes because the changes are larger and more complex than in a hand-coded version. Overall changes to the reconfiguration interface and the initial implementation standup well against hand-coding. Although the abstract interfaces cannot be used for all the modifications in these case studies, the design of Bullpen isolates changes enough to make them less error prone.

Some of this advantage stems from the fact that Bullpen is a more mature piece of software than the hand-coded versions created for each scenario. Although each hand-coded component started from scratch for each scenario, my experience helped me to improve the design for each successive scenario somewhat mitigating this effect. Regardless, more mature code is an advantage of using a tool, such as Bullpen.

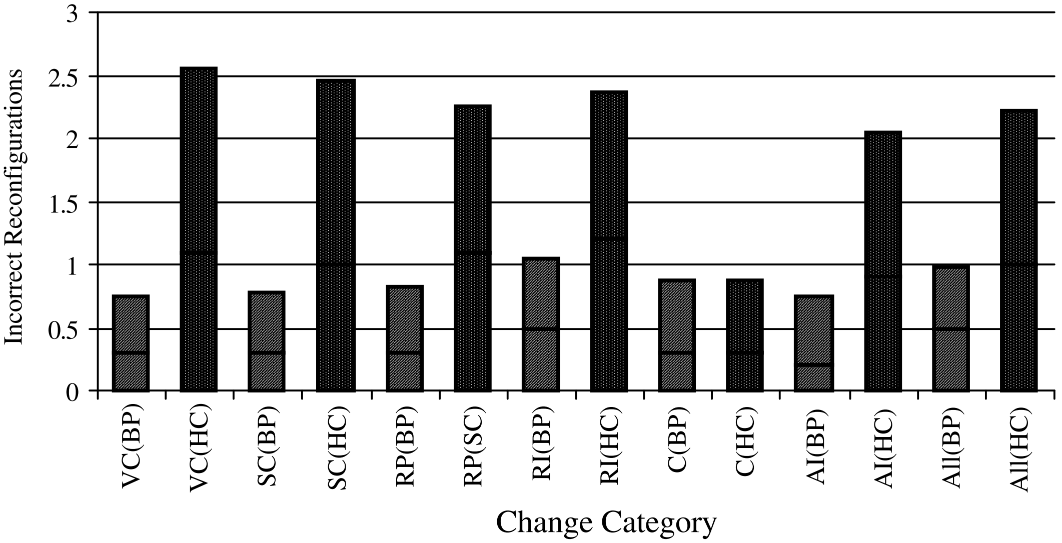

Figure 6.15: Case Study Incorrect Reconfigurations (Average and Standard Deviation Range)

The other and I believe dominant cause is that Bullpen is easier to use and reason about than high-level source code. A number of defects were the result of my forgetting about an interaction case while hand-coding that was automatically taken care of by Bullpen. Typically, these defect repairs were large and time consuming.

As with effort and complexity, Bullpen was more reliable in the categories of change that are user-driven and prototyping driven. In effort and complexity, the results of initial construction were only slightly worse than these categories. As far as reliability goes, initial construction was decidedly less reliable than the user and prototyping-driven categories. Because initial construction is a combination of all categories of requirements change and was not done completely through the Bullpen abstract interfaces, this is not surprising.

Expressiveness

I address expressiveness indirectly. If Bullpen was not expressive enough for the DVE domain then it would show in one of two ways. In the absolute worse case, I would not be able to implement compensating reconfiguration requirements. I did not encounter any cases like this during the case study. I used the HLA scenario as a reality check to insure that I was not avoiding the types of compensating reconfiguration requirements that are necessary, but impossible or difficult for Bullpen. In fact, the other two scenarios more fully exercised the expressiveness of the interface than the HLA scenario.

