

**VIRTUAL ENVIRONMENT ARCHITECTURES:
INTEROPERABILITY THROUGH SOFTWARE
INTERCONNECTION TECHNOLOGY**

P. David Stotts
University of North Carolina, Chapel Hill

James M. Purtilo
University of Maryland, College Park

Author contact information:

Prof. David Stotts
Dept. of Computer Science
CB #3175
Univ. of North Carolina
Chapel Hill, NC 27599-3175
phone: 919-962-1833
fax: (919) 962-1799
stotts@cs.unc.edu

Prof. James Purtilo
Computer Science Department
University of Maryland
College Park, Maryland 20742
phone: (301) 405-2706
fax: (301) 405-6707
purtilo@cs.umd.edu

This research is funded by contracts with Office of Naval Research.

VIRTUAL ENVIRONMENT ARCHITECTURES: INTEROPERABILITY THROUGH SOFTWARE INTERCONNECTION TECHNOLOGY

ABSTRACT: Virtual reality (VR) is emerging as an important approach to the modeling and simulation of complex systems. But software technology for scientists to build VR-based applications fosters development of closed applications each built from scratch. A scientist's ability to merge models and systems once developed is solely dependent upon their ability to 'hack' software, since the principles of VR system interconnection are poorly understood and no software engineering guidelines have ever been developed for use in VR applications.

We are studying the area of interoperation among VR systems (that is, the virtual environment), in order to discover essential principles governing their construction and effective use. Our approach focuses upon the control properties of interfaces between VR applications: existing VR applications are being examined in order to expose commonalities, and our abstractions of VR control behavior are being specified in terms of the software bus model of interconnection.

We are constructing prototype implementation of new software interconnection tools that embody the principles of VR interoperation we discover; these are used to evaluate application of these principles back in the domains from which our test problems are drawn. As a result of this research, scientists who use VR technology will have a sound basis for leveraging existing resources in new applications, and similarly, they will have an appropriately abstract framework for specifying how multiple models should be merged operationally.

This paper presents our vision for virtual environment infrastructure, and describes the current state of our work in progress. There are several direct applications of our VE exploration to collaboration infrastructure. First, some of the VR systems we are using allow collaborative interactions of multiple users. Secondly, our investigations involve combining the Polyolith software interconnection technology with Trellis, a multi-user, collaborative hypermedia technology.

KEYWORDS: Interoperability, module interconnection, collaboration protocols, virtual reality, formal specifications, software bus

This research is funded by contracts with Office of Naval Research.

1 OVERVIEW

Implementation of a virtual reality (VR) application — whether to support simulation, training or teleoperations — is typically a costly investment, in time and personnel. Focusing upon the software itself, the system may not be large in size, as compared with other industrial applications, but its complexity is certainly large when considering timing, control and display problems to be overcome.

The cost of VR software alone is sufficient for us to consider mechanisms for reusing their systems once built. By establishing architectures for VR development, the community will be able to benefit from commonalities among application components and development tools alike. But even beyond simple reuse of components lies the potential for interoperation between the many VRs being envisioned. Once two VRs applications have been fabricated — one a radar operator trainer and the other a fighter simulator, for example — how can we enable a pilot and radar operator to interact in their respective environments *without* the cost of developing a third entire system? Or, more to the point of this workshop, how can we enable *several* concurrently active instances of the flight simulation VR system to interact with each other, and to all interact with the radar operator VR system as well?

As another example, consider an architectural VR system built for visualizing new ship designs; it allows an individual to construct a 3-D model of a new ship and to "walk through" the ship, experiencing the design from a ship-dweller's perspective. Consider also a second VR system built to simulate a group conference; it allows a collection of individuals physically remote from one another to experience a face-to-face meeting in a virtual 3-D environment. How can we combine these two systems (without the expense and time required to development a separate third system) so they interoperate, allowing a group of physically separated people to virtually walk through a ship "together," as a group?

Adequate answers to these questions require more than just arranging for compatibility between the underlying programs and communication systems. We refer to the framework that enables multiple VR applications to interoperate as a *virtual environment*, or "VE", and note that the task of merging VEs themselves is indeed more than the simple splicing of a few program units: the developer must understand relationships between the two systems in terms of essential data structures; representation of knowledge or choice of schemas for large databases; type of data being exchanged; assumptions of event ordering and causality; timing, synchronization and the protocols for interaction and communication; and also software dependencies inherited from choice of underlying hardware and communication media.

Especially when the original VEs have been developed separately, we need a mechanism that will help us relate the *abstractions* from each VE to one another. Only when we can efficiently synthesize a model of the two can we also achieve reliable and low-cost interoperation between the underlying systems.

The promise of interoperation between separately-derived applications in virtual environments is

very ambitious. When researchers are able to unite diverse models, they will also be able to unite ideas and inspirations. They will not be forced to build from scratch every aspect of the domains in which they work. Besides the usual promise of reuse of software (lowering cost of development, as key components need not be redeveloped), there is also the reuse of key *requirements*, that need not be rediscovered. This is especially important in areas where the application may involve safety requirements.

But behind the promise is a sad reality, that the body of computing science does not yet fully contain the principles behind integration of abstract models – principles that would guide users in building VR applications. Currently VR applications are built from scratch (hacked), and reuse of software is only coincidental (relying upon the programmer to remember lines of code that can be edited into some new use.) The focus of integration is solely in terms of the code employed; it is enabled only by construction (by programmers enforcing use of a common language, set of data structures, and communication mechanisms.)

