View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by NSU Works

Nova Southeastern University
NSUWorks

CEC Theses and Dissertations

College of Engineering and Computing

2002

A Technique for Visualizing Software Architectures

Jon M. Inouye
Nova Southeastern University

This document is a product of extensive research conducted at the Nova Southeastern University College of Engineering and Computing. For more information on research and degree programs at the NSU College of Engineering and Computing, please click here.

Follow this and additional works at: http://nsuworks.nova.edu/gscis_etd

Part of the Computer Sciences Commons

Share Feedback About This Item

NSUWorks Citation

Jon M. Inouye. 2002. *A Technique for Visualizing Software Architectures.* Doctoral dissertation. Nova Southeastern University. Retrieved from NSUWorks, Graduate School of Computer and Information Sciences. (603) http://nsuworks.nova.edu/gscis_etd/603.

This Dissertation is brought to you by the College of Engineering and Computing at NSUWorks. It has been accepted for inclusion in CEC Theses and Dissertations by an authorized administrator of NSUWorks. For more information, please contact nsuworks@nova.edu.

A Technique for Visualizing Software Architectures

by

Jon M. Inouye

A Final Dissertation Report
submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy

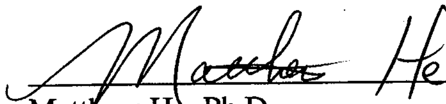
Graduate School of Computer and Information Sciences
Nova Southeastern University

2002

We hereby certify that this dissertation, submitted by Jon M. Inouye, conforms to acceptable standards and is fully adequate in scope and quality to fulfill the dissertation requirements for the degree of Doctor of Philosophy.

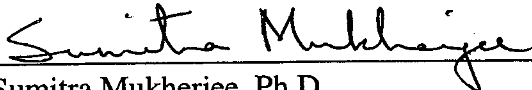
Michael J. Laszlo, Ph.D.
Chairperson of Dissertation Committee

4/12/02
Date



Matthew He, Ph.D.
Dissertation Committee Member

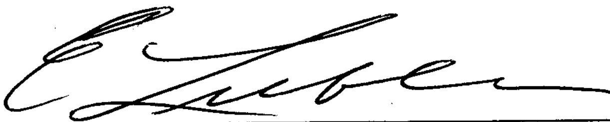
5/30/02
Date



Sumitra Mukherjee, Ph.D.
Dissertation Committee Member

5/29/02
Date

Approved:



Edward Lieblein, Ph.D.
Dean, School of Computer and Information Sciences

6-4-02
Date

The Graduate School of Computer and Information Sciences
Nova Southeastern University

2002

An Abstract of a Dissertation Submitted to Nova Southeastern University in Partial Fulfillment of the Requirements for the Degree of Doctor of Philosophy

A Technique for Visualizing Software Architectures

by
Jon M. Inouye

March 2002

Software architecture appeared in the early 1990s as a distinct discipline within software engineering. Models based on software architecture attempt to reduce the complexity of software by providing relatively coarse-grained structures for representing different aspects of software development. A software architecture typically consists of various components and connections arranged in a specific topology. Elements of the topology can serve as abstractions on (for example) modules, objects, protocols or interfaces. The meaning of the topology depends on viewpoint.

Software architectures can be described using an architecture description language (ADL). The key goals of ADLs are to communicate alternate designs to the different individuals involved in software development (such individuals are referred to as "stakeholders"), to detect reusable structures, and to record design decisions.

A major problem in software architecture has been the difficulty of creating different representations of an architecture to accommodate differing viewpoints of stakeholders. Ideally, different viewpoints would be conveyed in a way that is both comprehensive enough for specialists but consistent enough for generalists. The representation problem has been one of reconciling and integrating different viewpoints.

This dissertation provided a solution to the representation problem by creating a tool for three-dimensional visualization of software architectures using the Virtual Reality Modeling Language (VRML). Different architectural viewpoints were first defined in an ADL called the Visually Translatable Architecture Description Language (VTADL). When VTADL was translated into VRML, software architectures were embodied within three-dimensional "worlds" through which stakeholders may navigate. Each viewpoint was a separate VRML world. A viewpoint could be related to other viewpoints, representing different facets of software architectures, to reflect different stakeholder requirements. Traceability from design to requirements was possible through VRML hyperlinks from the visualized architecture.

The goal of the dissertation was to develop a prototype for demonstrating the visualization technique. Based on the successful results of two visualization case studies, we concluded that the goal was achieved.

Refinement of the prototype into a polished visualization tool was recommended. In future research, the refined version should be used for realistic evaluation of the technique in an actual software development environment.

Acknowledgements

The completion of this dissertation signified a major transition in my life, an endeavor requiring far more concerted thought and effort than originally anticipated. I am grateful to the faculty, staff, colleagues, family, and friends who supported me throughout this endeavor.

I am honored that Dr. Michael J. Laszlo consented to be my dissertation advisor, and deeply thankful for his patience and guidance. Gratitude is extended to committee members Dr. Sumitra Mukherjee and Dr. Matthew He, for their guidance and support; to undergraduate faculty members and colleagues Raisa Szabo, Terrell Manyak, Karen O'Brock, and Tim Margush for their poignant observations and advice; to staff members Barbara Edge, Dr. Diane King, Candy Fish, Will Ferri, Sharon Brown, and Mark Powell, who each encouraged and supported me in their own way. Classmates and fellow Ph.D. candidates Larry Shaffer and Peter Raethe supplied invaluable moral support during moments when I began doubting myself. I am very lucky to have two wonderful parents, Dr. and Mrs. Mitsuo Inouye; their encouragement also fueled the completion of the process.

A special thanks is extended to the Dean of the Graduate School of Computer and Information Sciences, Dr. Edward Lieblein, for his academic advice and unwavering assistance; to Dr. Naomi D'Alessio, Director of the Department of Math, Science and Technology; and Dean Norma Goonan, Ph.D., of the Farquhar Undergraduate Center, for providing a stimulating and intellectual environment in which I could pursue my educational goals.

Finally, I am grateful to the late Dr. Rollin S. Guild, who lent his creativity, insight, and good humor above and beyond the call of duty. This dissertation is dedicated to the memory of Rollin Guild.

And to those I may have unintentionally omitted, let me issue a blanket apology and justify the omission as a case of architectural data hiding!

Table of Contents

Signature (Approval) ii
Abstract iii
Acknowledgments iv
List of Tables vii
List of Figures viii

Chapters

I. Introduction
Introduction 1
Problem Statement 4
Goal 10
Relevance and Significance 12
Barriers and Issues 18
Limitations of the Research 24
Research Questions 27
Definition of Terms 29
Summary 40

II. Review of Literature
History and Foundation of Software Architecture 42
Software Architecture in the Software Development Process 53
Software Architecture and Design Patterns 56
Software Architecture Description Languages 58
Software Architectural Viewpoints 64
Visualization of Software Architectures 68
General References 75
Summary of Knowns and Unknowns 77
Contribution to the Field of Software Architecture (Software Engineering) 79

III. Methodology
Research Methods
Overview of Procedures to be Employed 81
Specific Procedures to be Employed 85
Description of Procedures
Procedure Step One 86
Procedure Step Two 100
Procedure Step Three 119
Procedure Step Four 121
Procedure Step Five 122
Formats for Presenting Results 124
Projected Outcomes 124
Resource Requirements 125
Reliability and Validity 125
Summary 126

IV. Results
Analysis 127
Findings 129
Summary of Results 130

V. Conclusions, Implications, Recommendations and Summary
Conclusions 132
Implications 133
Recommendations 134
Summary 135

Appendices
A. BNF Specification for VTADL 140
B. Lex Specification for VTADL 144
C. Yacc Specification for VTADL Parser Shell 147
D. Subroutine Design: Data Structures and Algorithms for Geometric Modeler 154
E. Example of a VTADL Source File 160
F. Complete Yacc Specification for VTADL Parser Shell 165
G. Test Cases 223
H. Case Study Report Template 260
I. Case Studies 262

Reference List 314

List of Tables

Tables

1. Summary of Case Studies 123

List of Figures

Figures

1. VTADL-To-VRML Translation 89

2. Architecture Definition (Template) 90

3. Properties of the Component 92

4. Template for a Connection List 95

5. View-List Template of a VTADL File 96

6. Basic VTADL-To-VRML Compilation Process 100

7. Node in Architecture Linked List 102

8. View List Node Definition 106

Chapter I

Introduction

Introduction

The emerging field of software architecture, within the broader context of software engineering, capitalizes on the proclivity of human beings to draw simplified, highly abstracted "first sketches" of a system prior to detailed design [Perry & Wolf, 1992; Shaw & Garlan, 1996]. Rather than dismissing these first sketches as informal exercises, software architects have classified and analyzed the preliminary abstractions. Software architects have even formalized the representation of architectures by using formal languages [Allen, 1998; Abowd et al., 1995].

Software architecture focuses on the structure and interrelationships between software components. The architectural approach describes the components and connections with relatively coarse granularity, hiding design details to achieve simplicity. An architecture tries to capture the essence of the software system, so that those involved in the development process (referred to as "stakeholders") may readily comprehend the system's purpose. The approach is a way to regain intellectual control over the growing complexity of software.

Software architecture allows stakeholders to evaluate alternate designs at an early stage, in order to make informed decisions about the later phases of software development.

Thus, software architecture serves as a framework, or "architectural baseline" [Jacobson et al., 1999], to develop a more detailed design. The process of defining a software architecture traditionally followed the requirements specification, but preceded the detailed design and implementation phases of the software life cycle.

In recent years, the concept of software architecture has grown to encompass the entire software development life cycle [Abowd et al., 1997; Jacobson, Booch, & Rumbaugh, 1999]. Software architecture is now seen as a reference model that evolves along with software development. The reference model ideally reflects changes to the software architecture over time. Through spotting inconsistencies in architectural design, detecting invalid interfaces between software components, or discovering other mistakes in the software architectural reference model (such mistakes are referred to as "architectural mismatches"), software errors can be detected and eliminated as early as possible. Architectural design errors, if caught early in the software development life cycle, can result in substantial savings in the development cost, since such errors caught later in development (during implementation or maintenance) are many orders of magnitude more expensive to fix.

Software architectures are represented in textual form using an Architectural Description Language (ADL). The motivation behind an ADL is to facilitate communication on the overall structure to stakeholders [Bass et al., 1998]. Ideally, ADLs represent the different viewpoints of several stakeholders in order to increase joint

understanding and agreement on the overall design. However, as Hofmeister, Nord, & Soni of Siemens Corporation have observed, software architects "…can't yet succinctly describe which design details are important over all domains and system sizes" [Hofmeister, Nord, & Soni, 2001].

The textual ADL may not be understandable to all stakeholders in a problem domain. For instance, nontechnical management in the organization may not readily grasp the structure merely by perusing the ADL even when the ADL is accompanied by natural language descriptions. As the structural complexity increases, so does the need for an effective architectural representation.

Stakeholders desire that specialized viewpoints communicate effectively to the intended audience. On the other hand, stakeholders require that the viewpoints be consistent with one another (that is, elements of one view can be mapped to one or more elements of another view), and do not diverge to such an extent that the views can no longer be integrated or comprehended by a nonspecialist. These apparently contradictory demands on an architectural representation can lead to problems when using an ADL as a medium of communication.

Thus, representation of a software architecture in a form that is readily grasped by stakeholders continues to be a fundamental problem in software engineering [Bass, Clements, & Kazman, 1998; Egyed, 1999].

Another important goal of the ADL is to standardize the structural representation in a language independent of system implementation, so that knowledge of abstract patterns can be detected, reused, and transferred to future projects. An effective ADL

can even be used for training purposes, dramatically reducing the learning curve for new project members.

Good architectural designs allow the completed software system to be maintained without degrading the performance of the system. In other words, our architecture description should be unambiguous and consistent so that we can change or replace components without having adverse or unforeseen impacts on other parts of the system [Jacobson, Booch, & Rumbaugh, 1999].

Any architectural description must incorporate not only the topology, but also the correspondence between software requirements and elements of the structure [Shaw et al., 1995]. Thus, the original reasoning behind aspects of the architecture can be traced. When linked to original requirements, an ADL file can serve as an archive of design decisions, design rationale, and reusable patterns.

Problem Statement

This research focused on the problem of representing and communicating the software architecture to different stakeholders involved in the software development process. Ideally, different viewpoints on the software architecture would be conveyed in a way that is both comprehensive enough for specialists and general enough to stakeholders trying to consolidate the different views (e.g., project managers or executives). In reality, as the software system becomes more complex and the viewpoints on the system become numerous, difficulties in integrating viewpoints begin to emerge.

Based on the desired stakeholder viewpoint, the components and links in the software topology would be represented differently.

An analogy may be drawn with the architectural viewpoints to the structure of a physical building. The building contractor, architect, electrician, interior designer, landscaper, plumber, and occupants will all have different views of the same building. Each person needs to view separate components and relationships, requiring different spatial, hierarchical, textural, and qualitative views. When taken together, the different views comprise the building's entire architecture. In like fashion, each viewpoint of the software architecture requires a different combination of component/connection topology and spatial displacement. The elements of each abstraction will be interpreted differently; each interpretation is modified based on the particular nomenclature and mindset of the stakeholder. In other words, an architectural viewpoint must be distinctive and separable enough to do justice to the specialist, yet consistent enough so that the view may be integrated with other views without incompatibility errors in the modeling.

Incompatibility between views is referred to as "viewpoint mismatch" [Egyed, 1999; Egyed & Medvidovic, 1999]. For an example of viewpoint mismatch, consider the blueprint of the physical architecture of a five-story building. The electrician's view is comprised of the wiring and electrical outlets; only the general outline of the building is provided for reference. The plumber's view, on the other hand, consists of pipes and pumping motors along with the general building outline; a cursory view of electrical power generators is provided to the plumber as a safety measure. Let us suppose that, when integrating the electrician's and plumber's view, it is found that the plumber's view

consists of electrical power generators that are not found in the electrician's view. Or that the electrician's view inadvertently contains the outline of doors and corridors which do not exist. These viewpoint mismatches may lead to serious design flaws and expensive fixes during construction of the actual building.

A view may contain more than one software architecture. Incompatibilities between software architectures within a view may also occur and are called "architectural mismatches" [Garlan, Allen, & Ockerbloom, 1994]. For instance, a view may consist of a software architecture modeling a process with two inputs and a single output, and another architecture modeling a process with three inputs and five outputs. One of the outputs of the first architecture is directed as input into the second architecture. However, the second architecture requires all three inputs in order to function at all. Here, then, is an incompatibility between the two architectures. Visualization of both software architectures will assist us in detecting such mismatches.

This dissertation proposed a solution to the representational problem by allowing interaction with a three-dimensional, virtual reality "world" modeling the software architecture. The research began by designing an ADL capable of representing different viewpoints, then provided a compiler which translated the ADL into a three dimensional representation in VRML (Virtual Reality Modeling Language).

The presence of three dimensions in a complicated representation, accompanied by the capacity for user-directed movement in a three-dimensional medium, have been shown empirically to enhance comprehension of complex structures [Ware & Franck, 2000].

The stakeholder can navigate (move about) through the software structure, examining patterns and relationships not readily discernible from the original ADL. Other research in software visualization indicates that visual identification of software flaws and even the discovery of reusable design patterns are often possible from the 3D representation of software structures [Feijs & de Jong, 1998]. Furthermore, as Dutch researchers Feijs and de Jong enthusiastically point out, "...walking through or rotating a complex design is exciting, almost like having the design in your hands."

The use of different viewpoints to explore requirements and design specifications is nothing new [De Michelis et al., 1998]. However, the *virtual reality visualization* of software structures from different qualitative as well as spatial viewpoints had not been fully explored.

Different viewpoints may occupy different initial positions in a relational hierarchy of software components. That is, based upon the viewpoint, certain objects and relationships will be emphasized and other objects/relationships kept hidden. For example, software quality assurance testers may need portions of the full hierarchy, from high-level requirements components down to code-level modules, in order to trace compliance with code to requirements. Marketing managers or vice-presidents, however, would probably be interested only in a high-level perspective on the architecture; such stakeholders desire to observe how the finished product may function, and thus only a few high-level components need to be rendered, with lower-level code modules kept hidden. Computer programmers would need to view only select low-level code modules; irrelevant higher-level components would be kept hidden. Code modules

unrelated to the computer programmer's current activity would not be rendered from the programmer's viewpoint.

Based on the desired viewpoint, stakeholders will visually navigate through the portion of the "world" which represents their application domain. However, the stakeholder (based on security clearances) would have access to an expanded representation beyond his or her immediate domain. Certain visual icons may represent portals, or entranceways, to other branches of the world not yet visible to the person navigating through the world. The portal may be an entranceway either to another portion of the architecture at the same level of hierarchy or to a lower-level portion of the architecture with a more detailed viewpoint of the software complexity (previously kept hidden in order to maintain a higher level of abstraction). Conversely, the portal may be an entrance to a higher-level world, with several components in the virtual world kept hidden in order to represent a higher level of abstraction.

For example, when the visual portal (such as a three-dimensional sphere) is selected by a pick device (a mouse), another branch of the world will be entered. A department manager, viewing higher-level components in the software structure, may for whatever reason desire to navigate to lower-level portions of the "world," closer perhaps to the code modules normally viewed by programmers.

Furthermore, the objects in the world may represent hyperlinks to real-world requirements documents, design documents, audio recordings of design conferences or management interviews, video snapshots, and so on.

It should be noted that recently, an important distinction has been made between the definitions of "view" and "viewpoint" by the Institute of Electrical and Electronic

Engineers [IEEE, 2000]. A *view* is defined by IEEE Standard 1471-2000 as an abstraction on a particular model, based on the concerns of a particular stakeholder or group of stakeholders. A *viewpoint* is the template or specified format for describing the abstraction. The viewpoint would specify, for example, how the elements of the notation are interpreted and configured, in concurrence with the mode of thinking and semantics of the stakeholder. The viewpoint would also specify data hiding and decomposition – what data is to be emphasized at a higher level, and which details of the data are to be kept hidden for a lower level.

Thus, a view is the instantiation of a viewpoint (using the IEEE definition) in the same way that an object in the object-oriented model is the instantiation of a class.

Goal

The goal of this research was to devise algorithms for converting an ADL into effective VRML representations based on the desired viewpoint. The VRML representations were intended to enhance comprehension on the overall design and to improve communications between diverse stakeholders. In other words, the tangible goal was to allow for visualization of more than one viewpoint, and to allow the stakeholder to toggle between multiple viewpoints within a participatory medium. Using hyperlinks from certain aspects of the architecture, stakeholders may trace the rationale behind certain designs or may verify the correctness (or flaw) of architectural elements. More importantly, the ADL, along with the ADL's visualization in virtual reality, may serve as a repository for reusable patterns in future projects.

As far as the constraints of this research were concerned, the goal was obtained when we developed a tool consisting of an ADL capable of representing multiple architectural viewpoints, a translator from the ADL to VRML representation, and a successfully translated VRML representation from the architecture originally described by the ADL. Aspects of the VRML representation were required to be traceable (using hyperlinks) back to the requirements documentation.

Several case studies of different representations in more than one architectural style were conducted to demonstrate the viability of the approach. Among the case studies was a representation of a structured program topology ("Program Call-and-Return") and a Linux conceptual and concrete view.

The actual use of the tool in a commercial development environment was not explored and was considered a topic for future research.

Relevance and Significance

David Garlan and Dewayne Perry provided a comprehensive overview of key research areas in software architecture [Garlan & Perry, 1995]. As will be elaborated in the literature review, these research areas have only intensified within the past six years.

Perhaps one of the most significant trends in software engineering since the late 1990s has been the ascent of the Unified Modeling Language (UML) into an industry standard for object-oriented design, analysis, and specification.

UML was originally not a true language, but actually a graphical notation for representing object-oriented models (classes, objects, relations, methods, state charts to model behavior, etc.). UML was standardized by the Object Management Group into a semiformal language [Egyed & Medvidovic, 1999]. Prior to UML, three methodologies were in widespread use in the object-oriented programming community: OMT (invented by James Rumbaugh), the Booch methodology (by Grady Booch), and OOSE (by Ivar Jacobson).

Each method had strengths and weaknesses. OMT emphasized the design phase but was poor in analysis; OOSE was strong in modeling behavior but weak in other areas; and Booch's method was strong in design but weak in the analysis phase [Quatrani, 2000]. The three competing methodologies were unified when all three software engineers (Booch, Jacobson, and Rumbaugh) joined forces at the Rational Corporation during the mid-1990s. They unified their methodologies and combined the strengths of each methodology into the Unified Modeling Language [Booch, Rumbaugh, & Jacobson, 1999].

Because the Unified Modeling Language has become the de facto standard in software design, UML has been proposed by numerous authors as a standard tool for software architecture description [Egyed, 1999; Medvidovic, Rosenblum, Robbins, & Redmiles, 2000; Jacobson, Booch, & Rumbaugh, 1999]. However, UML is a visual language, not a textual one, and was originally intended for object-oriented design rather than exclusively architecture description. The UML notation in its current form does not model configurations of architectures explicitly; the explicit modeling of configurations is one of the criteria for defining an ADL [Medvidovic & Taylor, 2000]. Furthermore, UML does not adequately describe the details of connector or component attributes to the extent of dedicated ADLs such as UniCon or Wright. Typically, architecture descriptions are implemented in UML using extensions to the language [Medvidovic, Rosenblum, Robbins, & Redmiles, 2000; Selic, 2001]. However, the problem is that these extensions are not standard to the native UML and, hence, the argument for a dedicated architecture description language appears to have merit.

UML has other weaknesses in modeling software architectures. As Alexander Egyed pointed out, UML is not formal or complete in integrating different views [Egyed, 1999]. That is, UML's ability to define views may be ambiguous, and the capability for defining interrelationships between views does not exist. Medvidovic and Taylor of the University of Southern California's Center for Software Engineering [Medvidovic & Taylor, 2000] have also pointed out that an ADL has a formal semantic theory with respect to the architectural domain, an underlying framework to the language. Since UML was originally intended to assist in the modeling of design classes and not software components or connectors, the separation of object-related concerns from architectural

concerns would complicate rather than simplify the design task. In other words, using UML for software architecture would make architectural design more difficult, not less complex.

Some authors have suggested that UML and ADL can complement rather than compete with one another [Gomma & Wijesekera, 2001; Medvidovic, Rosenblum, Robbins, & Redmiles, 2000]. UML can serve as the object-oriented and analysis tool, with ADLs focusing in a complementary way on the architectural views. These same researchers are also striving to incorporate architecture description language capabilities in the next version of UML (UML 2.0).

While three-dimensional visualization of UML notation remains a distinct possibility for future research, this dissertation had decided to use a dedicated ADL as the preliminary definition for an architecture description. The reason was due to the original purpose of UML: UML was invented as an object-oriented design tool and does not, in its current form, explicitly define software architectures with the versatility of ADLs.

Architecture description languages have so far been relegated to research environments and have not succeeded commercially, despite the potential of the languages to assist in software development [Jazayeri, Ran, & van der Linden, 2000].

Other active research areas include formal methods for software architectures, architectural analysis techniques, architectural recovery for legacy systems, architectural codification, and the recording of architectural expertise for future use. Most importantly (from the standpoint of this dissertation), Garlan and Perry discussed the need for more research in architectural tools and environments. This need was echoed by Shaw and

Garlan [Shaw & Garlan, 1996] at the very beginning of the software architecture discipline.

Very few tools existed for the flexible, and three-dimensional, visualization of multiple views on software architecture. This research noticed the powerful capability of VRML to model worlds containing a variety of user-defined structures [Ames et al., 1997]. A natural association was made by the author of this dissertation between the world-modeling capabilities of the virtual reality paradigm and the nature of a software architecture viewpoint. A major component of the tool developed in this dissertation was to map a viewpoint definition contained in an architecture description language to a VRML world.

Feijs and de Jong [Feijs & de Jong, 1998] are one of the few software engineers to visualize software architecture using VRML. They emphasized the need for flexible architecture visualizations with multiple viewpoints and software traceability:

"Although helping remember complex structures and making appealing images are alone not strong reasons for pursuing 3D, making design objects consistent with an object's intended role, choosing new viewpoints, and identifying new design metaphors are serious motivation and should be investigated."

While Feijs and de Jong did use VRML as the medium for representing software architecture, they based their visualization on a relational database containing connectivity information.

To the best knowledge of this research, no compiler had been created which directly translated a dedicated architecture description language into VRML. Furthermore, very little or no research had been done on the representation of interrelated

architectural viewpoints in a three-dimensional, virtual reality medium. This research allowed the definition of separate but integrated viewpoints within an ADL, and also translated the architecture description to a corresponding visualization in VRML. Each viewpoint was represented as a self-contained, separate world (a separate reality), hopefully reflecting the stakeholder's concerns in an interactive, exploratory medium that would enhance understanding about the stakeholder's world. The viewpoints were integrated through hyperlinks (experienced by stakeholders as portals), as though entering the door to a separate room in an office building.

Perhaps the maxim that the specialist lived in his or her own world can be taken literally in this visual representation. But now the specialist's world was visible for all to see, in three-dimensional color, movement, and with realistic lighting, appearing literally in the midst of a clear blue sky. So perhaps the excuse that the stakeholder could hide in his or her own world no longer applied.

This dissertation contributed to the field of software architecture by providing a three-dimensional visualization tool to represent multiple viewpoints of a software architecture. The visualization attempted to make the software architecture more comprehensible by immersing the stakeholder in a virtual world representing that architecture. User-directed movement, along with three dimensions, had been shown to increase comprehension of complex structures. The tool also provided traceability from objects in the visualized architecture to source documentation, so that stakeholders may trace aspects of the architecture back to the rationale behind the design.

This research was significant since very little effort had been made to visualize viewpoints on software architectures using VRML. This research also attempted to

integrate viewpoints using the techniques of virtual reality.

Barriers and Issues

Software architects such as Philippe Kruchten [Kruchten, 1995], Alexander Egyed [Egyed, 1999; Egyed, 2000], Christine Hofmeister [Hofmeister, Nord, & Soni, 2000] and Rich Hilliard [Hilliard, 2001] have discussed at length the problems of modeling different viewpoints on the software architecture. These authors presented different solutions on the viewpoint representation problem.

Philippe Kruchten tried to solve the viewpoint representation problem by decomposing the architecture description process into four main views: the logical, process, development, and physical views.[1] The logical view dealt with the concerns of stakeholders involved with the object-oriented development. The process view dealt with process synchronization and concurrency issues, while the development view represented the static organization of the software (what we would call the static software architecture), and is perhaps closest to the software architectural representation discussed in this dissertation. Kruchten's physical view represented the physical configuration of the hardware and mapped the software onto the hardware components. A fifth "view" established the possible use case scenarios to model the system behavior.

Hofmeister, Nord and Soni also divided a system under development into independent views, yet their views did not exactly coincide with Kruchten. Hofmeister et al. proposed the code, module, execution and conceptual views. The code view dealt with implementation code, while the module view was a slightly higher-level abstraction, where components represented modules. The execution view dealt with mapping

---

[1] Please Note: The terms *view* and *viewpoint* were not necessarily differentiated in earlier literature until IEEE Std. 1471-2000, the first standard for architecture description.

modules into the runtime view of the software architecture. The conceptual view consisted of high-level conceptual designs and how they interrelated.

Rich Hilliard discussed modeling views and software architectures using UML. Alexander Egyed, using UML as a modeling tool, discussed the modeling of viewpoints in a much broader, more universal sense. Egyed crystallized many of the problems that exist in viewpoint modeling, and suggested future research in the area of viewpoint modeling.

"This deficiency (in viewpoint modeling) would not exist," Egyed wrote, "if we could have a few perfect views that could be used by all stakeholders...which were precise enough but still easy enough to use. These views, unfortunately, do not exist. Instead, we are confronted with a number of loosely coupled, sometimes quite independent views. This is not really what we want. Nuseibeh wrote that 'multiple views often lead to inconsistencies between these views -- particularly if these views represent, say, different stakeholder perspectives or alternative design solutions.' "

In order to develop a tool for representing different viewpoints on one or more software architectures, we first had to clarify exactly how we wanted to represent a view and the associated semantics of our particular research model behind the view, and how we wanted to represent one or more software architectures within that view.

In other words, the barrier we first had to overcome was how to represent the view in the ADL. Second, we had to decide how we wanted to map the view defined in the ADL to the Virtual Reality Modeling Language (VRML). Different software architectures for different aspects of the system were defined in the ADL. These software architectures had one or more architectural styles. Given the architectures, a

given viewpoint may contain one or more architectures, or even parts of architectures, within the view. Certain styles of architectures determined a particular rendering of the objects in VRML. That is, there was a distinct mapping from the ADL to VRML objects (spheres, cones, etc.) based on architectural style.

The exact appearance of the VRML objects was an aesthetic consideration. This research intended to make the aesthetics as palatable as possible but left the polished, professional graphics to a future version. In a future version of the compiler, some allowances may even be made for the viewer to customize the objects. Once again, the aim of this research was to build a prototype to demonstrate the workability of the visualization technique rather than seek aesthetic perfection.

Alexander Egyed and his colleagues mentioned the need for view integration tools to automatically check for errors during the composition of views. While a view integration tool that can check for viewpoint mismatch would be a valuable one, this research did not intentionally follow this road of development.

Ultimately, the actual view integration was left to the stakeholder or software architect. The tool developed by this dissertation intended to make the task of view integration easier by translating each view into a virtual reality world.

The development of an architecture description language capable of being translated into VRML resulted in the Visually Translatable Architecture Description Language (VTADL) and was loosely based on ASDL [Eixelsberger & Gall, 1998]. The motivation for creating VTADL was to obtain a relatively simple language for defining essential architectural information (i.e., components, connectors, interfaces, and style). VTADL allowed the software architect to ignore details about the interface behavior of

components and to concentrate on architectural structure. Language features were added to VTADL both to constrain the structural topology and to make explicit certain visual attributes of an architecture within VRML.