If the interfaces were lacking in expressiveness to a smaller degree, that would manifest itself in the effort metrics. A programmer would require more SLOC to implement a requirements change than with a more expressive environment. The Procedural Bullpen from the Fly-off scenario is a perfect example. The interface was not expressive enough and consequently the size of the change was three times that of hand-coding for one case study. Bullpen has been shown to require far fewer SLOC to implement requirements changes than manual programming, therefore Bullpen is expressive enough for the military DVE domain.

Reaction Time

Reaction time is also an important aspect of performance. Abstract interfaces generally impose an execution speed penalty as a cost of an easier to understand interface. For Bullpen to be a worthwhile tool, the reaction time penalty must be within acceptable limits.

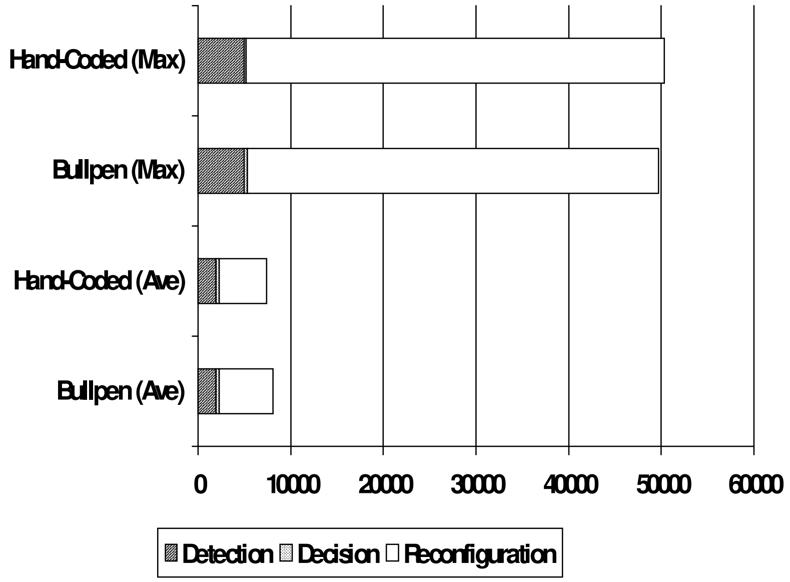

Figure 6.16: Average and Worst-Case Reaction Time (Milliseconds)

Reaction time varied greatly and is non-deterministic because of the nature of distributed programs. It is dependent on the hosts and VEs involved as well as the reconfiguration decisions. Changing the maximum wait for an acknowledgment can have a huge effect on the reaction time in cases where numerous VEs must respond to queries from the compensating reconfiguration code.

I saw no difference between reaction times based on the type of requirements change. The size of the DVE did have an effect; the HLA scenario did have the longest reaction times. The distribution of reaction times is so wide that I will discuss reaction time from the perspective of the entire case study, rather than divided into categories and scenarios as I have other results.

Generally, Bullpen responds slower to acknowledgment messages which accounts for the twelve percent average difference in reconfiguration time. However, the variance is large and many times Bullpen reconfigured faster than the high-level language version. In the worst case total reaction time, Bullpen was actually two percent faster.

The worst case comes from the HLA scenario where the reconfiguration included starting and initializing numerous VEs before they could take ownership of orphaned entities. The decision time was not significant when compared to the total time. The difference between the hand-coded version and the Bullpen version in the worst case did not affect the total reaction time. In fact Bullpen was overall faster in this one case due to other factors in the system.

The average reaction time comes from approximately four hundred executions in the HLA scenario. Although the decision time for Bullpen software was sometimes five times slower than the hand-coding software, the difference was on the average less than one hundred milliseconds. One hundred milliseconds is considered instantaneous to human observers. [Mac94] Therefore, for short reconfigurations the difference is not noticeable. For longer reconfigurations, the difference is more pronounced, but still not significant.