The challenge addressed by our research is to derive those principles of VE development that would guide programmers in construction of VR applications. Programmers should be given software architectures within which components they build can be both used and reused; they should be given analysis tools sufficient for verifying that critical requirements in an integrated VE are maintained; and should have expressive means for prescribing the interaction they desire between multiple VR applications in a composite system. Once a configuration of VR components has been rapidly and abstractly expressed by the programmer, a rich software infrastructure should enable generation of a valid implementation automatically. Moreover, the clarity of design and correctness of automatic implementation must not come at the expense of any run time performance.

Our approach to this task is to study example VR systems looking for commonalities, make abstractions and design an appropriate architectural VE framework from this study, and to design VE specifications and structure analysis methods based on existing interconnection technology and existing formal specifications and process flow work. Our premise is that the interconnection of VE models can be accomplished by use of an enhanced *software bus* abstraction. A software bus model of software interconnection organizes interfacing decisions in such a way that they can be easily reused and leveraged across many applications and domains. Currently bus implementations are helping users accommodate heterogeneity in choice of programming languages, host operating systems and communication mechanisms; bus-based interconnection systems are now being used by ARPA's Prototyping Technology program and also the Domain Specific Software Architectures program. We are extending this model of interconnection to accommodate the additional dimensions of concern to VE developers, in particular the models which drive distributed interactive simulation.

This paper presents our vision and preliminary progress to date, towards understanding the principles of software infrastructure underlying domain specific VEs. In the next section, we motivate the study with some example scenarios, and afterwards lay out our specific research activities. Following that we present background material on technologies contributing to this research, on both the software bus model of interconnection and (Trellis).

2 MOTIVATING EXAMPLES

Nobody can really leverage multiple VRs together yet, so there is not much to look at that exactly illustrates the key problems. But it is our vision that people should be able to leverage multiple tools within a VE. Users and researchers remote from important resources should be able to inject their model (and all the knowledge and operational capabilities it brings with it) into someone else's universe, and have "the right things" happen.

But we don't have to look far to find analogous problems. We illustrate some of the issues being addressed in terms of a simple integration problem between multiple simulators.

In order to illustrate the issues driving our research, consider the problems of interconnecting two VR applications. Consider a VR-based flight simulator, and a VR-based radar operator system used for training. What are the problems to be overcome in integrating the two, so that pilot actions will be reflected upon the radar operator's "screen", and that the operator's analysis and directions can be used to guide the pilot?

- **Data relations:** One program may have embedded within the code itself, but the other may be driven by a backend database system. The very manner by which important data values themselves are accessed by the two programs may be very different.
- **Data representation:** However data values are accessed, they may have very different representation within the respective computer systems. The data structures important to the pilot system may be a polar coordinate system, geared to helping a graphics display quickly update the image of "external" features (e.g. other aircraft or missiles, and ground features such as an airstrip or radar control area) with correct perspective during flight. Yet the radar trainer system may represent data as records in a database, indexed by a sequence number for fast retrieval — based upon the operator's actions, different sequences of simulated events may thus be portrayed on his screen. An integration of the two systems would need to resolve such conflicts in data representation.
- **Control representation:** Less understood than issues of data representation are those of control representation. Illustrating this, the pilot system may be designed so that images are presented to the graphic display using data streams (a continuous sequence of primitive data); or the pilot system may have each event communicated to all the necessary software components using a broadcast communication paradigm. In contrast, the radar trainer may consist of a centralized computer system that accesses subsystems using a procedure call paradigm. Interconnecting these two VR applications will require existence of a 'software glue' that can accommodate translation between the two paradigms; and that in turn will require a robust control specification technology for even deciding what the interactions should be at all.
- **Event mapping:** With each of these systems running separately, many "events" that occur are of course simulated by the computer. But in an integrated reality, events from one system must be mapped and translated for presentation as stimulus to the other system.

Because most VR systems are currently built *without* an architectural orientation, the mere act of identifying which abstract events could be mapped to another reality can be a difficult task. How the events are named and then redirected will be an important issue to sort out.

- **Time mapping:** The pilot system may be a true real-time system, in that the ‘simulation’ time and ‘real’ time are intended to coincide. (Clearly, a pilot in air combat should never see a message on his HUD saying “Targeting subsystem is garbage collecting ... please wait.” Hence, he should not train in a system that might say such a silly thing.) In contrast, for training purposes the radar operator may operate in a virtual time frame, that is, long idle periods may be speeded up, or busy periods slowed down, for purposes of education or elucidation. Finding an abstract way to characterize these two extremes, and then arbitrate a common time scale between them, will be an important task, and one affecting how we set up our control specification framework.
- **Synchronization:** In the same way that the time scales between multiple VR applications must be mated, so must the interconnection system provide for identification and ordering of important events. The radar operator should not be shown any images of a bogey in his airspace after the pilot has successfully neutralized the threat.

Indeed, many technical issues arise in considering the interconnection of two existing systems such as above. Presented as such, it may actually seem more cost effective *not* to deal with interconnection of these systems, and to instead develop a third new system (containing the functionality of these two) from scratch. Therefore, we should also illustrate what are the *benefits* from merging these two systems as well.