VTADL was formally defined using BNF (see Appendix A). The BNF specification was used as input to the parser generator called yacc (yet another compiler compiler), which generated the compiler for a Unix environment. The resulting parser was a program shell capable of recognizing the VTADL language, but not yet capable of generating code in the target language (VRML). Final code generation into VRML depended upon the implementation of an effective geometric model representing each architectural style. The geometric model was implemented using abstract data types and associated operations on the data types. The intermediate geometric model was the medium through which the architectural structure defined by VTADL was mapped to the visual objects within VRML. Hence, the implementation of the intermediate geometric model was a major barrier to overcome.

In considering the geometric model, we also considered the concept of an *architectural style* [Shaw & Clements, 1997]. Architectural styles in software architecture are analogous to the various styles in building architecture (e.g., Gothic, Victorian, Modern, etc.). An architectural style defines how the components and connectors of the architecture may be used, with constraints on the topology and instantiation. Examples of the better-known styles are the program call-and-return, pipelined, and layered styles.

Two possible architectural styles were represented in VRML: the call-and-return and the layered architectural styles. The components and connections of the call-and-

return architectural style were represented using an n-ary tree; component attributes were stored in the fields associated with the vertices, and connections were defined by tree edges. The layered style used linear arrays. Pertinent architectural information was stored in each cell of a given array.

As the compiler scanned and parsed the source language, component/connector identifiers and attributes were stored in the appropriate data structure based on the style (the parser knew the style at the very beginning of the parse, since the style was declared at the start of the architectural description). When all the components, connectors and views were inserted into the data structure, the code generation phase of the compiler program then traversed the data structure and used logic about the topological constraints of the style to generate VRML commands. The VRML commands defined the geometric shape, position, scaling, color, and other visual attributes of the objects in three-dimensional space.

In other words, using the intermediate data structure for the given architectural style, the code generation phase mapped the ADL components and connectors into an appropriate VRML representation.

VRML as a target language has numerous advantages in the representation of software architectures. VRML could represent a number of diverse objects in a user-defined three-dimensional world. Texture-mapped and reflective surfaces, multiple light sources, shading, and object animation are but a few examples of the capabilities of this virtual reality paradigm [Ames et al., 1997]. VRML is an immersive medium, in the sense that the user (stakeholder) is considered to be occupying a position in the world, and may be surrounded by a visually rich environment (including interactive

representations of other stakeholders who may be seen as avatars). The user is capable of navigating through the world, viewing objects interactively and from a variety of positions (including inside the object itself). The virtual reality worlds definable by VRML were used to represent viewpoints; each viewpoint defined in the architecture description language was mapped to a separate VRML file by the VTADL-to-VRML compiler.

While VRML as a medium for user interaction and visualization had numerous advantages, the VRML medium also had disadvantages which should be mentioned. Chief among them was the often inconsistent rendering qualities of many VRML add-ons. Visual qualities such as shading, motion and lighting would vary from add-on to add-on. This dissertation used (and recommended) the VRML add-on called Cortona for Windows environments (95, NT, 2000).

Another disadvantage was the need for add-ons at all. Recent advances in XML technology [Web3D, 2001] intend to incorporate the functionality of VRML as tags within XML. The use of XML tags for virtual reality visualization of software architectures will eliminate the need for separate add-ons, and is a recommended avenue for future research.

Limitations of the Research

The communicative quality and aesthetic appeal of the visualized architectures were major considerations. It was anticipated that the visual appearance could be "tweaked" as needed, by adjusting cosmetic attributes (component colors, shapes, etc.) of the VRML rendering of the architecture. However, it was understood that "degree of communication" and "aesthetic appeal" were highly subjective in nature. This research conducted no stakeholder survey to determine the psychological appeal of different renditions of the same software architecture.

The following constraints were based on the fact that a prototype, rather than a full commercial product, was developed:

1. *Static, rather than dynamic, architectures were visualized.* This dissertation focused on static software architectures, and the associated views of the static architectures. The task of representing static architectures alone was a challenging one; representing dynamic architectures (such as state-based architectures modeling behavior) was reserved for future research.

2. *Not all properties of each architecture were visualized.* For example, although components and connections were visualized, explicit interfaces defined in the ADL were not shown as separate entities. Directionality (bidirectional or unidirectional) was ignored; only the presence of a connector itself, with no arrows, indicated a relation.

3. *Only two architectural styles, the program call-and-return style and the layered style, were visualized.* Although an arbitrary network and pipelined architectural

styles were definable in the ADL, only the call-and-return and the layered styles were visualized.

4. *Software architecture topologies were acyclic (without cycles) and coarse-grained.* Again, the motive was to demonstrate a "proof of concept" behind the visualization of software architectures in VRML. A given software architecture was limited to twenty-five components and twenty-four connectors. A given architecture requiring more detail than twenty-five components could be decomposed into smaller sub-architectures; each sub-architecture could be referenced either in the same view, or in another view (as defined by the architecture in the original VTADL file). In addition, a maximum of twenty-five architectures were allowed in a given view.

While research into visualizing complex, three-dimensional structures with tremendous numbers of components had taken place [Parker, Franck, & Ware, 2000], software architectures typically were coarse-grained. In other words, it did not benefit the software architect to incorporate tremendous numbers (on the order of hundreds or thousands) of components and connectors in a representation, even if paging techniques were used. This would defeat the premise of software architecture – simplicity. Therefore decomposition into other architectural views would take place long before the limit of twenty-five components or twenty-four connectors was reached. In such a manner, hundreds or thousands (or more) components could be represented by decomposing them into simpler but interrelated architectures and architectural views. In fact, the classic empirical studies on human comprehensibility by Miller [Miller, 1956] had shown that human beings were comfortable grasping no more than seven or so entities at a time.

5. *The VTADL-to-VRML visualization tool was a prototype only.*

The effectiveness of the VTADL visualization tool in a real-world development environment was not explored in this dissertation and is an avenue for future research. Case studies were conducted, however, in order to demonstrate the workability and future potential of the prototype of the VTADL-to-VRML compiler.

Research Questions

This research provided a solution to the representation problem (as described in the first section) by developing an architecture description language and a language translator to visualize the architecture in VRML. The resulting tool was intended as a "proof of concept" rather than a full-scale commercial project.

Although no user survey was conducted and no formal software engineering quality metric study was performed to determine how well the tool contributed to reducing long-term software development costs (this is an avenue for possible future funding and research), the initial question for which the tool was devised is an important one:

How can one effectively represent different viewpoints on software architectures so that they are more comprehensible and more engaging? This research proceeded on the assumption that three-dimensional movement in a realistic, virtual world assisted in the comprehension of complex structures [Ware & Franck, 2000]. VRML was determined to be an excellent medium for immersing the stakeholder's views, allowing user-controlled, three-dimensional navigation and participation (see the “Barriers and Issues” section). The technique for visualization, as described in detail in Chapter Three (Methodology), devised an architecture description language called VTADL for specifying the architectures and viewpoints on the architectures, a compiler for mapping the specifications into an intermediate geometric model, and an algorithm for instantiating the contents of the geometric model into VRML files.

A second question was intimately related to the first one:

Since each view addresses the specialized concerns of different stakeholders, how does one consistently integrate the separate views in a way that addresses the more general concerns of broader stakeholders, yet avoid the pitfalls of viewpoint mismatch (i.e., flaws uncovered in relating the viewpoints)? Integrating viewpoints and the discovery of viewpoint mismatch during view integration were discussed at length by Egyed [Egyed, 1999; Egyed, 2000].

This dissertation addressed these research questions by providing a visualization tool to represent software architectures and the integrated viewpoints on software architectures. However, the VTADL compiler did not attempt to automatically detect viewpoint or architectural mismatches; the mismatches were left for the stakeholder to detect manually (or rather, visually!) through observation of the visual representation.

Definition of Terms

Architecture Mismatch: The incompatibility between elements of different architectural structures, often caused by invalid assumptions about the architecture. Architecture mismatch may occur when component reuse is attempted; the reused component may not interface properly with a new architecture due to erroneous assumptions about how the component connected to other elements in the architecture.

ADL: See Architecture Description Language.

Architecture: See Software Architecture.

Architecture Description Language: A language intended to specify the structure of a system by describing the system's architecture; the description includes specifying the components, interconnection between components, and the constraints on the topology. Also included in the language would be the capability to describe the rationale behind elements of the architecture, and the behavior of the architecture under specified conditions.

Architectural Framework: (from Hofmeister et al., 1998) A specification or template for the architectural structure and flow of control/data. Part of the specification would be in the form of a partial implementation, ready for use in a specific problem domain.

Architectural Style: A definition of how the components and connectors of an architecture can be used, with constraints on the topology and instantiation. Typical architectural styles are the program call-and-return, pipelined, layered, and event-based styles.

BNF: Backus-Naur Form, a formal language for defining the syntax and grammar of a language.

Call-and-Return Architectural Style: (Also "Main-program-and-subroutine") The classical programming paradigm. Each component represents a program or subroutine; each link represents a call (control flow) to a lower-level routine. The decomposition of a program is thus represented hierarchically.

Class: A set of elements having common characteristics; the domain which defines an object. An object is an instantiation of a class.

Component: An encapsulated part or unit; a fundamental element of a software architecture. In a graphical representation of the software architecture, a component is represented as a node which, depending on the viewpoint, can be interpreted as a module, process, chunk of code, workflow, etc.

Compiler: A computer program for translating statements from a source language to a target language. Often the source language is a higher-level language (such as C, C++,

Fortran, etc.) and the target language is an object language such as machine code. For this dissertation, the source language is a text-based architecture description language called VTADL, and the target language is the Virtual Reality Modeling Language (VRML).

Connector: (also referred to as "connection") A link between software components. Based upon the viewpoint, the link may be defined as a relation, an association, data flow, control flow, inheritance, etc.

Control Flow: In the parlance of an architecture description, control flow refers to the flow of control from one component to another. A connection may typically be defined as the flow of control from a server to a client.

CORBA: Common Object-Oriented Request Broker Architecture. A specification (established by the Object Management Group) to allow distributed objects to request services from one another, independent of system or locale.

Data Flow: An architectural style where the system is viewed as a series of transformations on input data. Data is seen to flow along a connector to a component, undergoing transformation within the component, and emerging from the component in altered form. Data flow may also simply refer to the flow of data along a connector from one component to another (as compared to a control flow).

Design Pattern: (from Gamma, Helm, Johnson, & Vlissides, 1995). Patterns are "descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context. A design pattern names, abstracts, and identifies the key aspects of a common design structure that make it useful for creating a reusable object-oriented design."

Formal Language: A language generated from an alphabet using a set of production rules. The advantage to a formal language is that it is defined mathematically and, hence, mathematical techniques can be used to prove the correctness of the assertions made in the formal language. Ambiguity and incompleteness are eliminated in a correct formal language description. Formal specifications in software engineering use a formal language to describe the requirements of the system; formal specifications can also be proved to be correct using mathematical techniques.

4+1 View: The four views (and the fifth "view", the use case modeling) of software architecture as proposed by Philippe Kruchten of the Rational Corporation. These views included the logical view (the object model), process view (architecture at execution time), physical view (implementation onto hardware), and the development view (static software architecture within the development environment).

Geometric Model: An intermediate representation of the geometry of a graphical object for later display in a visual medium. The geometric model may consist of data structures

such as arrays, stacks, queues, matrices, etc., used for representing the geometric properties of objects.

IEEE: The Institute of Electrical and Electronics Engineers. A nonprofit, international organization involved with topics of interest to electrical or electronics engineering or related professions (such as applied physics and computer science). IEEE is widely recognized for setting standards and for publication of research.

IEEE Standard 1471-2000: A standard established by the Institute of Electrical and Electronic Engineers for the recommended practice of describing software architectures. This standard is perhaps the first major standard by a recognized standards body dealing with software architecture. It attempted to establish definitions for what was meant by a software architecture, a software architecture description, a software view and viewpoint, and so on, by establishing a broad consensus among many practitioners and academicians in software engineering.

Interface: In a general sense, an interface is the common face (or surface) shared by intersecting objects. In software architecture, the interface is the point at which a connection touches a component, analogous to a wall socket; the manner in which a component deals with a connector. An interface is typically assigned a role defining how the component and connector interact.

Layered Architectural Style: An architectural style in which components are seen as hierarchical layers; a lower layer provides a service to a higher layer. Connections between layers are kept hidden. The OSI Seven-Layer Network is an outstanding example of the layered style.

Legacy System: An existing system for which documentation may be minimal, missing or nonexistent. A software architecture may be extracted from a legacy system as a means of comprehending the older system.

Lex: For "lexical generator" or "scanner generator." Lex is a popular program that generates lexical analyzers. Lex is normally used in conjunction with yacc.

Lexical Analyzer: Also referred to as "lexer" or "scanner". The part of a compiler program that reads through the source program, establishing the valid tokens (such as keywords, operators, and variables or labels) in the language, and building a symbol table for later reference. Invalid strings are noted as errors by the lexical analyzer.

Model: A representation of a system, showing only those features of interest in order to reduce complexity; an abstraction.

Object: A member of a class which has existence (occupies space, persists through time, etc.). The instantiation of a class.

OMG: Object Management Group. An international consortium intended to set standards. OMG is most widely known for creating the CORBA standards for distributed object services.

Parser: Sometimes called "syntax analyzer." A phase of the compiler which accepts tokens from the lexer and, based on the grammar and the symbol table, determines whether or not the token's appearance is valid in the language. The parser creates a "parse tree" representing the source code statement; the parse tree is later traversed to generate the target code.

Regular Expression: A string from an alphabet generated by production rules defining the grammar of the language. Many programming languages are generated by regular expressions.

Regular Language: A language consisting entirely of valid regular expressions.

Scenario: A description of a possible sequence of events between actors, thus describing the behavior of a system. Scenarios are instantiated by use cases.

Semantics: The meaning of words or phrases, as opposed to their order (syntax).

Software Architecture: A discipline within software engineering that attempts to simplify the complexity of a software system by describing the overall system in terms of coarse-

grained structures, such as components, connections between components, and configurations of components and connections.

Software Homeostasis: The propensity of a software system to automatically "restore to normal or desired or equilibrium state when something occurs to upset or disturb that state" (from M. Shaw, *Sufficient Correctness and Homeostatis in Open Resource Coalitions*).

Software Traceability: (sometimes also called "requirements traceability"). The capability of tracing a path from a component in the requirements phase of a software life cycle model to one or more components in the later phases of software development (such as design, coding, and testing); also, the capability of tracing a path backwards from a component in a later phase, such as design, to a particular requirement. Software traceability is a technique for insuring conformance of design to requirements and insuring software quality control during maintenance.

Stakeholder: An individual or group with a "stake" (interest) in the success of the software development project. A stakeholder may be the client, manager, user, developer, or anyone else with an interest in the success of the software development project.

Style: See "Architectural Style".

Sufficient Correctness: (from M. Shaw): "The degree to which the system developer aspires to establish that the system meets its specifications, given constraints of time, cost, and limited knowledge."

Syntax: A description of the allowed order of words in a language; the form of words in the language.

Token: A string of characters from a valid alphabet in the language that form the building blocks of the language. Examples of tokens may be keywords, an arithmetic operator, a variable name, integer, and so on.

Topology: A description of the way components of the architecture are connected, implemented by a connectivity matrix.

Traceability: See software traceability.

UML: "Unified Modeling Language," a visual language for modeling various aspects of software development (especially analysis and object-oriented design). UML was a unification of the earlier object-oriented modeling techniques by Booch, Rumbaugh, and Jacobson. Modifications to UML have been proposed to make the language an architecture description language.

Use-Case: A software development technique intended to describe behavior of a system. A use-case is an instantiation of a particular scenario (see "scenario").

View: (from Egyed, 1999): "A piece of the model (used to represent some aspect of the real world) which is still small enough for us to comprehend and which also contains relevant information about a particular concern." A view is an abstraction on a model, hiding details which would make the model more complex, but revealing only those details of concern to us from a certain perspective.

Viewpoint: A template specifying a view on one or more software architectures. The viewpoint tells us the format for how a view is described. A viewpoint is to a view in much the same way as a class (in object-oriented programming) is to an object.

Viewpoint mismatch: An inconsistency in the way viewpoints are modeled. For example, suppose viewpoints are supposed to have a one-to-one mapping between components in their representations. If one viewpoint has many more components than the other, there is a mismatch.

Virtual Reality: A term referring to techniques for the realistic simulation of reality, typically using advanced three-dimensional computer graphics. Virtual reality is a paradigm that defines an entire world through which certain rules (navigational, movement, lighting, etc.) logically apply. Simulation is based on the assumption that a

viewer is immersed in this world and is considered to occupy a particular viewpoint within the world.

VRML: Virtual Reality Modeling Language. A versatile language intended to model worlds in three dimensions, using the virtual reality paradigm. VRML is designed to display its results on the Internet by means of an Internet browser; hence, VRML is platform independent.

Yacc: For "yet another compiler-compiler." Yacc is a popular program for generating compilers based on a grammar with semantic actions.

Z (Pronounced "Zed"): A formal language used for system specifications. It has been used to formalize architectural descriptions.

Summary

Software architecture is an emerging discipline within software engineering. Models based on software architecture attempt to reduce the complexity of a software system by representing the system with coarse-grained structures. A software structure could be represented by components and connections arranged in a specific topology. An architectural style defines the constraints on the topology and instantiation of the structure during run-time. Depending on the stakeholder viewpoint, elements of the topology are interpreted differently; a component, for example, may be an abstraction representing a program module, object, concept, or database.

Software architectures may be described using a text-based architecture description language (ADL). The key goals of an ADL are to communicate alternate designs between different stakeholders, to detect reusable structures, and to record design decisions. ADLs serve as tools to assist in analytical reasoning about the preliminary software design, to insure software quality early in software development.

A major problem in software architecture has been the difficulty of creating different representations to accommodate the contrasting viewpoints of stakeholders. A set of viewpoints should be conveyed in a way that is both comprehensive enough for specialists but understandable to generalists. The representation problem has been one of integrating different viewpoints without losing consistency (viewpoint mismatch) and without errors in relating architectural structures (architectural mismatch).

This dissertation provided a solution to the representation problem by creating a tool for three-dimensional representation of architectural viewpoints.

The tool consisted of an architecture description language (VTADL) to first describe the software architectures and viewpoints on the architectures; and a VTADL-to-VRML compiler to translate each viewpoint into a separate virtual reality world.

This research was significant since no compiler existed to translate a dedicated architecture description language into VRML. To the best knowledge of the dissertation author, very little or no research had been conducted on representing software architectural viewpoints in virtual reality.

An additional benefit of the VTADL-to-VRML compiler was the support for software traceability, the capability of tracing a path from elements of the architecture to associated requirements documentation. Using the VRML visualization, a stakeholder could trace the rationale behind the design using hyperlinks from elements of the visualization to source documents.

Chapter II

Review of Literature

History and Foundation of Software Architecture

The motivation of software architecture as a discipline within software engineering was the need to manage complexity by reducing complex structures to smaller, simpler parts. At the most fundamental level, a software architecture consists of discrete components and connections between the components, a form of decomposition of a system into coarse-grained parts.

For example, a complex system may be broken down into smaller pieces, perhaps four or five subsystems at the next level. Each of the four or five subsystems would, in turn, be broken down. At any given node in this hierarchy, no more than seven or so subsystems would lie directly beneath the parent. Traversing the tree structure, one could confront and manage a small number of subsystems at a given level. This decomposition process would continue until the leaves are reached, at which point we have reached the lowest level of detail.

The instinctive need for human beings to handle complexity by breaking the complexity down into a manageable number of pieces was empirically confirmed by psychologist George Miller [Miller, 1956] in his influential paper, *The Magic Number Seven, Plus or Minus Two: Some Limits on Our Capacity for Processing Information*.

Miller discovered that the average human being can handle no more than seven or so discrete entities of information at a given moment.

One of the earliest software engineers to note the advantages of decomposition was Edgar Dijkstra [Dijkstra, 1968]. Dijkstra was involved with the "THE" project, an operating system developed in the Netherlands. By using a "separation of concerns" during development of the operating system, the implementation and testing phase became less cumbersome, with the resultant savings in development time.

David Parnas's classic paper, *On the Criteria to be Used in Decomposing Systems into Modules* [Parnas, 1972] studied the criteria for breaking a larger software module into smaller modules. Among the criteria that Parnas used were the modifiability of independent modules, comprehensibility, and reduced development time (since independent modules could be developed in parallel by more than one software development team). Depending on the priorities of the developer, a software module could be decomposed in vastly different ways. Parnas's paper was one of the earliest papers to mention the benefits of information hiding – when decomposition is performed, the details below the current module are suppressed.

Hierarchical, modular decomposition with the use of data hiding of lower-level details set the stage for the emergence of software architecture. A precursor to software architecture description languages were the module interconnection languages (MILs), which were in vogue during the 1980s [Prieto-Diaz & Neighbors, 1986]. MILs focused on describing the interface between software components (or modules). The concentration on describing module interfaces was intended to assist in detecting errors in

integrating modules (which in software architecture is referred to as "architectural mismatch"), or to assist in reusing modules.

The paper by Perry and Wolfe [Perry & Wolfe, 1992] served as the catalyst for the emergence of the architectural approach. Earlier approaches akin to the architectural approach (representation of the key components and connections of a software system) were performed in an ad hoc manner. Perry and Wolfe analyzed these "first sketches" not as throwaway "rough drafts," but as important, albeit coarse-grained, designs. Their early model of software architecture consisted of elements (components), forms (connectors), and rationale (the reason behind the topology, based on system requirements). With software architecture now defined and acknowledged as a powerful new method of abstraction, other researchers refined and elaborated on the technique.

David Garlan and Mary Shaw [Garlan & Shaw, 1993] wrote a paper summarizing many of the key concepts of the new field. The same authors wrote the first highly influential book on software architecture [Shaw & Garlan, 1996], covering the gamut of software architectural concepts and methods. The authors emphasized the classification of architectures into different styles.

Architectural styles in software architecture are equivalent to the various styles used in the architecture of physical structures (e.g., Gothic, Victorian, Modern, etc.). Architectural styles were defined by the authors as the set of design rules which govern the types of components and connectors used, and the relational constraints between components and connectors. Garlan and Shaw provided several case studies demonstrating the use of architectural styles in software architecture. One of these case studies, based on Parnas's KWIC (Key Word in Context), has become a well-known

standard to demonstrate the idea of "decomposition." A future case study in architectural visualization may visualize the KWIC decomposition.

A novel architectural description language (ADL) called UniCon was also introduced and discussed by the authors. UniCon was capable of representing interconnection properties (protocols, messages, etc.) as well as components.

Shaw and her colleagues at the Software Engineering Institute at Carnegie Mellon University further developed UniCon and implemented it as one of the first workable architectural description languages [Shaw et al., 1995]. The focus was on developing tools to describe various architectural styles, with the profound hope that software architecture could help elevate software engineering into a "science of software" on a par with more traditional engineering disciplines.

With both foresight and confidence in the new approach, Shaw and Garlan proposed the educational curriculum for software architects (as a branch of software engineering).

Mary Shaw and Paul Clements further explored the idea of architectural styles in *A Field Guide to Boxology* [Shaw & Clements, 1997]. In this important paper, the authors provided one of the first comprehensive classifications of the different architectural styles. Prior to *A Field Guide to Boxology*, architectural styles were defined in an informal fashion. The styles were classified into the data-flow (based on the movement of data through components and connectors), call-and-return (such as a main program which calls subroutines), independent processes, data-centered repository (database or client/server), data sharing, and hierarchical styles (such as the Java virtual machine). What is most significant about this paper, however, is that the authors

described the criteria for applying each style to a particular problem. For example, if the problem involved transformations on continuous data streams, the pipelined architectural style was recommended.

In *Comparing Architectural Design Styles* [Shaw, 1995], Mary Shaw studied eleven different designs of the same automobile cruise-control system. She discovered that all eleven designs fell into one of four generic architectural styles: object-oriented, state-based, feedback-control or real-time styles. Each of the different styles focused on different aspects of the problem. In other words, the presence of different viewpoints was noted within each design. A given viewpoint would emphasize certain components and relations, and suppress other components/relations.

After analyzing the manner in which design styles were selected to model the solution, Shaw concluded that a systematic means was needed to establish the relations between multiple viewpoints (and to ensure consistency between viewpoints). Also needed were techniques to evaluate which architectural style would be most appropriate to the problem.

David Garlan was instrumental in developing an object-oriented architectural design tool called Aesop, as described in the paper, *Exploiting Style in Architectural Design Environments* [Garlan, Allen, & Ockerbloom, 1994]. Aesop was a generic architectural model capable of being instantiated into the desired architectural style; i.e., instantiated into a pipelined, call-and-return, or event-based style. Using a generic approach, rather than a style-specific architectural model, had several advantages – one was not limited to just one style and thus had more design versatility. The disadvantage

was that the stylistic constraints had to be incorporated into the methods of instantiated objects. The use of methods tended to obscure invariant properties of certain styles.

Software architectural techniques were compared to object-oriented design and design patterns [Monroe, Kompanek, Melton, & Garlan, 1997]. The strengths of architectural design – enhanced comprehensibility through data hiding, ability to communicate design decisions to different stakeholders, etc. – have been covered in the other literature and repeated here.

Noteworthy was the authors' discussion on the weakness of architectural design. The weakness was evident when one system used one architectural style, but another system used a different architectural style. Reusability between systems, or even compatibility between systems, was made more difficult by differing styles. The authors concluded that the three models were complementary; one model does not subsume the other.

Alexander Egyed and his co-authors discussed the idea of architectural families, in which a generic software architecture could be reused by many different software products [Egyed, Nikunj, & Medvidovic, 1999]. The idea of architectural families was to accelerate reuse of designs, dramatically reducing development time. Traditional architectural families were defined around software components; Egyed and his co-authors presented a method to design around software connectors. A taxonomy of software connectors was used to define and constrain the software families.

The difficulties of reusing software components were discussed in *Architectural Mismatch: Why Reuse is So Hard* [Garlan, Allen, & Ockerbloom, 1994]. The authors defined "architectural mismatch" as the incompatibility between software components.

Architectural mismatch may occur when off-the-shelf components are used to build a system, and the components cannot communicate with each other. Garlen, Allen and Ockerbloom related their experiences with architectural mismatch while developing the Aesop system, a tool for experimenting with software environments. They found that incorrect assumptions about the architecture, or miscommunication in the architectural description, were primary causes of mismatch.

The authors proposed several solutions to the architectural mismatch problem. One proposed solution was to devise techniques that make the assumptions about the architecture explicit, leaving no room for ambiguity. Another proposed solution was to construct the software components from "orthogonal subcomponents"; i.e., from independent modules. The authors recommended developing tools to guide the design process, and developing techniques for surmounting mismatches once the mismatches occur.

Within the past few years, the use of formal methods in software architecture has come to the fore. Formal methods make the assumptions about architectures explicit, eliminate ambiguities, and allow for logical reasoning about the architecture. With formal methods we can prove the "correctness" of an architecture (if our initial assumptions are valid). The disadvantages to formal methods lie in the costs to produce a formal description. The skilled personnel involved in writing a formal description must be knowledgeable in discrete mathematics as well as in the problem domain; such personnel must either be trained (a costly investment) or hired from the ranks of trained mathematicians or computer scientists (also a costly investment).

Formalizing Style to Understand Descriptions of Software Architecture [Abowd, Allen, & Garlan, 1995] applied the Z formal language to describe the components, connections and properties of software architectures. Significantly, the authors mathematically formalized the idea of an architectural style.

The authors presented the argument that the cost of producing a formal architectural description for a product family is worthwhile. The investment is in producing a *family of systems*, rather than an individual system. Formalization of the properties and styles of product families would be applicable to a wide spectrum of products.

The advantages and disadvantages of Z notation to software architectures are also explored by the dissertation work of one of the co-authors of the above paper, Robert J. Allen [Allen, 1997].

Since the inception of software architecture in the early 1990s, the initial flurry of activity defining the nature of this new discipline has given rise to books about the applications of software architecture to large-scale development projects. Noteworthy among the recent "applications" books is *Software Architecture in Practice* [Bass, Clements, & Kazman, 1998]. The authors were software engineers with the Software Engineering Institute (SEI) of Carnegie Mellon University. The process of developing and refining a software architecture was viewed as an iterative process that must be integrated throughout the software life cycle. This application-oriented model was referred to as the "architecture business cycle." Software architectures were evaluated from the standpoint of software quality assurance and the degree of reusability.

Applications of software architecture to real-world projects were described through numerous case studies. The A-7E Avionics System used software architecture to simplify the overall structure of the system and insure reusability. Most interesting was a case study on the architecture of the World Wide Web, and the impact that this architecture had on the organization that designed it (the Web's structure far outgrew its anticipated size). The architecture of the CORBA (Common Object Request Broker Architecture), designed by the Object Management Group to achieve interoperability between different software products, was also examined in light of the goals of the businesses that supported the standard.

Other case studies included an air traffic control system, a flight simulator, and a corporation (CelsiusTech) that implemented a product line based on reusable software architecture.

In a similar vein, but taking a more European perspective, *Software Architecture for Product Families* [Jazayeri, Ran, & van der Linden, 2000] described the work of ARES (Architectural Reasoning for Embedded Systems). The ARES project was funded by the European Commission and lasted from December 1995 to February 1999; it involved six European partners, three industrial and three academic.

Each research group reported on their successes or failures in the use of software architecture. The emphasis of the research was architecture-centric software development. Software architectures were developed with reusability in mind; specifically, the ability to create entire product families based on similar architectures.

Software architecture as a distinguishable branch of study within software engineering has matured to the point where the Institute of Electrical and Electronics

Engineers have released their first standard, the *IEEE Recommended Practice for Architectural Description of Software-Intensive Systems, IEEE Std. 1471-2000* [IEEE, 2000]. IEEE Standard 1471-2000 is significant in that it is the first standard by a major organization to address software architecture explicitly. A consensus on various definitions and practices (such as on software architecture, views, viewpoints, architecture description, and so on) was reached by a standards committee, the Architecture Working Group, comprised of influential members of the software architecture community from both industry and academia. IEEE Standard 1471-2000 was as generic as possible in advising how architectures should be described; no one architecture description language was recommended over another. Instead, a standardized template containing the necessary properties of an architecture (components, connectors, configuration, etc.) was provided.