6.4 Summary

I have determined that using Bullpen as opposed to hand-coding compensating reconfiguration functionality results in writing fewer lines of code. It also takes less time to do the detailed design, coding and unit test using Bullpen. When building compensating reconfiguration components, hand-coding results in changes being made in more locations than using Bullpen. The total number of defects detected during integration test was higher for the hand-coded versions than using Bullpen. The repairs were also more time consuming, both per defect and overall for the hand-coded version. Incorrect reconfigurations, which are the most serious types of defects were fewer in the Bullpen produced code, than in the hand-coded version.

Bullpen's strength lies in handling requirements changes that are made through its abstract interfaces. Those changes are part of the Virtual Configuration, System Configuration, and Reconfiguration Policy categories. Reconfiguration Interface and Conditions category changes are more difficult for Bullpen to handle. Bullpen was still superior to hand-coding when handling Reconfiguration Interface changes on the average, but the advantage was less pronounced that in the other categories. In almost every case hand-coding was superior to Bullpen when dealing with changes to the conditions handled.

During initial construction, which is an amalgamation of all the types of changes, Bullpen still requires fewer lines of code, and less time. The code modifications were concentrated in fewer locations and results in fewer defects that are easier to repair. The advantages are not as notable during the initial construction as they are when handling requirements changes, but throughout the lifecycle of the compensating reconfiguration component the advantages are significant.

Throughout the lifecycle Bullpen requires a little more than half the effort of hand-coding to implement requirements changes when using both size and time to determine effort. By using the locations touched, number of defects and defect repair time to judge complexity, using Bullpen is about one-third as complex as hand-coding. Bullpen produced compensating reconfiguration was approximately three times as reliable as the code produced by hand-coding.

The performance penalty for using Bullpen is not apparent in terms of expressiveness. Bullpen can handle all realistic requirements and more. As far as reaction time goes, Bullpen is slower than hand-coded compensating reconfiguration. The overhead is not significant in this case study, averaging twelve percent. However, the DVE builders must balance the advantages of using Bullpen with the disadvantages for the DVE in question.

7. CONCLUSIONS AND FUTURE WORK

7.1 Conclusions

The hypothesis of this research is that using a rule-based tool to build compensating reconfiguration software for DVEs is superior to traditional hand-coding techniques. A rule-based tool that automatically tracks virtual and system information is easier to use, less complex, and performs sufficiently for practical use. To demonstrate this approach I built a rule-based tool called Bullpen and conducted a case study comparing it to manual programming.

Bullpen combines a rule-based interface for the decision-making logic, with abstract representation of the virtual state and system state to create compensating reconfiguration components for DVEs. Bullpen is designed to allow the expression of requirements in ways that more closely resemble the way that users state requirements to the DVE builders.

In the military DVE environment, using Bullpen shows a significant advantage over manual programming throughout the lifecycle of a project. Bullpen is superior to hand-coding during the initial set-up of the DVE. In the kinds of requirements changes most common to the military DVE domain using Bullpen is much easier than manual programming. When handling less common types of requirements changes, Bullpen's advantages are not so much in evidence. The ease of development gained by using Bullpen is paid for by slower reaction time, but expressiveness and reliability do not suffer.

Initial Implementation

Bullpen yields significant advantages over hand-coding when building an initial version of compensating reconfiguration. About one-half the effort expended to build compensating reconfiguration by hand is all that Bullpen requires to get a component running. The interface used by the DVE builder is less complex than the high-level source code used in hand-coding. The builder must touch less than one-third of the locations with Bullpen than he must using hand-coding techniques. Using Bullpen results in fewer defects that are easier to fix. Due to the less complex interface, the code produced by Bullpen has a lower defect rate, especially when the most serious defects are taken into account.

Most Common Requirements Changes

Bullpen truly differentiates itself from hand-coding when specific types of requirements changes are introduced to the project. These are the types of changes that are most likely to be found in the military DVE domain. In this domain it is common for the virtual world to change numerous times throughout the development of the project. It is also common for the resources available to change, but not as much as the virtual world. DVE builders may also want to prototype the allocation of resources and reconfiguration policy. Prototyping results in changes to the system configuration and reconfiguration policy requirements of the project.

