Zubin: A Software Engineering Environment for Interconnecting Legacy Virtual Environments

Donald J. Welch Jr. *

dwelch@cs.umd.edu

Department of Computer Science

University of Maryland

James Purtilo †

purtilo@cs.umd.edu

Institute for Advanced Computer Studies and

Department of Computer Science

University of Maryland

November 25, 1996

Abstract

As the power and utility of virtual reality environments increases, so do the potential benefits found from combining several such environments. But doing so presents the developer with a host of difficult software engineering issues. This paper explores what some of these issues are within the VE domain, relates our successes to date in overcoming these problems by means of various automated tools, and suggests ways to apply our results other target domains.

Keywords: Software Development Methodology, Rapid Prototyping, Legacy Systems, Virtual Environments

*Contact Information. Address: Department of Computer Science, A. V. Williams Bldg. University of Maryland, College Park, MD 20742-3255; Phone: (301) 405-2722; Fax: (301) 405-6707

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

1 INTRODUCTION

Interconnecting legacy programs is laborious and prone to errors. There is a significant amount of new development associated with the interconnection software as well as the modifications required to the legacy software. Interconnected systems by their nature are more complicated than their component parts. Building interconnected computer systems from existing applications requires that developers overcome many types of problems not normally encountered when developing new software from scratch. We are intensely interested in finding methods and support tools that will lower the cost of building with legacy systems, while at the same time improving the quality of any resulting products. Our approach in this research has been to focus on a variety of special domains, wherein issues for dealing with legacy systems can be examined in depth, with the intent of abstracting these lessons later. Focusing on domain-specific software engineering was resurfaced by Glass and Vessey as a possible way to gain breakthroughs in software engineering research.[5]

One of these special domains under consideration is that of building virtual environments (VEs). Interconnected Virtual Environments (IVEs) are multi-user distributed VEs. The key factor that differentiates IVEs from the rest of the distributed VEs is that they are made up of legacy VEs that may not have been designed to interact with other programs. The separate VEs of an IVE do work together in a cooperative environment, but the world they represent has been built to meet their individual needs and not to match their sibling VEs. IVEs have all the usual problems associated with program interconnection plus other issues such as coordinate system translations, synchronization, partitioning the virtual world and object grouping. In addition, the component VEs may themselves be distributed or even IVEs.

This added complexity, especially the mapping across virtual worlds makes it more difficult for the end-users to understand and communicate what they want the IVE to do. A user may be familiar with the operation of one or more component systems but may not be able to visualize the complete interconnected system or the nuances of such a new system.

Prototyping can help reduce the risk of building the "wrong system". A working prototype of the system or the parts of the system that are the least understood can help the user understand what the final version of the system will look like. Better yet is an executable specification of an IVE. With an executable specification the user can point to a system and say, "I want it to do that," and the developer can look at the system and understand what "that" is.

The problem with this approach usually occurs when the user doesn't understand why it will take so long to build a production system when he can see what he wants already running. Trying to retro-fit quality into a prototype to transform it into production code can be problematic. Normally the prototype is built to get something running quickly, and will suffer many major modifications throughout its lifecycle. Most successful prototyping is used to learn more about the system and apply that knowledge to the design and implementation of the production system to achieve a quality product.

The purpose of this paper is to illustrate what we are developing to help with the interconnection of legacy VEs, as well as to relate our experiences to the issues encountered while building general interconnected applications. This paper discusses requirements for tools that support interconnection of virtual environments. We continue by explaining how our environment meets these requirements with an interconnection generator, interconnection runtime abstraction, and tailored methodology. To illustrate some of our points we will use the example of interconnecting a number of existing military trainers and simulators into a large war game distributed across the Internet.

2 REQUIREMENTS

We break the requirements down into two sections. First we discuss the requirements for a general interconnection software engineering environment, then we discuss the additional requirements for an environment tailored to IVE domain.

2.1 Interconnection Requirements

The environment must be powerful enough to create fully functioning system. An executable specification must be able to show all facets of the application for it to be used as a complete specification or as the basis for production development.