The standard was not without its critics. Rich Hilliard, a member of the original Architecture Working Group which established IEEE Standard 1471-2000, stated in his position paper, *IEEE Standard 1471 and Beyond*, that too much emphasis was placed on conceptual modeling while lacking a "larger context needed in most practical, industrial-strength applications" [Hilliard, 2001]. However, Hilliard admitted that the standard was "worth appreciating relative to current work in architectural research and practice."

In 1995, David Garlan and Dewayne Perry, among the pioneers (along with Mary Shaw) of software architecture, summarized several active areas of research [Garlan & Perry, 1995]. The areas of research they pinpointed were architectural description languages (i.e., the problems associated with representing different viewpoints), formal software architecture, architectural analysis techniques, architectural development

methods, architectural archeology, architectural codification and guidance, and architectural tools and environments. Most of these areas of research have accelerated since 1995.

Though this dissertation research focused primarily on software architecture representation (textually and visually), the research itself may be classified into three of the topics mentioned by Garlan and Perry. This dissertation can be considered an architectural tool, and the resulting description and visualization can be considered architectural description languages and architectural guidance. The resulting tool can, arguably, be used to derive and visualize legacy code, placing the research into a fourth research topic as well (software archeology).

Software Architecture in the Software Development Process

As mentioned in Chapter I, software architecture has grown to encompass the entire software development life cycle, incorporating elements of the requirements, design, implementation and maintenance phases into the architectural framework.

The works of object-oriented pioneers Grady Booch, Ivar Jacobson, and James Rumbaugh are highly significant to software architecture. Their most recent work, *The Unified Software Development Process* [Jacobson, Booch, & Rumbaugh, 1999], described an object-oriented and architecture-centric software development process. Their visual modeling language for representing software design, called the Unified Modeling Language (UML), combined the earlier methods of the authors into a single method. UML has rapidly become an industry standard in the modeling of software design. *The UML User Guide* [Booch, Rumbaugh, & Jacobson, 1999] gives specific details on how UML is used to model object-oriented development using an iterative, evolutionary life cycle model.

The earlier work by object-oriented pioneer Grady Booch [Booch, 1994] supplements the unified method book. Booch's overall discussion on the nature of modularity, hierarchy, and the object-oriented view of the world is outstanding and should be read for its own sake. Booch's classified bibliography is highly comprehensive (as of the early to mid-1990s).

Assuring the quality of the software architecture as it evolves along with the software development life cycle is of paramount important. Abowd et al. discussed techniques for evaluating software architectures in *Recommended Best Industrial Practice for Software Architecture Evaluation* [Abowd et al., 1997]. Prior to this paper,

scant research was focused on insuring the quality of software architectures during the development process.

Len Bass, Paul Clements and Rick Kazman of the Software Engineering Institute at Carnegie Mellon wrote the seminal work on incorporating software architecture in real-world applications. *Software Architecture in Practice* [Bass, Clements, & Kazman, 1998] discussed in depth how the architecture may be incorporated into the business cycle of development. Significantly, the authors incorporated numerous case studies from actual development environments: the development of the World Wide Web, CORBA, and an air traffic control system were explored from an architectural perspective. Successful cases where reusability was achieved through software architecture were discussed.

In *Taming Architectural Evolution*, authors van der Hoek et al. described a novel architectural evolution environment called Mae [van der Hoek, Rakic, Roshandel, & Medvidovic, 2000]. Mae modeled changes to the software architecture in a way analogous to more established configuration management techniques. This paper is of interest because the authors described how changes to software architecture could be modeled as the overall system evolves. There is a possibility that the visualization techniques developed by this dissertation could be used to model software evolution.

Software architecture is increasingly used to model legacy systems, especially when minimal or no documentation exists. Jazayeri et al. [Jazayeri, Ran, & van der Linden, 2000] provided several case studies in "software architecture recovery," also known as software archeology. The overriding theme of the authors was the derivation

or recovery of general architectural properties, so that the software architecture could describe an entire product family.

Ivan Bowman performed a notable architectural recovery on the Linux operating system. In *Conceptual Architecture of the Linux Kernel* [Bowman, 1998] and *Concrete Architecture of the Linux Kernel* [Bowman, Siddiqi, & Tanuah, 1998], Ivan Bowman created two-dimensional visualizations of the conceptual and concrete software architectures of the Linux operating system kernel. The software architectures were formulated with only minimal or nonexistent references.

Software architectural recovery is a form of reverse engineering. Ted Biggerstaff of Microsoft Corporation [Biggerstaff et al., 1993] called the problem of deriving the human concepts behind computer code the "concept assignment problem" in understanding existing program code. In other words, aspects of the program code are mapped to the conceptual model. In software architectural terms, this is equivalent to deriving the conceptual view from the implementation model.

Biggerstaff isolated two types of concepts – human-oriented concepts and program-oriented concepts. While program-oriented concepts tend to be more precise and are logically recognizable by an automated parser, human-oriented concepts are fuzzier and more ambiguous. The authors developed a program understanding assistant called DESIRE (for "Design Recovery"). The DESIRE reasoning system provided a domain-specific, a priori knowledge base to allow inference of formal information types from informal information types, and vice versa.

The authors believed that a completely automated design recovery system would not be possible, due to instances of incomplete, contradictory or nonexistent knowledge

about the program code. But the authors did conclude that a recovery assistant such as DESIRE could greatly accelerate and simplify the manual derivation of concepts from legacy code.

The Internet has affected how software architectures evolve, and how we may judge the correctness of software architectures. Traditional software architectures were relatively static. In a technical report published by Carnegie Mellon university, entitled *Sufficient Correctness and Homeostasis in Open Resource Coalitions: How Much Can you Trust your Software System?*, Mary Shaw pointed out that software architectures on the Internet are dynamic and constantly evolving, demanding an updated criteria on how we gauge the correctness of software architectures [Shaw, 2000].

Dr. Shaw proposed the idea of "sufficient correctness" when dealing with the time constraints and limited information of the Internet. Distributed software systems could automatically monitor and correct their own behavior when a deviation from a norm is detected, much like an air conditioning system uses a thermostat to maintain a sufficiently correct temperature. The monitoring and correction of dynamically evolving architecture was called "software homeostasis" by the author. She argued that software homeostasis could be used to verify software architectural quality with the sufficient correctness of a fuzzy system.

Software Architecture and Design Patterns

Design Patterns: Elements of Reusable Object-Oriented Software [Gamma, Helm, Johnson, & Vlissides, 1995] is a compendium of design patterns intended for reuse by object-oriented developers. It is a "bible" of known object-oriented design patterns and

represents an effort to standardize reusable patterns within the object-oriented software design community.

Many of the ideas behind design patterns are pertinent to software architecture. However, Hofmeister et al. point out [Hofmeister, Nord, & Soni, 2000] that design patterns may be relevant to architecture or they may be relevant only to detailed design. Design patterns may be considered architectures when the design patterns describe interactions between architectural elements. Design patterns are relevant only to detailed design (and not architecture) when the design patterns describe interactions within architectural elements. In other words, when the design patterns describe detail more fine-grained than the architectural elements, they are no longer part of software architecture.

Christopher Alexander [Alexander et al., 1979; Alexander, Ishikawa, & Silverstein, 1977], in a two-volume work about using patterns in the construction of buildings (*The Timeless Way of Building* and *A Pattern Language*) influenced an analogous pattern language movement in object-oriented programming. Alexander et al. take an almost Taoist approach to defining patterns for architectural construction; they indicate that the best and most universal patterns come from the inner knowledge locked within the human being, flowing naturally from the essence of the requirements. Since the architecture of physical artifacts consists of relating components to one another, and the overall philosophy of reusing patterns to build successful structures is such a natural and eloquent one, it was inevitable that software engineers would see the analogy to the development of software components.

Software Architecture Description Languages

The architecture description language defined by this dissertation research, VTADL (Visually Translatable Architecture Description Language) was patterned after a language called ASDL (for Architecture Structure Description Language). ASDL was originally developed to assist in architecture recovery from legacy systems; the language was intended to convey basic architectural properties (configuration, basic components, information interchange), while ignoring properties such as interface behavior [Eixelsberger & Gall, 1998].

Elements defined by the language seemed unambiguous and straightforward to visualize. However, the language did not extend to defining viewpoints. Additional language syntax was needed to allow different architectural styles and explicit positions (e.g., left, right, top, bottom) when positioning could be ambiguous. Thus, a new language was devised (VTADL) and is defined in more detail in Chapter Three and the Appendix.

Though mentioned in the History and Foundation section of this literature review, the 1995 paper by Shaw et al. was seminal in the field of software architecture description languages and is worth repeating here. *Abstractions for Software Architecture and the Tools to Support Them* [Shaw, DeLine, Klein, Ross, Young, & Zelesnik, 1995] was one of the first papers to discuss the idea of software architecture description languages for describing different architectural styles, and also for implementing the architecture beyond the design. One of the first ADLs, called UniCon, was presented. The authors expressed the hope that ADLs, along with other software

architectural tools, could lead to a science of large-scale systems, just as there is a science of algorithms.

By the late 1990s, enough ADLs had been created to merit a classification of the languages based on various criteria. In two important papers, Nenad Medvidovic and his co-authors provided such a classification.

Domains of Concern in Software Architectures and Architecture Description Languages [Medvidovic & Rosenblum, 1997] classified the architectural domains of architecture description languages. For example, refinement and simulation were two different application domains. The ADL called "C2" was used to represent simulation and event filtering, while Darwin was used for hierarchical views on the architecture. The ADL called "MetaH" was ideal for static analysis of a parser or for security analysis of an architecture.

In *A Classification and Comparison Framework for ADLs* [Medvidovic & Taylor, 2000], the authors classified architecture description languages according to key criteria. According to the criteria, an ADL must be capable of describing (to varying degrees) the components, connectors, and configuration of an architecture. An ADL must also have tool support. Furthermore, each type of description must possess, to greater or lesser extent, key features such as interfacing capability, typing, semantics, constraints, evolution, and non-functional descriptive capability. Using this classification scheme, the authors compared languages such as ACME, C2, Aesop, Darwin, Metcalf, Rapide, SADL, UniCon, Weaves, and Wright.

The use of formal languages for architecture description appeared almost at the very beginning of software architecture as a discipline. The dissertation by R.J. Allen, *A*

Formal Approach to Software Architecture [Allen, 1997], and the paper by Abowd et al. [Abowd, Allen, & Garlan, 1995] were discussed at length in the History and Foundation section of this literature review.

However, a fascinating formal language using a chemical model to describe software architectures was discussed by Inverardi and Wolf. In *Formal Specification and Analysis of Software Architectures Using the Chemical Abstract Machine Model* [Inverardi & Wolf, 1995], the authors reported on their research with CHAM, the Chemical Abstract Machine Model, a formal language based on chemical reactions. Since CHAM was a formal language, it could be used to reason and prove the correctness of architectures; CHAM could also be used to formalize the idea of architectural styles. The authors concluded that CHAM might be "...one useful tool in the software architect's chest of useful tools," since no one formal language could be all things to all architectures.

Using an architecture description language to describe and visualize a neural network was discussed by Inouye in *An Architecture Description Language for Visualizing Neural Network Designs* [Inouye, 2001]. The style used for the neural net description was not implemented in this dissertation.

Rapide was an architecture description language intended for event-driven and concurrent systems [Luckham, Kenney, Augustin, Vera, Bryan, & Mann, 1995]. Components were viewed as states and connectors were interpreted as transition arcs between the states.

By describing architectural families instead of single architectural structures, software development time could be significantly reduced. In *Software Connectors and Refinement in Family Architectures* [Egyed, Nikunj, & Medvidovic, 1999], the authors expanded upon the idea of architectural families by including generic software connectors (traditional architectural families were defined using software components). The idea of architectural families is to accelerate reuse of designs and to dramatically reduce overall development time. Egyed et al. presented a taxonomy of software connectors to define and constrain the architectural families. The authors presented a case where using software connectors could increase the flexibility of product families and allow for the automation of architecture refinement. They mentioned the development of automated tools to detect architectural mismatch and to ensure consistency between architectures.

With the wide number of versatile ADLs available by the late 1990s, some software engineers began to consider ways of integrating the various languages. The idea of a canonical "ADL toolkit" was proposed by Garlan et al. in *Towards an ADL Toolkit* [Garlan, Ockerbloom, & Wile, 1998]. The ADL toolkit used an ACME language environment. ADL translators were capable of translating ACME to UML, ACME to MetaH, C2 to ACME, and so on. Using ACME as the intermediary, several ADLs could be translated to one another. However, the idea of translating an ADL to ACME, and ACME to VRML, has not been explored and would involve geometric and computer graphics issues. The translation of an ADL to VRML was, of course, the topic of this dissertation.

A very recent and interesting trend has been the idea of representing a known ADL using the Extensible Markup Language (XML). The authors at the University of Southern California's Center for Software Engineering represented several ADLs, including ACME, C2, SADL, and Darwin, using XML tags [Dincel, Roshandel, & Medvidovic, 2000]. The authors expressed the goal of making ADLs interact with one another through connectors in the XML environment.

The reasoning and philosophy preceding IEEE Standard 1471-2000 (the first standard on software architecture from a recognized organization) was discussed in *Toward a Recommended Practice for Architecture Description* [Ellis, Hilliard, Poon, Rayford, Saunders, Sherlund, & Wade, 1996]. The authors reported on the work of their Architecture Planning Group (chartered by the IEEE) to standardize the practice of software architectural descriptions. The IEEE standard was ultimately finalized and released four years later (in 2000) as IEEE Std. 1471-2000.

The *Recommended Practice for Architectural Description of Software-Intensive Systems* [IEEE, 2000] stands as an important milestone in software architecture description. A consensus on the definition of software architecture, view, viewpoint, on what information constituted an architecture description, and so on, was reached by members from both industry and academia in the field of software engineering.

Like any standard, some of the definitions or explanations in the standard – no matter how rigorously described – could still be open to interpretation. The *IEEE Std. 1471 Frequently Asked Questions (FAQ)* [Maier, Emery, & Hilliard, 2001] document was one attempt to correct this. The authors pointed out that the standard was non-restrictive, not enforcing a particular ADL or viewpoint template. The standard simply set

guidelines for what should be included in an architectural description: common sense items such as identification of stakeholders, definition and selection of viewpoints, documentation standards for viewpoints, recording viewpoint inconsistencies, and recording rationale behind the architectural decisions.

Rich Hilliard's position paper, *IEEE Std. 1471 and Beyond* [Hilliard, 2001] discussed the ramifications and use of the new standard, attempting to dispel many of the misconceptions.

Software Architectural Viewpoints

Phillipe Kruchten, of the Rational Corporation, defined software architecture in terms of different views on the software development process [Kruchten, 1995]. Krutchen's views were developed in response to the problem of representing software architectures for the different stakeholders involved in the development process. He claimed that any view of the software architecture could be folded into one of four possible classifications of software viewpoints. The *logical view* describes an object model of the environment when object-oriented design methods are used; as an alternative, the logical view can be represented by an entity-relationship diagram in traditional structured programming. The *process view* describes the architecture at execution time, covering aspects such as concurrency and synchronization. The *physical view* describes the mapping of software implementation onto the hardware, whereas the *development view* describes the static software components during development time.

Each of the four views can be illustrated using a fifth view, the *use case scenarios* where the interrelationships between components of the view are instantiated.

An equally effective but alternate set of views was developed by Soni, Nord and Hofmeister of the Siemens Corporation and described in their paper, *Software Architecture in Industrial Applications* [Soni, Nord, & Hofmeister, 1995]. The authors studied several software systems built by the Siemens Corporation and discovered that multiple architectural views were desirable. They classified the views into conceptual architectures, module interconnection architectures, execution architectures, and code architectures. These views differed from the 4+1 View Model as proposed by Kruchten,

and arguably could not be folded into one of Kruchten's views. The contribution of this paper was to show that alternate view models beyond the 4+1 View Model were viable.

Christine Hofmeister, Robert Nord and Dilip Soni applied their model of software architectural views to actual products at the Siemens Corporation as described in *Applied Software Architecture* [Hofmeister, Nord, & Soni, 2000]. It was interesting and highly informative to see how the authors abstracted the general software architecture of each commercial product into their defined views.

In *A Three-Faceted View of Information Systems*, authors Giorgio de Michelis et al. [de Michelis, Dubois, Jarke, Matthes, Mylopoulos, Schmidt, Woo, & Yu, 1998] do not refer specifically to software architecture, but to managing software change in general. Software change is managed through the use of software traceability, the capability of following a path between requirements documentation and software components established later in the development process. This paper was included in the literature review due to the classification of software change into three major sources: system-related views, group collaborative views, and organizational views. Since the visualized architectural structures of this dissertation (represented within VRML worlds) were hyperlinked to requirements, organizational, or other documents, the three viewpoints mentioned by these authors were of note.

In *Integrating Architectural Views in UML* [Egyed, 1999], Alexander Egyed examined the nature of views in software architecture and how architectural views could be implemented in the Unified Modeling Language (UML). Egyed presented the argument that views should be both independent and related to one another from the standpoint of a consistent model of software. When different views within a model are

inconsistent (e.g., cardinality within the different views are inconsistent, components mean different things, etc.), we have what is called *viewpoint mismatch*, a more general form of architectural mismatch. Egyed discussed the generic types of viewpoint mismatch based on different possible views and proposed methods to detect and correct them. The possibility of automating the task of detecting viewpoint mismatch was proposed as a major benefit to software architecture.

Because UML alone was limited in representing architectural views, Egyed and Medvidovic proposed that architectural description languages such as C2 and Wright be integrated into UML using extensions [Egyed & Medvidovic, 1999]. While this solution was notable, their more general insights were of greater value.

Egyed and Medvidovic argued that three things were necessary for successful view integration: mapping, transformation, and differentiation between the views. "When we talk about the need to integrate views," the authors wrote, "we are really talking about the need of having a system model integrated with its views."

Dr. Egyed described a technique to integrate views using architectural patterns [Egyed, 2000]. Architectural patterns were used to map existing views to one another, or to transform one view into a separate view. The author also demonstrated how architectural patterns could be used to evaluate the correctness of architectural views.

In *Viewpoint Modeling* [Hilliard, 2001], the author proposed that a view should first be defined in a given viewpoint language, and then translated or conformed to UML. Hilliard (one of the committee members instrumental in forming IEEE Standard 1471-2000, the standard for architectural description discussed in detail in the section on

architectural description languages) proposed a standard template for modeling architectural views.

In this dissertation, however, views were first described in VTADL and then translated into VRML.

Visualization of Software Architectures

The focus on techniques to visualize software architecture is relatively recent, having begun in earnest in 1997.

The paper *The Artistry of Software Architecture* [Boasson, 1998], along with the paper *3D Visualization of Software Architectures* [Feijs & de Jong, 1998], served as catalysts for this research.

Boasson pointed out the aesthetics inherent in software architecture. Software design is similar to artistic design in the sense that designers are producing complex structures that may have appeal to our sense of form; a great masterpiece of software design, the author argued, is no less valuable than other works of art that have appeared in cultural history. Another similarity between software design and artistic design is that designers are not limited by physical constraints. However, software designers are limited by utilitarian constraints (just like physical architecture and other engineering disciplines). Boasson emphasized the need for formal mathematical methods to analyze software representations in order to reduce the chaos of large-scale system development.

The paper by Loe Feijs and Roel de Jong of the Phillips Corporation discussed a visualization tool which converted an architectural representation (stored in a relational database) into a VRML file. The authors expressed the need for a tool to generate not only a 3D visualization of a software architecture, but also hyperlinks from the visualized components to relevant multimedia documents. This dissertation intended to develop a tool to allow traceability from aspects of the visualization to hyperlinked multimedia documents.

In another paper, Loe Feijs and his co-authors at the Phillips Corporation used mathematical relations as the basis to represent software architecture [Feijs, Krikhaar, & van Ommering, 1998]. Tools were developed to extract relations from the structural information contained in source code and documentation. The extracted relations were stored in a Microsoft Access database and visualized as graphs with a visualization tool called TEDDY. However, in this paper and the one mentioned earlier, Feijs et al. did not visualize an architecture directly from an architecture description language but, rather, from a relational database.

Though the papers by Boasson and Feijs et al. served as the inspiration to the author of this dissertation, the motivation behind architectural visualization can be traced to earlier work.

David Harel presented a strong argument that complicated computer systems could be better understood by visualizing the systems using a small number of graphical elements that are topological, rather than geometric, in nature [Harel, 1988]. He referred to the diagrams as "visual formalisms," and provided an example of a visual formalism called a higraph. A higraph combines the idea of a Venn diagram (which represents sets) with the idea of a hypergraph (for representing relations that are not always binary). The resulting visualization was quite versatile when visualizing concepts such as the Cartesian product and finite-state machines.

Harel believed that the formalisms would encourage more visual modes of thinking in managing the intricacies of complex software.

R. C. Holt discussed the use of directed graphs to represent software architectures [Holt, 1996]. The directed graphs are "typed" (or "colored") in the sense that the graphs

may have more than one type of edge. Holt then used binary relational algebra to formalize the idea of architectural styles. This paper is important (but not necessarily seminal) from the standpoint that the paper demonstrated how the mathematics of binary relations can be used to formalize a description of the architecture of a system. In the words of the Holt: "These visual formalisms provide the advantage that they bring the mathematical formalism close to the mental images that many people visualize."

The stated goal of the authors of *Architectures with Pictures* [Buhr & Casselman, 1992] was to reconcile different ways of conceptualizing software architectures. Buhr and Casselman provided a preview on many of the motivations for visualizing software architecture; their prescient work appeared before the emergence of software architecture as a distinct branch of study.

The authors visualized software architecture using a two-dimensional notation. The visual notation was comprised of "wired" and "wireless" models. With wired architectures, the relations between components were represented as static wires; with wireless architectures the relations between components were seen as dynamically linked. The authors demonstrated how more intricate interrelationships between components could be represented by factoring the wiring diagrams into simpler diagrams (a form of visual decomposition).

The wired and wireless models were viewed in terms of contracts (establishing how independent objects interact to accomplish tasks), roles (the job that the components play when entering and leaving contracts), wires, softlinks (the dynamic link in wireless models), and timethreads (a time-based path through an architectural design intended to offer a behavioral trace).

Hence, this versatile and early model of software architecture visualization provided not only topological information, but software traceability to the rationale behind the design.

An architectural visualization of the Linux operating system [Bowman, 1998; Bowman, Siddiqi, & Tanuah, 1998], available on the Internet, was studied as an example of how an architecture could be visualized in the plane. This dissertation research intended to use a portion of the planar Linux visualization and project various components and connections into a three-dimensional, virtual world using VRML.

The empirical studies of Colin Ware and Glenn Franck on the effectiveness of three dimensions in visualizing software architecture are of great significance to this dissertation. In *Visualizing Object Oriented Software in Three Dimensions* [Ware, Hui, & Franck, 1993], the authors compared the human comprehension (among several human subjects) of a complex structure visualized in two dimensions to the same structure visualized in three dimensions. It was found that the three-dimensional visualization substantially reduced the error rates in the comprehension of the complex structure.

The same authors refined their experimental studies on three-dimensional visualization in *Evaluating Stereo and Motion Cues for Visualizing Information Nets in Three Dimensions* [Ware & Franck, 2000]. Experiments were conducted on a number of human subjects to measure how well the subjects could trace a path through a complex two-dimensional and three-dimensional graph. The error rate of tracing the path was significantly reduced (by a factor of three) when using three dimensions. Furthermore, allowing the subject to rotate his or her head relative to the three-dimensional structure resulted in dramatically increased comprehension of that structure.

Visualization of Large Nested Graphs in 3D: Navigation and Interaction [Parker, Franck, & Ware, 2000] is a remarkable paper that discussed the techniques for visualizing very large graphs.

When viewing large graphs with a tremendous number of nodes and arcs (at least thirty or more), the images become muddled and complex. The authors defined the generic "focus-context" problem with large graphs: it is desirable for information to be provided for the overall large structure, yet important to be able to view arbitrarily small details. The traditional techniques for solving the focus-context problem were discussed (rapid zooming, distortion techniques, component hiding, multiple windows, and three-dimensional visualization). The authors were partial to three-dimensional visualization, and described a system called NV3D that allowed visualization of very large graphs with nested structures. Visualization of dynamic behavior was achieved using a "snake," the animation of arc behavior over time to allow tracing of execution threads. Snake animation could be superimposed over the larger static structure to attract user attention and provide an execution history of the behavior.

NV3D was being applied to a commercial environment at Nortel Corporation, with applications to software training (visualization of software architectures to indoctrinate new programmers), code management, and visualization of execution paths through the program structure.

While the NV3D software may resemble the research of this dissertation, it is important to point out the differences: this dissertation used an architecture description language as a starting point and mapped elements described in the ADL to elements of the architectural structure in VRML based on the architectural style. One viewpoint

defined in the ADL was mapped to one world defined by a separate VRML file. The VRML worlds may be linked to one another, and elements within each architecture may be hyperlinked to source documentation. The intention of this dissertation was to map the architectural description language to a virtual-reality environment using the VRML language, and not to serve as a sophisticated 3D visualizer of large-scale graphs. The dissertation took a purely software architectural approach and operated on the assumption that the graphs would be relatively coarse-grained at the outset, with details kept hidden. Lower-level details in a potentially large structure would be unveiled only as needed (the traditional "component hiding" approach was taken). Furthermore, the VTADL-to-VRML compiler was intended as a relatively inexpensive application capable of being viewed on the Internet.

Algorithms for the effective modeling of hierarchical graphs in three dimensions were discussed by Sugiyama, Tagawa and Toda in *Methods for Visual Understanding of Hierarchical System Structures* [Sugiyama, Tagawa, & Toda, 1981]. The algorithms provided here were of special interest to the visualization of software architectures. Sugiyama et al. emphasized algorithms which would make graphs more "readable." The authors defined several criteria for determining a readable graph. A hierarchical graph was considered readable if it had a minimal number of crossed edges per level (since crossed edges make for a more complicated graph), if the graph's long span edges were straight, if the layout of edges was balanced, and if the layout of nodes in the graph were close to one another. Sugiyama et al. provided both theoretical and heuristic methods to insure that their criteria for readability could be met.

The research of this dissertation did not use the algorithms by Sugiyama et al., since the VTADL-to-VRML geometric algorithms were already implemented prior to reading the paper. However, the algorithms provided in the paper may be of use in future research where large number of nodes and more arbitrary hierarchical structures are encountered.

Papakostas and Tollis provided two algorithms for drawing orthogonal graphs in three dimensions in which edges do not cross, fulfilling a criterion for graph readability as defined by Sugiyama et al. [Papakostas & Tollis, 1999]. The algorithms were analyzed for time complexity based on graph size and were shown to be linear-time in efficiency. This dissertation did not use their algorithms but the existence of this fundamental and graph theoretic research was noted for future versions of the VTADL-to-VRML intermediate geometric model.

A standard reference on VRML is the *VRML 2.0 Sourcebook* [Ames, Nadeau, & Moreland, 1997]. This book discussed the virtual reality paradigm, VRML in the context of the Internet, and provided detailed syntax and examples on the language itself.

The wider literature on the target language of this dissertation research, VRML, is relatively informal. The Web3D Consortium Home Page is the definitive source of up-to-the-minute information on VRML and related technologies [Web3D Consortium, 2001]. The Web3D Consortium is a non-profit trade association consisting of both corporations and individuals involved with modeling three-dimensional graphics on the Internet (some of the larger corporations include Sony and Phillips). The Web3D Consortium inherited the standardization and specification tasks from the San Diego Supercomputer Center in the late 1990s. The Web3D site is comprehensive, providing

the original specification for VRML 1.0 and the most recent version, VRML 2.0 (referred to as "VRML 97"). In 2001, the functionality of VRML was incorporated into the Extensible Markup Language (XML) using a new standard defined by the Web3D Consortium, X3D. X3D is intended as a successor to VRML, since X3D can be run directly on an Internet browser without a VRML add-on. Existing VRML applications, however, will remain compatible to X3D. The X3D standard was released on May, 2001; a specification update was released on November 25, 2001.

General References

Key references used in the development of the ADL-to-VRML compiler include the classic compiler text widely known as the "dragon book" [Aho, Sethi, & Ullman, 1986], an outstanding reference on lex and yacc [Levine, Mason, & Brown, 1992], and the "bible" of VRML, the *VRML 2.0 Sourcebook* [Ames, Nadeau, & Moreland, 1997].

The compiler text by Aho et al. was used to gain a better understanding of the ADL-to-VRML translation. The book *lex and yacc* by Levine et al. was invaluable when implementing the shell of the actual compiler for the ADL. The *VRML 2.0 Sourcebook* by Ames et al. was invaluable for understanding the target language, VRML, and the visualization capabilities of the target language.

The widely read text by Hearn and Baker, *Computer Graphics* [Hearn & Baker, 1986] was used for quick references on matrix algorithms for geometric translations and transformations, and for geometric modeling for computer graphics.

While programming the data structures and subroutines using the C language, a textbook by J. Antonakos and K.C. Mansfield, *Practical Data Structures Using C/C++*

[Antonakos & Mansfield, 1999] provided excellent description on standard algorithms and abstract data types, greatly assisting in the implementation of the intermediate geometric model and VRML code generator.

During research into architecture descriptions with UML, two reference works were used outside of academic literature. Pierre-Alain Muller's *Instant UML* [Muller, 1997] served as a handy reference when studying UML tools and concepts; and Terry Quatrani's *Visual Modeling with Rational Rose 2000 and UML* [Quatrani, 2000] provided a vendor-specific reference on the popular Rational Rose implementation of UML.

Summary of Knowns and Unknowns

Architecture description languages (ADLs) may be text-based or graphical and are used explicitly for the purpose of representing software architectures. Different ADLs place emphasis on different aspects of software architecture. UniCon focused more on connectivity, for example, than ADLs that preceded it [Shaw et al., 1995]. Rapide [Luckham et al., 1995] was an ADL for event-driven and concurrent system architectures, while Wright was one of the first formal architecture description languages.