In these three categories of requirements change, Bullpen far surpassed hand-coding in ease of use and reliability. The programmer time required to implement these types of changes in Bullpen was less than half the time it took to hand-code. The interface is easier to use. The alterations to the code were grouped closer together and as an indirect measure of complexity, the code had fewer defects that were easier to fix. Bullpen produced code contained about one quarter of the most serious defects that the hand-coded versions did.

Reconfiguration Interface

Bullpen is designed to handle most efficiently the most frequent categories of change in military DVEs. Bullpen includes design decisions that achieve these goals at the expense of handling changes in the reconfiguration interface and the conditions handled. Bullpen's ability to handle changes to the reconfiguration interface did not suffer nearly as much as did its ability to adapt to requirements changes in the conditions handled.

In case studies where requirements changes were the product of reconfiguration interface changes Bullpen shows a some advantages over hand-coding. It requires about two-thirds of the effort that hand-coding does. The complexity, as determined by these metrics, is about half of what is found when making changes to hand-coded versions. However, the defects that did exist in the Bullpen version were more insidious than those in the manually built component.

Conditions Handled

The types of requirements change in which using Bullpen is a definite disadvantage is conditions handled. By most conditions requirements changes hand-coding was easier and took less time. Hand-coding also resulted in fewer defects that were easier to diagnose and mend. In normal use, it would be rare for major changes such as adding a new condition to occur. In projects where which conditions to handle is in question at the start, hand-coding may be a better choice than Bullpen. When requirements changes concerning condition handling are a significant portion of all requirements changes then using Bullpen is not advantageous.

Interconnection Infrastructure Service

Bullpen can create components that participate in the DVE as a support utility, or Bullpen can be integrated into the interconnection infrastructure providing compensating reconfiguration as a service to the DVE. If it is part of the interconnection infrastructure, then it must be able to provide a full range of service without recompiling the interconnection infrastructure. Newer programming languages such as Java™ allow dynamic class loading which can help avoid recompilation for some types of changes. Bullpen provides an interpreted interface for the reconfiguration rules, virtual and system state, the relationship between entities and the use of abstract virtual states. Using these interfaces, Bullpen can handle a wide array of requirements changes without recompiling. This makes it uniquely suited for incorporation into the interconnection infrastructure.

When handling requirements changes in which Bullpen did not require recompiling, it truly differentiated itself from hand-coding. It took less than half the effort that manual programming did to create equivalent functionality. The complexity of the changes was about one-third of hand-coding. The code produced was only one-fifth as likely to have the most serious defects in it.

Bullpen provides abstract interfaces that make producing compensating reconfiguration simpler and easier. When these interfaces are used to fabricate changes to compensating reconfiguration, Bullpen is superior to hand-coding. When requirements force alterations to Bullpen outside of these interfaces, it does not compare as well against hand-coding. Based on the normal use for the military DVE domain, DVE builders who use Bullpen will expend less effort, make simpler modifications, and create code that is more reliable. The advantages do not come with a grave reaction or expressiveness penalty.

7.2 Future Work

DVE Related

This case study makes the case for further investigation of Bullpen. The next step to proving its worth and gaining its acceptance in the military DVE community would be a trial during an actual DVE execution. This requires modifying the RTI to support compensating reconfiguration. I hope to get support for this work from the Defense Modeling and Simulation Office, HLA's proponent.

Another approach is to build a research interconnection infrastructure that is self-reconfiguring and use it for large DVE executions in a research environment. I have had preliminary discussions with the Naval

Postgraduate School and my colleagues at the United States Military Academy about a joint endeavor to build such an interconnection interface. Staging the executions as games is one way we hope to get users to participate in our case studies.

Bullpen currently works in the DVE environment where the events are not synchronized and events are never rolled back. Generalizing Bullpen to handle all types of distributed simulation, and not just virtual simulation would make it a much more useful tool. To handle the more general domain of distributed simulation, the reconfiguration logic in Bullpen must change, and Scoreboard would need to be more sophisticated to facilitate time management.