- **Low cost:** *If* a robust interconnection technology is found to be adequate for merging existing systems, then developers will not have to pay the extensive price of *rediscovering the expertise* embodied in the existing programs. What is important about VR systems and simulators is not just the code for display or interaction, but rather the *knowledge* that the code embodies — the model of the system, as expressed in an operational form by programmers. Throwing out the code will often find you throwing out the accumulated experiences with the system as well.
- **Large scale:** As more ‘realities’ can become integrated by the software, more human experts may participate simultaneously in large problem solving efforts. Larger scale simulations are enabled, and more tools are made available to people in the field than would have possible using single reality technology.
- **Broad applicability:** Once a VR application has been developed for one scenario, the interconnection infrastructure should make it more broadly applicable, or at least accessible. Once an expert system using VR for fault diagnosis in the field has been developed, and should some particularly hard problem be encountered, then interconnection technology for VR systems should allow the human experts — who may be remote from the component being studied — to become just as involved in the problem as the technician. The remote expert should be able to experience the same images and

- **New functionality:** Perhaps the most important aspect of interconnection is that it simplifies creation of ‘realities’ that are not currently feasible, and by this we refer to realities involving multiple agents. After a VR flight simulator for pilot training has been developed, interconnection technology should be able to link several of these simulators into the same ‘reality’, for group training. But because of this technology’s ability to accommodate heterogeneity, we can also envision creation of “instructor realities”, to be linked into the group simulation. Users playing these roles may be free to fly through air combat simulations *without* the bother of simulating an aircraft, invisible to the other participants, and not subject to the usual laws of physics: in real time, the instructor as ‘ghost’ would be free to observe the simulation as if a participant, but from perspectives not normally available, hopefully improving the quality of training and evaluation.

In the other scenario mentioned earlier — the “group walk-through” system — we have similar problems. Again, instead of traditional (simple) data types and data structures needing to be exchanged, we have data *and behavior* from each system that must be understood by the other. For example, in a ship walk-through system, we not only have data that will define where walls are (physical coordinates in 3-space), but we have other information that will tell whether the walls are solid (realistic) or permeable (a feature of a VE that surpasses reality). In a group conferencing system, we not only have data telling what participants look like, and who is currently speaking, but we also have protocols that define the allowed interactions among participants (like Robert’s Rules of Order). When combined, we have interesting interactions among these behavioral constraints. For example, in a group walk-through VE, an interaction between two group members that is permissible in the normal conferencing environment might be inadmissible “inside” the ship VE if a “solid” wall stands between the two group members.

Data plus behavior is a (simplified) definition of *object*; our work must produce, then, specification methods for defining the salient features of the objects in each VE system to be combined. In addition to objects, the relationships among objects must be encoded in a way that can be exploited by both execution environments and by analysis techniques. The existing software interconnection technology we are building on has primarily concentrated on *interface* specifications, where the interfaces are static data descriptions. The expansion into *behavioral abstractions* as interface enhancements is open for scientific study and design of good techniques.

Before significant development can be done on interconnecting VEs, before major new systems can be assembled by reusing existing systems, and before developers will become more efficient in producing VEs for general use, the principles that should guide such developments must be uncovered, codified, and established in an appropriate interconnection framework. These principles, largely unknown at this time, will be a major contribution of our work and will produce a more complete understanding of the nature of virtual environments. While we believe we have a nucleus of the technology that ultimately will provide the leverage we envision, we are equally sure that some new concepts and new techniques will have to be developed to augment the current technology (which is based on programming language interconnection, and collaboration protocols). This identification of principles will proceed from the obvious point: a detailed study of good existing examples of VEs.

Figure 1: Trellis client/server system architecture.

3 BACKGROUND: COLLABORATION AND INTERCONNECTION TECHNOLOGIES

Our project in VEs is leveraging several existing technologies. Basically, we are studying current VR applications (at the University of North Carolina, and also at the Naval Research Laboratory in Washington D.C.) looking for common elements that can serve as basic abstractions for interoperable environments. Around this base of common abstractions we are building specification and interconnection techniques for use within the current Polyolith framework. Existing VR systems will have their behavior extracted, abstracted, and specified with methods based on application of Trellis, a process-based collaboration protocol prototyping system, similar to a current project called IDTS. The Trellis (IDTS) and Polyolith projects are briefly described in the following sections.

3.1 TRELLIS: COLLABORATION PROTOCOLS AND PROTOTYPES

The Trellis project [StFu89, FuSt89] has investigated for the past several years the structure and semantics of human computer interaction in the context of hypertext/hypermedia systems, program browsers, visual programming notations, and collaborative process models. In this discussion, we refer to an information structure in Trellis as a *hyperprogram*. Due to the unique features combined in the Trellis model, a hyperprogram integrates user-manipulatable information (the hyperlinked structure) with user-directed execution behavior (the process). We say that a hyperprogram *integrates task with information*. In particular, the newest Trellis model integrates *collaborative* tasks with information.

Figure 2: Concurrently displayed Neptune images classified by features.

The Trellis model treats a hyperprogram as an annotated, colored, timed, place/transition net (PT net) that is used both as a graph (for static information) and as parallel automaton (for dynamic behavior). We will assume in this paper that the reader has some familiarity with PT nets; for background on PT net theory, the Trellis references given can be consulted. In Trellis, the places of a PT net are annotated with fragments of information (text, graphics, video, audio, executable code, other hyperprograms); these annotations are termed the *content* elements of the hyperprogram. A hierarchy is created by allowing the content element of a place to be itself an independent hyperprogram.

To use a Trellis model (as hypertext, for example) we provide visual interfaces (clients) to give a user a tangible interpretation of the PT net and its annotations. When a token enters a place during execution of the engine, the content element for that place is presented for viewing or other consumption in the client. Any enabled transitions leading out of that place are shown next to the displayed content element as selectable *buttons*, or hot spots in the client interface. Selecting a button (with a mouse, usually) will cause the engine to fire the associated net transition, moving the tokens around; this leads the client to display new content information, and so on.

Figure 1 shows the high-level client/server architecture of a Trellis-based implementation. Trellis clients and engines execute as independent Unix processes, communicating via RPC. The Trellis engine determines the static structure and dynamic collaborative behavior of a hyperlinked information base; the individual clients determine the context-dependent and user directed presentation of that information.