The Unified Modeling Language (UML), a graphical notation for representing object-oriented designs and behavioral analysis (such as scenarios and use cases), has become a de facto standard in the software development industry for modeling software development. However, UML is not considered an ADL since UML does not model configurations of architectures explicitly, one of the criteria for defining an ADL [Medvidovic & Taylor, 1999]. UML does not adequately describe the details of connector or component attributes to the extent of dedicated ADLs. Attempts to model components or connectors with UML have been made through using extensions to UML, but these architectural extensions are not native to the language. UML has no capability for modeling relationships between views.

Hence, there continues to be a need for ADLs, at least until UML 2.0 contains the capability for explicit and intentional architectural descriptions. However, even UML 2.0 will still be a two-dimensional representation. It will still be designed with object-oriented, notational design in mind, rather than as a virtual medium inviting exploration of three-dimensional structures.

Feijs & de Jong claimed (from a subjective standpoint) that visualizing software architectures with VRML can make the representation of software architectures more engaging and more comprehensible, perhaps even assisting in detecting reusable patterns in the architecture. However, the authors did not visualize an ADL explicitly; rather, they defined the software architecture through a relational database and translated the connections into VRML.

Parker et al. presented strong, empirical evidence that using three-dimensional visualizations, along with user-directed movement within the three dimensions, greatly enhanced the user's comprehension of complex structures in comparison to planar representations [Parker, Franck, & Ware, 2000]. This evidence, along with related research [Ware, Hui, & Franck, 1993; Feijs & De Jong, 1998] form the basis for translating an ADL into the VRML format.

The inherent capability of VRML in representing distinct worlds using the virtual reality paradigm is a powerful yet natural medium for modeling different stakeholder viewpoints [Ames et al., 1998]. VRML is designed for rendering on an Internet browser, and so is platform-independent and available at minimal cost.

Surprisingly, only a handful of researchers have used VRML to represent software architectures [Feijs & de Jong, 1998]. To the best knowledge of this researcher, no research has yet focused on using VRML as a medium to explicitly create viewpoint models on software architectures. The inherent capability of VRML to represent a distinct world using the virtual reality paradigm is a prime advantage, along with VRML's availability on the Internet. A world can easily be interpreted as a viewpoint

containing multiple structures. Each structure could be readily understood to be software architectures contained within the world representing the view.

Furthermore, to the best knowledge of this research, no compilers existed which directly translated an ADL to VRML.

The ability to represent viewpoints using three-dimensional interaction, along with traditional decomposition techniques, can lead to easily comprehended and aesthetically engaging software architectures.

Contribution to the Field of Software Architecture (Software Engineering)

With the shortcomings of UML in mind, and the continuing need for research in architecture description languages, this research contributed to the field of software architecture by creating a new ADL (called VTADL, for Visually Translatable Architecture Description Language) capable of representing separate but integrated viewpoints on software architectures. VTADL allowed a flexible and relatively simple description of one or more viewpoints, and one or more architectures within each viewpoint. VTADL allowed the selective hiding or revealing of elements of an architecture, allowed for the representing of multiple architectural styles, and allowed for hyperlinks of any element of any architecture within a view, either to another view or to a desired file external to VTADL.

This research also contributed to the field of software architecture by translating the views and architectures defined by VTADL into a visual medium, VRML. Each view defined in VTADL was translated into a separate VRML file; the views (and the associated VRML files) were related to one another by the VTADL definitions.

Any relatively complex architectures defined in the architecture description file could be made more interesting and hopefully more comprehensible by visualizing them in three-dimensional, colorful worlds, rendered using the virtual reality paradigm. Although the visualizations were of static software architectures, the stakeholder was free to navigate through the visual representation, able to glance at the structures from different perspectives; the stakeholder was also able to navigate through other viewpoints not necessarily his own as easily as clicking a mouse, perhaps exploring alternate worlds and gaining a better understanding of the whole.

Chapter III

Methodology

Research Methods

Overview of Procedures Employed

In Chapter I, the stated goal of this research was to develop a new technique for software architecture visualization using virtual reality. Ware and Franck demonstrated that the presence of three dimensions in the visualization of complex structures, when accompanied by user-directed movement, increased the comprehension of those structures [Ware & Franck, 1993]. VRML is a highly flexible medium, intended for representing realistic worlds in three-dimensional space. When the VRML file is instantiated, a viewer may navigate through an engaging world, exploring objects from a variety of locations.

A natural association was made by the author of this dissertation between the virtual reality worlds modeled by VRML and the representation of a viewpoint. It was determined that each VRML file would be a separate view (abstraction) containing one or more software architectures. The views (and the architectures within each view) were defined by the software architect through means of an architecture description language.

The overall strategy of this research was to first develop an architecture description language; second, to develop a compiler capable of translating the architecture description language into VRML; and third, to run case studies demonstrating the visualization. The developed software was a prototype demonstrating the "proof of concept" and not a polished, commercial product.

The strategy was realized in five major steps (or phases). The procedures behind each step are described as follows:

Procedure Step One defined a simple yet effective architecture description language, capable of being visualized. The language was capable of describing software components, connectors, configurations, and architectural styles; the language could also allow for the definition of multiple views on one or more architectural structures, conforming as much as possible to the architecture description standards defined by IEEE 1471-2000 [IEEE, 2000]. The language could allow for hyperlinks from elements of an architecture to external files, so that the rationale behind the architecture could be traced to source documentation.

The reasoning behind features of the language was documented within the context of current research in the field of architecture description languages. The result of Procedure Step One was a formal description of the language in BNF (Backus-Naur Form).

Procedure Step Two began once the grammar of the architecture description language was defined. This step was the design phase of a compiler that ultimately translated the regular expressions in the source ADL to visualized objects in VRML. In Procedure Step Two, a mapping was defined between the objects (such as architectural components or connectors) in the ADL to the objects within VRML. This mapping was achieved by a geometric model, which defined the topology of an architecture using a connectivity matrix.

The mapping from ADL architecture to the geometric model was determined by the architectural style as defined in the ADL (with the types of style as described originally by Shaw and Garlan [Shaw & Garlan, 1996]).

Procedure Step Two also defined the overall data structures required to implement the geometric model, and the basic algorithms needed to traverse the data structures to generate VRML code.

The BNF for the source ADL was used as input to a parser generator, yacc (acronym for "yet another compiler-compiler"). The tokens of the source language were defined and fed into lex (a lexical analyzer generator). The end result of Step Two, then, was a parser shell capable of recognizing the source language, and a lexer capable of accepting tokens and feeding them to the parser. In other words, the syntax of the source language was defined, but the full semantics had yet to be implemented as C code in the form of functions and subroutines embedded in the parser shell.

Step Three involved coding and initial testing of the subroutines that operated on the data structures representing the geometric model. These subroutines comprised the semantics of the compiler. After the subroutines were tested, they were integrated into the parser shell.

The subroutines for traversing the data structures and generating VRML code were successfully integrated into the compiler; the compiler was then fully implemented and was ready for acceptance testing.

Step Four performed acceptance testing of the compiler on a number of ADL test files.

Step Five conducted actual case studies, starting with the rudimentary and progressing into more intricate cases. Step Five represented the culmination of this research into software architecture visualization, and resulted in a prototype capable of translating an architecture description language into a three-dimensional visualization in the Virtual Reality Modeling Language.

As mentioned in Chapter I, this research proceeded only as far as the development of a prototype. The actual effectiveness of the visualization software in a real software development environment was left as a topic for future study.

Specific Procedures to be Employed

For each test or case study, the following items of documentation were provided:

(1) Documentation on the purpose, background, and pertinent details of the test or case study;

(2) The VTADL source file(s);

(3) The actual visualizations by means of a color screen printout. An explanation of the view and architectures within the view were provided.

These items were incorporated in the template in Appendix H.

Description of Procedures

Procedure Step One

The purpose of Step One was to specify a general architecture description language, capable of meeting the basic requirements of an ADL as specified in the *IEEE Recommended Practice for Architectural Descriptions* [IEEE, 2000]. The ADL was also required to fulfill the fundamental criteria as specified by the key literature [Bass, Clements, Kazman, 1998; Jazayeri, Ran, & van der Linden, 2000; Medvidovic & Taylor, 2000; Shaw & Garlan, 1996]. Fundamentally, an ADL described the overall coarse-grained structure of software. The language had to be capable of describing components, connections between components, and configurations of topologies. General properties of components (such as role, associated process, interfaces, etc.) or connectors (relation type, weight, directionality, connectivity definition, etc.) also had to be specified in the language.

We desired the capability to describe architectural styles. An architectural style defined how the components and connectors were used, with constraints on the topology and instantiation. Examples of the better-known styles were the pipelined, main-program-and-subroutine, object-oriented, layered, and state-based styles. Though VTADL allowed for several different styles, only two were visualized in VRML for this dissertation: program-call-and-return and the layered styles.

It was possible to combine styles using a "heterogeneous" style, where elements of an architecture may consist of more than one style (e.g., a component in a program call-and-return style could represent a pipeline). Heterogeneity had the drawback that, by

combining too many styles, the original reason for software architecture – simplicity – was lost. VTADL did not allow for direct representation of heterogeneous styles; however, architectures of more than one style could be visualized in a view, juxtaposed in proximity to one another for easy comparison.

Another important goal of an ADL was to standardize the structural representation into a language independent of system implementation, so that knowledge of successfully applied patterns could be detected, reused, and transferred to future projects. Bass, Clements and Kazman [Bass et al., 1998] reasoned that ADLs shared features of requirements, programming and modeling languages, yet were distinct from all three. Requirements languages were rooted in the problem domain, while ADLs concentrated on the solution space. Programming languages mapped all architectural components to the execution space, while ADLs attempted to hide the execution and concentrate on structure. Finally, modeling languages referred to the behavior of the whole system, rather than the parts. ADLs concentrated on component structure and connections.

We modified the ADL criteria by adding a further constraint: that the language be translatable into visual form in a way that lucidly and unambiguously communicated the architectural descriptions to the stakeholders.

It was intended that the ADL not be domain-specific. In other words, the language was required to be generic in nature; the language was required to be relatively easy for a non-specialist to understand, and was required to contain non-ambiguous structures capable of being visualized. For example, if a structural hierarchy was present,

the language must have explicitly stated whether one component was a child of another component, to the left or right of another component, etc.

The ADL was required to focus on the static representation of a software architecture, rather than the state-based notations modeling the dynamic behavior.

Finally, the ADL was required to allow for rapid and simple description of an architecture.

An existing architecture description language called ASDL, for "Architecture Structure Description Language" [Eixelsberger & Gall, 1998] was used as a model in the design of the ADL for this dissertation. ASDL emphasized important static structural properties, while hiding many behavioral details, a key feature in the design of our own language. Although ASDL was originally intended for architectural recovery from legacy systems, the authors pointed out that ASDL had general use to a wide variety of domains.

However, the capability for representing different views on one or more software architectures was needed, per the stated goals of this dissertation. Other features, such as the ability to define hyperlinks on selected elements, the ability to selectively show or hide elements of an already defined architecture, and the ability to show multiple architectures within the same view were also be required. Using ASDL as a starting point, a new language incorporating the additional features was developed.

With the aforementioned language characteristics in mind, a new language was defined on May, 2001. The new architectural description language visualized by this dissertation was called VTADL, an acronym for Visually Translatable Architectural Description Language. The unique feature of VTADL was that it allowed for a flexible

description of both views and architectures within views. VTADL was a regular language whose expressions were recursively generated from the grammatical production rules in an LR (left-to-right) derivation. VTADL was translated into VRML by an LR parser (see Figure 1).

Appendix A contains the full grammatical specification of VTADL in BNF (Backus-Naur Form).

Figure 1: VTADL-To-VRML Translation

To reiterate the definition of a view: a view is an abstraction of a model (in this case, an architectural structure), based upon the concerns and interests of a stakeholder.

The analogy of a physical building architecture to a software architecture noted in Chapter I is worth repeating here, but with the caveat that the analogy is not meant to be taken too literally. Often, software may be far more untenable than a physical structure since software itself is fundamentally an immaterial abstraction, which may be changed at the whim of stakeholders.

If we were to provide a blueprint of an electrician's view of the building, the diagram would probably hide most of the physical details except for the overall dimensions of the walls, doors, etc. The electricians' diagram would emphasize the wiring, power sources, and overall cable circuitry. A plumber's view would contain

details about the piping, while suppressing most if not all information about the wiring circuitry (except in isolated cases where intersection with electrical wiring would present a potential hazard to the plumber).

The VTADL source file consisted of two parts: the architecture list, which listed all architectures, and the view list, which listed the views on one or more of the architectures. A view may have used one or more architectures from the architecture list; however, a view may not have used an architecture not defined in the architecture list.

See Figure 2 for the body of an architecture definition.

Figure 2. Architecture Definition (Template)

Architecture <Arch-Name>
type <Style>
{
 ComponentList
 {
   Component <Name-1>;
    .
    .
    Component <Name-n>;
  }
  ConnectionList
  {
   Connection <ConnName-1>
   .
   .
   Connection <ConnName-n>
   }
}

Note that the reserved words in the language are in bold font.

An architecture definition was assigned a unique name, and an architectural style was assigned for the entire structure that followed. The architectural styles available in VTADL were:

1. The program call-and-return style (PROGRAM selection), where components represent program modules and connections represent calls to subroutines;

2. The object-oriented style (OBJECT selection), where components represent classes or objects, and connections represent an association or relation;

3. The pipelined style (PIPELINE selection), where a component represents a transformation on data flow, and a connection represents a flow of data;

4. The layered style (LAYER selection), where components represent a service to a higher layer, and connections are hidden. The highest layer represents a service to the overall system.

Of the four style selections, only two were implemented in the VTADL-to-VRML compiler: the program call-and-return and the layered style.

The architectural style determined how the components and connections were visualized. In other words, the style mapped the architectural structure to a particular geometric model which was, in turn, rendered in VRML. The architecture definition itself consisted of a component list, defining the components in the topological order that they should appear in, and a connection list, which established the connectors between the defined components. Each component in the component list had a set of properties, depending on the architectural style constraints; each connector also had a set of properties.

A component in a component list was first given a unique component name. The component name had to be unique only within the component list. Following the name, the component was assigned a component type. The component type may have been defined as a Processing Element, a Program Component, a Conceptual Component, an Object Component, or a Data Repository Component. For the present implementation, a Program Component was used (for the program call-and-return or layered styles).

Component properties were defined using the following template (see Figure 3):

Figure 3: Properties of the Component

```
Component <Comp-Name>
ComponentType <Type-of-Component>;
Properties:
        CompRole: <Role-of-Component>;
        ChildOf:  <Component-Parent>;
        Layer:  <Layer-Name>;
        Process: <Process-Name>;
        InterfaceList:
                Interface <Relative-Position>  <Interface-Name-1>;
                { InterfaceRole: <Role-Selection>; }
                .
                .
                Interface <Relative-Position>  <Interface-Name-n>;
                {  InterfaceRole: <Role-Selection>; }
```

Following the reserved word, **Properties**, the component attributes were defined. **CompRole** defined the Component Role, or the role that the component played in any interaction with another component. The component may have served roles as **Input** (an input node to a network), **Output** (an output node to a network), **Root** (root node of a hierarchy), **CmpProducer** (producer in a client server relation), or **CmpConsumer**

(consumer in a client-server relation). A component role must always have been indicated.

The **ChildOf** property need not have been indicated, however. The **ChildOf** property was defined when a component was a child to another component in a hierarchy (such as a program call-and-return or layered relation). If the component was a child, then the name of the parent component was provided.

The **Layer** property was used to define an additional identifier for the component when the component was used as a service in the layered style, or when the component occupied a defined layer of nodes in the network style (the network style will be implemented in a future version of VTADL). The layer definition was optional, but may be defined as **Linput** (for layer input node in a network), **Loutput** (for layer output node), or may be assigned an alphanumeric name.

In the **Process** property (optional), a component may be assigned a process type. The process types were for use with neural network nodes; the process type selections were **Sigma** (for summation node) or **Threshold** (for the threshold function of a processing element).

A component had one or more interfaces defined for the component. Connections to that component must have been instantiated through an interface name. In other words, the interface name was the medium between the component and a connection to the outside world. An interface name was analogous to an electrical "socket" which accepts attachments from electrical plugs.

At the end of the component properties section, the **InterfaceList** was defined. The interface list contained the definitions of interface names and interface properties.

An interface in the interface list was defined using the reserved word, **Interface**, followed by the relative interface position and the interface name.

The relative interface position must have been selected from one of six possible locations: **Top**, **Bottom**, **Left**, **Right**, **Front**, **Back**. The component was visualized as occupying a position within a cube; the interface was at the center of one of the six faces of the cube. For instance, the **Top** position could be considered to occupy the center of the top face, and so on. A maximum of 25 interface names were allowed for any given component. However, in the current implementation of the VTADL-to-VRML compiler, the interfaces were not visualized as separate objects in VRML.

For each interface, an interface role was defined. The interface role determined how the interface was used by the component with respect to the connection. In the current version of VTADL, there were only two possible interface roles: **Producer** and **Consumer**. An interface was a producer if it supplied data or sent control signals to the connection; an interface was a consumer if it received data from the connection or received control signals.

Once the components were defined within the component list, a connection list was used to define the connections between the previously defined components. The connection list must always have followed the component list, since the connections were defined using the interface names from the component properties.

Figure 4 gives the template for the connection list of a software architecture.

Figure 4: Template for a Connection List

```
ConnectionList
    {
      Connector <Connect-Name-1>
        ConnectType <Connect-Type>  <Connect-Direction>;
        Connect( <From-Interface-Name> , <To-Interface-Name> );
          .
          .
      Connector <Connect-Name-n>
        ConnectType <Connect-Type>  <Connect-Direction>;
        Connect( <From-Interface-Name> , <To-Interface-Name> );
    }
```

A given connection was defined using the **Connector** reserved word, followed by the connector name. After the connector name, the connector type was specified as one of three options: **DataFlow** (meaning that the connection was used to transport data), **ControlFlow** (the connection was used for control purposes), or **Associates** (meaning that the connection was used to establish a relation in the conceptual model or object-oriented model). Following the connector type, the directionality of the connector was established as either **Unidirect** (the connection was unidirectional, flowing in one direction) or **Bidirect** (the connection was bidirectional, flowing in either direction).

The actual connection was established using the **Connect** reserved word. Within the parenthesis of the Connect clause, the **From-Interface-Name** was the name of the interface from which the connector originated, and the **To-Interface-Name** was the name of the interface which served as the destination for the connector.

It should be noted that an interface name within an architecture should be unique. That is, an interface name should be associated with only one component, otherwise an error condition may result.

A given architecture was not instantiated unless it was incorporated within a given view. In the second section of a VTADL file, a **View-List** was built, defining one or more views. A main view was always required, even though the main view may have been empty (the trivial case). Within a given view, preferably one or more architectures could be used. If more than one architecture was used by the view, the first architecture specified in the ordering within the view-list was rendered first; the second, third, and nth architectures were displaced into the background in successively more distant and step-wise fashion.

Figure 5 gives the template of the view-list section of a VTADL file.

Figure 5. View-List Template of a VTADL File

```
ViewList
 {
   ViewMain
    {
      { UsingArch <Arch-Name-1>; }
      .
      .
      { UsingArch <Arch-Name-n>; }
      .
      .
     }
    .
    .
    View  <View-Name-1>
       {
         { UsingArch <Arch-Name-1>; }
         .
         .
```

```
        { UsingArch <Arch-Name-n>; }
      }

    View  <View-Name-n>
      {
        { UsingArch <Arch-Name-1>; }
        .
        .
        { UsingArch <Arch-Name-n>; }
      }
}
```

The main view was defined using the **ViewMain** reserved word. If the view was not the main view, the reserved word **View** was used, followed by the view name. The architectures within a view were defined with the **UsingArch** reserved word, followed by the name of the architecture. The name of the architecture must have been previously defined in the architecture list section of the VTADL file, otherwise an error condition was generated by the compiler. The same architecture name could be used more than once within a view; different versions of the same architecture could represent different aspects of the same structure. When several architectures were used within a view, the architectures were successively translated further into the distance in a step-wise fashion.

For any architecture specified within the view, components and connections could be selectively displayed or hidden. Hyperlinks could also be established from specified components or connections to external files.

After the **UsingArch** clause and architecture name, the hyperlinks and show/hide clauses were established within a pair of brackets { ... } for the architecture being used. The referenced component or connection identifiers must have been valid for the architecture.

For the architecture being used, the **Components** reserved word indicated which components were to be shown. If the word **All** was used, all components were shown; however, if individual component names were listed, only the listed component names were shown. The component names were listed with one or more spaces between them. The word **Connections** was used to indicate the desired connections to be shown. If the word **All** followed **Connections**, then all connections were shown. However, if one or more individual connection names were listed, only those connections were shown (and the remainder kept hidden).

If a given component or connection was selected for display, a hyperlink could be established from that component or connection to a file. The following statement was used whenever a hyperlink was desired:

HyperLinkOn <Component-or-Connection> **ToFile** <File-Name>**;**

A hyperlink may be only established once for any component or connection in the architecture. The hyperlink may be either to a view file or to an external file name. If the hyperlink was to a view file, the view file name may be used without quotes; if an external file name was desired, the file name must have been defined as a literal, with url or directory path included if the file name resided in a location other than the default location.

Each view in the view-list generated a separate VRML file, using the view name as file name, followed by the extension, "wrl." The main view always generated the file, "Main.wrl."

So far, we have defined a VTADL file as consisting of the architecture list, which defined each architecture using a style, components, connectors, and a topology; and a view-list, which listed the main view and user-defined views, with each view using one or more architectures. We have shown how each architecture specified within a view may have components or connectors selectively shown or hidden, with hyperlinks on elements in each architecture to either other views or to external files.

Appendix E provided a simple example of a VTADL file which defined two architectures: ExampleProgram, a hierarchical structure with a root and three children, (four components and three connections) in the call-and-return style; and ExampleLayer, with four components in the layered style.

The main view contained both architectures, using all components and connections. A hyperlink existed from the root in ExampleProgram to the source VTADL file (a text file named "Example.txt"). Another hyperlink was established from the top layer of ExampleLayer to a view named "SecondView."

SecondView contained three versions of ExampleLayer. Version one used the first and second layers, hiding the other layers; version two used the first, third and fourth layers; and version three used all four layers. The first layer of version one contained a hyperlink to the main view; the second layer of version one contained a hyperlink to the source VTADL file, Example.txt.

Example.txt was used as one of several test cases to demonstrate that the program was working according to requirements. The resulting VRML target files, and the accompanying visualizations, were described in Appendix G.

Procedure Step Two: Design of Geometric Model (Data Structures and Algorithms)

Figure 1 illustrated the general translation process from VTADL source file to VRML target file. Figure 6 decomposes the translation process into the well-known subdivisions of a compiler [Aho, Sethi, & Ullman, 1986]. Since the text-based source file was translated into a graphical language, an intermediate geometric model was included to map the architectural structures defined in the source language to VRML objects. At an even higher level of abstraction, the intermediate geometric model mapped each view defined in the VTADL source file to a separate world defined by a separate VRML file. That is, one VRML file was generated for each view defined in the VTADL source file.

Figure 6. Basic VTADL-to-VRML Compilation Process

VTADL Source file ===> Lexical Analyzer ===> Tokens ===> Parser

Parser ===> Geometric Modeler ==> VRML Generator

VRML Generator ===> VRML View ===> Visualization of Views

The VTADL source file was required to be a text file. The VTADL file was scanned by a lexical analyzer (or "lexer"), which built the symbol table, detected tokens, and passed the tokens to the parser. If any symbol or string was invalid (not allowed in the string specifications of the language) the lexer generated an error message for the

symbol. If the symbol or string was valid, the token was passed to the parsing program (or "parser").

The parser builds a parse tree during the first pass of the parsing process; during a second pass, the parse tree is traversed to validate the syntax of the grammar. If the syntax is invalid (e.g., a token existed in an order where the token violated the grammar of the language), an error message is generated indicating the parsing is unsuccessful.

If the parse is successful, the statements in the source file are grammatically correct. However, the semantics of the statements must also be generated. By semantics is meant the corresponding actions taken by the target language, VRML, for the source language statement (Appendix C shows the parser shell, the yacc specification of VTADL without the semantic actions to generate VRML code).

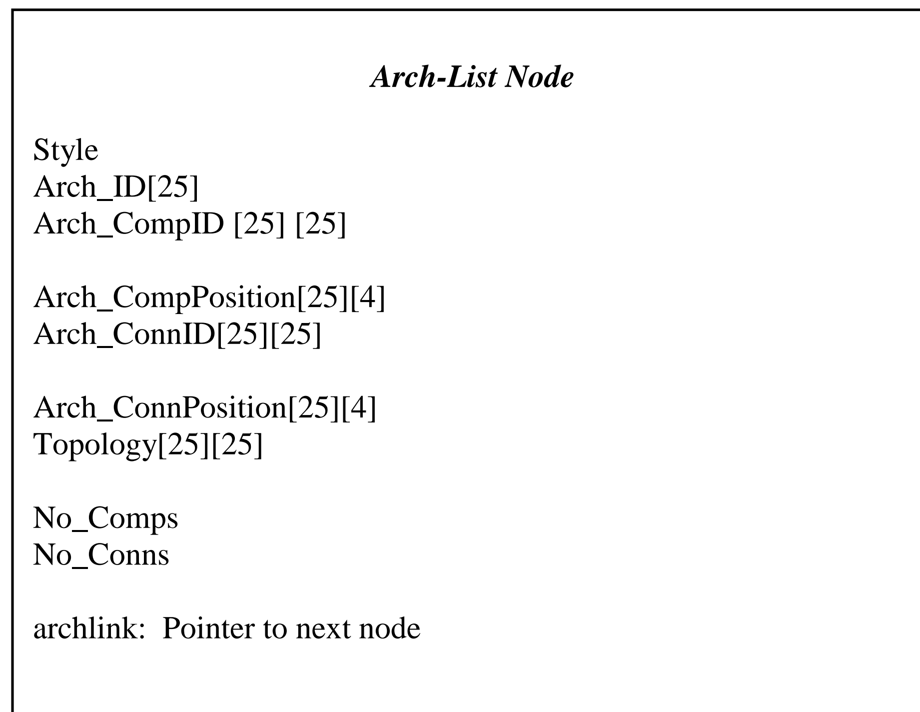
The corresponding actions (statements) in VRML were generated by various subroutines which were called as semantic responses to VTADL statements. The subroutines called in response to VTADL statements served two basic tasks: first, they stored properties of components and connections defined in the source language to an intermediate geometric model; second, the subroutines traversed the intermediate geometric model, performed space-related calculations, and generated the corresponding VRML code.

The intermediate geometric model was implemented as data structures in the C programming language. What we have referred to as the "geometric model" was in reality a collection of abstract data types such as multidimensional arrays, pointer variables, and linked lists. In this section, we defined these abstract data types and described the algorithms that would later operate on the data structures.

The first section of the VTADL source file was the architecture list. The list of architectures was represented as a linearly linked list, with each node representing one architecture. An architecture described in VTADL was stored in an architecture node; the architecture node was inserted into the linked list in the order that the architecture appeared in the architecture list of the VTADL file.

A given architectural node contained information that identified the name of the architecture and defined the style, components, connectors, and topology (connectivity). A count of the number of components and connectors in the architecture was also included in the architecture node. Figure 7 illustrates a node in an architecture linked list.

Figure 7: Node in Architecture Linked List



For each architecture, the architectural style determined how the components and connectors were rendered in VRML. In a program call-and-return style, components were rendered as spheres and connectors were rendered as cylinders. The enforced topology of the call-and-return style was a three-dimensional tree, with child components under a parent rotated around the y-axis in an imaginary circle. All spheres at a given level (height) in the tree hierarchy were given the same color and radius. Spheres at higher level numbers were assigned successively smaller radii and different colors.

For the layered style, components were rendered in VRML as successively larger cones with successively more space between each layer beyond the base layer. Even-numbered layers (with the base layer being 0) were assigned one color, while odd-numbered layers were assigned a second color. Since the layered style hid information about connections, the connection information defined in VTADL for layered styles was not rendered in VRML.

The algorithms for rendering the two styles will be given later in this section.

Referring again to Figure 7, Arch_ID is the 25 character name of the architecture defined in the architecture list node. No_Comps is a counter variable telling how many components were used in the architecture. No_Conns is a counter variable telling how many connectors were used in the architecture.

Arch_CompID[25][25] was a two-dimensional array containing the identifying names of components. Each component could be a maximum of 25 characters. There could be a maximum of 25 components in the architecture. To access the name of the first component, an index of 0 was used: Arch_CompID[0]. To access the name of the last component, an index of No_Comps - 1 was used, or Arch_CompID[No_Comps-1].

In likewise fashion, Arch_ConnID[25][25] was a two-dimensional array containing the identifying names of connectors.

Topology[25][25] was a two-dimensional, 25 x 25 array of integers. The Topology matrix was used to represent the connectivity between components, and provided the index of the connector used for connecting one component to another. A row number of Topology represented the index of the component from which the connection originated; a column number of Topology represented the index of the destination component to which the connection terminated. Specified at the row and column intersection was an integer representing the index of the connector that linked the originating component (row) to the destination component (column). If no connector existed at the row and column intersection, a negative one (-1) was placed at that location.

Arch_CompPosition[25][4] was a matrix storing the relative coordinate positions and level of a component in a hierarchy. The row number was the index (an integer from 0 to 24) identifying the component within the architecture. The column numbers indicated the x, y, and z positions relative to the top of the architecture itself, and the level number within the hierarchy. The root was assigned the level number 0, the immediate children of the root were assigned the level number 1, and so on. For example, Arch_CompPosition[0][0] would be the relative x-position of the root (which had the component index of 0); Arch_CompPosition[0][1] would be the relative y-position of the root; Arch_CompPosition[0][2] would be the relative z-position. Finally, Arch_CompPosition[0][3] would be the level number of the root, in this case 0.