Beyond DVEs

Compensating reconfiguration is also appropriate for a broader class of problems in distributed systems. For example, it can be used to implement dynamic space-sharing on clusters of non-dedicated workstations. Dynamic space-sharing is a policy that can change the number of processors allocated to a distributed application during execution. Implementation on a cluster of non-dedicated workstations requires the dynamic reconfiguration of the application in three cases: when a workstation being used by the application fails, when owner activity is detected on a workstation, and when the job scheduler decides to reduce or increase the number of processors allocated to the application. [Cho97]

Assuming that the distributed applications have been built to run using a runtime infrastructure that supports dynamic reconfiguration such as POLYLITH, [Pur94] a compensating reconfiguration component can be built using Bullpen to implement space sharing in the following way. The conditions that Scout must handle are: a new job, job completion, failure of a component, creating a checkpoint, and owner activity. Coach would assume the role of job scheduler. Scoreboard would track the information on the workstations, applications, and components of the applications.

For Scout detecting conditions would mostly be trivial. Scout would detect the new job, job completion and owner activity conditions by simply receiving a message from the component. Scout would notify the components to create a checkpoint at a specified time interval. The most sophisticated condition for Scout is component failure, which Scout has successfully accomplished in DVEs.

Coach would provide the job scheduler function through CLIPS rules. As in the DVE environment it would rely on the information provided by Scoreboard. The activities required for reconfiguration such as rollback would be encoded into the reconfiguration rules.

Scoreboard would track the workstations as it tracks hosts in a DVE. It would handle applications the way it handles VEs. Applications are made of up components that provide atomic units of functionality. The components in this case are equivalent to DVE entities that are the atomic units of functionality.

References

[Art89] Yeshyahu Artsy and Raphael Finkel. Designing a Process Migration Facility. *IEEE Computer*, 22(9):47-56.

[Bah96] Hubert Bahr and Ronald DeMara. A Concurrent Model Approach to Scaleable Distributed Interactive Simulation. In *Proceeding of the 15th Distributed Interactive Simulation Standards Workshop*, 1996.

[Bar90] Mario Barbacci, Dennis Doubleday, and Charles Weinstock. Application-Level Programming. In *Proceedings of the 10th International Conference on Distributed Computing Systems*, pages 458-464, 1990.

[Ben90] M. Ben-Ari. *Principles of Concurrent and Distributed Programming*. Prentice Hall, New York, 1990.

[Bro97] Wolfgang Broll. Bringing People Together---An Infrastructure for Shared Virtual Worlds on the Internet. In *Proceedings of the 6th WET-ICE*. June 1997, pages 199-204.

[Bru97] Don Brutzman, Mike Zyda, K. Watsen, and Mike Macedonia. Virtual Reality Transfer Protocol(VRTP) Design Rationale. In *Proceedings of the 6th WET-ICE*. June 1997, pages 179-186.

[Cal91] John Callahan and James Purtilo. A Packaging System For Heterogeneous Execution Environments. *IEEE Transactions on Software Engineering*, 17(6):626-635, Jun 91.

[Cal94] James Calvin and Daniel Van Hook. AGENTS: An Architectural Construct to Support Distributed Simulation. In *Proceedings of the 11th Distributed Interactive Simulation Standards Workshop*, Sep 94.

[Cal95] James Calvin, Carol Chiang, and Daniel Van Hook. Data Subscription. In *12th Distributed Interactive Simulation Standards Workshop*, Orlando, FL, September 1995.

[Che88] David Cheriton. The V Distributed System. *Communications of the ACM*, 31(3):314-333, Mar. 1988.

[Cho97] Abdur Chowdhury, Lisa Nicklas, Sanjeev Seitia, and Elizabeth White. Supporting Dynamic Space Sharing on Clusters of Non-dedicated Workstations. In *Proceedings of the 17th International Conference on Distributed Computing Systems*, Baltimore, MD, May 1997.