Additionally, the environment must be able to quickly create a working prototype/specification of the interconnection. If the tool is to be useful, it must be substantially quicker and cheaper to build an application using this environment than through other methods. Since numerous iterations will be necessary to tighten up the requirements, the time required to modify the application must be significantly less than manual coding.

The environment must be capable of incorporating design decisions as more is understood about the system. The performance of the application can be severely impacted by basic design decisions, and these decisions must be testable while in the early phases of the development, if the code will be used as the basis for production code.

It must create robust modifiable code that can serve as the base for a production system. Exploiting the work done in prototype development can reap large benefits, if the pitfalls can be avoided. The code produced must expose hinge points (places where change is likely throughout the life-cycle) to the software engineers.

There are two types of reuse and both should be exploited in this environment.[9] First, and most common is software reuse. Proven code can be integrated into a system with less effort, providing higher quality.[2] Second, is process reuse. The process of successfully creating code can be re-used to create code that is similar, high-quality, and requires less effort. The best way to capture and reuse a successful process is with a code generator.

2.2 IVE specific requirements

The interconnection software must maintain a minimal copy of the system state. This gives the interconnection software the ability pass only changes around the IVE, handle dynamic reconfiguration in both normal and abnormal circumstances, and insure that the state remains consistent without significantly adding to the message traffic. This implies that it must also maintain the hierarchy and grouping of the data.

The interconnection software must translate the coordinate systems used by the component VEs as part of message passing. The VEs will most likely use different coordinate systems and translating between them can be computationally intensive.

All events in the component VEs will not map nicely to single events among the other VEs. The interconnection software must perform this mapping in a seamless way to the component VEs, otherwise the modification of the legacy code becomes extensive.

Large IVEs may run for long periods of time with components entering and leaving the virtual world gracefully, and as a result of failure. The IVE must not only continue to function but continue to provide a consistent virtual world to the rest of the users.

2.3 Related Work

Research in distributed virtual environments has been mostly confined to multi-user virtual environments built from scratch to work together or distributing the processing of VE tasks to improve performance. The *EM toolkit* and *DIVE* environment are two of the best known systems that provide an environment for building the entire distributed VE. [12] [6] *Virtual Design* is another complete VE software engineering environment that also focuses on using a wide array of input data and hardware. [1] *DGPSE+* is an experimental environment supporting distributed graphics support. [10] None of these systems supports the reuse of legacy VEs.

This work draws on the experience with code generators, interconnection tools, and packagers. Hofmiester and Purtilo [7] have built the SURGEON system for dynamic reconfiguration and Chen and Purtilo [3] have built tools for mapping events in event based interconnected systems.

3 ZUBIN: THE SOFTWARE ENGINEERING ENVIRONMENT

We have built an environment towards meeting these requirements. It has three facets: a tailored methodology that exploits the software tools, an interconnection generator (Zubin) to create the application specific parts of the interconnection runtime and the interface software, and a runtime interconnection abstraction (Isthmus).

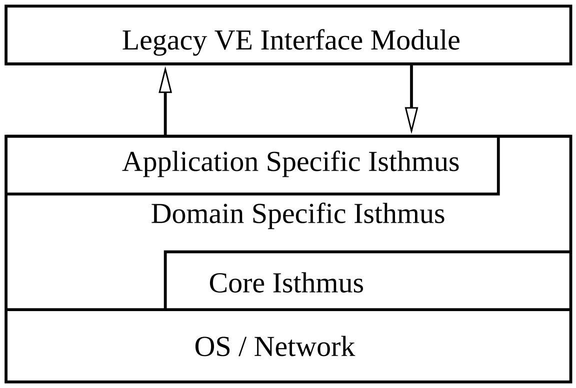

Figure 1: Isthmus Architectural Layers

3.1 ISTHMUS - Runtime Interconnection Abstraction

Isthmus is the runtime environment that serves to abstract the interconnection details from the user. Isthmus is inspired by the Polylith software bus.[11] Each instantiation of Isthmus contains the code necessary to provide the general interconnection services, the IVE domain specific services and the application specific services. For example Isthmus is capable of passing messages reliably (TCP) or unreliably (UDP), using single-casting or Multicasting. These services are common to most interconnection environments. The requirement to store the system state is common to IVEs. The translation between different coordinate systems, although in general, a common requirement in IVEs is very application specific due to the wide array of coordinate systems in use in legacy VEs.

