Interconnecting Distributed Legacy Systems: Virtual Environment Domain Example

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

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

October 1, 1996

Abstract

As the power and utility of virtual reality environments increases, so do the potential benefits found from combinding several such environments. But doing so presents the developer with a host of difficult distributed systems 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.

1 Overview

Building distributed 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 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.

One of these special domains under consideration is that of building virtual environments (VEs). The problems encountered in building VEs are not at all foreign to the distributed computer system developer, but we have found they may be solved with less effort by applying some domain-specific software engineering techniques. 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 more issues encountered while building general distributed applications. This paper discusses requirements for tools that support interconnection of virtual environments, and will highlight the system issues such as mixing different communication protocols, distributed event generation, heterogeneous architecture and dynamic application structure.

*Lieutenant Colonel Welch is in the U.S. Army and is studying at UMCP through the Army's advanced civil schooling program.

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

The IVE Domain For our purposes VEs which include virtual reality, simulation, and telepresence are interactive systems that provide 3D-graphics, and possibly other output to the user. Interconnected Virtual Environments (IVEs) are multi-user distributed virtual environments. 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 each other. The separate VEs of an IVE do work together in a cooperative environment, but the world they represent is tailored to their individual needs and requires mapping to the rest of the IVE. Besides looking at reality differently, these legacy VEs will not necessarily be written with the same programming language, run on the same operating system or hardware, use similar design architecture, or use the same interprocess communication techniques. The component VEs may themselves be distributed or even be IVEs which increases the complexity of the interconnection.

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.[13] [4] *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.[9] None of these systems supports the reuse of legacy VEs.

Three Part Solution Our approach is a three-pronged attack on the problem. We first define an abstract interconnection target for all the VEs. This layer of abstraction shields the developer from the details of interconnection and allows each VE to communicate with only one entity — the runtime. The developer can think in terms of functional requirements for the message transport, not protocols. The details of melding disparate architectures, coordinate systems and programming languages is handled by the runtime. Using an interconnection runtime abstraction as a target allows us to capture the development process with an interconnection generator and description language, the second prong of our attack. Finally, to provide a proper framework for the application of these two tools we have developed a tailored methodology that focuses the developers on the salient characteristics of the IVE and allows them to exploit the power of the environment.

Example Problem VEs offer a chance to prove our techniques work on sticky domain specific problems without departing from general purpose interconnection. To illustrate what is required to interconnect virtual environments consider this example. The U. S. Army wants to take existing VR training devices and integrate them into a battlefield simulator that allows small units to practice against each other without the expense of a field training exercise. One training device is for a TOW, a wire-guided anti-tank missile. This simulator's primary mission is training the gunners; no maneuver is involved. The other devices are the simulators for M1A1 Abrams tanks. The goal is to quickly build an IVE in which the TOW gunners can practice against the the tanks as they take evasive action. The tanks have a complicated virtual world which includes the terrain they maneuver over, and other friendly and enemy armored vehicles. The TOW's view of the terrain must generally coincide with the tank's, but is much simpler because they do not maneuver. These two types of VEs share a similar virtual world, but their software is very different.

This IVE showcases a raft of problems that the developers must solve to meet the user's requirements.

Each system uses different event names and requires different message content to react to each event. The coordinate systems used are different: the tanks use geographic coordinate systems while the TOWs use polar coordinates in meters and radians with the origin at the TOW location. Even though each VE appears to use clock time, the IVE requires that the TOW appear in different locations along the tank's axis of advance to maximize training opportunities. Finally, the tank simulators must now coordinate with each other to avoid collisions and contention over moving obstacles in the virtual world.

This presents the programmer of an IVE with the need to integrate disparate VE software into a coherent distributed virtual environment. The legacy VEs that make up the IVE will have events, states and objects that the interconnection software must map appropriately. In addition, IVEs require such standard interconnection services as, dynamic reconfiguration, message passing, data storage and data consistency.

The rest of this paper is structured as follows: we lay out the requirements for interconnecting virtual environments in Section 2. We have organized these requirements into a runtime interconnection abstraction (Section 2.1), an interconnection generator (Section 2.2) and a methodology to put the tools in context (Section 2.3). A discussion of our implementation of the software engineering environment is contained in Section 3, divided into three sub-sections: the runtime interconnection software *Isthmus* (Section 3.1), the interconnection generator *Zubin* and description language *VIDL* (Section 3.2), and our tailored methodology (Section 3.3). We conclude our findings in Section 4 and discusses where we are headed with this research.