[Car93] Christer Carlsson and Olof Hags. DIVE - A Platform for Multi-User Virtual Environments. *Computers and Graphics*, 17(6):663-669, 1993.

[Cri91] Flavio Cristian. Understanding Fault-Tolerant Distributed Systems. *Communications of the ACM*, 34(2):57-78, Feb. 1991.

[DMSO97a] Defense Modeling and Simulation Office. *High Level Architecture Rules*, Aug. 1997. Version 1.2.

[DMSO97b] Defense Modeling and Simulation Office. *High Level Architecture Interface Specification*, Aug. 1997. Version 1.2.

[DMSO97c] Defense Modeling and Simulation Office. *High Level Architecture Object Model Template*, Feb. 1997. Version 1.1.

[Dis94] DIS Steering Committee. *The DIS Vision, A Map to the Future of Distributed Simulation*. May 1994.

[Fag97] R. Fagin, J. Halpern, Y. Moses, and M. Vardi. Knowledge-based Programs. *Distributed Computing*, 10(4):199-225, Aug. 1997.

[Fri91] Ophir Frieder and Mark. Segal. On Dynamically Updating a Computer Program: From Concept to Prototype. *Journal of Systems Software*, 14(2):111-128, February 1991.

[Gia93] J. Giarratano. *CLIPS User's Guide*. NASA Johnson Space Center, Information Systems Directorate, Software Technology Branch. 1993.

[Gre96] Mark Green. Shared Virtual Environments: The Implications for Tool Builders. *Computers and Graphics*, 20(2):185-189, 1996.

[Gri93] William Griswold and David Notkin. Automated Assistance for Program Restructuring. *ACM Transactions on Software Engineering and Methodology*, 2(3):228-269, July 1993.

[Gup93] Deepak Gupta and Pankaj Jolote. On-Line Software Version Change Using State Transfer Between Processes. *Software Practice and Experience*, 23(9):949-964, September 1993.

[Gup96] Deepak Gupta, Pankaj Jalote, and Gautam. Barus. A Formal Framework for On-line Software Version Change. *IEEE Transactions on Software Engineering*, 22(2):120-131, February 1996.

[Hof91] Christine Hofmeister and James Purtilo. Dynamic Reconfiguration of Distributed Programs. In *Proceedings of the 11th International Conference on Distributed Computing Systems*, pages 560-571, 1991.

[Hof93] Christine Hofmeister, Elizabeth White, and J. Purtilo. SURGEON: A Packager for Dynamically Reconfigurable Distributed Applications. *Software Engineering Journal*, 1993.

[Kra85] Jeff Kramer and Jeff Magee. Dynamic Configuration for Distributed Systems. *IEEE Transactions on Software Engineering*, 11(4):424-435, April 1985.

[Kra90a] Jeff Kramer and Jeff Magee. The Evolving Philosophers Problem: Dynamic Change Management. *IEEE Transactions on Software Engineering*, 16(11):1293-1306, Nov. 1990.

[Kra90b] Jeff Kramer, Jeff Magee and A. Young. Towards Unifying Fault and Change Management. In *Proceedings of the IEEE International Workshop on Distributed Computing Systems in the 90's*, pages 57-63, 1990.

[LeB85] Thomas Le Blanc and Stuart Friedberg. HPC: A model of Structure and Change in Distributed Systems. *IEEE Transactions on Computers*, C-34(12):1114-1129, Dec. 1989.

[Mac94] Michael Macedonia, Michael Zyda, David Pratt, Paul Barham and Steven Zeswitz. NPTSNET: A Network Software Architecture for Large Scale Virtual Environments. *Presence*, 3(4), Fall 1994.