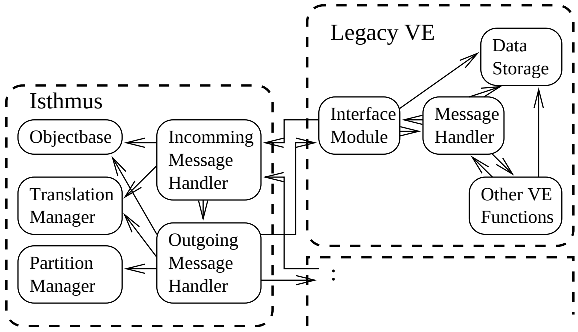

Figure 2: Interconnection Software Architecture

All communication in the system is between the interface modules and Isthmus. When an event occurs in a legacy VE, it uses the interface module API to send a message to Isthmus. Upon receipt of message Isthmus decides what action must be taken for each other VE in the IVE and performs those tasks.

Isthmus keeps track of the system state that is salient to the IVE in its objectbase, therefore only changes need be passed around the IVE. The Isthmus objectbase uses a simple locking scheme to maintain data coherency in the system. For performance all coherency resolution that can be done in the component VEs is done there, but some must be resolved in the interconnection. In this case the component VE will use a synchronous messages to “ask permission”, before it makes the change to its version of the state. Grouping of objects is also provided by the Isthmus. When virtual objects are grouped either permanently or temporarily, that grouping can be transparent to the component VEs.

The interconnection software does the coordinate system translation. In some cases it is advantageous to do the (usually computationally intensive) translations in Isthmus, others in the interface module. By placing the translation burden on Isthmus, the component VEs are spared this interconnection overhead. However, this means that multicasting cannot be used, and a message must be generated and sent to each component VE. If translation is done in the interface module then message traffic can be reduced by using multicasting, however, the systems running the component VE must be able to handle this load. The interconnection software can also be built with a hybrid system, where some messages are multicasted and translated by the interface modules and some messages are translated by Isthmus.

Isthmus also decides which messages a VE is interested in, and which it can ignore. This helps to reduce network traffic and the load associated with handling the messages in the component VEs. Isthmus uses the definitions of spatial areas of interest and functional interests to make this decision.

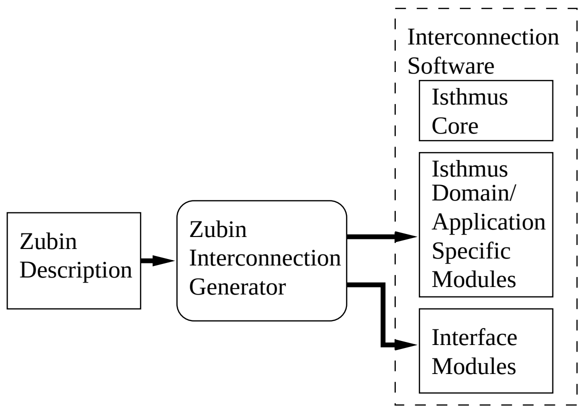

Figure 3: Using Zubin to Create Isthmus

3.2 ZUBIN: the Interconnection Generator and Description Language

The runtime interconnection abstraction is highly customized to fit the design of the legacy VEs and the requirements of the IVE. The runtime and interface modules can be built by hand, but we feel that this design and coding process can be effectively captured and reused by an interconnection generator. The interconnection generator is similar to application generators that allow reuse of processes in domain specific applications built from scratch. The difference is, of course, that Zubin does not have control over how the entire application built, but only generates the software that glues together the legacy applications.