We mentioned coordinate positions relative to the top of the architecture. For each architecture, the origin (or top) of the architecture was considered to be the center of the root node.

Arch_ConnPosition[24][4] was a matrix storing the relative coordinate positions of a connector. The x, y and z coordinates specified the coordinates for the center of a cylinder representing the connector.

As an example, Arch_ConnPosition[0][0] would specify the x-position for a connector with the identifying index of 0. Arch_ConnPosition[0][1] would specify the y-position for the connector. Arch_ConnPosition[0][2] would specify the z-position.

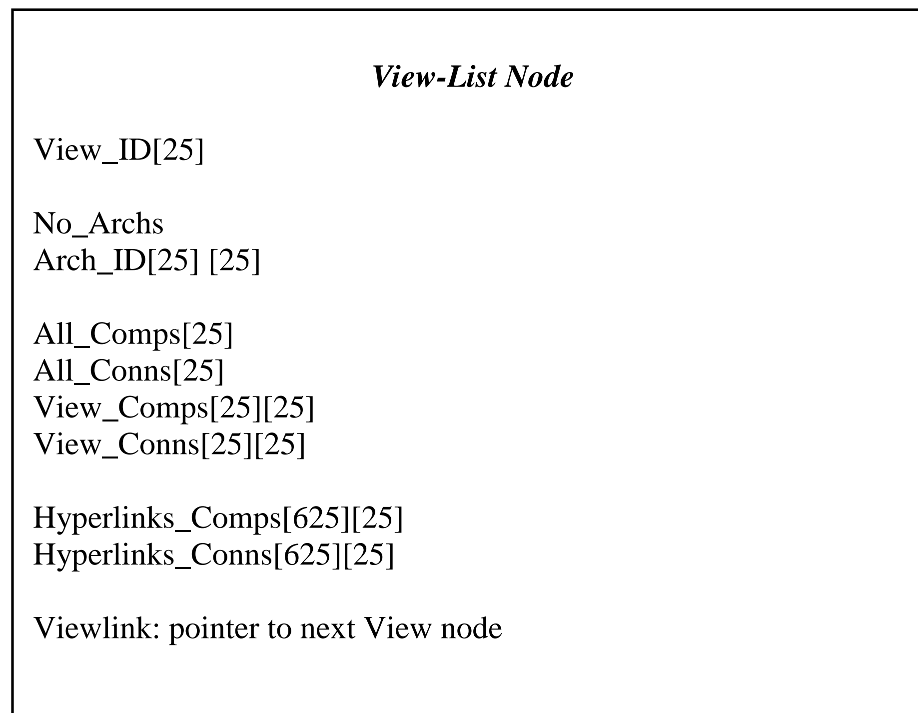
Arch_ConnPosition[0][3] would specify the rotation (in radians) of the cylindrical connector around the y-axis. This value would place the connector in its correct position as a child to a parent. The algorithms for connector rotation and position calculations will be given later in this section.

The arrays and matrices contained in the architecture node defined only a single architecture from the architecture list of the VTADL file. A pointer variable, Archlink, pointed to the next node in the linked list of architecture nodes. If there were no successor nodes, Archlink was set to null.

The second part of the VTADL file was the view-list. The view-list was required to contain the main view ("ViewMain") and could contain one or more user-defined views. The view-list was implemented as a linear linked list. Each node of the linked list represented one view. The first node in the linked list was the required main node, followed by any user-defined view nodes. The views in the linked list were inserted in the order that they appeared in the VTADL source file.

A view could use one or more architectures from the arch-list defined at the beginning of the file. It should be noted that a view could contain no architecture, but this was considered the trivial case. All the properties of a single view were included in a node of the view-list. Figure 8 describes the properties of a view-list node.

Figure 8: View List Node Definition



A view was represented as a world in VRML. Each architecture within a view containing multiple architectures must be related in some way for the view to make logical sense. The view could be effective or ineffective depending on how well the user defined the original VTADL file. The process of eliminating ambiguity or viewpoint mismatch was a manual and not an automated process in the current implementation.

View_ID was a character array that identified the view name of the node. No_archs was a count of the number of architectures contained within the view.

A software architecture was instantiated by being included within a view by the **UsingArch** <ArchID> statement. The architecture names defined by each UsingArch statement were successively stored in the Arch_ID[25][25] array. Arch_ID was a two-dimensional array that contained a maximum of 25 architecture names; each name could be a maximum of 25 characters. The row index of Arch_ID indicated the architecture, while the columns indicated the character positions of the name itself.

Each architecture identified in the Arch_ID could have a maximum of 25 components or connectors. Any component or connector in an architecture used within a view may be selectively shown or hidden. The array All_Comps[25] was used to indicate whether a given architecture (identified by an index ranging from 0 to 24) would display all components. For example, if All_Comps[5] was set to 1, then all components of the architecture whose index was 5 would be displayed; otherwise, if All_Comps[5] was set to 0, only selected components would be shown. In likewise fashion, the array All_Conns[25] was used to indicate whether all connectors would be shown for the architecture.

If selected components were desired for display, the View_Comps[25][25] array was used. The row number of View_Comps was the index of the architecture, and the column number was the index of the component of the architecture to be viewed. If the array element specified by the row and column contained a 1, the component was rendered in VRML; if the array element specified by the row and column contained a 0, however, the component was kept invisible.

A similar scheme was used to display or hide selected connectors. The View_Conns[25][25] array was used, with the row number being the index of the architecture, and column number being the index of the connector. If the array element at the intersection of row and column was 1, the connector was viewed; otherwise the connector was hidden.

Hyperlinks could be specified from any component or connector in an architecture to another view file, or to an external file.

The array, Hyperlinks_Comps[625][25] allowed for a hyperlink from a specified component in an architecture to a file name. An architecture within Hyperlinks_Comps was accessed by means of an index, which may range from 0 to 24. A component within that architecture may also be accessed by an index (0 to 24). Since there were a maximum of 25 architectures and a maximum of 25 components per architecture, Hyperlinks_Comps contained a maximum of 625 rows. Given the architecture index and component index, we could calculate the row index of Hyperlinks_Comps by using the hashing formula described by Equation 1:

$$\text{Row index} = (\text{index of architecture})*25 + (\text{index of component}) \quad \textbf{(1)}$$

The resulting row index was an integer in the range from 0 to 624.

As an example, to access the tenth component (component index = 9) of the architecture whose identifying index was 4, the row index = (4) * 25 + 9 = 109. The hyperlink file name was accessed at row index 109 of Hyperlinks_Comps.

Once the row had been located (which represented the architecture and component within that architecture), the 25 columns allowed for a file name with a maximum of 25 characters. If a file name was used without quote marks and without an extension, the compiler assumed that a view file was being hyperlinked, and stored a VRML file name with the extension, ".wrl." However, if the file name was used with quote marks around it, a literal string was assumed, and the file name was stored as a literal at the calculated row number. The file name would be the destination of the hyperlink from that component.

Hyperlinks_Conns[625][25] was the array for establishing hyperlinks from connectors, and followed a similar scheme described for components. See Equation 2 for the calculation scheme for the row of Hyperlinks_Conns:

Row index = (index of architecture)*25 + (index of connector) **(2)**

The pointer variable, "Viewlink," pointed to the next view node (if any) in the linked list. Viewlink was set to "null" if there was no succeeding view node.

Description of Geometric Algorithms on Data Structures

At the highest level, the algorithms that operate on the intermediate geometric model have two main phases:

1. *Build Geometry Phase*. Parse the VTADL source file and build the architecture list and view list data structures;

2. *Code Generation Phase.* Traverse the view list from first to last node, using the architectures specified for each view. While generating a view, search the architecture list for arch-node details. Render the architectures within the view. Generate a separate VRML file for each view.

The algorithms for Phase One and Phase Two are briefly described (in pseudo-code) as follows:

Phase One: Algorithm for the "Build Geometry Phase"

Parse the source file using the parser.

While the End of File ("$") symbol is NOT encountered:

/* Read through the architecture list */

While the architecture list has another architecture defined in it:
    Store the architecture name in Arch_ID.
    Store the style in style_var
    Initialize component, connection, interface counts and arrays.

        For each component in component list:
            Store component name to comp_name_array.
            For each interface name in properties section:
                Store interface name to interface_name_array.
                Store component ID associated with interface name to interface_comp.

End For each interface name.
Add 1 to comp_index, the component counter.
End For each component in component list.

For each connector in the connector list:
Store connector name to conn_name_array.
Define connection using interface1 to interface2:
Find component index for interface1:
Search array interface_comp for interface1.
If component index found, use component index
as source component in the connection,
else generate ERROR and EXIT.
Find component index for interface2:
Search array interface_comp for interface2.
If component index found, use component index
as destination component in the connection,
else generate ERROR and EXIT.

Place source component index ("from_index") and destination
component index ("to_index"), along with connector index,
in a temporary array, "from_to[conn_index][3]".

from_to[conn_index][0] = index of connection name.
from_to[conn_index][1] = index of originating component.
from_to[conn_index][2] = index of destination component.

Add 1 to conn_index.

End for each connector.

/* Call subroutine to create new arch-node and insert in linked list */
/* Note: Details of Insert_arch_node are provided in Appendix D. */

Insert_arch_node(component, connector, and connectivity arrays).
This subroutine moves the architectural values into the node values.
The routine also generates the component and connector positions for
an architecture relative to the topmost component, whose center is
viewed as the origin (0.0, 0.0, 0.0) of the architecture.

End While arch list has another architecture.

```
/* For view-list section of VTADL, create view linked list */

While view-list has another view defined in it:
    Store view name in View_ID.
    (Note constraint:  The very first view must be called "ViewMain").
    Initialize temporary view component, view connection, and hyperlink
    arrays.

    Search the view-list for View_ID.
        If View_ID already exists, generate an ERROR and EXIT.

    Store view name in a global view name array called View_Names.
        View_Names array will be used to generate a legend of views in each
        VRML file during VRML code generation.

    For each architecture specified in the "Using Architecture" command:
        Search for Arch_ID from architecture linked list.
        If architecture NOT found (-1), generate ERROR and EXIT.
        If architecture found, proceed:

        Components Show or Hide Section:
            If user specified "ALL" components:
                Set All_Comps array to 1 for the current architecture.
            If user specified selected components:
                Set All_Comps array to 0 for the current architecture.
                For each selected component specified for viewing:
                    Search for the specified component in the component array
                    of the current architecture node.
                    If component not found, generate ERROR and EXIT.
                    If component found:
                        Set temporary array, view_comps[curr_arch_index][comp_index]
                        to 1 for the architecture index and component index.
                End for each selected component.

        Connectors Show or Hide Section:
            If user specified "ALL" connectors:
                Set All_Conns array to 1 for the current architecture.
            If user specified selected connectors:
                Set All_Conns array to 0 for the current architecture.
                For each selected connector specified for viewing:
                    Search for the specified connector in the connector array
                    of the current architecture node.
                    If connector NOT found, generate ERROR and EXIT.
                    If connector found:
                        Set temporary array, view_conns[curr_arch_index][conn_index]
```

to 1 for the architecture index and connector index.
End for each selected connector.

Hyperlinks section:
For each selected element specified for a hyperlink:
Set the element ID to test_string.
Search the component array using test_string.
<u>If found</u>, the element is a component.
<u>If NOT found</u>:
Search the connector array using test_string.
<u>If NOT found</u>, ERROR and EXIT.
<u>If found</u>, the element is a connector.

/* At this point, the element was determined as a comp or conn */

<u>If element was a component</u>:
Set hyperlink index to access the hyperlink component array using Equation 1:
Row index of Hyperlinks_Comps is:
(index of architecture)*25 + (index of component)

<u>If the file name was specified without quotes</u>, we assume the file name is a view file and add the extension ".wrl".
<u>If the file name was specified within quotes</u>, we use the string as a literal and assume that an extension was given.

Store the file name for hyperlink in the array of hyperlinked components, Hyperlinks_Comps[row index][ ].

<u>If the element was a connector</u>:
Set hyperlink index to hyperlink connector array using Equation 2:
Row index of Hyperlinks_Conns is:
(index of architecture)*25 + (index of connector)

<u>If the file name was specified without quotes</u>, we assume the file name is a view file and add the extension ".wrl".
<u>If the file name was specified within quotes</u>, we use the string as a literal and assume that an extension was given.

Store the file name for hyperlink in the array of hyperlinked connectors, Hyperlinks_Conns[row index][ ].

End for each selected element specified for a hyperlink.

End for each architecture specified in the "Using Architecture" command.

/* Call subroutine to Insert View into view linked list */
Insert_View_Node (view_name, architecture_array, all_comps, all_conns, view_comps array, view_conns array, hyperlink_comps, hyperlink_conns, count of architectures)

Add 1 to number of views.

End While view-list has another view.

End While End-of-file NOT encountered.

Phase Two: Code Generation Phase (Generate VRML Files)

```
/* At this point, the parse of VTADL file has been completed. */
/* Data structures representing architectures and views have been built.  */
/* The view list must be traversed from first to last node, architectures within */
/* each view must be rendered, and the VRML output file(s) will be generated. */

/* Note:  Each view occupies one VRML file.  */


/* Traverse the View linked list; generate one VRML file per node. */

While there are more view nodes in the view linked list:
    Store the view name as VRML file name with extension ".wrl".
    Get the number of architectures count from the view node data.
    Generate the VRML heading.
        Headings include View Name and VRML legend.
        Use global View_Names array to generate legend.
        Legend includes listing of all view file names with highlighted spheres
        next to each view name.  No sphere is drawn near the view name
        that is the current view file.

            For each view name:
                Establish hyperlink from the sphere for that view name
                to the VRML file.
            End for each view name.

        For each architecture in the "used-architecture" array of view node:
            Using the Arch_ID, get the arch node from arch list.
            If unable to get the arch node, generate ERROR and EXIT.
            Using the origin of the architecture (call-and-return or layered),
            set the x, y, and z displacements for the architecture:
                x_displacement = old_x_displacement - 25.5;
                y_displacement = old_y_displacement + 15.0;
                z_displacement = old_z_displacement - 50.0.
            The effect of these displacements is to display each successive architecture in
            a given view as being farther back, moved farther to the left, and slightly
            raised.

            Write VRML architecture header using displacements.
            Traverse Topology matrix in breadth-first fashion, using a
            queue, as follows:
            Initialize queue for node traversal.
            Start with the very first row of Topology matrix:
                Curr_root = 0, set the current component index to 0.
```

Insert Curr_root into queue.
While queue is NOT empty:
    Remove item from queue (item is component index).
    Store item into Curr_root.
    Using Curr_root as index, get positions from
        Arch_CompPosition:
    root_x = Arch_CompPosition[Curr_root][0].
    root_y = Arch_CompPosition[Curr_root][1].
    root_z = Arch_CompPosition[Curr_root][2].
    Level = Arch_CompPosition[Curr_root][3].

    If the Style of the current architecture is "Call-and-Return":
        We assume an acyclic, tree-like structure.
        We do not assume that Topology matrix contains solely diagonal elements.

    Based on the level (0, 1, 2, ... etc.), set the spherical color of VRML component.

    Based on the level, set the radius of the sphere. The lower levels (higher level numbers) have successively smaller radii.

    If all_comps array set to 1 for component, render the sphere at the position root_x, root_y, root_z.

    If all_comps array set to 0 for component, check view_comps array for component:
        If view_comps set to 1, render sphere at position (root_x, root_y, root_z) in VRML;
        otherwise, do not render sphere.

    If the component was rendered in VRML, check for a hyperlink at that component. If there is a hyperlink, generate the VRML command for a hyperlink to the file name.

    /* Next we render the connection found in the row, column */
    /* intersection of Topology matrix */

    Using Curr_root value as the row:
        For all columns of Topology matrix for that row:
        If Topology[Curr_root][column] NOT (-1)
            Column represents component index of child of curr_root.
            Store Topology[Curr_root][column] to Conn_index.
            /* Column represents destination component index */
            /* and Topology(Row, Column) is index of connector. */

/* Each Column NOT -1 is therefore a child of Curr_root. */
Using Conn_index as index, and the connector position array from the arch node, set connector positions:

Conn_x = Arch_ConnPosition[Conn_index][0]
Conn_y = Arch_ConnPosition[Conn_index][1]
Conn_z = Arch_ConnPosition[Conn_index][2]
/* Delta_sum contains rotation for cylinders */
Delta_sum = Arch_ConnPosition[Conn_index][3]

If all_conns array set to 1 for connector,
set height of connector based on level number;
rotate around y-axis by delta_sum * (no_children – 1);
render the cylinder at position conn_x, conn_y, conn_z.

If all_conns array set to 0 for connector,
check view_conns array for connector:

If view_conns set to 1,
set height of connector based on level;
rotate around y-axis by delta_sum * (no_children -1);
render cylinder at position.

otherwise, do not render connector.

If the connector was rendered in VRML, check for a hyperlink at that connector. If there is a hyperlink, generate the VRML command for a hyperlink to the file name.

If the Style of the Architecture is "Layered," we assume components are rendered as cone-like objects using VRML extrusion nodes. Traversal of Topology matrix is diagonal since we assume a layered architecture. If not diagonal, we generate an error.

For the component represented by curr_root, we first check all_comps to see if the component is rendered.

If all_comps is set to 1, render the component as an extrusion node, scaled and colored for the level;
otherwise if all_comps is set to 0:

If view_comps is set to 1, render the component as an extrusion node with proper scaling and coloring for the level.
If view_comps is set to 0, do not render and hide the component.

If the component was rendered, check for hyperlinks and establish the hyperlink in VRML code.

Insert the column index representing the child component into the queue.

End For all columns of Topology matrix for that row:

End While queue is NOT empty.

End for each architecture in "used architecture" array of view node.

End While there are more view nodes.

Procedure Step Three: Coding and Code Testing

The routines for translating VTADL to VRML were incorporated in the yacc file, "vtadlv1.y." The complete yacc specification was listed in Appendix F. The corresponding lex file which fed tokens to the parser was "vtadlv1.l," and was listed in Appendix B.

The parser was generated by using the following command in a Unix environment:

```
% yacc -d vtadlv1.y
```

This command generated the default file, "y.tab.c" and a header file, "y.tab.h." The file "y.tab.c" was the C source code for the parser.

The lexer was generated by using the command:

```
% lex vtadlv1.l
```

The lex command generated the default file, "lex.yy.c," the C file for the lexer. After the C code files representing the parser (y.tab.c) and the lexer (lex.yy.c) were generated, the two C files were compiled using the gcc compiler command:

```
% gcc -o vtadl_run y.tab.c lex.yy.c -lm -ll -ly
```

The lexer and parser C programs were compiled and linked into an executable object file, "vtadl_run."

The vtadl_run program represented the actual compiler and must be executed in a Unix environment, but the VRML files that vtadl_run generates may be run on any platform that has a VRML 97 add-on to an Internet browser.

The vtadl compiler may be executed as follows:

```
% vtadl_run < datafile.txt
```

The example command line above uses a VTADL source file named "datafile.txt" as input to the compiler. One or more VRML files may be generated as a result of the VTADL-to-VRML compilation process.

The VTADL-to-VRML compiler was validated and passed the preliminary testing phase. Procedures Step Four and Five will discuss the testing and case studies in more detail.

Procedure Step Four: Acceptance Testing of ADL Files

More intricate testing was conducted using the Case Study Report Template, with results given in Appendix G.

Procedure Step Five: Case Studies

Two case studies were conducted to demonstrate the workability of the VTADL-to-VRML visualization tool. Table 1 provides a summary of the two case studies. Each case study first documented the background of the case study, described the VTADL interpretation of the architecture(s), and provided the VRML visualization of the VTADL source file.

Case Study One used a case study originally discussed by Shaw and Garlan in *Software Architecture: Perspectives on an Emerging Discipline* [Shaw & Garlan, 1996]. Several solutions were offered to the problem of designing an architecture to control a mobile robot. Two of the solutions involved a layered style and a call-and-return style. These solutions were translated into VTADL representation, and then compiled into one or more VRML files.

Case Study Two was the most complex of the case studies and could be considered the capstone of this dissertation. The Linux operating system kernel was visualized using VRML. The views on the Linux kernel were established by two VTADL files, which were translated into the corresponding VRML visualizations. Case Study Two was based on the work of Bowman et al., who performed a software architecture recovery of Linux with minimal documentation [Bowman, 1998; Bowman, Siddiqi, & Tanuah, 1998]. Bowman's visualization was conducted in the plane in a static medium. This dissertation used Bowman's planar visualizations, redefined the architectures using VTADL, and then translated the views into separate VRML worlds representing the Linux kernel.

Results and analyses of the case studies were provided in Chapter IV and Appendix I.

Table 1. Summary of Case Studies

Case Study Name	**Description**
Case Study One: Mobile Robot Architecture	Multiple views of the architectures for a mobile robot included both layered and program call-and-return styles. Case study originally from Shaw and Garlan [1996], re-visualized in VRML.
Case Study Two: Linux OS	Case study visualizing Linux operating system using VRML; based on original work by Bowman [1998].

Formats for Presenting Results

A report template for conducting the case studies was provided in Appendix H.

The format for presenting results included a documented description on the goals and stakeholder requirements for the case study, the VTADL source file, a representative sample of translated VRML files (due to the length of VRML files, not every file was included), and screen printouts of key views for each visualized VRML file. Sample files referenced by the architecture (through hyperlinks) were also selectively shown.

Projected Outcomes

The goal of this research was attained when the architectures and viewpoints defined by the ADL were automatically translated into VRML. Two case studies were conducted to demonstrate that the goal had been reached.

It was anticipated that the visualization tool would greatly assist stakeholders in comprehending different aspects of a software system. However, a human factors survey on the impact of VRML visualization on various stakeholders, and the effectiveness of the tool on an actual development project or environment, were not explored in this research.

Resource Requirements

A VTADL source file must be compiled in a Unix environment using the compiler program, vtadl_run. The compiler may create one or more VRML files, with each file having the extension "wrl."

In order to view the VRML files, the local computer system should have an Internet browser (either Internet Explorer or Netscape Navigator). A VRML add-on to the Internet browser is also required. Parallelgraphic's Cortona is recommended for Windows 95, Windows NT, or Windows 2000 environments.

Reliability and Validity

The validity of the VTADL-to-VRML compiler was confirmed during the test and integration procedures.

Source VTADL files were created for both trivial and more complex software architectures, in two architectural styles (call-and-return and layered). The visualization of components, connectors, and the topology of the styles were required to accurately reflect the source VTADL files. The capability to selectively show or hide architectural elements or to establish hyperlinks was also verified. Reliability of the visualizations was confirmed by checking the architectures in each view against the VTADL specifications.

Appendix G provided a representative sample of two test cases.

Summary

This chapter discussed the methodology by which the research in visualization of software architectures was conducted.

The first part of this research created an architecture description language capable of being visualized in three dimensions. The grammar of the language was specified in BNF.

The second part of this research used compiler generation tools (yacc and lex) to create a parser shell based on the BNF specification. The parser shell was capable of recognizing the language but did not generate VRML code. The second part of the research also designed data structures and algorithms capable of representing an intermediate geometric model of the software architecture. Using an architectural style, the architectural elements defined in the source file were mapped to the appropriate geometric model. Code generating algorithms were designed to traverse the geometric model and create the corresponding objects in VRML.

The third part of the research coded the algorithms and data structures defined in the second part, and integrated the coded subroutines within the compiler shell. The result of the third part was a completed compiler program.

The fourth part performed acceptance testing of the completed compiler, while the fifth part conducted case studies to demonstrate the workability of the prototype. The test cases were documented in Appendix G, while the case studies were documented in Appendix I.

Chapter IV

Results

Analysis

This chapter presents the results of two case studies using the VTADL-to-VRML visualization tool. In earlier literature, the software architectures for a mobile robot and the Linux operating system kernel were modeled in the plane (and without using architecture description languages) by the original authors.

In this dissertation research, both case studies were first represented using an architectural description language (VTADL), then translated into VRML using the visualization techniques developed by this dissertation. Each viewpoint was ultimately represented as a three-dimensional, virtual world.

Case Study One was performed originally in *Software Architecture: Perspectives on an Emerging Discipline* [Shaw & Garlan, 1996]. Four alternate architectural approaches were provided by Shaw and Garlan to model the control mechanism of a mobile robot: a control loop, layered, implicit invocation, and blackboard architecture. In this dissertation, the first three solutions were represented in VTADL using the call-and-return and layered styles; however, the fourth solution (blackboard architecture) was ignored, since the blackboard data repository style was not implemented in the compiler.

Case Study Two used the visualization study of the Linux kernel conducted by Bowman et al [Bowman, Siddiqi, & Tanuan, 1998]. The authors originally visualized the Linux kernel in the plane using a tool called the Portable Bookshelf. This dissertation research visualized the Linux kernel using both call-and-return and layered styles within

VRML worlds. However, only the IPC (Interprocess Control) module of Linux was visualized to the lowest level of detail.

The goal of this dissertation was to develop a prototype to demonstrate the "proof of concept" of this visualization technique; namely, to demonstrate that viewpoints on a software architecture could be visualized in three dimensions using the interactive medium of virtual reality. The software architecture, and the viewpoints on the architecture, were first represented in a new software architecture description language presented by this dissertation (VTADL); the architectural viewpoints described by the language were translated into virtual worlds, with each viewpoint defined by a separate VRML file. The separate views were integrated using hyperlinks between the VRML worlds. Traceability to requirements was also demonstrated using hyperlinks from elements of the VRML representation to source documentation (both text and HTML).

The visualized case studies were documented in Appendix I.

Although the prototype worked according to the specifications stated at the beginning of this research, not all aspects of the architecture description language (VTADL) were implemented in the target visualization language. For example, interfaces were not visualized in VRML.

Only the layered and call-and-return architectural styles were implemented in the compiler. Important styles such as the pipelined and data repository styles were ignored in the current version. Heterogeneous styles, while desirable for more complex architectures, were not implemented in the prototype.

Each architectural style was translated into VRML using only a small number of visualized objects. The call-and-return style used different sphere colors and sphere sizes

to represent program modules (a size and color were assigned for each level in the call-and-return tree). Cylinders were used to represent the connectors. The layered style used cone-like extrusions to represent components in a layered architecture. While the limited object set served the purposes of the prototype, a more diverse object set would be desired in any visualization tool beyond the prototype.

Findings

A prototype was developed to translate a software architecture (and the viewpoints on the architecture) described in an ADL into VRML. Details of the prototype and its development were included in the appendices. Appendix G demonstrates the functional correctness of the prototype, while Appendix I demonstrates the application of the prototype to well-known or existing software systems.

This dissertation demonstrated that software architectures could be represented in VTADL; that the VTADL-to-VRML compiler could be successfully used to translate architectural viewpoints described in VTADL into the target VRML files; and that the VRML files could be viewed using a standard Internet browser and a VRML add-on to the browser.

Based on the case studies described in Appendix I, the stated goals of this dissertation were achieved.

Summary of Results

In Chapter I, a fundamental problem in software architecture was pinpointed: how does one represent different viewpoints on the software architecture in a way that is detailed enough for a stakeholder who is a specialist, but comprehensive enough for a generalist? A ramification of this problem was to integrate the different viewpoints in a consistent way, avoiding the pitfalls of viewpoint and architectural mismatch.

This dissertation proposed a solution to the representational problem by developing a visualization tool to allow interaction with a three-dimensional, virtual reality world modeling the software architecture. The research began by designing an ADL capable of representing different architectural viewpoints. A compiler was then developed to translate the ADL into one or more VRML files representing the corresponding viewpoints on the software architecture.

The stated goal was to develop a prototype to demonstrate a "proof of concept" of the visualization tool. Aesthetic concerns, the user response to the visualization, or the impact of the visualization tool on software productivity were not explored in this dissertation. The latter issues will be the topic for future research.

The test cases of Appendix G demonstrate the functional correctness of the prototype. The two case studies of Appendix I demonstrate that the prototype could be used to visualize the software architectures of well-known or existing systems (namely, the software architecture of a mobile robot and the architecture of the Linux operating system kernel).

Based on the narrow constraints established at the beginning of this dissertation research (Chapter I), the results of the test cases (Appendix G), and the two case studies (Appendix I), we concluded that the stated goals of this dissertation were achieved.

While the limited number of architectural styles and visualized objects served the purposes of a prototype, more visualization capabilities would be required for a truly versatile tool. It was recommended that the refined version of the visualization tool be tested and evaluated in an actual software development environment.

Chapter V

Conclusions, Implications, Recommendations, and Summary

Conclusions

Based upon the analysis of results in the fourth chapter, we conclude that the prototype does indeed fulfill the role of serving as a "proof of concept" of the visualization technique. The immediate goals of this dissertation were achieved.

From a purely subjective standpoint, the three-dimensional visualizations in the case studies appeared to be more engaging than the original static visualizations in the plane. The claim that the VRML visualizations are more engaging to stakeholders can be supported by subjective, anecdotal evidence, but cannot be conclusively proved without objective, empirical evidence. A secondary claim that the visualization tool could increase software development productivity (through improving communication between stakeholders) must also be validated by a controlled experiment, measuring the impact of the tool in a software development environment. As specified at the very beginning of this dissertation, the actual impact of the visualization tool on users in a software development environment should be addressed in future research.

Implications

Few tools exist for the three-dimensional visualization of software architecture. To the best knowledge of this research, no compiler existed prior to this dissertation to translate an architecture description language into VRML. Furthermore, the representation of integrated viewpoints on a software architecture *using virtual reality* had not been previously explored. The prototype developed by this research demonstrated that an ADL could model several viewpoints on a software architecture, and that the viewpoints could be translated from the ADL into VRML based upon the architectural style specified in the ADL. The resulting tool is an inexpensive, three-dimensional technique to visualize software architectures.

An additional contribution that the tool made to software architecture was to allow software traceability from the three-dimensional visual representation to the original design rationale. This capability was made possible using hyperlinks from elements in the virtual reality representation to requirements documentation.