For example, consider figure 2, which shows two screens from α Trellis, an early Trellis prototype implemented for Unix platforms and the SunView window system. This example is an image browsing index constructed as part of a NASA experiment at Goddard Space Flight Center. Images of Neptune and Phobos are linked and cross linked in the net structure according to common characteristics. The index exists as an instance of the Trellis engine; as such, it is accessed through various Trellis clients. Three α Trellis browsing clients are visible, with the graphical editing client on the top-right, overlapping the windows of the text browsing client on the top-left, and an image-rendering client shown at the bottom (displaying over the Internet on a different workstation running X windows). The index allows collaborating astronomers to share images and annotate the network of data with comments and research results.

Other demonstrated applications of Trellis include a browser for concurrent programs written in CSP [StFu90a] (the basis for our IDTS techniques that we will adapt for the VE work); a simulation system for teaching parallel algorithms, illustrated with a Dining Philosophers network [StFu92]; timed multi-user hypermedia [StFu90b]; and a system for the representation, enactment, and improvement of software development processes [StFu93]. In addition to applications, we have adapted a concurrent program verification technique called *model checking* [CIES86] to produce analysis methods for hyperprogram verification, that is, for demonstrating that Trellis nets exhibit, when browsed, the reader-document interactions that are desired by their authors [StFR92].

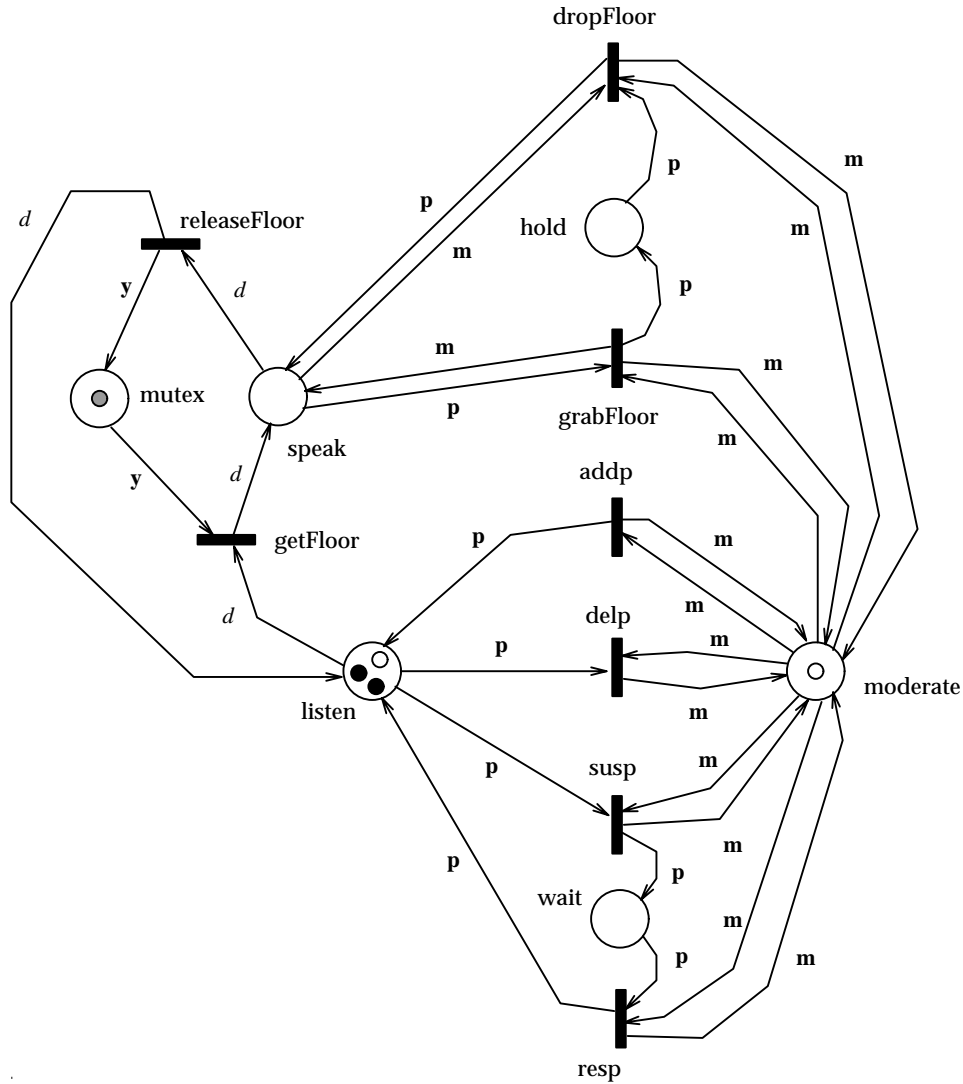


Figure 3: Simple moderated meeting protocol.

Collaboration protocols

Multiple clients concurrently active on one Trellis engine may exchange information via an attribute/value pair mechanism in the engine. The relationships among the different clients is expressed within the net structure as defined by the various token colors and color combination rules on the net components. The full explanation of the use of Trellis in collaborative system prototyping has appeared previously [StFu93]. We briefly illustrate this feature of Trellis here for completeness of the discussion.