We have an interconnection generator and description language called Zubin for this domain that generates the source code for Isthmus and the component interface modules. Zubin uses the Isthmus core runtime and adds customized services to it as called for in the formal description. It also generates the interface modules in the native programming language for each component VE.

Zubin is designed to create readable source code. The source code can be used just as an API to quickly create the prototype or it can serve as the foundation of the production system, and as a result modules will have to be modified by software engineers. Therefore the software engineer can declare user-defined constants and types in the description language that will end up in the source code. Zubin also supports the inclusion of comments in the formal description that are transferred to the source code of Isthmus and the component interface modules.

The Zubin description gives a mostly non-procedural description of the software glue that holds the IVE together. There are some places in the IVE description where design decisions are recorded. It also contains places where code escapes can be inserted in the description. The reasons we departed from a purely non-procedural specification was the need to support iteration and the complexity of many of the algorithms used in VEs. The first specification of the IVE can contain very little design relying on the defaults. As a prototype is built and information is discovered, that information can be incorporated into the description before the next iteration is built and tested.

A Zubin interconnection description is broken down into sections that concentrate on a specific functionality that the interconnection is to exhibit. The design of the description language allows developers to concentrate on the different functions of the IVE independently while Zubin provides the integration framework much like CDSL does for more general applications.[4]

Since many different coordinate systems will be in use in the IVE, the translation algorithms to convert coordinates between each coordinate system in use are described in the *Translation Section*. The algorithms are defined as code escapes because the translation of coordinate systems is not always a single mathematical formula. The spherical earth does not transfer well to a flat map, therefore converting geographic coordinates requires conditionals and lookup tables.

Interconnection and configuration information for Isthmus are described in the *Component Section* of the description. In addition, the initialization information required for component VEs to join the IVE is also related here. It also contains information about the different types of component VEs that may participate in an IVE session. Much of the information here will go into the generation of the interface modules.

The environmental states, such as weather or lighting conditions are defined in the *Environment Section*. Declaration of environmental states automatically causes Zubin to generate events that change the states.

Partitioning and message routing is described in the *Areas of Interest Section*. Areas of Interest may be declared based on spatial constraints (the most popular), or functional. This partitioning — to reduce message traffic — can be handled by using multicasting which is useful in large IVEs, or by the Isthmus selectively sending messages to the component VEs.

The *Objects Section* describes the objects of the virtual world that can be manipulated by any of the component VEs. Objects are described using inheritance which provides less repetitive descriptions, but more importantly allows more intuitive descriptions of things such as collisions. A set of default attributes automatically is included with each object. These attributes vary with the object type but primarily include the location (in three dimensions), the orientation (3 degrees of freedom), and the size (min and max along each dimension) of the object.

Figure 4 shows an example of a virtual object description in the Zubin description language. This simple example shows the description of a tank virtual object. The tank is declared to be an active object, therefore Isthmus is aware that the tank can move

```
OBJECT obTank ACTIVE:
# Comments that go in the source code
# for the Tank Virtual Object
ATTRIBUTES :
  FLOAT atTurretDir, # in radians
  FLOAT atDir,
  BOOLEAN atDrawPlow = FALSE;

RULES rnMove(atX, atY)
  FOR ALL ACTIVE
  {* if ((atX + atMaxSizeX) >=
              ($atX - $atMinSizeX) &&
      (atX - atMinSizeX) >=
              ($atX + $atMaxSizeX) &&
      (atY + atMaxSizeY) >=
              ($atY - $atMinSizeY) &&
      (atY - atMinSizeY) >=
              ($atY + $atMaxSizeY))
   *} VIOLATION evCrash(atObId, $atObId);

HOLDS
  obTurret;

CAN HOLD
  obMinePlow[atDrawPlow = TRUE:FALSE];
```