The translation of architectural views into VRML was offered as a creative solution to the broader problem of representing contrasting viewpoints on the software architecture. The solution was intended to enhance communication among stakeholders who were specialists in different domains and stakeholders who were generalists. Stakeholders who were generalists, for example, could be managers with an interest in coordinating the different specialties.

Recommendations

The limitations of the prototype should be noted. The current version of the visualization tool needs to be developed beyond a prototype in order for the effectiveness of the tool to be gauged in a software development or training environment.

In the prototype, only two architectural styles were implemented; each style used a limited number of visualized objects. The small number of styles and lack of diversified objects limited the scope of software architectures that could be visualized. Heterogeneous styles were not implemented. Component interfaces, connector directionality, and a greater diversity of visualized objects within each architectural style should be incorporated within the visualization tool.

At present, the visualization technique models only static architectures. Dynamic architectures should be represented using event-based architectural styles. VRML has the capability to animate objects; this capability should be utilized to animate the evolution of software architectures over time, or to animate dynamic relationships between architectural elements.

The resulting visualizations from the compiler are intuitively more engaging than images sketched in a plane. Using VRML, the user can navigate through a three-dimensional world, with the sensation of exploring or manipulating different objects within the world.

However, the actual impact on stakeholders must be examined in a consistent and scientific manner. The effectiveness as a training tool could be determined through analysis of user surveys. The effectiveness of the tool in modeling real-world software architectures could be determined through quantifying software productivity

improvements, improvements in architectural comprehension, and enhanced stakeholder communication.

Summary

Software architecture is an emerging discipline within software engineering. Models based on software architecture attempt to reduce the complexity of a software system by representing the system with coarse-grained structures. A software structure could be represented by components and connections arranged in a specific topology. An architectural style defines the constraints on the topology and instantiation of the structure during run-time. Depending on the stakeholder viewpoint, elements of the topology are interpreted differently; a component, for example, may be an abstraction representing a program module, object, concept, or database.

Software architectures may be described using a graphical or text-based architecture description language (ADL). The key goals of an ADL are to communicate alternate designs between different stakeholders, to detect reusable structures, and to record design decisions. ADLs serve as tools to assist in analytical reasoning about the preliminary software design, to insure software quality early in software development.

A major problem in software architecture has been the difficulty in creating different representations to accommodate the contrasting viewpoints of stakeholders. A set of viewpoints should be conveyed in a way that is both comprehensive enough for specialists but understandable to generalists. The representation problem has been one of integrating different viewpoints without losing consistency (viewpoint mismatch) and without errors in relating architectural structures (architectural mismatch).

This dissertation provided a solution to the representation problem by creating a tool for three-dimensional representation of architectural viewpoints.

The tool consisted of an architecture description language (VTADL) to first describe the software architectures and viewpoints on the architectures; and a VTADL-to-VRML compiler to translate each viewpoint into a separate virtual reality world.

This research was significant since no compiler existed (prior to this dissertation) to translate a dedicated architecture description language into VRML. To the best knowledge of the dissertation author, very little or no research was being conducted on representing software architectural viewpoints in virtual reality.

This research noticed the powerful capability of VRML to model worlds containing a variety of user-defined structures [Ames et al., 1997]. A natural association was made by the author of this dissertation between the world-modeling capabilities of the virtual reality paradigm and the nature of a software architecture viewpoint. A major component of the tool developed in this dissertation was to map a viewpoint definition contained in an architecture description language to a VRML world. Furthermore, VRML offered an inexpensive, easily accessible means (via the Internet) of presenting software architectural visualizations.

An additional benefit of the VTADL-to-VRML compiler was the allowance for software traceability, the capability of tracing a path from elements of the architecture to associated requirements documentation. Using the VRML visualization, a stakeholder could trace the rationale behind the design using hyperlinks from elements of the visualization to source documents.

The goal of the dissertation was to devise algorithms for translating an ADL into effective VRML representations based on the desired viewpoint. The VRML representations were intended to enhance comprehension on the overall design and to improve communications between diverse stakeholders. In other words, the tangible goal was to allow for visualization of more than one viewpoint, and to allow the stakeholder to toggle between multiple viewpoints within a participatory medium. The ADL, along with the ADL's visualization in virtual reality, would serve as a repository for reusable patterns in future projects.

As far as the constraints of this research were concerned, the goal was considered to be reached when we accomplished the following:

1. Developed a prototype consisting of an ADL capable of representing multiple architectural viewpoints;
2. Developed a translator from the ADL to VRML representation;
3. Demonstrated the successfully translated VRML representation from the architecture originally described by the ADL;
4. Demonstrated software traceability (using hyperlinks) from aspects of the VRML representation back to the requirements documentation.

In addition, two architectural styles (the layered and call-and-return) were implemented in the compiler. Visualization of heterogeneous styles, connector directionality, component interfaces, and dynamic architectures were not implemented in the current version of the compiler.

The development and validation of the prototype consisted of five phases.

The first phase of the research was to create an architecture description language capable of being visualized in three dimensions. The new language was called VTADL

(Visually Translatable Architecture Description Language). The full grammar of the new language was specified in Backus-Naur Form (BNF).

The second phase used compiler generation tools (yacc and lex) in a Unix environment to create a parser shell based on the BNF specification. The parser shell at this stage was only capable of recognizing the syntax of the language, and could not generate target VRML code. The second phase also involved the preliminary design of data structures and algorithms representing the intermediate geometric model of the software architecture. Using an architectural style, the architectural elements defined in the source file were mapped to the appropriate geometric model. Code generating algorithms were designed to traverse the geometric model and create the corresponding objects in VRML.

The third phase of the research implemented the algorithms and data structures (defined in the second phase) in the C programming language. Coded subroutines were integrated and tested within the compiler shell, resulting in a completed compiler program.

The fourth phase performed acceptance testing of the completed compiler program (documented in Appendix G), while the fifth phase performed case studies using the final visualization tool (documented in Appendix I).

Case Study One modeled the software architecture for an autonomous mobile robot. Several different solutions were extracted from earlier literature, and re-represented in VTADL. The VTADL-to-VRML compiler was used to generate a VRML file for each view of the architecture.

Case Study Two modeled the software architecture for the Linux operating system kernel in VTADL. Using the compiler, VTADL was translated into several VRML files representing a hierarchy of integrated views on the kernel. Although all modules of the Linux kernel were represented at a high level of abstraction, only the IPC (Interprocess Control) module was visualized to the lowest level of detail.

Hyperlinks to source documentation were used in both case studies, and all views were integrated through the use of VRML portals.

Based on the test cases and two case studies, the prototype demonstrated that the viewpoints on a software architecture could be represented in an architecture description language, then visualized in three dimensions using the techniques of virtual reality (VRML). Further, the prototype demonstrated that the viewpoints could be integrated using hyperlinks, and that software traceability could be established between elements in the virtual world to source documentation.

We concluded that the goals of the dissertation were achieved, but emphasized the limitations of the prototype and constraints placed on the research. The foremost constraint was that no study was undertaken to measure the effectiveness of the visualization tool.

The prototype should be developed into a more polished, versatile tool suitable for use by software developers or trainers. The effectiveness of the refined visualization tool could then be realistically evaluated in an actual software development environment.

Appendix A

BNF Specification for VTADL

(Visually Translatable Architecture Description Language)

BNF Specification for VTADL

(Visually Translatable Architecture Description Language)

Please Note: Terminal tokens are capitalized in bold.

<vtadl> ::= <archmain> <viewpart>

<archmain> ::= **NULL** | <archmain> <archdef>

<archdef> ::= <architecture> <archname> <architect_style>
<leftmark> <partslist> <connslist> <rightmark>

<architecture> ::= **ARCHITECTURE**

<leftmark> ::= **LEFTBRACKET**

<rightmark> ::= **RIGHTBRACKET**

<architect_style> ::= **ARCHSTYLE** <stylechoice>

<stylechoice> ::= **PROGRAM** | **OBJECT** | **PIPELINE** | **LAYER**

<archname> ::= **ID**

<partslist> ::= **COMPLIST** <leftmark> <compbreakdown> <rightmark>

<connslist> ::= **CONNLIST** <leftmark> <connbreakdown> <rightmark>

<compbreakdown> ::= **NULL** | <compbreakdown> <middlemark>

<middlemark> ::= **COMPONENT** <compname> **COMPTYPE** <comptype> <semimark>
<proplist> <intlist>

<proplist> ::= **PROPERTIES COLON** <propdetails>

<propdetails> ::= <component_role> <child_of> <layer_no> <process_def>

<component_role> ::= **COMPROLE COLON** <comp_selection> <semimark>

<comp_selection> ::= **INPUT** | **OUTPUT** | **ROOT** | **CMPPRODUCER** | **CMPCONSUMER**

<child_of> ::= **CHILDOF COLON** <child_selection> <semimark>

<child_selection> ::= **NULL** | <child_id>

<layer_no> ::= **LAYER COLON** <layer_selection> <semimark>

<layer_selection> ::= **NULL** | **LINPUT** | **LOUTPUT** | <layer_id>

```
<layer_id>          ::= ID

<process_def>       ::= PROCESS COLON <process_selection> <semimark>

<process_selection>      ::= NULL | SIGMA | THRESHOLD

<child_id>          ::= ID

<intlist>           ::= INTERLIST COLON <interbreakdown>

<interbreakdown>    ::= NULL | <interbreakdown> <middleinter>

<middleinter>       ::= INTERFACE <inter_direct> <intername> <semimark>
                        <leftmark> INTERFACEROLE COLON
                        <inter_role_choice> <semimark> <rightmark>

<inter_direct>      ::= TOP | BOTTOM | LEFT | RIGHT | FRONT | BACK

<inter_role_choice>      ::= PRODUCER | CONSUMER

<connbreakdown>     ::= NULL | <connbreakdown> <connmiddle>

<comptype>          ::= CMPPE | CMPPROGRAM | CMPCONCEPT
                        | CMPOBJECT | CMPDATA

<connmiddle>        ::= CONNECTOR <connname> CONNTYPE <connecttype>
                        <connectdirect> <semimark> <connend>

<semimark>          ::= SEMICOLON

<compname>          ::= ID

<connname>          ::= ID

<connecttype>       ::= DATAFLOW | CONTROLFLOW | ASSOCIATES

<connectdirect>     ::= UNIDIRECT | BIDIRECT

<connend>           ::= CONNECT LEFTPARENS <interface1> COMMA
                        <interface2> RIGHTPARENS <semimark>

<interface1>        ::= ID

<interface2>        ::= ID

<intername>         ::= ID

<viewpart>          ::= VIEWLIST <leftmark> <mainview>
                        <userviews> <rightmark>

<mainview>          ::= VIEWMAIN <leftmark> <vmainbody> <rightmark>

<vmainbody>         ::= NULL | <useArchSection> <vmainbody>

<useArchSection>    ::= <leftmark> <usePart> <useProp> <rightmark>
```

```
<usePart>          ::= USEARCH <useArchID> <semimark>

<useProp>          ::= <leftmark> <CompConn> <HyperPart> <rightmark>

<useArchID>        ::= ID

<CompConn>         ::= <CompSelects> <ConnSelects>

<CompSelects>      ::= COMPONENTS <CompOptions> <semimark>

<CompOptions>      ::= ALL | <CompOptList>

<CompOptList>      ::= NULL | <CompNext> <CompOptList>

<CompNext>         ::= ID

<ConnSelects>      ::= CONNECTIONS <ConnOptions> <semimark>

<ConnOptions>      ::= ALL | <ConnOptList>

<ConnOptList>      ::= NULL | <ConnNext> <ConnOptList>

<ConnNext>         ::= ID

<HyperPart>        ::= NULL | <Hypercase> <HyperPart>

<Hypercase>        ::= HYPERLINKON <HypConnComp>
                       TOFILE <Filename> <semimark>

<HypConnComp>      ::= ID

<Filename>         ::= ID | QSTRING

<userviews>        ::= NULL | <Viewdef> <userviews>

<Viewdef>          ::= VIEW <viewname> <leftmark> <vmainbody> <rightmark>

<viewname>         ::= ID
```

Appendix B

Lex Specification for VTADL

Lex Specification for VTADL

(Visually Translatable Architecture Description Language)

```
%{

#include "y.tab.h"
#include <string.h>

%}

id        [a-zA-Z][a-zA-Z0-9]*
qstring   \"[^\"\n]*[\"\n]

%%

\n       ;
[\t ]+   ;

Architecture     { return ARCHITECTURE; }
"{"              { return LEFTBRACKET; }
"}"              { return RIGHTBRACKET; }
"("              { return LEFTPARENS; }
")"              { return RIGHTPARENS; }
";"              { return SEMICOLON; }
","              { return COMMA; }
":"              { return COLON; }
Input   { return INPUT; }
Output  { return OUTPUT; }
Root    { return ROOT; }
Producer { return PRODUCER; }
Consumer { return CONSUMER; }
CompProducer  { return CMPPRODUCER; }
CompConsumer  { return CMPCONSUMER; }
Left     { return LEFT; }
Right    { return RIGHT; }
Front    { return FRONT; }
Back     { return BACK; }
Top      { return TOP; }
Bottom   { return BOTTOM; }
Component    { return COMPONENT; }
Properties   { return PROPERTIES; }
CompRole     { return COMPROLE; }
ChildOf      { return CHILDOF; }
Connector    { return CONNECTOR; }
NA       { return NA; }
Style       { return ARCHSTYLE; }
Program      { return PROGRAM; }
Object       { return OBJECT; }
Pipeline     { return PIPELINE; }
Layer     { return LAYER; }
LayerInput { return LINPUT; }
LayerOutput { return LOUTPUT; }
```

```
ComponentList    { return COMPLIST; }
ConnectionList  { return CONNLIST; }
ComponentType  { return COMPTYPE; }
ConnectType  { return CONNTYPE; }
PE     { return CMPPE; }
Cprogram { return CMPPROGRAM; }
Cconcept { return CMPCONCEPT; }
Cobject  { return CMPOBJECT; }
Dataflow   { return DATAFLOW; }
Controlflow  { return CONTROLFLOW; }
Associates  { return ASSOCIATES; }
Unidirect    { return UNIDIRECT; }
Bidirect     { return BIDIRECT; }

Interface   { return INTERFACE; }
InterfaceList  { return INTERLIST; }
InterfaceRole  { return INTERFACEROLE; }

Connect        { return CONNECT; }

Process    { return PROCESS; }
Sigma  { return SIGMA; }
Threshold { return THRESHOLD; }
ViewList  { return VIEWLIST; }
ViewMain  { return VIEWMAIN; }
UsingArch { return USEARCH; }
All { return ALL; }
Components  { return COMPONENTS; }
Connections { return CONNECTIONS; }

HyperLinkOn  { return  HYPERLINKON; }
ToFile  { return TOFILE; }
View   { return VIEW; }

{id}         { yylval.string = strdup(yytext);
               return ID;
             }

{qstring}    {  yylval.string = strdup(yytext+1);  /* skip open quote */
                if(yylval.string[yyleng-2] != '"')
                  printf("/nUnterminated character string/n");
                else  /* remove close quote */
                  yylval.string[yyleng-2] = '\0';
                 return QSTRING;
             }

"$"  { return 0; }

%%
```

Appendix C

Yacc Specification for VTADL Parser Shell

Yacc Specification for VTADL Parser Shell

(without routines for Geometric Modeler and VRML Generator)

For the complete Yacc specification with all subroutines, see Appendix F.

```
vtadl:    archmain viewpart
          ;

archmain:  /* empty */

           | archmain archdef
           ;

archdef:    architecture archname architect_style leftmark
            partslist connslist rightmark
            ;

architecture:  ARCHITECTURE
               ;

leftmark:  LEFTBRACKET
           ;

rightmark:  RIGHTBRACKET
            ;

architect_style:  ARCHSTYLE stylechoice
                  ;

stylechoice:  PROGRAM
              |
              OBJECT
              |
              PIPELINE
              |
              LAYER
              ;

archname:  ID
           ;


partslist:  COMPLIST leftmark compbreakdown rightmark
            ;

connslist:  CONNLIST leftmark connbreakdown rightmark
            ;

compbreakdown:  /*  empty  */
```

```
                |
                compbreakdown middlemark
                ;


middlemark:  COMPONENT compname COMPTYPE comptype semimark proplist intlist
             ;

proplist:  PROPERTIES COLON propdetails
           ;

propdetails:      component_role child_of layer_no process_def
                  ;

component_role:   COMPROLE COLON comp_selection semimark
                  ;

comp_selection:   INPUT
                  |
                  OUTPUT
                  |
                  ROOT
                  |
                  CMPPRODUCER
                  |
                  CMPCONSUMER
                  ;

child_of:     CHILDOF COLON child_selection semimark
              ;

child_selection:  /* Empty */
                  |
                  child_id
                  ;

layer_no:   LAYER COLON layer_selection semimark
            ;

layer_selection:  /* Empty */
                  |
                  LINPUT
                  |
                  LOUTPUT
                  |
                  layer_id
                  ;

layer_id:   ID  { $$ = $1; }
             ;

process_def:  PROCESS COLON process_selection semimark
              ;

process_selection:  /* Empty */
                    |
                    SIGMA
```

```
                    |
                    THRESHOLD
                    ;

child_id:     ID    { $$ = $1; }
              ;

intlist:  INTERLIST COLON interbreakdown
           ;

interbreakdown:  /* empty */
                 |
                 interbreakdown middleinter
                 ;

middleinter:      INTERFACE inter_direct intername semimark
                  leftmark INTERFACEROLE COLON
                  inter_role_choice semimark rightmark
                  ;

inter_direct:  TOP
               |
               BOTTOM
               |
               LEFT
               |
               RIGHT
               |
               FRONT
               |
               BACK
               ;

inter_role_choice:  PRODUCER
                    |
                    CONSUMER
                    ;

connbreakdown:   /* empty */
                 |
                 connbreakdown connmiddle
                 ;

comptype: CMPPE
          |
          CMPPROGRAM
          |
          CMPCONCEPT
          |
          CMPOBJECT
          |
          CMPDATA
          ;


connmiddle:       CONNECTOR connname CONNTYPE connecttype
                  connectdirect semimark connend
```

```
                  ;

semimark:  SEMICOLON
           ;

compname:  ID
           ;

connname:  ID
           ;

connecttype:  DATAFLOW
              |
              CONTROLFLOW
              |
              ASSOCIATES
              ;

connectdirect:  UNIDIRECT
                |
                BIDIRECT
                ;



connend:  CONNECT LEFTPARENS interface1 COMMA interface2 RIGHTPARENS
                  semimark
          ;

interface1:  ID
             ;

interface2:  ID
             ;

intername:  ID
            ;

viewpart:   VIEWLIST leftmark mainview userviews rightmark
            ;

mainview:   VIEWMAIN leftmark vmainbody rightmark
            ;

vmainbody:  /* NULL */
            |
            useArchSection vmainbody
            ;

useArchSection:   leftmark usePart useProp rightmark
                  ;

usePart:  USEARCH useArchID semimark
          ;

useProp: leftmark CompConn HyperPart rightmark
          ;
```

```
useArchID:  ID
            ;

CompConn:  CompSelects ConnSelects
            ;

CompSelects:  COMPONENTS CompOptions semimark
             ;

CompOptions:  ALL
              |
              CompOptList
              ;

CompOptList:  /* NULL */
              |
              CompNext CompOptList
              ;

CompNext:     ID
              ;

ConnSelects: CONNECTIONS ConnOptions semimark
             ;

ConnOptions:  ALL
              |
              ConnOptList
              ;

ConnOptList:  /* NULL */
              |
              ConnNext ConnOptList
              ;

ConnNext:    ID
             ;

HyperPart:  /* NULL */
            |
            Hypercase HyperPart
             ;

Hypercase:        HYPERLINKON HypConnComp TOFILE Filename semimark
                  ;

HypConnComp:      ID
                  ;

Filename:         ID

                  | QSTRING
                  ;

userviews:  /* NULL */
            |
```

```
           Viewdef userviews
           ;

Viewdef:  VIEW viewname leftmark vmainbody rightmark
          ;

viewname:  ID
           ;
```

Appendix D

Subroutine Design: Data Structures and Algorithms for Geometric Modeler

Subroutine Design: Data Structures and Algorithms for Geometric Modeler

This section provides the pseudo-code description of the subroutines referred to by Chapter 3 (Methodology). The subroutines are used by Phase One and Phase Two of the section titled, "Description of Geometric Algorithms on Data Structures."

/* Subroutines follow */

/* The following subroutine uses the architecture descriptive arrays and variables */
/* and inserts them into the linked list representing the architecture list. */
/* The first parameter is a pointer to the current node; the second parameter is the */
/* architecture name; the third is the arch. style; the fourth is the component array; */
/* the fifth is the connector array; the sixth is an array representing connectivity; */
/* the last two parameters are the number of components and number of connectors. */

Insert_Arch_Node (with parameters: pointer to current arch-node, in_arch_ID, style, comp_array, conn_array, from_to array, no_comps, no_conns)

If current node is NULL, the list is empty.
Create a new node and move parameters to corresponding storage locations in the arch node. A temporary array containing connectivity information, "from_to" array, is read into Topology matrix in the arch node.

Using the style variable and Topology matrix, we calculate the relative positions of components and connectors using subroutine:

Calculate_Arch_Positions (arch node pointer).

Since arch node is first node, set archlink, the pointer to next node, to NULL.

Return from subroutine.

If current node is NOT NULL, the list is not empty.
We traverse the arch linked list to the very end, checking as we go along that the arch ID is not already contained in the linked list. We insert the new node at the end of the linked list:

Let curr_node point to first node.

While curr_node.archlink NOT NULL
Compare curr_node.arch_ID to in_arch_ID.
If they are the same, generate ERROR and exit. Otherwise, proceed.
Get the next node in linked list:

Let curr_node = curr_node.archlink.
End While curr_node.archlink is NOT NULL

At this point, we have reached the end of the linked list without duplicates.
We also assume that curr_node.archlink is NULL.

Create a new node, new_archnode.
Move the subroutine parameters to the corresponding locations in the new node. I.e., in_arch_ID is moved to the new node's architecture name, component and connector arrays are moved to new node's arrays, from_to array is moved to new node's Topology matrix, and so on.

Using the style variable and Topology matrix, we calculate the relative positions of components and connectors using subroutine:

Calculate_Arch_Positions (arch node pointer).

Let curr_node.archlink = new_archnode. We add new node to the linked list.

Return from subroutine.

End Insert_Arch_Node.

```
/*  The following subroutine uses the connector, component and Topology matrices        */
/*  of the arch-node to calculate the relative positions of each component and connection. */
/*  The arrays of the arch-node representing the connector and component positions are    */
/*  updated with the calculated positions.  We assume a Call-and-Return architectural      */
/*  style (a tree hierarchy with root and no cycles).   */

/*  Additional Note:  We traverse the tree using a depth-first traversal; a queue is used     */
/*  to store the index of the component nodes as we traverse the tree.                        */
/*  The row number of the Topology matrix (from 0 to number-of-components - 1) represents  */
/*  the index of the current root; the column numbers represent the indices of the children  */
/*  for the row index, ONLY when Topology[row][column] does not contain (-1).           */

/*  The value contained in Topology[row][column], when not (-1), represents the           */
/*  index of the connector attaching the source component (row index) to the destination */
/*  component (col index).  */

/*  The effect of the depth-first traversal using the queue is to access the rows in the      */
/*  order based on the queue, and to access the columns in order from 0 to max_cols.   */
```

Calculate_Arch_Position (pointer to arch_node)

<u>If the arch_node is NULL</u>, return (-1). Otherwise, proceed.

```
/* Note:  curr_root is index of current root.
Set curr_root to 0, the index of the root node.

Initialize position arrays:  Arch_CompPosition, Arch_ConnPosition
Initialize queue.

Insert very first root (curr_root = 0) into queue.

While queue is NOT EMPTY:
    Remove item from queue.
    Let curr_root = item removed from queue.

    /* Count the number of children of curr_root using the columns  */
    /* of Topology matrix.  */

    Set number_of_children to 0.
    For column = 0 to (Number_of_Components - 1)
        If Topology[curr_root][column] NOT (-1)
            Add 1 to number_of_children
            /* We determine level of child node */
            If Arch_CompPosition[column][3] contains a (-1),
                    Arch_CompPosition[column][3] =
                    Arch_CompPosition[curr_root][3] + 1
    End For column = 0 to (Number_of_Components - 1)

    /* We use the number_of_children count to determine delta_theta,                */
    /*  the number of degrees rotation around y-axis in a right-handed coordinate  */
    /*  system.  For example, if there are three children, delta_theta = 2*PI /3.       */

    /* First get the existing positions (x,y,z) from arch-node */

    root_x = Arch_CompPosition[curr_root][0]
    root_y = Arch_CompPosition[curr_root][1]
    root_z = Arch_CompPosition[curr_root][2]

    If  number_of_children >= 1
        delta_theta = (2.0 * PI)  / (number_of_children)
    Otherwise
        delta_theta = 0.0.

    As we record the accumulated rotation of each child along y-axis, we use
    delta_sum, where delta_sum = delta_theta * (current_child - 1).

    /* Initialize current_child counter */
    current_child = 0
```

```
For i = 0 to (Number_of_Components - 1)
    If Topology[curr_root][i] NOT = (-1)
        Add 1 to current_child
        delta_sum = delta_theta * (current_child - 1)
        /* Get the current level */
        Level = Arch_CompPosition[curr_root][3]
        Set the Connector_Height and Sphere_Radius based on Level.
        The higher the level, the smaller the Connector_Height and
        smaller the Sphere_Radius.

        /* Displace component along y based on Connector_Height */
        orig_x = 0.0
        orig_y = 0.0 - Connector_Height
        orig_z = 0.0

        /* Now rotate new center around z-axis by 60 degrees (-PI/3.0)  */
        child_x = (-orig_y) * sin(-PI/3.0)
        child_y = orig_y * cos(-PI/3.0)
        child_z = orig_z

        /* Now rotate around y-axis by delta_sum */
        old_child_x = child_x
        old_child_y = child_y
        old_child_z = child_z

        child_x = old_child_z * sin(delta_sum) + old_child_x * cos(delta_sum)
        child_y = old_child_y
        child_z = old_child_z * cos(delta_sum) - old_child_x * sin(delta_sum)

        /* Translate back to position under original root */
        child_x = child_x + root_x
        child_y = child_y + root_y
        child_z = child_z + root_z

        /* Store the calculated positions to arch-node position array */
        Arch_CompPosition[i][0] = child_x
        Arch_CompPosition[i][1] = child_y
        Arch_CompPosition[i][2] = child_z

        /* Now we establish connection positions given root position and */
        /* child position.  */

        /* First translate back to the origin */
        conn_orig_x = 0.0
        conn_orig_y = (0.0 - Connector_Height) / 2.0
```

conn_orig_z = 0.0

/* First rotate the leg (cylinder) center-point around z-axis */
conn_x = 0.0 * cos(-PI/3.0) - conn_orig_y * sin(-PI/3.0)
conn_y = 0.0 * sin(-PI/3.0) + conn_orig_y * cos(-PI/3.0)
conn_z = conn_orig_z

/* We are ready to rotate leg center-point around y-axis by delta_sum */
/* We have calculated the delta_sum for this rotation; */
/* However, the actual rotation of the cylinder will be performed */
/* by the VRML Generator from this current position. */
old_conn_x = conn_x
old_conn_y = conn_y
old_conn_z = conn_z

/* First translate back to location under current root node */
conn_x = conn_x + root_x
conn_y = conn_y + root_y
conn_z = conn_z + root_z

/* Get the connector name */
conn_id = Topology[curr_root][i]

/* Store the positions, along with delta_sum, to connector position array */
Arch_ConnPosition[conn_id][0] = conn_x
Arch_ConnPosition[conn_id][1] = conn_y
Arch_ConnPosition[conn_id][2] = conn_z
Arch_ConnPosition[conn_id][3] = delta_sum

/* Insert the child, i, into queue */
Insert_queue_item(i)

End For i = 0 to (Number_of_Components - 1)

End While queue is NOT EMPTY.

Return from subroutine.

End Calculate_Arch_Position.