2 Software Engineering Interconnected Virtual Environments Requirements

To specialize modern software engineering techniques for IVEs we must first understand the requirements for those techniques in this domain. Before exploring the approach of combining a runtime abstraction, interconnection generator, and tailored methodology to solve this problem, we will describe the requirements of these tools.

2.1 Runtime Interconnection Abstract

Abstractions have proved valuable in making interconnections easier to build.[11] The interconnection of virtual environments can be quite complex, because the legacy VEs may run on specialized hardware using different operating systems. They may also be written in different programming languages that represent data differently. IVEs require a myriad of communication types: fast but not necessarily reliable, reliable, uni-cast, multicast, and broadcast. The runtime interconnection environment serves as an abstract decoupling agent. Hiding the details behind an abstraction improves quality and productivity. Heterogeneity in language and architecture is allowed because all communication is with the runtime environment, not other component VEs. It also makes it easier to take advantage of technology changes such as new communication protocols when maintaining the system.

The general services that you would expect to find in any interconnection domain include these:

- **Dynamic Reconfiguration** An IVE must have the freedom to join and leave the virtual world at appropriate times, and to have different mixes of players without changing the software. The structure of the application cannot be coded into the interconnection software.

- **Objectbase** The interconnection must have an objectbase that provides non-persistent storage, and insures data coherency among the component VEs. It must be customizable to meet the requirements of the specific IVE, and must support completely transparent grouping of virtual objects. [10]
- **Interfaces** Since the legacy VEs will usually be written in different programming languages and run on different systems the interconnection software must include interfaces that minimize changes to the legacy software. For performance and ease of integration the interface module must be written in the same programming language as the legacy VE and not impose more interprocess communication overhead.
- **High Performance** The runtime must provide high performance. Most VEs are pushing performance limits already to meet response time requirements so the runtime cannot induce a system bottleneck. At the very least the interconnection software must be flexible enough to distribute the load appropriately between the component VEs and the runtime.
- **Distributed Implementation** In addition, large IVEs may require that the runtime environment itself be distributed to meet performance requirements. This must be an option when implementing the abstraction.
- **Event Name Mapping** Events in the legacy VEs will not be named the same, and not all events will correspond perfectly with other events in the rest of the VEs. The runtime must handle event mapping.

When focusing on the IVE domain, more specific requirements for the runtime abstraction manifest themselves:

- **Partitioning** Large IVEs require the IVE to be partitioned to minimize unnecessary message traffic. In our example the IVE would be partitioned spatially. Vehicles that are more then 10 kilometers apart in the virtual world would not need to be kept up-to-date on each other's location.[6]
- **Multiple Protocols** The runtime interconnection software must provide very fast — usually unreliable — messages along with reliable messaging. A message with a location update would be sent using a fast protocol. If the message was lost, the next update would follow quickly and not be disastrous to the system. A message announcing a "kill" on a tank would, have to be sent reliably, regardless of speed. One component VE thinking that a tank was alive, while another believed the tank had been destroyed would be disastrous for the IVE.[7]
- **Temporal Mapping** Both system performance and design can induce *anachronisms* or time errors. The runtime interconnection software must provide mechanisms to minimize these effects or eliminate these errors. Since our example IVE is used for two different purposes, simulation time will not always match. During one lengthy exercise for the tanks, numerous TOW gunners could be trained by "teleporting" the TOW around the virtual battlefield.[3]
- **Synthetic Events** Some events may be created in the interconnection that affect the IVE. These events must be recognized and handled. A simple example is a collision between two tanks. The tank simulators were designed so that the only tank under human control was the simulator itself. In the IVE some of the other tanks are under human, and therefore, unpredictable control. The IVE would have to detect a collision and spawn the appropriate event and messages.
- **Prioritized Messages** By its nature an IVE creates many messages. Processing these messages takes cpu time and if enough messages build up in a VE's queue it may process a message too late and induce an inconsistency. The IVE must provide a message queue to the VEs with the most important messages first while still maintaining causal relationships.
- **Coordinate Translation** Each component VE may

use several coordinate systems. The runtime must translate locations so that component VEs always receive locations in coordinate systems that they can handle.

2.2 Interconnection Generator

The interconnection generator must capture the process of interconnecting VEs and allow the developers to reuse this process. Once the specification of the IVE is complete and basic design decisions have been made the interconnection generator must create the interconnection software.