Figure 4: Example Virtual Object Description

```
evMove (INT obId, INT X, INT Y, FLOAT dir):
  MESSAGE SYNCH rnMove;
    FROM cmTankSim, cmBradleySim;
    CONTENT (INT obId, INT X, INT Y,
             FLOAT dir);

  MESSAGE ASYNCH;
    TO cmTankSim SPATIAL;
    PRIORITY REGULAR;
    RELIABILITY UNRELIABLE;
    CONTENT (INT obId, INT X, INT Y,
             FLOAT dir);
```

Figure 5: Example Event Description

about the virtual world influencing other virtual objects. Those attributes that are not part of the default attribute set are declared here, and their initialization values may also be declared here. One rule is shown, and this states that no two active objects can occupy the same space at the same time. Only two dimensions are used since tanks usually stay on the ground. All default attributes are used in this calculation. If this rule is violated, Isthmus is told to generate the "Crash" event. Finally object grouping is declared through the HOLDS and CAN HOLD declarations. There are other features of virtual object description that aren't touched on here.

The *Events Section* contains descriptions of the functionality of the IVE. Each event in the IVE has a list of messages that are sent or received when the event occurs. The messages can be synchronous, which require confirmation from Isthmus or asynchronous. The developer also designates whether or not a message is to be filtered based on areas of interest and the content of the message is defined. Events that have a many-to-one mapping can also be described here.

Figure 5 shows an example of an event description in the Zubin description language. This event, the moving of a virtual object in the virtual world shows there are two messages associated with this event, there may be many more as messages are tailored for the VE they with which they communicate. The first message is from the VEs to Isthmus. It is a synchronous message, which means that Isthmus must check the "Move" rule before it can acknowledge the message. If the rule is not violated, the appropriate "to" messages will be sent to the other VEs. This message is an update of the object location and receipt is not critical to the IVE therefore it is an asynchronous method that is sent by unreliable (less bandwidth) means.

3.3 The Zubin Methodology

To properly tie these tools together a tailored methodology is required to exploit the strengths of these approaches. This methodology focuses on extracting the salient features of the existing systems and combining those with the new requirements of the IVE to create a description of the interconnection. This description is used to create prototype systems quickly. The system is then assessed by the user to check that the new system does what it needs to. After a number of iterations, the requirements should be stable and both user and software engineer should be confident that the system represents an executable specification for the IVE. At this point the executable specification can be used to build a production system. The IVE code already carries a high degree of assurance that it accurately translates the description into interconnection code. Techniques can now be applied to optimize the simple interconnection code so that performance requirements can be met. These techniques can be automatic or manual.