Figure 3 shows a colored PT net that encodes the protocol for a simple moderated meeting. In this net, the moderator has one color of token, and the participants have a different token color (in the more complex version of this protocol, every participant has a different color). The protocol allows the addition and deletion of participants from the meeting; it allows participants to get the floor for speaking, and it allows the moderator to grab the floor from a speaking participant. Behavior of the net with colors when executed is controlled by the expression on the net arcs.

xTrellis

Max tokens/place = 12

red1 = <5>

SELECT TOKEN COLOR

black	magenta1	green1	pink1	maroon1	red1	orange1	purple1
blue1	magenta2	green2	pink2	maroon2	red2	orange2	purple2
blue2	magenta3	green3	pink3	maroon3	red3	orange3	purple3
blue3	magenta4	green4	pink4	maroon4	red4	orange4	purple4

TOKEN: [●] [🔥]

PLACE: [○]

TRANSITION: [—]

TRANSITION: [|]

ARC: [↘]

CUT: [✂]

```

graph TD
    mutex((mutex)) -- pink1 --> speak((speak))
    speak -- pink1 --> mutex
    speak --> grabFloor[grabFloor]
    grabFloor --> hold((hold))
    hold --> dropFloor[dropFloor]
    dropFloor --> speak
    dropFloor --> resp((resp))
    resp --> addp[addp]
    addp --> delp[delp]
    delp --> swapMod[swapMod]
    swapMod --> susp[susp]
    susp --> resp
    speak --> releaseFloor[releaseFloor]
    releaseFloor --> speak
  
```

NET ID: [NET ID] [NEW]

moderate

addp
delp
susp
swapMod

===== M O D E R A T E =====

You are the moderator and you are now in a moderating state.

You should be able to grab the floor even if someone else is speaking.

You can also add, delete or suspend participants.

listen

getFloor

===== L I S T E N =====

You are either a participant or the moderator, and ...

You are in a listening state.

You should be able to get the floor if nobody is speaking by selecting the corresponding button at the left.

Other transitions that might appear are not relevant in this state.

SELECTO SELECTO Console

Figure 4: Prototype of moderated meeting in Trellis.

Italicized symbols are color variables; Roman font indicates color constants. For example, the arcs coming into the transition labeled “addp” indicate that a token of color “m” (moderator color) is required in place “moderator” to invoke the *add participant* action. When invoked, the action will place an “m” colored token back in place “moderator” and will also deposit a token of color “p” (participant color) into place “listen.” In contrast, the transition labeled “getFloor” has a arc coming in from place “listen” marked “d”; this indicates that any color is acceptable to fire the event, and when fired, the getFloor action deposits a token of that same color “d” in the place “speak.”

A Trellis prototype (in the new χ Trellis system) for this meeting protocol is shown in figure 4. Several X-windows-based clients are shown interacting with one engine. One client is the xTed editor, showing a graphical representation of the engine structure. The text windows come from two instances of the browser client, one for the moderator color and one for the participant color. Buttons are visible next to the text windows to allow invocation of engine actions hypertextually.

Using the attribute/value pair facility of the engine, some actions are hidden in the participant interface (for example, the “delp” action requires tokens from both the “moderator” place and the “listen” place; however, we do not show the button for “delp” in the participant client, so only the moderator can actually invoke the operation).

3.2 PROJECT POLYLITH

Heterogeneity is a natural result of the diversity in problems we attempt to solve; the promise of increased performance leads us to specialize the tools we employ. But this diversity challenges us as well: it limits our ability to combine tools with one another. Programs written in different languages cannot be interfaced easily; data produced on one computer architecture may not be represented in a way that is usable by programs on another architecture; and differences in operating systems may prevent programs in a network from cooperating to solve a problem, even though there are physical connections between the hosts. Whenever we are inhibited from leveraging our tools, then we spend our time *re-solving* old problems instead of solving new ones.

The Polyolith Project demonstrates that not only can programmers build applications for heterogeneous computing environments, but they can do so without giving up the apparent simplicity of homogeneous environments. Our software organization allows programmers to create applications whose components are written in different languages; distribute their programs across a network of diverse computers having different operating systems; and vary the choice of media or protocols used for communication among the application components. Even though each of these activities has been addressed separately in the past, our research unifies the capabilities and demonstrates how all can be made available simultaneously.

The Polyolith research results are expressed in terms of both distributed systems and software engineering. Each choice of programming language, architecture and operating system defines a unique software *domain*, e.g., the ANSI C language on a Vax running 4.3 BSD Unix fixes one particular domain. A *module* is a source program unit drawn from a single domain, e.g., a C function from the above domain could constitute a module. A *heterogeneous program* (or *configuration*) is an application built from modules drawn from different domains.

In order to construct a heterogeneous application, programmers first need a basis for understanding the abstract relationships between domains. There are many relationships to consider. Each domain has its own type model (e.g., even the semantics of `integer` can be very different between procedural languages on the same host); its own way of representing data (e.g., `integer` uses a certain number of bytes in memory, having a fixed order and encoding, all of which may be incompatible with a different host’s architecture); its own control model (e.g., the way execution proceeds, such as by order of statements in imperative languages, or by searching as in logic-based languages); its own way of transmitting parameters; its own I/O conventions and assumptions; and so on. Without a comprehensive framework to organize these issues, programmers must employ ad hoc techniques for interfacing. For example, if a Fortran programmer wants to call a Lisp-based tool elsewhere in the network, then he must deal with *both* Fortran and Lisp data types; he must know the representation of data on each architecture; he must know how to exe-

cute a remote process through the network by establishing communication channels; and he must know how to trap a Fortran subroutine call and transform it into a remote function invocation. In general, building a heterogeneous program without support is costly, prone to error and one that few are either willing or able to do. For these reasons, programmers *reimplement* modules (hence regaining homogeneity) instead of dealing with heterogeneity.