[Mag89] Jeff Magee , Jeff Kramer and Morris Sloman. Constructing Distributed Systems in Conic. *IEEE Transactions on Software Engineering*, 15(6): 663-675, Jun. 1989.

[Min96a] Naftaly Minksy. Independent On-Line Monitoring of Evolving Systems. In *Proceedings of the 18th International Conference on Software Engineering*. Mar. 1996.

[Min96b] Naftaly Minsky. Law-Governed Regularities in Object Systems; Part 1: An Abstract Model. *Theory and Practice of Object Systems (TAPOS)*, 2(4), 1996.

[Min97] Naftaly Minsky and Partha Pal. Law-Governed Regularities in Object Systems; Part 2: A Concrete Implementation. *Theory and Practice of Object Systems (TAPOS)*, 3(4), 1997.

[Not88] David Notkin, Andrew Black, Edward Laxowska, Henry Levy, Jan Sanislo, and John Zahorjan. Interconnecting Heterogeneous Computer Systems. *Communications of the ACM*, 31(3):258-273, March 1988.

[Pol95] Michael Polis, Stephen Gifford, and David McKeown. Automating the Construction of Large-Scale Virtual Worlds. *IEEE Computer*, 28(7):57-65, July 1995.

[Pur94] James. Purtilo. The POLYLITH Software Bus. *ACM Transactions on Programming languages*, 16:151-174, Jan 1994.

[Rus95] Mark Russinovich and Zary Segall. Fault-Tolerance for Off-The-Shelf Applications and Hardware. In *Proceedings of the 25th International Symposium on Fault Tolerant Computing*, pages 67-71; Pasadena, CA, June 27-30 1995.

[Seg93] Mark Segal and Ophir Frieder. On-The-Fly Program Modification: Systems for Dynamic Updating. *IEEE Software*, 10(3):53-65, March 1993.

[Sie96] J. Siegel. *CORBA Fundamentals and Programming*. Wiley Computer Publishing Group, New York, 1996.

[Sin95] Gurminder Singh, Luis Serra, Willie Png, Audrey Wong, and Hern Hg. BrickNet: Sharing Object Behaviors on the Net. In *Proceedings of the IEEE Virtual Reality Annual International Symposium*, pages 19-25; Research Triangle Park, NC, March 11-15 1995.

[Shi95] James Shiflett, W. Lunceford, and Ruth Wells. Application of Distributed Interactive Simulation Technology Within the Department of Defense. *Proceedings of the IEEE*, 83(8):1168-1178, August 1995.

[Ste97] David Stewart, Richard Volpe, and Pradeep Khosla. Design of Dynamically Reconfigurable Real-Time Software Using Port-Based Objects. *IEEE Transactions on Software Engineering*, 23(12): 759-776, Dec. 1997.

[Sti97] Martin Stytz, Terry Adams, Brian Garcia and Steven Sheasby. Rapid Prototyping for Distributed Virtual Environments. *IEEE Software*, 14(5): 83-92, September/October. 1997.

[Wat97] R. Water, D. Anderson and D. Schwenke. *Design of the Interactive Sharing Transfer Protocol*. In *Proceedings of the 6th WET-ICE*. June 1997, pages 140-147.

[Wea96] Richard Weatherly, Annette Wilson, Bradford Canova, Ernest Page, Anita Zabek, and Mary 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.

[Wer96] Matthias Werner, Andreas Polze, and Mrioslaw Malek. The Unstoppable Orchestra: a Responsive Distributed Application. In *Proceedings of the Conference on Configurable Distributed Systems*, pages 154-160, Annapolis, MD, May 6-8 1996.

[Whi96] Elizabeth White, Kenneth Frosch, Vincent Laviano, Michael Heib, and Mark Pullen. Interfacing External Decision Processes to DIS Applications. In *6th Conference on Computer Generated Forces*, Orlando, FL, July 1996.

[USDAT95] Under Secretary of Defense (Acquisition and Technology), *Modeling and Simulation Master Plan*, Jan. 1995.