The input to the interconnection generator is an IVE specific description language. This language must be as non-procedural as possible. Code escapes are unavoidable in some areas but their use must be minimized. The language must be closely linked to the runtime interconnection abstraction and provide an integration framework while allowing the developer to focus on major functions of the interconnection by themselves.

The requirements and therefore the specifications will probably be both incomplete and incorrect the first time. An cycle of analysis – specification/design – implementation – test may be iterated many times during the life cycle of the IVE. The description language must support this cycle. It must create a testable interconnection from incomplete descriptions and create the interconnection with a minimum of hand-coding so that most work will be captured in the descriptions and not lost with each iteration.

2.3 Methodology

To effectively use these tools requires a tailored methodology that discovers the necessary information so that the developers can create the IVE specification. This methodology then guides the use of the tools and provides a framework for the IVE throughout the rest of its life-cycle. The methodology must take into account the unique features of connecting legacy systems as well as the particulars of IVEs.

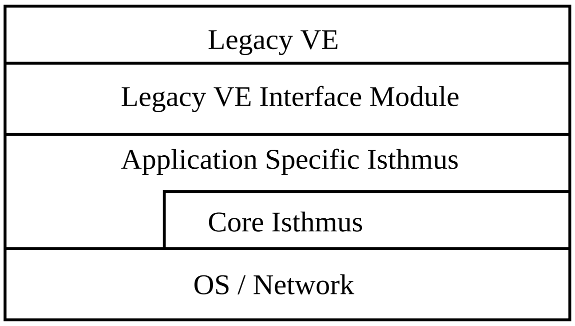

Figure 1: Layers of a Interconnected Virtual Environment

3 Project Overview

To develop our ideas on software engineering of virtual environments (SEIVE) we built a runtime interconnection environment, description language and interconnection generator, and developed a methodology to guide their use.

3.1 Isthmus – Interconnection Runtime Environment

We have developed *Isthmus*, a reification of the runtime interconnection abstraction. *Isthmus* is inspired by the Polylith software bus.[11] Each instantiation of the *Isthmus* contains the code necessary to provide both the general and IVE specific services. Each component VE has its own interface module. All communication in the system is between the interface modules and the *Isthmus*. This double layering means that the interaction between the legacy code and the interconnection software is minimal and simple (Figure 1). The *Isthmus* can be thought of as a software bus much such as a hardware bus except that the software bus does not just pass messages, it performs a host of services while passing the messages.

In its simplest sense when the *Isthmus* receives a message from a component VE it relays the message to all the other component VEs. The *Isthmus* has its own naming convention for all events in the IVE, therefore, the interface modules map the component VE event name to the IVE event name.

The *Isthmus* must also keep track of the system state in its objectbase. This way the *Isthmus* can handle whichever storage model the component VEs use; messages can include just changes or the complete state. Because the message content will normally have to be translated each time a message is sent or received, minimizing the content is desirable because it enhances performance. In addition to virtual objects, global system states are also tracked in the *Isthmus* objectbase.

In addition, when interconnecting legacy VEs, we want minimum intrusion into the legacy code, therefore only sending changes to the system state as opposed to collecting a complete state prior to sending a message better meets our needs. For these reasons a complete system state will have to be maintained on the *Isthmus*.

The *Isthmus* objectbase uses a simple locking scheme to maintain data coherency in the system. If our tank tries to move an obstacle that another tank is already moving the *Isthmus* will not allow it. The *Isthmus* is not sophisticated enough to handle simultaneous manipulation of virtual objects. This is a hard problem with no universally accepted solution.[12] [5]

Since each component VE has its own coordinate system, locations must be translated before they are received by the legacy VEs. Where this translation takes place is a performance decision. By placing the translation burden on the *Isthmus*, the component VEs are spared this interconnection overhead. However, this means that multi-casting 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 multi-casting, however, the systems running the component VE must be able to handle this sometimes computationally intensive load. The *Isthmus* can also be built with a hybrid system, where some messages are multi-casted and translated by the interface modules and some messages are translated by the *Isthmus*.

To minimize overhead, determining the legality of changes to the system state will be done in the component VEs. However, when legality cannot be completely determined in the component VE, it will use synchronous messages to "ask permission." If the system state is legal, the *Isthmus* sends back a confirmation message. If not, a number of options are available to include the generation of other events. If in our example a tank tries to go through a hill, the component VE knows the physics and will not allow this but, if two tanks try to move to the same location the *Isthmus* will have to handle it.