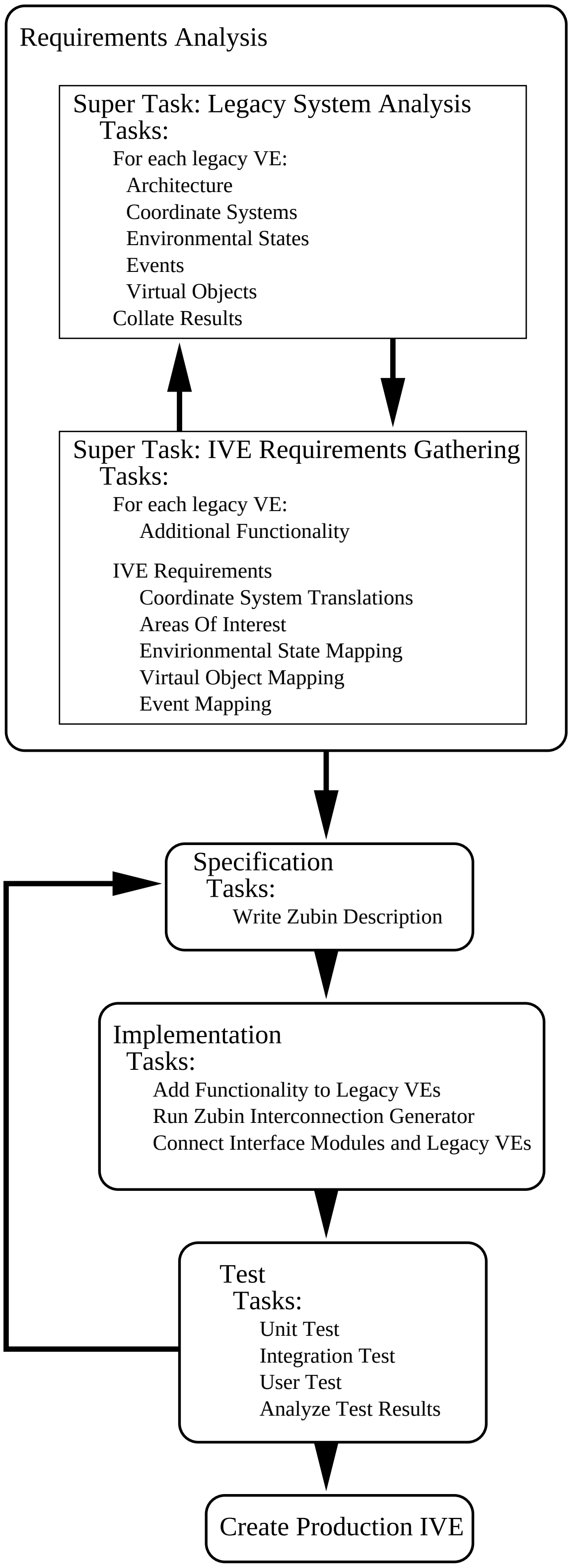

Figure 6: Zubin Methodology

3.3.1 Phase 1: Requirements Analysis

The first phase has two major super tasks: Legacy System Analysis and IVE Requirements Gathering, each with their own task list. The former concentrates on understanding the legacy systems within the context of the IVE, while the latter focuses on understanding how the total system is to act. These two activities cannot be performed in isolation as they are heavily influenced by the results of the other.

VEs are highly complex software. One of the driving factors during this phase is to reduce the data that a VE uses to its minimal form for passing around the IVE. For example, weather is a highly complex phenomena that can use many data structures to compute. The VEs will most assuredly not handle weather the same way. For this IVE to work, weather has to be reduced to the lowest common denominator. A set of states or a small set of numbers that each VE can use to map to their weather systems so that they are all sharing a virtual world that meets the needs of the IVE users.

Super-task a: Legacy System Analysis The tasks of Legacy System Analysis start with understanding the basics such as architecture, system and programming language. Each VE may use numerous coordinate systems, a primary coordinate system must be identified that the VE will use to communicate with the rest of the IVE. The environmental states found in the legacy VE must be identified for collating during IVE requirements gathering. The legacy VE events also have to be examined to determine which of them are salient to the rest of the IVE. The virtual objects present in each legacy VE must be identified and a determination made as to which ones need to be shared with the rest of the IVE.

The objects are classified by their type: active, inactive and environmental. Normally environmental objects will stay within the confines of their VE while active and inactive objects must be shared. To illustrate the classification, a tank would be active, it moves by itself and can change other virtual objects by shooting or running into them. A trailer would be inactive, it can move around the battlefield, but only when moved by an active object (truck). The trailer can be grouped or ungrouped with other objects. A hill would be an environmental object, it exists, but cannot be changed or moved by other objects.

Some non-obvious factors may end up in messages. Some VEs predict the action of virtual objects in the absence of updates to keep the realism as high as possible. These VEs will not only require updates on virtual object's position and orientation, but on the forces acting on the object.

Once each individual legacy VE has been examined, the results are collated to make the final determination about which parts of the component VEs will be shared with the rest of the IVE.

Super-task b: IVE Requirements Gathering One of the tasks that must be accomplished as part of this super-task is determining what, if any functionality must be added to each legacy VE. Once the functionality is identified, the design, implementation and test of this added functionality is handled using a traditional maintenance process, which is beyond the scope of this paper.