Fortunately, differences between domains have been considered in previous research, and mechanisms have been developed to help overcome the barriers imposed by heterogeneity. Many previous systems provided some form of interconnection capability [BDW89, HaMS88, HaNo86, JoRT85, LiSh88, MaKS89, NoBL88, Perr89, Snod89, WLRT91]. However, their foci were on the interconnection mechanisms themselves (such as data coercion operations or communication protocols) and not on the software engineering environment in which the mechanisms would be employed to solve large scale problems. There are some fundamental engineering concerns that previous systems do not (and, to be fair, were never intended to) address. Specifically, either the systems support only one form of interconnection, or they force their programmers to choose their interfacing mechanism at the same time they implement each module. The former is too restrictive for general application, and the latter distributes configuration information among each component, increasing its complexity. For example, programmers may need to program non-local data references differently depending upon where the data resides; they may need to provide code to marshal data as parameters and to make specific system calls depending upon the nature of the communication medium used for accessing a non-local resource; and they may need to provide code to assist in the coercion of data to match the representation needed by other modules in the application. The more details that a programmer must track, the greater is the likelihood of error. This increases development and maintenance costs, and reduces opportunity for software reuse.

We have devised an interconnection framework that *separates* interface programming from intra-module programming, while still providing access to the mechanisms that make heterogeneous programming possible. This model is the *software bus* organization. Intuitively, a software bus presents a standard interface into which modules may be “plugged.” In the same way that a hardware bus presents a standard for electrical characteristics and signal protocols — so boards consistent with that standard may be plugged together — a software bus interconnects software components whose internal properties may remain private as long as their interfaces match the bus standard. It is easier to interface a module to the bus than to all other previously developed modules.

More specifically, a software bus is a communication facility between separately-specified modules. The *abstract bus* is a specification of the services provided by this facility, and the *bus implementation* shows how those specifications are to be realized for a particular set of programming languages, host platforms and communication media. Using a software bus, we can encapsulate decisions concerning the interfacing of modules, rather than distribute those decisions among the application modules themselves.

The bus implementation incorporates existing communication and interconnect mechanisms, providing a comprehensive approach to the construction of heterogeneous programs where more than

one form of diversity is present. A site implementor is responsible for mapping each domain into the abstract bus specification, and for showing how the domain's type model, data representation, and control mechanisms relate to the bus standard. This correspondence would only need to be established once, and thereafter programmers would be free to use modules from that domain within their configurations.

What is most visible to the programmer, however, is that the behavior of a software bus can be customized to serve each application. A programmer can accomplish this by describing the application's structure in terms of a module interconnection language (MIL). Once a configuration is written in terms of the MIL (and modules are written in terms of the programmer's implementation language of choice), then the bus is responsible for invoking all processes on the appropriate machines, establishing communication channels, and, during the execution, assisting in relocation and coercion of data.

The software bus model (along with an experimental implementation, called Polyolith) is described in [Purt94]. Polyolith demonstrates that the software engineering benefits of bus organization can be achieved without performance loss as compared to the same design built using traditional methods. Use of bus organization in parallel and distributed systems is described in [PuRG88, PuJa91]. As a carrier of research ideas, the Polyolith platform has been distributed to support research at more than sixty other universities and laboratories.

A software bus organization has been adopted by ARPA as the basis for research on module interconnection formalisms, in both the Software Prototyping Program, and the Domain Specific Software Architecture Program. Likewise, Eureka Software Factory (a \$10M ESPRIT software environment effort) is founded entirely upon a software bus organization. Finally, emerging standards efforts, examining tool to tool interoperability, are considering software bus organization as the basis for expressing interconnection obligations to be encompassed by the standard.

4 RESEARCH THEMES FOR VE ARCHITECTURES

Our investigation into VEs will use not only the software bus concept from Polyolith, but also will use the collaboration protocol representation and analysis techniques from Trellis in developing useful VR system specifications for the bus. In this section, we first outline the major questions that motivate our investigation, and then summarize the major steps we are taking to produce answers. After that, we explain in more detail the technical framework for our VE interoperability investigation.

Summary of questions and method

Our study is driven by an overriding general hypothesis: in its current form, software interconnection technology provides a basis, but not yet a fully applicable technology, for combining existing VEs into new VEs; preservation and communication of the semantic *functions* of data objects is

the primary missing component. This hypothesis brings to mind several sub-issues that require investigation before a general solution framework can be designed and proven in principle.

- Are there abstractions (data objects, operations, procedures, events) that appear repeatedly in, or are common across, examples of current VEs?
- Can such common abstractions be used to provide current software interconnection techniques with enough specification leverage to allow interconnection of existing VEs?
- If not, what specifically is the information not captured in current software interconnection methods that is required for VE interconnection?
- What extensions to current interconnection technology (specifications and generation methods) will capture this missing information in exploitable format?
- Can these common abstractions and specifications be used to develop a high-level interoperability framework in which to express *future* VE developments?
- How can the performance requirements of VR systems be maintained in an interconnection framework? Issues of distribution across networks and heterogeneous execution platforms will create disparities in the apparent time clocks of the objects in a composite VE, for example.

The basic steps in our investigation are:

1. Study good examples of existing VEs, looking for common elements that can be abstracted into a general high-level software architecture for VEs.
2. Develop specifications for the data and semantic components of the objects in the abstract VE framework, probably by combining the formalisms of our current process modeling work (Trellis) with our current software bus technology (Polyolith/Polygen).
3. Develop analysis methods and generation methods for exploiting these specifications for interconnection of VEs.
4. Experiment with the new techniques; this will entail adapting the example VEs (as necessary) to the interconnection framework, and subsequently producing combined VEs to prove our interoperability principles.

Step One: Study Existing VE Examples

Before designing a software architecture for VE interoperability, we must examine current VR systems and identify their major components, actions, and structures. We have several candidate VEs for detailed study; these are mostly drawn from the graphics lab at UNC Chapel Hill, but we may be incorporating systems at NRL as well. These VE examples are being reverse engineered,

looking for and cataloging common or frequently occurring data structures, object, procedures, and events. From the catalog of elements in VR systems, we are defining abstractions that seem to be points of commonality and thus can serve as points of abstraction for interconnection, collaboration, and interoperability in a VE.