Grouping of objects is also provided by the *Isthmus*. When virtual objects are grouped either permanently or temporarily, that grouping is transparent to the component VEs. For example if the tank moves its turret does too, which is also in the objectbase. Should the tank pick up a soldier all behaviors of that tank and actions on that tank are reflected in the state of that soldier until he dismounts the tank. Some object groupings will impose different behavior and attributes of the grouped objects. A tank carrying mine-clearing equipment looks and behaves differently from one that is not.

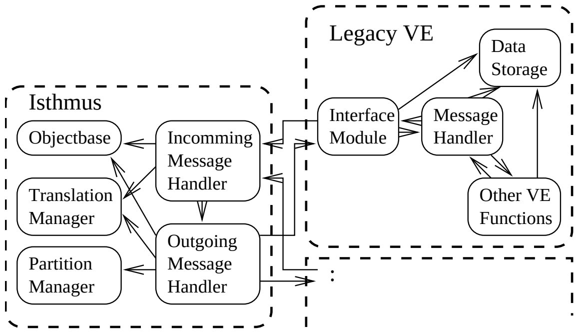

Figure 2: Dynamic Architecture View of Isthmus and Typical Legacy VE

Figure 2 illustrates the objects and message paths between the *Isthmus* and legacy VE. For simplicity this illustrates the *Isthmus* when it runs on only one system; it becomes a bit more complicated when the *Isthmus* itself is distributed. The legacy VE may be designed using a structured architecture and programming language, but even so we find it convenient to model all VEs in object-oriented terms.

To illustrate the workings of the *Isthmus* we will look at the example of a tank simulator moving in the virtual world. As it moves it sends a location update message only to the *Isthmus*. The *Isthmus* receives the message and checks to insure that the change in location is legal (no collisions with other objects in the system) and sends a confirmation message back to the tank. It will then store the new location and determine which other component VEs must be told about this change to the system state. Only component VEs with objects located in the spatial area of interest must be notified. For those, the *Isthmus* will translate the coordinates to the appropriate coordinate system and send the message using the appropriate communication protocol and priority.

3.2 Zubin and VIDL: 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 process can be captured and reused by using 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 the whole application is not being built, but merely the software that glues together the legacy applications. [8]

We have an experimental interconnection generator called *Zubin* for this domain that uses our domain specific description language dubbed *VIDL* to generate the source code for the *Isthmus* and interface modules. *Zubin* uses the *Isthmus* core runtime and adds customized services to it as needed. It also generates the interface modules in the native programming language for the component VE. The design of *VIDL* allows developers to concentrate on the different functions of the IVE independently while the description language and *Zubin* provide the integration framework much like CDSL does for more general applications.[2]

VIDL is designed to create readable source code. We acknowledge that after *Zubin* is run, the developer will have to look at the generated source code and possible modify it. Having understandable source code will be very useful when deciding where in the legacy VE to place the calls to the interface module. Therefore the software engineer can declare user-defined constants in the description language that will end up in the source code. *VIDL* also supports two types of comments: description comments and source code comments. The source code comments are placed in the generated source code by *Zubin*.

The *VIDL* description gives a mostly non-procedural description of the software glue that holds the IVE together. It has sections for declaring the configuration of the IVE. 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 reason we departed from a purely non-procedural specification to a mesh of specification and design was we felt the need to support iteration. The first specification of the IVE can contain very little design. As a prototype is built and information is discovered, that information can be captured in the *VIDL* and the next iteration can be built and tested.

Interconnection and configuration information for the *Isthmus* are described in the *Interconnection Section* of the description. Whether or not the *Isthmus* is distributed, where it resides is described here. In addition, the initialization information required for component VEs to join the IVE is also related here.

The *Component VEs Section* 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.

Partitioning is described in the *Areas of Interest Section*. Areas of Interest may be declared based on spatial constraints (the most popular), functional designations, or temporal constraints. This partitioning — to reduce message traffic — can be handled by using multi-casting which is useful in large applications, or by the *Isthmus* selectively sending messages.

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 parameterized 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.

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 based on the event. 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. Temporal mapping is accomplished here by using the *ADJUST* declaration. It is another code escape, because by its nature temporal mapping can be very complicated and application dependent. Before the message is sent the commands in the *ADJUST* algorithm are executed. Those commands may include the generation of additional events. The use of these recursive events gives the engineer great flexibility in handling this onerous problem.