Appendix E

Example of VTADL Source File: "Example.txt"

Appendix E

Example of a VTADL Source file: "Example.txt"

```
Architecture ExampleProgram
Style Program
  {
      ComponentList
         {
               Component Alpha
                ComponentType Cprogram;
                Properties:
                   CompRole:  CompConsumer;
                   ChildOf: ;
                   Layer: ;
                   Process: ;
                InterfaceList:
                  Interface Bottom AlphaSocket;
                     { InterfaceRole: Consumer;  }

               Component Beta
                ComponentType Cprogram;
                Properties:
                     CompRole: CompProducer;
                     ChildOf: ;
                     Layer: ;
                     Process:  ;
                     InterfaceList:
                     Interface Top BetaSocket;
                     { InterfaceRole: Producer;  }

                Component Gamma
                ComponentType Cprogram;
                Properties:
                     CompRole: CompProducer;
                     ChildOf: ;
                     Layer: ;
                     Process: ;
                     InterfaceList:
                     Interface Top GammaSocket;
                     { InterfaceRole: Producer; }

                Component Delta
                ComponentType Cprogram;
```

```
                Properties:
                    CompRole: CompProducer;
                    ChildOf: ;
                    Layer: ;
                    Process: ;
                    InterfaceList:
                    Interface Top DeltaSocket;
                    { InterfaceRole: Producer; }
          }

      ConnectionList
          {
             Connector Call1
               ConnectType Controlflow Unidirect;
               Connect(AlphaSocket, BetaSocket);
             Connector Call2
               ConnectType Controlflow Unidirect;
               Connect(AlphaSocket, GammaSocket);
             Connector Call3
               ConnectType Controlflow Unidirect;
               Connect(AlphaSocket, DeltaSocket);
          }
 }

Architecture ExampleLayer
Style Layer
 {
   ComponentList
     {
      Component LevelOne
        ComponentType Cprogram;
        Properties:
         CompRole:  CompProducer;
         ChildOf: ;
         Layer: L1;
         Process: ;
        InterfaceList:
          Interface Bottom Level1Socket;
             { InterfaceRole: Producer; }

       Component LevelTwo
        ComponentType Cprogram;
        Properties:
        CompRole: CompConsumer;
        ChildOf: ;
        Layer: L2;
```

```
      Process: ;
      InterfaceList:
        Interface Top Level2Socket;
          { InterfaceRole: Consumer; }

     Component LevelThree
      ComponentType Cprogram;
      Properties:
      CompRole: CompConsumer;
      ChildOf: ;
      Layer: L3;
      Process: ;
      InterfaceList:
        Interface Top Level3Socket;
          { InterfaceRole: Consumer; }

      Component LevelFour
       ComponentType Cprogram;
       Properties:
        CompRole: CompConsumer;
        ChildOf: ;
        Layer: L4;
        Process: ;
        InterfaceList:
          Interface Top Level4Socket;
            { InterfaceRole: Consumer; }
   }
   ConnectionList
    {
     Connector Service1
       ConnectType Dataflow Unidirect;
       Connect(Level1Socket,Level2Socket);

     Connector Service2
       ConnectType Dataflow Unidirect;
       Connect(Level2Socket,Level3Socket);

     Connector Service3
       ConnectType Dataflow Unidirect;
       Connect(Level3Socket,Level4Socket);
    }
}

   ViewList
     {
      ViewMain {    { UsingArch ExampleProgram;
```

```
                  { Components All;
                    Connections All;
                    HyperLinkOn Alpha
                    ToFile "Example.txt"; } }
              { UsingArch ExampleLayer;
                  {  Components All;
                     Connections All;
                     HyperLinkOn LevelFour
                       ToFile SecondView;  }  }
          }


View SecondView {
              { UsingArch ExampleLayer;
                  { Components LevelOne LevelTwo;
                    Connections All;
                    HyperLinkOn LevelOne ToFile Main;
                    HyperLinkOn LevelTwo
                     ToFile "Example.txt"; } }
              { UsingArch ExampleLayer;
                  { Components LevelOne LevelThree LevelFour;
                    Connections All; } }
              { UsingArch ExampleLayer;
                 { Components All;
                   Connections All; } }
              }
 }
$
```

Appendix F

Complete Yacc Specification for VTADL Parser Shell

(includes routines for Geometric Model and VRML Generator)