Step Two: Define an Architecture Framework

We are developing a prototype of a simple event-based simulation system, as a vehicle for modeling the software architectures emerging from study of the other particular VE's developed within this program, in cooperation with those other teams. Our effort will entail enhancing the software bus model to include domain modeling and simulation languages, so that we may study how to most effectively relate two disparate application models. From that set of relations, we will demonstrate how the run time interfacing mechanisms can be generated automatically, based upon our existing software packaging technology.

One of our goals is to let VE builders use ADLs to promote sharing and reuse. Another goal is to help participants discover commonalities between their VEs which might not otherwise be apparent without a rigorous architectural analysis. As a result of our efforts, VE builders will be able to use ADLs, promoting sharing, reuse and interoperation between the domains.

Our initial investigation of this technology, and hence its proof-of-demonstrations, will be performed between two sites already incorporating VE technology, the University of North Carolina at Chapel Hill and the Naval Research Labs. Having two previously disjoint facilities available will enable us to do more than just simulate our ability interconnect VR systems from multiple VEs.

Step Three: Develop specifications and analysis methods

In another software context (module re-engineering), the first author has designed a semi-automated formal specification extraction method, and both authors have recently begun a joint proof-of-concept implementation of this technology. Called IDTS (Interactive Derivation of Trace Specs), the techniques are based on the formal specification method of *traces* defined by Parnas. Given the particular suitability of trace specs to object-oriented event-based designs, and given that one major aspect of VEs that must be added to our software interconnection technology is the management of object actions and semantics, we expect the principles at work in IDTS to be applicable to to specification and verification needs in this VE project. We therefore offer a brief explanation of IDTS.

Most existing software systems have been developed without using formal methods. When new systems are constructed with formal methods, it is difficult to incorporate such existing code because of the lack of compatible specifications. We have developed a technique, called IDTS (Interactive Derivation of Trace Specs), for deriving Parnas' trace specifications [BaPa78, PaWa89] from existing code modules. The method of extraction involves browsing a formal model of

the system to develop an experiential sense for the behavior of the module, “guessing” a candidate specification, and finally, automated verification that the “guess” is correct for the system via model checking. Iteration of the technique builds a collection of trace specifications for the module.

Usage Scenario – We assume that a software developer has a software module (object) on hand and needs to produce trace specifications for it. The general outline of our approach is as follows:

- construct an operational model (Trellis) of the module in context of its usage; this involves modeling not only the software of the module itself, but also a working system (one believed to function correctly) in which the module is embedded;
- use the system model to describe (enumerate, exemplify) legal traces of operation invocations at the module’s interface;
- explore the model using a Trellis browsing interface; gain an intuitive feel for the legal traces from this manually controlled simulation and from looking for patterns in the generated sample traces;
- express a suspected trace pattern in the CTL* temporal logic language; this step is best characterized as making an “educated guess” at a candidate trace spec, where one’s education comes from browsing and example generation phases;
- use a CTL* model checking algorithm to verify the candidate spec; this step will tell if the CRL* formula expressing the trace property holds on the system model or not;
- build up a collection of valid specifications via iteration of this exploration over many candidate specs;
- iterate these steps over several usage models; the goal is to model enough systems employing the module to explore a comprehensive set of module behaviors and thereby capture a complete set of trace specs; for a special purpose module, one system may fully exercise its operations; for a more general module (e.g., a data structure like a b-tree) it is less likely that a single system will exercise all its operations fully.

IDTS is the union of three main technologies:

1. TRACE SPECS: The idea of specifying module behavior by describing the legal sequences of events at its interface was first given by Bartussek and Parnas [BaPa78, PaWa89]. Hoffman and Snodgrass produced a heuristic for complete and consistent sets of specs [HoSn88]. A modified predicate calculus is used to express valid sequences of module operations; for example, the trace spec

$$(\text{forall } T, i) (L(T) \text{ --> } L(T.\text{push}(i)))$$

for a stack states that it is always legal to append the "push" operation to a legal sequence of operations.

2. **TRELLIS MODELS:** The Trellis formalism [StFu89, StFu90a, StFu90b] above is used as a model of parallel program threads of control, and a browser is provided to allow the user to explore the possible executions of a program for which trace specs are being derived.
3. **MODEL CHECKING:** Clarke has developed an algorithms for determining efficiently if a property in a temporal logic notation (CTL*) is true on an annotated finite state machine [CIES86]. Called "model checking", we have showed how to apply this work to Petri nets for verifying the trace properties of Trellis models [StFR92]. To use this work in IDTS, we simply express candidate trace specs in CTL* and use the model checker to determine if the "guess" is correct or not on the Trellis model of the program under study.

Step Four: Demonstrate use via Prototyping Process

We will study how the emerging prototyping technologies can benefit VE builders. If the interconnection framework and mechanisms from the prior step provide any leverage at all, then it should be evident in the extent to which modern prototyping practices can be employed on top of the framework. We intend to evaluate the savings (and improvement in quality) afforded the developers of a VE who utilize our interconnection mechanisms to first model and prototype their design. In some cases, we believe our interconnection framework will enable prototyping activities that would not be possible at all without our mechanisms.

The approach will be to return to the example systems studied in step one. We will develop several interesting new system concepts that can be obtained by interoperation of the original example VR systems, and will then use the methods developed in steps two and three to generate appropriate interconnection specifications. The modified examples will then be interconnected and tested for both logical function and performance.

We will feed back information from these experiments into modifications of the interoperability framework design and operations, as appropriate.