The other tasks to be done as part of this super-task are the assimilation of the products of Legacy System Analysis into a coherent set of requirements for the interconnection software. Some of this assimilation will involve the mapping of items from the legacy VEs to each other through the interconnection software. Other parts include developing requirements for the way the IVE will function. The five major tasks to be accomplished here are:

Coordinate Systems Translations. A determination of which VE coordinate system to use in Isthmus must be made. Once decided, the software engineer need to only specify the translations between the Isthmus coordinate system and the component coordinate systems.

Areas of Interest. Message traffic can create a bottleneck both by taking up network bandwidth and using the component VE's processing time. Network bandwidth can be freed up by using Multicasting and the cpu burden can be eased by filtering out messages to components that do not need to receive them. The interconnection software uses areas of interest — both spatial and functional — to reason about message traffic reduction.

Environmental States Mapping. This is where the mapping between all the environmental states identified in the legacy VEs is made. The states must be distilled to their most basic components while still meeting the user's requirements for realism.

Virtual Objects Mapping. Object mapping involves identifying how each of the objects in the IVE will be represented in each VE. Objects will not always map directly to each other, in our example wargame only U. S. Forces are involved. To add realism a U.S. tank simulator may map to a Russian T-80 tank in the anti-tank missile trainer. In addition to mapping the objects, the coherency rules must be identified along with an algorithm for determining when they have been violated.

Event Mapping. Events must be looked at from three different perspectives. The largest task is to map the events from legacy VEs to each other and determine which events are salient to the IVE. For example, in one VE a tank turret rotation may be considered a move event for the turret object, while in another component this may be a change of direction event for the entire tank object. Events that will be created in the IVE and not originate in the component systems will have to be identified. An example is the "Crash" event in our example above. Some events will not have a one-to-one mapping between VEs. If one VE uses teleporting to move large distances this may not be acceptable because it appears unreal in another component, a solution may be that one large move maps to many small ones.

3.3.2 Phase 2: Specification

The major task of this phase is to take the products from phase 1 and put them into a formal preliminary description of the interconnection software. This formal description is written in the Zubin description language.

3.3.3 Phase 3: Implementation

This phase has three major tasks. Any functionality that must be added to the legacy VEs must also be completed by this point. The Zubin description is turned into into an instantiation of Isthmus and the interface modules are built. The interface modules are built in the legacy VE's programming language and provide a tailored application programming interface (API) to the rest of the IVE. The final task is to use the API that is provided by the interface module to manually connect the legacy VE source code to the interface modules, and thus the IVE.

3.3.4 Phase 4: Test

Once the legacy code has been modified and the executables built the system can be tested by the software engineers. Functionality added to the legacy VEs (if any) must be tested. Because we are reusing the process with a code generator, we have a high degree of assurance that the system meets the specifications as stated in the Zubin description. We can focus our testing on determining whether or not the specifications capture the true IVE requirements. The IVE can then be presented to the users to determine whether or not the developers have correctly captured the system requirements. In addition any changes to the requirements can be documented at this time.

With the additional knowledge gained at this point, the developers go back to the specification phase and modify the description. Design decisions can be put into the system such as where coordinate translation will occur at the component VE or in Isthmus. When the system has been rebuilt it can again be presented to the user. Eventually the system is an executable specification for the IVE.

3.3.5 Phase 5: Creating a Production System

Armed with this executable specification for the IVE the software engineers can then set about building a production quality system. There are two tracks that they can follow: starting from scratch, or reusing the prototype. Normally, trying to build quality and performance into a prototype system is a recipe for disaster. In this instance we feel that it is possible. The interconnection code generated by Zubin is robust code that has been used and tested numerous times. The interconnection code is modular and as long as the interfaces are kept intact it should be possible to keep changes isolated. As far as performance goes, the technology is emerging that allows simple representations of interconnection to be optimized to give almost hand-tuned performance.[8]

4 CONCLUSIONS

In our experiments so far we have found that the performance of Isthmus is comparable to hand-built interconnection code. This is not surprising, since our goal was to capture the process of manually creating code and automate it. In our development process we used several manually created IVEs as the "target" for Zubin while we created it.