The example event declaration (Figure 3) shows the messages that are passed to relay the information that a tank has moved. The tank sends a message to the *Isthmus* requesting a move to a location using its geographic coordinates.[1] If this is a legal move the *Isthmus* will send a message to all TOWs that are within the same spatial area of interest. This message will not require confirmation and will be in the TOWs polar coordinate system.

The *Virtual 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. In the tank descrip-

[1]The Military Grid Reference System (MGRS) geographic coordinate system allows location specification down to one meter with two five-digit coordinates.

```
evLocation :
  MESSAGE SYNCHRONOUS status;
  FROM enTank;
  PRIORITY REGULAR;
  RELIABILITY RELIABLE;
  CONTENT (INT vehicleId,
   STRING area,
   INT x_coord, INT y_coord,
   FLOAT orientation);

  MESSAGE ASYNCHRONOUS;
  TO enTow SPATIAL;
  PRIORITY REGULAR;
  RELIABILITY UNRELIABLE;
  CONTENT (INT vehicleId,
    FLOAT direction,
    INT distance);
```

Figure 3: Example Event Description

tion below we are able to say that two vehicles can't occupy the same space, as opposed to exhaustively listing all the objects in the simulation that can't occupy the same space. The objects are categorized as active (they can manipulate other objects), inactive (they can be manipulated by other objects) or environmental (they cannot be manipulated by other objects). Environmental objects will not normally be listed; interaction with them will be managed in the component VEs. The attributes of the objects are listed along with the other objects that it can be grouped with. When listing the potential grouping objects, the attributes that change when the objects are grouped is delineated. The coherency rules are declared here along with a list of events that are fired when a coherency rule is violated.

A simple example of the tank virtual object description is shown in Figure 4. The *COORDINATE SYSTEM* declaration indicates that the locations of all objects will be stored using the *csMGRS* coordinate system[2]. The *HOME VE* designates the *enTank* component VE as the controlling VE for the tank. The attributes that are important to the IVE are listed and in this case it is just the size of the tank itself. There are numerous default attributes generated such as a unique identifier that do not have to be declared here. The *HOLDS* declaration tells the software that the turret object is always grouped with a tank object. The coherency rules are declared in a C++ code escape and the *collision* event that is fired in the case of a coherency violation is listed.

States that are part of the IVE but don't belong to a virtual object are declared in the *Global States Section*. An example of global states are things such as weather or lighting conditions. Simulation time is another global state. If a central clock is to be kept then it is declared here.

The example global state declarations show one simulation clock declared that starts at 12:00. It advances with clock time by default. Two global conditions are declared with their possible states and the default state.

3.3 SEIVE Methodology

Our tailored methodology provides the framework for the tools and runtime environment. Because of the nature of IVEs this methodology differs from standard software engineering. The division into phases is more functional than chronological, although there is a causal ordering for some of the steps. Here is a quick overview of the phases (Figure 6), however the details of the analysis and the products of each phase are the topic of another paper.

[2]This coordinate system is described in the translation section.

```
COORDINATE SYSTEM : csMGRS;

OBJECT oVeh :
 ATTRIBUTES :
  INT x_coord;
  INT y_coord;

OBJECT oTank ACTIVE INHERITS FROM oVeh :
 HOME VE: enTank;
 ATTRIBUTES :
  INT size;
 HOLDS oTurret;
 COHERENCY RULES :
  {if (($x_coord + $size) >=
     ($all.oVeh.x_coord - $all.oVeh.size)
    && ($x_coord - $size) <=
     ($all.oVeh.x_coord + $all.oVeh.size)
    && (y_coord + size) >=
     ($all.oVeh.y_coord - $all.oVeh.size)
    && (y_coord - size) <=
     ($all.oVeh.y_coord + $all.oVeh.size))
   return FALSE;
  } VIOLATION evCollision;
```

Figure 4: Example Virtual Object Description

```
TIME : tCurrentTime = 12:00;

STATE : esTimeOfDay (day, twilight,
                     night) = day,
      : esWeather (clear, rain, snow,
                   fog) = clear;
```