5 CONCLUSION

Our investigation is not nearly complete, but from our initial study we are encouraged that the two technologies we are combining—Polylith software interconnection technology, and Trellis collaboration protocol representation and analysis—is leading us to a system for combining existing virtual reality systems into new systems with new functionality. The combination takes place within a framework of common abstractions and commonly specified behaviors called a *virtual environment* that includes descriptions of data sharing and collaboration constraints among the various VR entities participating cooperatively. These specs are modifications of the data transformation descriptions found in the Polylith software bus; since they must not only encode data

format, but operational behavior as well, the specifications resemble the collaborative process descriptions found in Trellis hyperprograms.

While the project is producing a working proof-of-principle prototype VE, equally as valuable will be the body of organizing principles we identify in our study of current VR work. We expect the project to result in an intellectual framework that will give developers the leverage to reuse past VR systems as well to correctly organize new VR systems for smooth and efficient interoperability.

ACKNOWLEDGEMENT

The authors would like to acknowledge several individuals that have contributed in various ways to our ideas. Richard Furuta, of Texas A&M University, was co-developer with the first author of the Trellis collaborative model described in this paper. We also thank Larry Schuette, of the Naval Research Lab, for several conversations that helped to clarify some notions about VRs, and for information about current work in the field.

REFERENCES

- [BaPa78] Using Assertions about Traces to Write Abstract Specifications for Software Modules. W. Bartussek and D. L. Parnas. Proceedings of the Second Conference on European Cooperation in Informatics, Springer-Verlag, New York, 1978, pp. 111-130.
- [BDW89] Barbacci, M., D. Doubleday, C. Weinstock and J. Wing. Developing applications for heterogeneous machine networks: The Durra environment. *Computing Systems*, vol. 2, pp. 7-35.
- [CIES86] Automatic verification of finite-state concurrent systems using temporal logic specifications. E. Clarke, E. Emerson and A. Sistla. *ACM Transactions on Programming Languages and Systems*, 8:244-263, 1986.
- [FuSt89] Programmable browsing semantics in Trellis. R. Furuta and P. Stotts. In *Hypertext '89 Proceedings*, pages 27-42. ACM, New York, November 1989.
- [HaMS88] R. Hayes, S. Manweiler, and R. Schlichting. A simple system for constructing distributed, mixed-language programs. *Software Practice and Experience*, vol. 18, (July 1988), pp. 641-600.
- [HaNo86] N. Habermann, D. Notkin. Gandalf: Software Development Environments. *IEEE Transactions on Software Engineering*, vol. 12, no. 12, (December 1986), pp. 1117-1127.
- [HoSn88] Trace Specifications: Methodology and Models. D. Hoffman and R. Snodgrass. *IEEE Transactions on Software Engineering*, 14(9), September 1988, pp. 1243-1252.

- [JoRT85] M. Jones, R. Rashid and M. Thompson. Matchmaker: An Interface Specification Language for Distributed Processing. *Proc. of 12th POPL*, (1985).
- [LiSh88] B. Liskov and L. Shira. Promises: linguistic support for efficient asynchronous procedure calls in distributed systems. *Proc of SIGPLAN Language Design and Implementation*, (June 1988).
- [MaKS89] J. Magee, J. Kramer and M. Sloman. Constructing distributed systems in Conic. *IEEE Transactions on Software Engineering*, vol. 15, (June 1989), pp. 663-675.
- [NoBL88] D. Notkin, A. Black, E. Lazowska, et al. Interconnecting heterogeneous computer systems. *CACM*, vol. 31, (1988), pp. 258-273.
- [PaWa89] The Trace Assertion Method of Module Interface Specification. D. L. Parnas and Y. Wang. Technical Report 89-261, Queen's University, Kingston, Ontario, October 1989.
- [Perr89] D. Perry. The Inscape Environment. *Proceedings of 11th International Conference on Software Engineering*, (1989), pp. 2-12.
- [PuJa91] J. Purtilo and P. Jalote. An environment for developing fault tolerant software. *IEEE Transactions on Software Engineering*, vol. 17, (1991), pp. 153-159.
- [PuRG88] J. Purtilo, D. Reed and D. Grunwald. Environments for prototyping parallel algorithms. *Journal of Parallel and Distributed Computing*, vol. 5, (1988), pp. 421-437.
- [Purt94] J. Purtilo. The Polyolith Software Bus. *ACM TOPLAS*, (January 1994).
- [Snod89] R. Snodgrass. *The Interface Description Language: Definition and Use*. Computer Science Press, (1989).
- [StFu89] Petri-net-based hypertext: Document structure with browsing semantics. P. Stotts and R. Furuta. *ACM Transactions on Information Systems*, 7(1):3-29, January 1989.
- [StFu90a] Temporal hyperprogramming. P. Stotts and R. Furuta. *Journal of Visual Languages and Computing*, 1(3):237-253, 1990.
- [StFu90b] Browsing parallel process networks. P. Stotts and R. Furuta. *Journal of Parallel and Distributed Computing*, 9(2):224-235, 1990.
- [StFu92] Hypertextual concurrent control of a lisp kernel. P. Stotts and R. Furuta. *Journal of Visual Languages and Computing*, 3(2):221-236, June 1992.
- [StFR92] Hyperdocuments as automata: Trace-based browsing property verification. P. Stotts, R. Furuta, and J. Ruiz. In *Proceedings of the 1992 European Conference on Hypertext (ECHT92: November 30-December 4, Milan, Italy)*, pages 272-281. ACM Press, New York, 1992.

- [StFu93] Modeling and prototyping collaborative software processes. P. Stotts and R. Furuta. In *Proceedings of the NATO Advanced Research Workshop on Integration of Information and Collaboration Models (Il Ciocco, Italy, June 6-11)*. 1993, to appear.
- [WLRT91]. Wileden, et alia. Specification-level interoperability. *CACM*, vol. 34, (May 1991), pp. 72-87