Our preliminary work verifies that Zubin requires less effort than creating equivalent interconnection code by hand. The process for using Zubin mirrors manual coding up through specification. A Zubin interconnection description is just another way of documenting the interconnection specification. Savings of effort begin to show during implementation and test. In a simple walkthrough IVE a description of about 400 lines was turned into approximately 5000 lines of C++ source code by Zubin. If nothing else we have saved the clerical work of creating 5000 lines of source code. The minimal effort to use the interface module API was present while creating both manual and automatic versions of the IVE. We saved effort testing, because we focused on overall IVE behavior, and did not spend the time verifying that the interconnection code met specifications. However, the real savings came when as a result of test, major changes were made to the IVE. Developers may reason about the IVE at an abstract level, and manipulate change there as well, rather than having to make "sweeping" changes only in terms of low-level code. In one instance we decided to change the coordinate system used by Isthmus, and putting this into the next iteration was trivial. This change affected almost every code module of the interconnection software; making the change manually required hours of editing and compiling.

We have combined an interconnection abstraction that has been automatically specialized for a specific application, an interconnection generator that creates the interconnection runtime as well as the applications programming interface with a tailored methodology that exploits the strengths of these tools to create a rapid prototyping environment for the virtual environment domain. The executable specification that results is useful to developers of production systems in two ways. By having an executable specification much more can be understood about the target system by the developers than just working from paper documents alone. The users have had an opportunity to see what the final system will look like, and both users and requirements engineers can have higher assurance that the correct system is being built. The interconnection code is robust since it is the result of process reuse, therefore it may be possible to modify the prototype to create a production quality system. Our system does not replace the difficult reasoning about IVEs. Instead it automates the more mundane aspects of IVE creation, allowing the software engineers to focus on the core problems facing the IVE builder. Our research continues as we evaluate the breadth of Zubin applicability, the expressiveness of our notation, and the extent to which our approach continues to capture the performance of hand-tuned code.

References

[1] Peter Astheimer, Wolfgang Felger, and Stefan Müller. Virtual Design: A generic VR System for Industrial Applications. *Computers and Graphics*, 17(6):671–677, November/December 1993.

[2] Victor R. Basili, Lionel C. Briand, and Walcélio L. Melo. How Reuse Influences Productivity in OO Systems. *Communications of the ACM*, 39(10):104–116, October 1996.

[3] Chen Chen and James Purtilo. Event Adaptation for Integrating Distributed Applications. In *Proceedings of the Conference on Software Engineering and Knowledge Engineering*, June 1995.

[4] Flavio DePaoli and Francesco Tisato. Cooperative systems configuration in CSDL. In *Proceedings of the International Conference on Distrubuted Computing Systems*, pages 304–311, Poznan, Poland, June 21-24 1994.

[5] Robert L. Glass and Iris Vessey. Contemporary Application-Domain Taxonomies. *IEEE Software*, 12(3):63–76, July 1995.

[6] Mark Green. Environment Manager. Technical report, Department of Computing Science, University of Alberta, February 1994.

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

[8] Tae-Hyung Kim. *Toward Optimizing Distributed Programs Directed by Configuration*. PhD thesis, University of Maryland College Park, September 1996.

[9] Hafedh Mili, Fatma Mili, and Ali Mili. Reusing Software: Issues and Research Directions. *IEEE Transactions on Software Engineering*, 21(6):528–561, June 1995.

[10] Zhigeng Pan, Jiaoying Shi, and Mingmin Zhang. Distributed Graphics Support for Virtual Environments. *Computers and Graphics*, 20(2):191–197, March/April 1996.

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

[12] Qunjie Wang, Mark Green, and Chris Shaw. EM - An Environment Manager For Building Networked Virtual Environments. In *Proceedings of the IEEE Virtual Reality Annual International Symposium*, pages 11–18, Research Triangle Park, NC, March 11-15 1995.