Figure 5: Example Environmental State Description

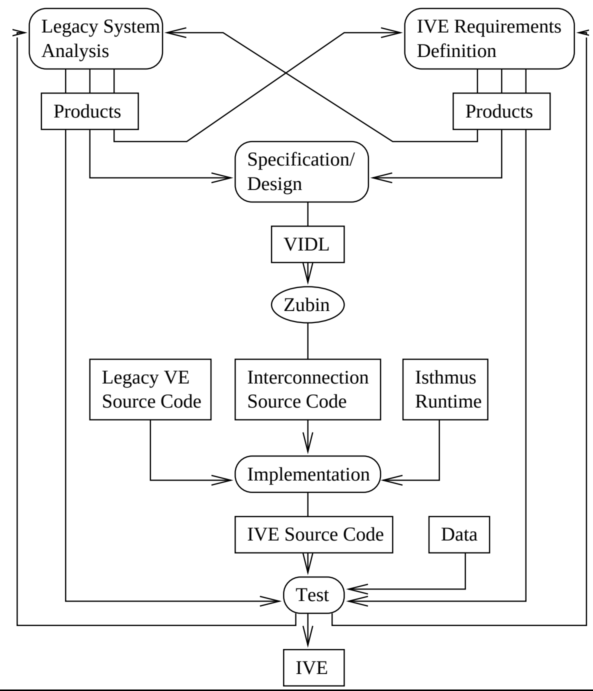

Figure 6: Flow Diagram for IVE Methodology

1. **Legacy System Analysis** This is the analysis of the legacy virtual environments. The architecture of the legacy software as well as the function of the VE must be understood. Some of the products of this phase are: a list of behaviors of the VE, a list of environmental states, a table of virtual objects and their grouping and coherency rules, and a table of events.

2. **IVE Requirements Definition** During this phase the results of Legacy System Analysis are synthesized into IVE wide products. The other step of this phase is to discover and document the desired characteristics of the IVE. This is much like the requirements analysis of new development. Parts of this phase rely on Legacy System Analysis, however some of that phase can't be properly done without knowledge of what the user wants the final IVE to do.

3. **Specification/Design** The products of the Requirements Definition phase are used to write the

VIDL description of the IVE. Along with specifying what the system should do some design decisions are made during this phase.

4. **Implementation** In this phase the *Zubin* is run, and the resulting interface source code is manually linked to the legacy VE.

5. **Test** There are assessments of the software that must come from the test phase: first does the IVE meet the specifications, second does the IVE meet the requirements, finally does the IVE do what the user wants, which may not have been captured by the requirements.

When building an IVE the focus can be towards interconnecting the existing VEs so that they perform basically the same functions, or to use existing VEs to build a system with each VE in the IVE more capable than it was. Our discussion has been more involved with the former, but this methodology supports the latter too, freeing the developer to deal with the VEs themselves.

4 Conclusion and Future Direction

The combination of a tailored methodology, interconnection generator and interconnection runtime appears to hold promise for developers of IVEs. Our experience to date shows that it requires less effort than interconnecting by hand, while providing higher quality. The *Isthmus* based architecture simplifies programming the interface to the point where we can automate the process of building the interconnection software. Automating this process allows the developer to focus on the larger conceptual problems involved in creating the IVE. Our continuing experiments in the IVE domain serve to refine the methodology, description language *VIDL*, interconnection generator *Zubin* and interconnection runtime *Isthmus*. In our on-going research, we anticipate exploring the use of these tools to support interconnection in other challenging domains, with the ultimate intent of contrasting and generalizing the techniques.

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] 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.

[3] James W. Duff, James Purtilo, Michael Capps, and David Stotts. Software engineering of distributed simulation environments. In *Proceedings of the Conference on Configurable Distributed Systems*, pages 202–209, Annapolis, Maryland, 1996. IEEE Computer Society Technical Committe on Distributed Processing.

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

[5] Mark Green. Shared Virtual Environments: The Implications for Tool Builders. *Computers and Graphics*, 20(2):185–189, March/April 1996.

[6] Michael R. Macedonia, Michael J. Zyda, David R. Pratt, Donald P. Brutzman, and Paul T. Barham. Exploiting reality with multicast groups: A network architecture for large-scale virtual environments. In *Proceedings of the IEEE Virtual Reality Annual International Symposium*, pages 2–10, Research Triangle Park, NC, March 11-15 1995.

[7] Micheal Macedonia and Micheal Zyda. A taxonomy for networked virtual environments. In *Pro-*

ceedings of the 1995 workshop on Networked Realities, Boston, MA, October 26-28 1995.

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

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

[10] Przemyslaw Pardyak and Brian N. Bershad. A group structuring mechanism for a distributed object-oriented language. In *Proceedings of the International Conference on Distrubuted Computing Systems*, pages 312–319, Poznan, Poland, June 21-24 1994.

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

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

[13] 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.