Complete Yacc Specification for VTADL Parser Shell

(includes routines for Geometric Modeler and VRML Generator)

```
%{
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <memory.h>
#include <math.h>

#define MAX_QUEUE_POSITION 120

int comp_count = 0;
int conn_count = 0;
int arch_count = 0;
int inter_count = 0;
int curr_row_count = 0;
int curr_view_index = 0;  /* Index of view names, for view_names_array */
int curr_arch_index = 0;
char arch_name[25] = " ";
char comp_name[25] = " ";
char conn_name[25] = " ";
char interface_name[25] = " ";
int interface_comp[25];   /* Component for Interface */
char first_conn_name[25] = " ";
char comp_name_array[25][25];
char test_string[25];  /* Temp string for testing */
char conn_name_array[25][25];
char inter_name_array[25][25];
char view_name[25] = " ";
char view_names_array[25][25];
char tmp_arch_ID[25][25];
int tmp_all_comps[25];
int tmp_all_conns[25];
int tmp_view_comps[25][25];
int tmp_view_conns[25][25];
char tmp_hyper_comps[625][25];
char tmp_hyper_conns[625][25];
char stop_var[10];
char use_arch_name[25] = " ";  /* Use arch name for usearch nodes */

char style_char[2];
int style_var = 1;
int test_result;    /* General variable used for test results */
int test_comp;      /* Check for component index */
int test_conn;      /* Check for conn index */
int i; /* Subscript for array */
int j; /* Subscript for array */
float Conn_Height = 5.0;
float Sphere_Radius = 0.5;
float Cylinder_Radius = 0.1;
```

```
float PI = 3.14159;

/* from_to[0] is connection
/* from_to[1] is index of source component
/* from_to[2] is index of destination component */
int from_to[25][3];

/* ==================================================    */
/* USED FOR TOPOLOGY MATRIX TRAVERSAL:                   */
/*                                                       */
int curr_root;   /* Current root of node traversal */
int node_count;  /* Count of nonempty nodes in row of Topology matrix */
int no_components; /* Number of components */
int child_count; /* Count of child nodes for curr root row of Top matrix */
int queue_empty_flag;  /* Queue empty? 1 = true, 0=false */
int conn_node;  /* connection node index */
float root_x;   /* xyz coordinate positions of root node */
float root_y;
float root_z;
float child_x;
float child_y;
float child_z;
float orig_x;
float orig_y;
float orig_z;
float conn_orig_x;
float conn_orig_y;
float conn_orig_z;
float old_child_x;
float old_child_y;
float old_child_z;
float old_conn_x;
float old_conn_y;
float old_conn_z;
float temp_child_x;
float temp_child_y;
float temp_child_z;
float conn_x;
float conn_y;
float conn_z;
float delta_sum = 0.0;     /* Angular sum for rotation */
float delta_theta = 0.0;   /* Angle (radians) for rotation */

double rads60degree;
float conn_level;
int conn_id;

/*====================================================== */

/* Arch-list Node */
typedef struct arch_node
 {
  int Style;                         /* Architecture Style, 1-4 */
  char Arch_ID[25];                  /* Architecture Name */
  char Arch_CompID[25][25];          /* Max 25 comp names, 25 chars each */
  double Arch_CompPosition[25][4];  /* Comp. position, x,y,z,level; */
  char Arch_ConnID[25][25];          /* Max 25 conn. names, 25 chars each */
```

```
  double Arch_ConnPosition[25][4];  /* Conn. Position  */
  int Topology[25][25];             /* Indices by comp row, col */
                                    /* Determines connector at (row,col) */
  int No_Comps;                     /* No. of comps in architecture */
  int No_Conns;                     /* No. of connects in architecture */
  struct arch_node *archlink;       /* link to next arch_node in list */
 } ANODE;

/* View-list Node */
/* Now includes multiple arch. array instead of usearch.
/* Also includes View_Comps, View_Conns, and Hyperlinks    */
  typedef struct viewnode
  {
   char View_ID[25];                 /* Name of View */
   int no_archs;                     /* No of Archs in View */
   char arch_ID[25][25];             /* 25 Possible Archs Max */
   int all_comps[25];                /* All components flag for each arch */
   int all_conns[25];                /* All connectors flag for each arch */
   int view_comps[25][25];           /* View tags for each arch and comp */
   int view_conns[25][25];          /* View tags for each arch and connector */
   char Hyperlinks_Comps[625][25];  /* Each arch uses 25 rows, cols */
   char Hyperlinks_Conns[625][25];  /* Each arch uses 25 rows, cols across */
                                    /* Index of arch is calculated (n-1)*25 +
                                    /* comp or component index */
   struct viewnode *viewlink;       /* Link to next viewnode */
  } VNODE;


/* Define node for linked lists */
typedef struct node
 {
   char data[25];
   struct node *link;
 } LNODE;

/* Queue variables */
int queue[MAX_QUEUE_POSITION];
int front_of_queue = MAX_QUEUE_POSITION - 1;
int rear_of_queue = MAX_QUEUE_POSITION - 1;
int queue_empty_flag;

 ANODE *r1 = NULL;   /* Arch node list empty */
 VNODE *r2 = NULL;   /* View Node list empty */
 ANODE *r3 = NULL;   /* Temp ref node for arch */

/* Subroutine declarations */

%}

%union {
     char *string;    /* string buffer for various names */
}

%token <string> ID QSTRING
%token ARCHITECTURE LEFTBRACKET RIGHTBRACKET COMPONENT CONNECTOR
%token SEMICOLON ARCHSTYLE PROGRAM OBJECT PIPELINE LAYER COMPLIST
%token CONNLIST COMPTYPE CONNTYPE CMPPE CMPPROGRAM CMPCONCEPT CMPOBJECT
```

```
%token CMPDATA DATAFLOW CONTROLFLOW ASSOCIATES UNIDIRECT BIDIRECT
%token INTERLIST INTERFACE INTERFACEROLE CONNECT LEFTPARENS RIGHTPARENS COMMA
%token PROPERTIES COMPROLE CHILDOF NA COLON PROCESS SIGMA THRESHOLD
%token INPUT OUTPUT ROOT PRODUCER CONSUMER LEFT RIGHT FRONT BACK TOP BOTTOM
%token CMPPRODUCER CMPCONSUMER LINPUT LOUTPUT VIEWLIST VIEWMAIN USEARCH
%token VIEW COMPONENTS CONNECTIONS HYPERLINKON ALL TOFILE

%type <string> archname compname connname interface1 interface2 intername
layer_id child_id useArchID
%type <string> CompNext ConnNext HypConnComp Filename viewname


%%

vtadl:    archmain viewpart
          { printf("\n Generating view files... \n");
            generate_view_files(r1,r2);
            printf("\n Tasks are complete! \n"); }
          ;

archmain:  /* empty */

             { show_arch_nodes(r1);      /* Show archs diagnostics */
               strcpy(view_name,"Main"); /* Store Main view */
               arch_count = 0;           /* Main arch count is 0 */
               curr_view_index = 0;      /* Init view count */
               for(i=0; i<=24; i++)
               {
                strcpy(tmp_arch_ID[i]," "); /* Init view */
                tmp_all_comps[i] = 0;       /* All comps */
                tmp_all_conns[i] = 0;       /* All conns */
                for(j=0; j<=24; j++)
                {
                 tmp_view_comps[i][j] = 0; /* View comps */
                 tmp_view_conns[i][j] = 0; /* View conns */
                 view_names_array[i][j] = ' ';
                }
               }
               for(i=0; i<=624; i++)   /* Init hyperlink arrays */
                {
                 strcpy(tmp_hyper_comps[i]," ");
                 strcpy(tmp_hyper_conns[i]," ");
                }
             }

           | archmain archdef
           ;

archdef:  architecture archname architect_style leftmark
          partslist connslist rightmark
           { insert_arch_node(&r1, arch_name, style_var, comp_name_array,
              conn_name_array, from_to, comp_count, conn_count); }
          ;

architecture:  ARCHITECTURE
               ;
```

```
leftmark:  LEFTBRACKET
           ;

rightmark:  RIGHTBRACKET
            ;

architect_style:  ARCHSTYLE stylechoice
                  ;

stylechoice:  PROGRAM  { style_var = 1; }
              |
              OBJECT  { style_var = 4; }
              |
              PIPELINE { style_var = 3; }
              |
              LAYER    { style_var = 2; }
              ;

archname:  ID  { comp_count = 0;
                 conn_count = 0;
                 inter_count = 0;


                 for(i=0; i<=24; i++)
                  {
                   from_to[i][0] = 0;
                   from_to[i][1] = 0;
                   from_to[i][2] = 0;
                   interface_comp[i] = 0;
                   for(j=0; j<=24; j++)
                    {
                    comp_name_array[i][j] = ' ';
                    conn_name_array[i][j] = ' ';
                    inter_name_array[i][j] = ' ';
                    }
                  }
                strcpy(arch_name,$1);
                }
           ;


partslist:  COMPLIST leftmark compbreakdown rightmark
                { printf("\n Comp List as Follows: \n");
                  for(i=0; i<= comp_count-1; i++)
                    {
                     printf("\n Component : %s \n",comp_name_array[i]);
                    }
                  printf("\n Interface List as Follows: \n");
                  for(i=0; i<= inter_count-1; i++)
                   {
                     printf("\n Interface: %s ",inter_name_array[i]);
                     printf("   for Comp No: %6d Comp Name: %s \n",
                     interface_comp[i],
                     comp_name_array[interface_comp[i]]);
                    }
                }
            ;
```

```
connslist:  CONNLIST leftmark connbreakdown rightmark
               { printf("\n ===================================");
               printf("\n *** Connection List as Follows: ***");
               for(i=0; i<= conn_count-1; i++)
               {
                printf("\n Conn No: %6d Conn Name: %s  From: %s To: %s \n",
                       from_to[i][0], conn_name_array[i],
                       comp_name_array[from_to[i][1]],
                       comp_name_array[from_to[i][2]]);
               }

               }
            ;

compbreakdown:  /*  empty  */
                |
                compbreakdown middlemark
                ;

middlemark:  COMPONENT compname COMPTYPE comptype semimark
             proplist intlist
             ;

proplist:  PROPERTIES COLON propdetails
           ;

propdetails:  component_role child_of layer_no process_def

component_role:  COMPROLE COLON comp_selection semimark

comp_selection:  INPUT
                 |
                 OUTPUT
                 |
                 ROOT
                 |
                 CMPPRODUCER
                 |
                 CMPCONSUMER
                 ;

child_of:     CHILDOF COLON child_selection semimark
              ;

child_selection:  /* Empty */
                  |
                  child_id
                  ;

layer_no:   LAYER COLON layer_selection semimark
            ;

layer_selection:  /* Empty */
                  |
                  LINPUT
                  |
```

```
                  LOUTPUT
                  |
                  layer_id
                    ;

layer_id:    ID  { $$ = $1; }
             ;

process_def:  PROCESS COLON process_selection semimark
              ;

process_selection:  /* Empty */
                    |
                    SIGMA
                    |
                    THRESHOLD
                    ;

child_id:     ID    { $$ = $1; }
             ;

intlist:  INTERLIST COLON interbreakdown
           ;

interbreakdown:  /* empty */
                 |
                 interbreakdown middleinter
                 ;

middleinter:     INTERFACE inter_direct intername semimark
                 leftmark INTERFACEROLE COLON
                 inter_role_choice semimark rightmark
                 ;

inter_direct:  TOP
               |
               BOTTOM
               |
               LEFT
               |
               RIGHT
               |
               FRONT
               |
               BACK
               ;

inter_role_choice:  PRODUCER
                    |
                    CONSUMER
                    ;

connbreakdown:   /* empty */
                 |
                 connbreakdown connmiddle
                 ;
```

```
comptype: CMPPE
          |
          CMPPROGRAM
          |
          CMPCONCEPT
          |
          CMPOBJECT
          |
          CMPDATA
          ;

connmiddle:  CONNECTOR connname CONNTYPE connecttype
             connectdirect semimark connend
             ;

semimark:  SEMICOLON
           ;

compname:  ID  { strcpy(comp_name,$1);
                 strcpy(comp_name_array[comp_count], comp_name);
                 comp_count = comp_count + 1;
                }
           ;

connname:  ID  { strcpy(conn_name,$1);
                 strcpy(conn_name_array[conn_count], conn_name);
                 from_to[conn_count][0] = conn_count;
                 conn_count = conn_count + 1;
               }
           ;

connecttype:  DATAFLOW
              |
              CONTROLFLOW
              |
              ASSOCIATES
              ;

connectdirect:  UNIDIRECT
                |
                BIDIRECT
                ;

connend:  CONNECT LEFTPARENS interface1 COMMA interface2 RIGHTPARENS
                  semimark
          ;

interface1:  ID {test_result =
                 find_interface_comp(inter_name_array, $1, interface_comp,
                 inter_count);
                 if(test_result >= 0)
                  {
                     from_to[conn_count-1][1] = test_result;
                  }
                 else
                  {
                   printf("\n Error! Interface does not exist! \n");
```

```
                  exit(0);
                 }
                }
             ;

interface2:  ID { test_result =
                 find_interface_comp(inter_name_array, $1, interface_comp,
                 inter_count);

                  if(test_result >= 0)
                 {
                   from_to[conn_count-1][2] = test_result;
                 }
                else
                 {
                  printf("\n Error! Interface does not exist! \n");
                  exit(0);
                 }

                }
             ;

intername:  ID { strcpy(interface_name,$1);
                strcpy(inter_name_array[inter_count], interface_name);
                interface_comp[inter_count] = comp_count - 1;
                inter_count = inter_count + 1;
              }
            ;

viewpart:  VIEWLIST leftmark mainview userviews rightmark
              { printf(" \n View List successful... \n ");
                printf(" \n curr_view_index:
                %d indexed views \n",curr_view_index);
                printf(" \n List as follows: ");
                for(i=0; i<=curr_view_index-1; i++)
                   {
                    printf("\n %s ",view_names_array[i]);
                   }
              }
           ;

mainview:  VIEWMAIN leftmark vmainbody rightmark
              { /* Check if view_name is duplicate */
                test_result = search_view_list(r2, view_name);

                if(test_result == 1)
                  {
                   printf("\n *** Error: View already exists! \n");
                   exit(0);
                  }

                strcpy(view_names_array[curr_view_index],view_name);

                curr_view_index = curr_view_index + 1;

                /* Insert Main Node! */
                insert_view_node(&r2,view_name,tmp_arch_ID,
```

```
                                tmp_all_comps,tmp_all_conns,
                                tmp_view_comps,tmp_view_conns,
                                tmp_hyper_comps, tmp_hyper_conns,
                                arch_count);

              printf("\n *** Main Node inserted *** \n");
              for(i=0; i<=arch_count-1; i++)
               {
                printf("\ntmp_arch_ID[%d] %s \n",i,tmp_arch_ID[i]);
               }
              /* above: insert view with parameters */

             /* reinitialize arrays, increment view counters */

               strcpy(view_name," "); /* Clear view name */
               arch_count = 0; /* Init arch count */

               for(i=0; i<=24; i++)
               {
                strcpy(tmp_arch_ID[i]," "); /* Init arch array */
                tmp_all_comps[i] = 0;       /* All comps */
                tmp_all_conns[i] = 0;       /* All conns */
                for(j=0; j<=24; j++)
                {
                 tmp_view_comps[i][j] = 0; /* View comps */
                 tmp_view_conns[i][j] = 0; /* View conns */
                }
               }
               for(i=0; i<=624; i++)   /* Init hyperlink arrays */
                {
                 strcpy(tmp_hyper_comps[i]," ");
                 strcpy(tmp_hyper_conns[i]," ");
                }

               }
              ;

vmainbody:  /* NULL */
           |
           useArchSection vmainbody
           ;

useArchSection: leftmark usePart useProp rightmark
                 { test_result = calculate_arch_position(&r3);
                   if(test_result == -1)
                   {
                    printf("\n calc failed! \n");
                   }
                 }
           ;

usePart:  USEARCH useArchID semimark
          ;

useProp: leftmark CompConn HyperPart rightmark
          ;
```

```
useArchID:  ID  {  strcpy(use_arch_name, $1);
                   r3 = r1;  /* r3 is temp arch node for ref */

                   /* search r3 for arch ID; use r3 as ref */
                   if(r3 == NULL)
                    {
                     printf("\nArch list is empty! Exiting... \n");
                     exit(0);
                    }
                   if(r3 != NULL)
                   {
                     item_found = 0;
                     do
                      {
                        strcpy(teststring,r3->Arch_ID);
                        if(strncmp(teststring,use_arch_name,
                           strlen(use_arch_name)) == 0)
                        {
                         item_found = 1;
                          }
                          else
                          {
                                r3 = r3->archlink;
                          }
                    } while( (r3 != NULL) && (item_found != 1) );

              }  /* end if r3 not NULL */

            if(r3 == NULL)
              {
                    printf("\n ERROR! *** Arch ID not found!  EXITING... \n");
                    exit(0);
              }


             /* Assume valid arch node...proceed!  */
              arch_count = arch_count + 1;
              curr_arch_index = arch_count - 1;
              strcpy(tmp_arch_ID[curr_arch_index],use_arch_name);

             }
            ;

CompConn:  CompSelects ConnSelects
            ;

CompSelects:  COMPONENTS CompOptions semimark
             ;

CompOptions:  ALL   { tmp_all_comps[curr_arch_index] = 1; }
              |
              CompOptList
              ;

CompOptList:  /* NULL */
              |
              CompNext CompOptList
```

```
              ;

CompNext:     ID  { strcpy(comp_name,$1);
                    /* Check if comp in architecture */
                    test_result =
                     find_comp_index(comp_name,r3->Arch_CompID,
                                                r3->No_Comps);
                    if(test_result == -1)
                    {
                      printf("\n ERROR! Invalid component in architecture!\n");
                      exit(0);
                    }

                    tmp_view_comps[curr_arch_index][test_result] = 1;
                  }
              ;


ConnSelects: CONNECTIONS ConnOptions semimark
             ;

ConnOptions:  ALL { tmp_all_conns[curr_arch_index] = 1; }
              |
              ConnOptList
              ;

ConnOptList:  /* NULL */
              |
              ConnNext ConnOptList
              ;

ConnNext:    ID  { strcpy(conn_name,$1);
                   test_result = find_conn_index(conn_name,r3->Arch_ConnID,
                                 r3->No_Conns);
                   if(test_result == -1)
                    {
                     printf("\n ERROR! Invalid connector in architecture!\n");
                     exit(0);
                    }
                   tmp_view_conns[curr_arch_index][test_result] = 1;
                 }
             ;

HyperPart:  /* NULL */
            |
            Hypercase HyperPart
             ;

Hypercase:  HYPERLINKON HypConnComp TOFILE Filename semimark
            ;

HypConnComp:  ID  {   strcpy(test_string, $1);
                      test_comp = find_comp_index(test_string, r3->Arch_CompID,
                                  r3->No_Comps);
                      test_conn = find_conn_index(test_string, r3->Arch_ConnID,
                                  r3->No_Conns);
                      if( (test_comp == -1) && (test_conn == -1) )
```

```
                         {
                           printf("\n ERROR! Invalid comp or
                                    conn in hyperlink!\n");
                           exit(0);
                         }
                      if(test_comp != -1)  /* Hyperlink on Component */
                       {
                         hyperlink_index = 25*(curr_arch_index) + test_comp;
                       }
                      if(test_conn != -1)  /* Hyperlink on Connector */
                       {
                         hyperlink_index = 25*(curr_arch_index) + test_conn;
                       }

                    }
             ;

Filename:    ID  {  strcpy(filename,$1);
                    strcat(filename,".wrl");
                     if(test_comp != -1)
                     {
                       strcpy(tmp_hyper_comps[hyperlink_index],filename);
                     }
                    if(test_conn != -1)
                     {
                       strcpy(tmp_hyper_conns[hyperlink_index],filename);
                     }
                  }
              | QSTRING
                  { strcpy(filename,$1);
                    if(test_comp != -1)
                     {
                       strcpy(tmp_hyper_comps[hyperlink_index],filename);
                     }
                    if(test_conn != -1)
                     {
                       strcpy(tmp_hyper_conns[hyperlink_index],filename);
                     }
                   }

             ;

userviews:  /* NULL */
            |
            Viewdef userviews
            ;

Viewdef:  VIEW viewname leftmark vmainbody rightmark
           {  insert_view_node(&r2,view_name,tmp_arch_ID,
                                tmp_all_comps,tmp_all_conns,
                                tmp_view_comps,tmp_view_conns,
                                tmp_hyper_comps, tmp_hyper_conns,
                                arch_count);
              /* above: insert view with parameters */
              /* Get ready for next view node */
              /* Fix: curr_view_index already increment by viewname */
```

```
              /* reinitialize arrays, increment view counters */

                strcpy(view_name," "); /* Clear view name */
                arch_count = 0; /* Init arch count */

                for(i=0; i<=24; i++)
                {
                 strcpy(tmp_arch_ID[i]," "); /* Init arch array */
                 tmp_all_comps[i] = 0;       /* All comps */
                 tmp_all_conns[i] = 0;       /* All conns */
                 for(j=0; j<=24; j++)
                 {
                  tmp_view_comps[i][j] = 0; /* View comps */
                  tmp_view_conns[i][j] = 0; /* View conns */
                 }
                }
                for(i=0; i<=624; i++)   /* Init hyperlink arrays */
                 {
                  strcpy(tmp_hyper_comps[i]," ");
                  strcpy(tmp_hyper_conns[i]," ");
                 }
              }
           ;

viewname:  ID  { strcpy(view_name,$1);

                  /* Check for duplicate view */
                  test_result = search_view_list(r2, view_name);

                  if(test_result == 1)
                    {
                     printf("\n *** Error: View already exists! \n");
                     exit(0);
                    }
                  strcpy(view_names_array[curr_view_index],view_name);
                  curr_view_index = curr_view_index + 1;
                }
            ;

%%

    LNODE *n1 = NULL;

    /*  ANODE *r1 = NULL;  Arch Node list empty */
   /*  VNODE  *r2 = NULL;  View Node list empty */
   /*  ANODE  *r3 = NULL;  Temp ref node for arch */

    int i,j;
    int from_index;
    int item_found;  /* 1 = found, 0 = not found */
    int to_index;
    int test_result;
    int hyperlink_index;
    char filename[25];   /* Hyperlink to filename */
    char teststring[25]; /* String to compare */
```

```
main()
  {
     do
       {
     yyparse();
       }
        while(!EOF);
  }

show_list(LNODE *ptr)
 {
       int counter = 0;
       printf("\n");

       while(ptr != NULL)
       {
             printf("Next node: %s",ptr->data);
             ptr = ptr->link;
             counter = counter + 1;
             printf("\n");
       }
       printf("\n");
       printf("Counter is %d",counter);

 }

 count_layers(LNODE *ptr)
 {
       int layer_count = 0;

       while(ptr != NULL)
       {
             ptr = ptr->link;
             layer_count = layer_count + 1;
       }

       return(layer_count);
 }

 add_layer_component(LNODE **ptr, char new_layer[25])
 {
    LNODE *p1, *p2;
      char teststring[25];

      p1 = *ptr;

      if(p1 == NULL)   /* if list is empty */
      {
            p1 = malloc(sizeof(LNODE));
            if(p1 != NULL)
                    {
                          strcpy(p1->data,new_layer);
                          p1->link = NULL;
                          *ptr = p1;
                    }
      }
      else    /* if list is not empty */
```

```
      {
        while(p1->link != NULL)
            {
                strcpy(teststring,p1->data);
                if(strncmp(teststring,new_layer,strlen(new_layer)) != 0)
                  {
                        p1 = p1->link;
                  }
                else
                  {
                     printf("\n Error -- duplicate layer name \n");
                     exit(0);
                  }
            }
            p2 = malloc(sizeof(LNODE));
            if(p2 != NULL)
            {
                  strcpy(p2->data,new_layer);
                  p2->link = NULL;
                  p1->link = p2;
            }
      }
}

/* Returns index of component corresponding to interface name */
find_interface_comp(char in_inter_array[25][25], char in_test_intername[25],
                    int in_inter_comp[25], int in_no_of_inters)
 {
   int i;
   int result_index;

   result_index = -1;

   for(i=0; i<=(in_no_of_inters-1); i++)
   {
       if(strncmp(in_inter_array[i],in_test_intername,
                  strlen(in_test_intername)) == 0)
           result_index = i;
   }

       if(result_index >= 0)
         return( in_inter_comp[result_index] ); /* Give index of component of
interface */

       if(result_index < 0)   /* Case where interface not found */
         return(-1);
 }

/* Initialize queue */
/* Example of correct usage:
/*  init_queue(test_queue);  */

 init_queue()
 {
       int i;

       for(i=0; i<= (MAX_QUEUE_POSITION - 1); i++)
```

```
       {
          queue[i] = -1;
       }
 }

 /* returns 0 if queue is not empty, 1 if queue is empty */
 /* Note that an array content of -1 means value is NULL */
 is_queue_empty()
 {
   if( (queue[front_of_queue] == -1) && (front_of_queue == rear_of_queue) )
         return(1);
   else
       return(0);
 }

void show_queue()
 {
       int i;

       printf("\n ");
       for(i=0; i<= (MAX_QUEUE_POSITION - 1); i++)
       {
             printf("  %4d  ",queue[i]);
       }
       printf("\n ");
 }

 insert_queue_item(int insert_item)
 {
        int switchvar;

     /*  Do case
     /*
     /*  Check case where in_rear == in_front */
     /*  WHERE CASE IS TRUE, in_rear == in_front:
     /*    Conceptually, two things are possible in this case:
     /*        Either the queue is empty or we are at the very first entry!
     /*
     /*    If in_rear == in_front AND queue[in_rear] is empty (-1)
     /*     then set both current in_rear, in_front to MAX_QUEUE_POSITION (199)
     /*        Place item at current location of in_rear
     /*           queue[in_rear] = insert_item
     /*     else
     /*        if in_rear == in_front and queue[in_rear] is NOT empty
     /*         then check if in_rear = 0
     /*               if in_rear is 0, generate ERROR message ("OUT OF MEMORY")
     /*               if in_rear is NOT 0,
     /*                 then in_rear = in_rear - 1
     /*                 Place item at new location of in_rear
     /*                   queue[in_rear] = insert_item
     /*
     /*  Check case where in_rear != in_front
     /*  WHERE CASE IS TRUE, in_rear NOT EQUAL in_front:
     /*      If in_rear != in_front
     /*       {
     /*          If in_rear > 0
     /*             in_rear = in_rear - 1
```

```
/*             Place item at new in_rear value
/*                       queue[in_rear] = insert_item
/*          else
/*             if in_rear = 0
/*                Place item at 0 position with warning
/*                queue[in_rear] = insert_item
/*                Print "Warning!  Memory out!"
/*       }
/*                                                 */


  if(  (rear_of_queue == front_of_queue) && (queue[rear_of_queue] == -1) )
        switchvar = 1;  /* Flags equal and empty contents */

  if( (rear_of_queue == front_of_queue) && (queue[rear_of_queue] != -1))
        switchvar = 2;  /* Flags equal but contents not empty */

  if(rear_of_queue != front_of_queue)
        switchvar = 3;  /* Flags not equal */

  switch(switchvar)
  {
  case 1:
        {
         printf("\n Case 1... \n");
         rear_of_queue = MAX_QUEUE_POSITION - 1;
         front_of_queue = MAX_QUEUE_POSITION - 1;
     queue[rear_of_queue] = insert_item;
         break;
        }
case 2:
        {
              printf("\n Case 2... \n");
              if(rear_of_queue == 0)  {
               printf("\n OUT OF MEMORY! \n");
             return(-1);
               break;  }
              else  {
               rear_of_queue = rear_of_queue - 1;
               queue[rear_of_queue] = insert_item;
               }
           break;
        }
  case 3:
        {
        printf("\n Case 3... \n");
              if(rear_of_queue > 0) {
                    rear_of_queue = rear_of_queue - 1;
                    queue[rear_of_queue] = insert_item;  }
              else
              { queue[rear_of_queue] = insert_item;
                printf("\n Out of Memory! ");
                return(-1);
              }
           break;
    }
```

```
     return(1);

      }  /* End of Switch */


}  /* End of insert_queue_item */


remove_queue_item()
{
      int switchvar;
      int queue_item;

/* Example of Use:
/*      curr_item = remove_queue_item(queue, current_front, current_rear);
/*         where queue is the array representing queue,
/*               current_front is front of queue,
/*               current_rear is rear of queue.
/*

/* DO CASE  switch(type)
/*
/*    Case where in_front == in_rear
/*         When CASE is TRUE, in_front == in_rear
/*         If queue[in_front] = -1
/*             return (-1), a NULL
/*         Else
/*            If queue[in_front] != -1
/*              queue_item = queue[in_front]
/*               queue[in_front] = -1;
/*            EndIf
/*         EndIf
/*      break
/*    Case where in_front != in_rear
/*        When CASE is TRUE, in_front NOT EQUAL in_rear...
/*           If queue[in_front] = -1
/*             return (-1), a NULL
/*           Else
/*             If queue[in_front] != -1
/*                queue_item = queue[in_front]
/*                queue[in_front] = -1
/*                If in_front > 0
/*                   in_front = in_front - 1
/*                EndIf
/*             EndIf
/*           EndIf
/*
/*
/*
/*
/*                            */

     if( (front_of_queue == rear_of_queue) && (queue[front_of_queue] == -1))
       switchvar = 1;  /* If front and rear are same and queue is empty */

     if( (front_of_queue == rear_of_queue) && (queue[front_of_queue] != -1))
```

```
        switchvar = 2;  /* If front = rear and queue is NOT EMPTY */

      if( (front_of_queue != rear_of_queue) && (queue[front_of_queue] == -1))
        switchvar = 3;  /* front != rear and queue empty */

      if( (front_of_queue != rear_of_queue) && (queue[front_of_queue] != -1))
        switchvar = 4; /* front NOT EQ rear and queue NOT EMPMTY */

     switch(switchvar)
       {

          case 1:
                  {
               return(-1);
                       break;
                  }
          case 2:
                  {
               queue_item = queue[front_of_queue];
                       queue[front_of_queue] = -1;
               return(queue_item);
                       break;
                  }

          case 3:
                  {
                        return(-1);
                        printf("\n Error condition... \n");
                        break;
                  }
          case 4:
                  {
                queue_item = queue[front_of_queue];
                        queue[front_of_queue] = -1;
                        if(front_of_queue > 0)
                              front_of_queue = front_of_queue - 1;
                return(queue_item);
                        break;
                }
       } /* End switch */

      return(queue_item);

}  /* End of remove queue function */

 /* Diagnostic: Show view nodes in view list */
 show_view_nodes(VNODE *vptr)
 {
      int view_counter = 0;
      int row = 0;
      int col = 0;
      char output_filename[25] = "viewout.txt";
      FILE *file_pointer;   /* File pointer.   */

      printf("\n");
      printf("\n Opening file: viewout.txt \n");
```

```
       file_pointer = fopen(output_filename,"w");

       while(vptr != NULL)
       {
         fprintf(file_pointer,"\n******** Next View ****** \n");
         fprintf(file_pointer,"\n View ID: %15s",vptr->View_ID);
         fprintf(file_pointer,"\n Number of Archs: %4d",vptr->no_archs);
         fprintf(file_pointer,"\n ====== Architecture List =========\n");

         for(row=0; row<=4; row++)
         {
         fprintf(file_pointer,"\n Arch ID:...%2d : %25s",
         row,vptr->arch_ID[row]);
         }
         vptr = vptr->viewlink;
         view_counter = view_counter + 1;
         printf("\n");
     }
        fprintf(file_pointer,"\n No of Views: %d",view_counter);
        printf("\n -- File Done! Closing File! -- \n ");
        fclose(file_pointer);
 }


/* Inserts new arch_node of type ANODE in arch_node list. */
insert_arch_node(ANODE **ptr, char in_arch_id[25], int in_arch_style,
                      char in_comp_array[25][25],
                      char in_conn_array[25][25],
                      int in_from_to[25][3],
                      int in_no_comps, int in_no_conns)
   {
    ANODE *p1, *p2;
      char teststring[25];
      int row;
      int col;
      int test_flag;

      p1 = *ptr;

      if(p1 == NULL)   /* if list is empty */
      {
            p1 = malloc(sizeof(ANODE));
            if(p1 != NULL)
                    {
                     strcpy(p1->Arch_ID,in_arch_id);
                     p1->Style = in_arch_style;
                     p1->No_Comps = in_no_comps;
                     p1->No_Conns = in_no_conns;

                    /* Initialize the Topology Matrix */
                    for(row=0; row<=24; row++)
                     {
                      for(col=0; col<=24; col++)
                        {
                          p1->Topology[row][col] = -1;
                         }
                     }
```

```
      /* Initialize the Comp and Conn matrices */
      for(row=0; row<= in_no_comps-1; row++)
       {
         strcpy(p1->Arch_CompID[row]," ");
         strcpy(p1->Arch_ConnID[row]," ");
       }

     /* Initialize Arch comp, connector Position matrices */
        for(row=0; row<= 24; row++)
        {
         p1->Arch_CompPosition[row][0] =  0.00; /* x pos */
         p1->Arch_CompPosition[row][1] =  0.00; /* y  */
         p1->Arch_CompPosition[row][2] =  0.00; /* z  */
         p1->Arch_CompPosition[row][3] = -1.00; /* level */
         p1->Arch_ConnPosition[row][0] = 0.00;  /* x  */
         p1->Arch_ConnPosition[row][1] = 0.00;  /* y pos */
         p1->Arch_ConnPosition[row][2] = 0.00;  /* z posit */
         p1->Arch_ConnPosition[row][3] = 0.00;   /* delta */
        }

     /* Load the Component and Connector matrices */
        for(row=0; row<= in_no_comps-1; row++)
        {
          strcpy(p1->Arch_CompID[row],in_comp_array[row]);
         }

        for(row=0; row <= in_no_conns-1; row++)
        {
          strcpy(p1->Arch_ConnID[row],in_conn_array[row]);
        }

        /* Now determine the topology matrix */
        /* Topology(row,col) contains index of connector */
        /* Topology(row,col) defines connection. */
        /* row=index of source comp, col=index of dest */

        for(row=0; row <= in_no_conns-1; row++)
         p1->Topology[in_from_to[row][1]][in_from_to[row][2]]
             = in_from_to[row][0];

/* Now we generate the comp and conn positions! */

/* For Call and Return architectural style */
        if(in_arch_style == 1)
        {
          /* curr_root = 0;   */
          /* Store x = 0, y = 0, z = 0
          /* For curr comp row, use indices of non-0 conns;
          /* Insert_queue_item(connector_index);
          /*   When row is complete:  */

          /* Set the component and connector positions */
          test_flag = calculate_arch_position(&p1);

          if(test_flag == -1)
            {
```

```
                        printf("\nERROR CALCULATING POSITION! Exit!\n");
                        exit(0);
                        }
            }

                      p1->archlink = NULL;
                    *ptr = p1;
                 }
}
else    /* if list is not empty */
{
/* Scan to end of existing list for available spot to insert */
  while(p1->archlink != NULL)
      {
      /* First check if any duplicates... */
      strcpy(teststring,p1->Arch_ID);
         if(strncmp(teststring,in_arch_id,strlen(in_arch_id)) != 0)
            {
               p1 = p1->archlink;
            }
          else
            {
               printf("\n Error -- duplicate component identifier \n");
               exit(0);
            }
      }

      /* Create a brand new node for insertion */
      p2 = malloc(sizeof(ANODE));
      if(p2 != NULL)
      {
         strcpy(p2->Arch_ID,in_arch_id);
               p2->Style = in_arch_style;
               p2->No_Comps = in_no_comps;
               p2->No_Conns = in_no_conns;

             /* Initialize the Topology Matrix */
              for(row=0; row<=24; row++)
                    {
                for(col=0; col<=24; col++)
                        {
                     p2->Topology[row][col] = -1;
                        }
                    }

                     /* Initialize the Comp and Conn matrices */
                    for(row=0; row<= in_no_comps-1; row++)
                    {
                          strcpy(p2->Arch_CompID[row]," ");
                          strcpy(p2->Arch_ConnID[row]," ");
                    }


      /* Initialize the Arch component and connector Position matrices */
      for(row=0; row<= 24; row++)
      {
      p2->Arch_CompPosition[row][0] =  0.00;  /* x position */
```

```
        p2->Arch_CompPosition[row][1] =  0.00;  /* y position */
        p2->Arch_CompPosition[row][2] =  0.00;  /* z position */
        p2->Arch_CompPosition[row][3] = -1.00;  /* level number */
        p2->Arch_ConnPosition[row][0] = 0.00;   /* x position */
        p2->Arch_ConnPosition[row][1] = 0.00;   /* y position */
        p2->Arch_ConnPosition[row][2] = 0.00;   /* z position */
        p2->Arch_ConnPosition[row][3] = 0.00;   /* delta sum radians */
        }

        /* Load the Component and Connector matrices */
                     for(row=0; row<= in_no_comps-1; row++)
                     {
                      strcpy(p2->Arch_CompID[row],in_comp_array[row]);
                     }

                      for(row=0; row <= in_no_conns-1; row++)
                      {
                       strcpy(p2->Arch_ConnID[row],in_conn_array[row]);
                      }

                     /* Now determine the topology matrix */
                     /* Topology(row,col) contains index of connector */
                     for(row=0; row <= in_no_conns-1; row++)
                       p2->Topology[in_from_to[row][1]][in_from_to[row][2]]
                       = in_from_to[row][0];

              /*  ==== x,y,z and delta_theta POSITIONS AND ANGLES ==== */
              /*  Determine positions of components, connectors */
              /* curr_root = in_from_to[0][1];
              /* Get the very first component in Topology Matrix  */

                      /* p2->Arch_CompPosition[curr_root][1] = 0.00;   */
                      /* p2->Arch_CompPosition[curr_root][2] = 0.00;
                      /* p2->Arch_CompPosition[curr_root][3] = 0.00;

                      /* p2->Arch_ConnPosition[curr_root][1] = 0.0;
                      /* p2->Arch_ConnPosition[curr_root][2] =
                      /*  -Conn_Height;
                      /* p2->Arch_ConnPosition[curr_root][3] = 0.0;
                      /* p2->Arch_ConnPosition[curr_root][4] = 0.0;
                      /*
                      /*  Calculate current Comp and Conn positions; */

                    test_flag = calculate_arch_position(&p2);

                    if(test_flag == -1)
                    {
                     printf("\nERROR CALCULATING POSITION! Exiting...\n");
                       exit(0);
                      }

        /* New node points to null; last node points to new node */
              p2->archlink = NULL;
              p1->archlink = p2;
        }
    }
}
```

```
/* This routine generates positions for arch comps and conns for
/* call-and-return-style */
/* We use arch_node for architecture info and view_node for hiding/showing and
/* hyperlink info.  */

  int calculate_arch_position(ANODE **tempptr)
  {
        ANODE *archptr;  /* pointer to arch node */

        int i;  /* For loop matrix */
        int level;

        archptr = *tempptr;

        if(archptr == NULL)
              return(-1);


        curr_root = 0;
        rads60degree = PI/3;
        Conn_Height = 15.0;

        archptr->Arch_CompPosition[curr_root][0] = 0.0;
        archptr->Arch_CompPosition[curr_root][1] = 0.0;
        archptr->Arch_CompPosition[curr_root][2] = 0.0;
        /* Level of first root node follows: */
        archptr->Arch_CompPosition[curr_root][3] = 0.0;

        init_queue();  /* Initialize queue used for traversal */

        /* Insert very first root into queue */
      insert_queue_item(curr_root);

        queue_empty_flag = is_queue_empty();

        /* While the queue is not empty, proceed... */
      while(queue_empty_flag != 1)
        {
         curr_root = remove_queue_item();
             node_count = 0;

             /* Count Number of children of current root in
             /* order to determine delta_theta */

             no_components = archptr->No_Comps;

             for(i=0; i<= (no_components - 1); i++)
             {
                   if(archptr->Topology[curr_root][i] != (-1) )
                   {
                         node_count = node_count + 1;

                       /* Set the level of the node, where root is 0 */
                       if(archptr->Arch_CompPosition[i][3] = -1)
                         {
                         /* Determine level */
```

```
            archptr->Arch_CompPosition[i][3] =
              archptr->Arch_CompPosition[curr_root][3] +1;
            }
      }
}

printf("\n Current node count is: %3d ",node_count);
printf("\n");

root_x = archptr->Arch_CompPosition[curr_root][0];
root_y = archptr->Arch_CompPosition[curr_root][1];
root_z = archptr->Arch_CompPosition[curr_root][2];

if(node_count >= 1)
      delta_theta = 2.0 * PI / node_count;

if(node_count == 0)
      delta_theta = 0.0;

/* Scan through child nodes of curr_root */
child_count = 0;
delta_sum = 0.0;

for(i=0; i<= (no_components-1); i++)
{
      if(archptr->Topology[curr_root][i] != (-1) )
      {
            child_count = child_count + 1;

            delta_sum = delta_theta * (child_count - 1);
            level = archptr->Arch_CompPosition[curr_root][3];

            if(level == 0)
            {
                  Conn_Height = 15.0;
                Sphere_Radius = 0.6;
            }
            if(level == 1)
            {
                  Conn_Height = 7.5;
                Sphere_Radius = 0.6;
            }
            if(level == 2)
            {
                  Conn_Height = 3.75;
                Sphere_Radius = 0.5;
            }
            if(level == 3)
            {
                  Conn_Height = 2.9;
                Sphere_Radius = 0.4;
            }
            if(level == 4)
            {
                  Conn_Height = 1.5;
                Sphere_Radius = 0.4;
            }
```

```
                    if(level == 5)
                    {
                          Conn_Height = 1.5;
                        Sphere_Radius = 0.35;
                    }
                    if(level >= 6)
                    {
                          Conn_Height = 1.0;
                        Sphere_Radius = 0.3;
                    }


                    /* Point at origin */
                    orig_x = 0.0;
/*   orig_y = 0.0 - 0.5 times Sphere_Radius - Conn_Height;  */
                    orig_y = 0.0 - Conn_Height;
                    orig_z = 0.0;

/* Rotate around z as follows:  */
/* 1. Assume rotation occurs for object of
/*    height Conn_height + Sphere Radius
/* 2. After rotation with height occurs around z,
/*    we rotate around y by delta_sum */

             child_x = (0)*(cos(-rads60degree)) -
             (orig_y)*(sin(-rads60degree));
             child_y = (0)*(sin(-rads60degree)) +
             (orig_y)*(cos(-rads60degree));
             child_z = orig_z;

             /* Rotate around y by delta_sum */

             old_child_x = child_x;
             old_child_y = child_y;
             old_child_z = child_z;

             child_z = old_child_z * cos(delta_sum)
             - old_child_x * sin(delta_sum);
             child_x = old_child_z * sin(delta_sum)
             + old_child_x * cos(delta_sum);
             child_y = old_child_y;

            /* Translate back to position UNDER THE ORIGINAL ROOT */
              child_x = child_x + root_x;
                        child_y = child_y + root_y;
                        child_z = child_z + root_z;

             /* Store Arch Positions and Level */

                archptr->Arch_CompPosition[i][0] = child_x;
                archptr->Arch_CompPosition[i][1] = child_y;
                archptr->Arch_CompPosition[i][2] = child_z;

             /* Now establish connection given the root position and the
             /* child position */
```

```
                /* First Translate to the origin */
                    conn_orig_x = 0.0;
                /*   conn_orig_y = (0.0 -Conn_Height)/2.0-Sphere_Radius;  */
                     conn_orig_y = (0.0 - Conn_Height)/2.0;
                     conn_orig_z = 0.0;

                /* First Rotate the leg center-point around z-axis */
                      conn_x = (0) * (cos(-rads60degree)) -
                               (conn_orig_y)*(sin(-rads60degree));
                       conn_y = (0) * (sin(-rads60degree)) +
                                (conn_orig_y)*(cos(-rads60degree));
                       conn_z = conn_orig_z;

                /* Now rotate the leg center-point around the y-axis */
                /* Rotate around y by delta_sum */

                            old_conn_x = conn_x;
                            old_conn_y = conn_y;
                            old_conn_z = conn_z;

                /* Translate back to location under current root node */
                            conn_x = conn_x + root_x;
                            conn_y = conn_y + root_y;
                            conn_z = conn_z + root_z;

                       conn_id = archptr->Topology[curr_root][i];

                       archptr->Arch_ConnPosition[conn_id][0] = conn_x;
                       archptr->Arch_ConnPosition[conn_id][1] = conn_y;
                       archptr->Arch_ConnPosition[conn_id][2] = conn_z;
                      /* rotate around z */
                       archptr->Arch_ConnPosition[conn_id][3] = delta_sum;

                       insert_queue_item(i);

                 }  /* end if Topology not -1 */

           }  /* end for i <= no_comps */

            queue_empty_flag = is_queue_empty();

       }  /* end while queue not empty */

 }   /* end calculate_arch_position */


 /* This routine is used for diagnostic purposes;
 /* It prints the selected contents of each arch_node */

show_arch_nodes(ANODE *ptr)
{
      int arch_counter = 0;
      int row = 0;
      int col = 0;
      printf("\n");
```

```
      while(ptr != NULL)
      {
            printf("******* Next Architecture ********* \n");
            printf("Arch ID: %s",ptr->Arch_ID);
            printf("\n ====== Style: %d",ptr->Style);
            printf("\n ====== No_Comps: %d",ptr->No_Comps);
            printf("\n ====== No_Conns: %d",ptr->No_Conns);
            printf("\n ====== Component List =========");

            for(row=0; row<=8; row++)
            {
              printf("\n........%d : %s",row,ptr->Arch_CompID[row]);
            }

            printf("\n ====== Connectors =============");

            for(row=0; row<=7; row++)
            {
              printf("\n........%d : %s",row,ptr->Arch_ConnID[row]);
            }

            printf("\n ====== (TOPOLOGY) =============");

        for(row=0; row<=8; row++)
            {
                 printf("\n");
                 for(col=0; col<=8; col++)
                 {
               printf(" %d ",ptr->Topology[row][col]);
                 }
            }

            ptr = ptr->archlink;
            arch_counter = arch_counter + 1;
            printf("\n");
    }
      printf("\n No of Archs: %d",arch_counter);
}

/* Search for a view node; return 1 if found, 0 if not found */
 int search_view_list(VNODE *vvptr, char in_view_id[25])
 {
       char teststring[25];

       if(vvptr == NULL)
             return(0);
       else
       {
              do
              {
                    strcpy(teststring,vvptr->View_ID);
                    if(strncmp(teststring,in_view_id,strlen(in_view_id)) == 0)
                          return(1);
                    vvptr = vvptr->viewlink;
              } while(vvptr != NULL);
              return(0);
       }
```

```
  }

   /*  Insert the view Node into View-List */

  void insert_view_node(VNODE **vptr, char in_view_id[25],
                              char in_arch_id[25][25], int in_all_comps[25],
                              int in_all_conns[25], int in_viewcomps[25][25],
                              int in_viewconns[25][25], char
in_hyper_comps[625][25],
                              char in_hyper_conns[625][25], int in_no_archs)

   {
    VNODE *vp1, *vp2;
      char teststring[25];
      int row;
      int col;

      vp1 = *vptr;

      if(vp1 == NULL)   /* if list is empty */
      {
            vp1 = malloc(sizeof(VNODE));
            if(vp1 != NULL)
            {
             strcpy(vp1->View_ID,in_view_id);

             /* Initialize the various arrays: */
             vp1->no_archs = 0;


             for(row=0; row<=24; row++)
              {
               vp1->all_comps[row] = 0;    /* All comps flags to 0 */
               vp1->all_conns[row] = 0;    /* All conns flags to 0 */

               strcpy(vp1->arch_ID[row],"         ");

               for(col=0; col<=24; col++)
                {
                  /* strcpy(vp1->arch_ID[row][col]," "); */
                  vp1->view_comps[row][col] = 0;
                  vp1->view_conns[row][col] = 0;
                }
              }

              /* Initialize the Hyperlink arrays */
              for(col=0; col<=624; col++)
              {
                strcpy(vp1->Hyperlinks_Comps[col]," ");
                strcpy(vp1->Hyperlinks_Conns[col]," ");
              }


              vp1->no_archs = in_no_archs;

              for(col=0; col<=24; col++)
              {
```

```
          strcpy(vp1->arch_ID[col],in_arch_id[col]);
          vp1->all_comps[col] = in_all_comps[col];
          vp1->all_conns[col] = in_all_conns[col];

             for(row=0; row<=24; row++)
             {
               vp1->view_comps[col][row] = in_viewcomps[col][row];
               vp1->view_conns[col][row] = in_viewconns[col][row];
             }
          }

                   for(row=0; row<=624; row++)
                   {
                      for(col=0; col<=24; col++)
                      {
                        vp1->Hyperlinks_Comps[row][col] =
                         in_hyper_comps[row][col];
                        vp1->Hyperlinks_Conns[row][col] =
                         in_hyper_conns[row][col];
                       }
                   }

             vp1->viewlink = NULL;
             *vptr = vp1;
       }
}
else    /* if list is not empty */
{
  /* Scan to end of existing list for available spot to insert */
  while(vp1->viewlink != NULL)
       {
           /* First check if any duplicates... */
           strcpy(teststring,vp1->View_ID);
           if(strncmp(teststring,in_view_id,strlen(in_view_id)) != 0)
             {
               vp1 = vp1->viewlink;
             }
           else
             {
                printf("\n Error -- duplicate component identifier \n");
                exit(0);
             }
       }

       /* Create a brand new node for insertion */
       vp2 = malloc(sizeof(VNODE));
       if(vp2 != NULL)
       {
          strcpy(vp2->View_ID,in_view_id);

          /* Initialize the variables and arrays */
                vp2->no_archs = 0;

                for(row=0; row<=24; row++)
                {
                      vp2->all_comps[row] = 0;  /* All comps flags to 0 */
```

```
                    vp2->all_conns[row] = 0;  /* All conns flags to 0 */

                    strcpy(vp2->arch_ID[row],"           ");

                    for(col=0; col<=24; col++)
                    {
                        /* strcpy(vp2->arch_ID[row][col]," "); */
                          vp2->view_comps[row][col] = 0;
                          vp2->view_conns[row][col] = 0;
                    }
              }

              /* Initialize the Hyperlink arrays */
              for(col=0; col<=624; col++)
              {
                   strcpy(vp2->Hyperlinks_Comps[col]," ");
                   strcpy(vp2->Hyperlinks_Conns[col]," ");
              }

              vp2->no_archs = in_no_archs;

              for(col=0; col<=24; col++)
              {
                    strcpy(vp2->arch_ID[col], in_arch_id[col]);
                    vp2->all_comps[col] = in_all_comps[col];
                    vp2->all_conns[col] = in_all_conns[col];

                    for(row=0; row<=24; row++)
                    {
                       vp2->view_comps[col][row] =
                       in_viewcomps[col][row];
                       vp2->view_conns[col][row] =
                       in_viewconns[col][row];
                    }
              }

              for(row=0; row<=624; row++)
              {
                    for(col=0; col<=24; col++)
                    {
                      vp2->Hyperlinks_Comps[row][col] =
                       in_hyper_comps[row][col];
                      vp2->Hyperlinks_Conns[row][col] =
                       in_hyper_conns[row][col];
                    }
              }

         /* New node points to null; last node points to new node */
               vp2->viewlink = NULL;
               vp1->viewlink = vp2;
         }
     }
 }


 find_comp_index( char test_comp[25], char in_comp_array[25][25], int
number_of_comps)
```

```
 {
   int i;
   int result_index;

   result_index = -1;

   for(i=0; i<=(number_of_comps -1); i++)
   {
       if(strncmp(in_comp_array[i],test_comp,strlen(test_comp)) == 0)
           result_index = i;
   }

       return(result_index);

 }


 find_conn_index( char test_conn[25], char in_conn_array[25][25], int
number_of_conns)
 {
   int i;
   int result_index;

   result_index = -1;

   for(i=0; i<=(number_of_conns -1); i++)
   {
       if(strncmp(in_conn_array[i],test_conn,strlen(test_conn)) == 0)
           result_index = i;
   }
       return(result_index);
 }



  /* The following routine generates the VRML view file for each view.
  /* It is the target file for the view description in each node.
  /* That is, each view node contains one or more architectures
  /* which are rendered in the VRML file.
  /*
  /* Sample call:   generate_view_files(ANODE in_arch, VNODE in_view);
  /*
  /*  1.  The viewlist VNODES are traversed, from first to last view node;
  /*  2.  For each view node:
  /*      Prepare file by first preparing the navigation toolbar
  /*      Establish anchor nodes for ALL OTHER views using
  /*      visual navigation icon.
  /*      The navigation icon consists of a sphere for all wrl files,
  /*      excluding the current view.
  /*      Use the view_nodes array for the list of nodes other
  /*      than current view node.
  /*      Generate the VRML codes for anchor nodes, translate
  /*      to the lower portion below the architectures.
  /*         View Name = View Name + ".wrl" extension.
  /*
  /*     3.  For the current view node:
  /*
```

```
  /*         3A.  Read the architecture array for arch ID
  /*         3B.  Search the arch list for arch node.
  /*         3C.  Determine height, H, of the arch from root to longest leaf.
  /*         3D.  Translate the architecture according to the algorithm:
  /*              If architecture 1, Z-Center is at 0.0.  Render.
  /*              If architecture > 1:
  /*                 If architecture number is even:
  /*                  new Z-Center is at old Z-Center -
  /*                  (1/2)* height of current arch n.
  /*                  new y-center is old y-center +
  /*                  (1/2) * height of earlier arch.
  /*                   Rotate architecture by 90 degrees around x-axis.
  /*                 If architecture number is odd:
  /*                 new Z-Center is at old Z-Center -
  /*                  (1/2) * height previous arch.
  /*                 new y-center is old y-center + (1/2) * height current
arch.
  /*                     Do not rotate around x-axis.
  /*                 NOTE:  In this fashion we create a
  /*                 staircase, orthogonal representation of multiple
  /*                 architectures within one view.
  /*
  /*    4.   For each arch node within the current view node:
  /*         Access the topological matrix and begin traversal
  /*         for call-and-return style.
  /*
  /*      Use traversal technique used earlier in calculate_arch_position( ).
  /*
/*         For the component:
/*            Get the position of the component.
/*          If All_flag, prepare-to-render is true.
/*          If not All_flag, but view_comps set to 1, prepare-to-render is
true.
/*            If prepare-to-render is true:
/*                Check hyperlink_comp array for component-id.
/*                    If found:  Render component at position as Anchor Node.
/*                    If not found: Render component as transform node sphere.
/*            If prepare-to-render is false, ignore, continue to next
component.
/*         For the connection:
/*             Get the position of the component.
/*             Rotate around z by -45 degrees.
/*             Rotate around y by delta_sum given.
/*             If All_flag, prepare-to-render is true.
/*             If not All_flag, but view_conns set to 1,
/*             prepare-to-render is true.
/*             If prepare-to-render is true:
/*                Check hyperlink_conn array for connector-id.
/*                 If found: Reander connector at position as Anchor Node.
/*                 If not found: Render connector as transform node cylinder.
/*             If prepare-to-render is false, ignore,
/*              continue to next connector.
/*
/*          Repeat for each arch node.
/*          Generate the full VRML file.
/*
/*
```

```
/*
/*     5.  Get the next view node and repeat for the view list until empty.
/*
/*
/*       */

  generate_view_files(ANODE *inarchptr, VNODE *inviewptr)
  {

        int prepare_to_render = 1;    /* Flag for rendering; 1=true, 0=false;
*/
        int search_result = 1;
        int number_of_archs = 0;
        int i;
        int item_located;
        float layer_height = 0.0;
        int number_of_layers;
        int row;
        int col;
        int level;
        int temp_arch_style;
        int temp_view_pointer = 0;  /* used as view_names_array index */

        float scale_value = 0.0;    /* variable used for VRML scale command */
        float position_value = 0.0; /* variable used for VRML cone position */
        float title_value = 0.0;    /* variable used for VRML title position */
        char quotemark = '"';

        float font_size = 0.0;
        float z_arch_displace = 0.0;
        float x_arch_displace = 0.0;
        float y_arch_displace = 0.0;
        float temp_y_legend = 0.0; /* used for legend title placement along y
*/
        float view_x;  /* viewpoint x position */
        float view_y;  /* viewpoint y position */
        float view_z;  /* viewpoint z position */

        char output_filename[25];
        char output_extension[25];
        char arch_test_string[25];
        char temp_view_name[25];

      /* print_view_string used for printing text string in
      /* Text node of VRML. */
      /* Contains a quote, text, quote */
        char print_view_string[27];  /* String contains quote View_ID quote */

        char temp_arch_id[25];
        char temp_view_id[25];

        FILE *file_pointer;   /* File pointer.   */

        ANODE *rtemp;         /* Reference node for architecture */

        /* Traverse the viewnodes from start */
        while(inviewptr != NULL)
```

```
{
    strcpy(output_filename," ");
    strcpy(output_extension,".wrl");
    strcpy(output_filename,inviewptr->View_ID);
    strcat(output_filename,output_extension);
    file_pointer = fopen(output_filename,"w");

    number_of_archs = inviewptr->no_archs;

    x_arch_displace = 0.0;  /* Init arch displacement within view */
    y_arch_displace = 0.0;
    z_arch_displace = 0.0;

    printf("\n Writing file: %s \n",output_filename);

/***    Print View File VRML Heading   ***/
fprintf(file_pointer,"#VRML V2.0 utf8");
fprintf(file_pointer,"\n#Viewpoint of Architectures");
fprintf(file_pointer,"\n#Generated by VTADL Version 1.0");

fprintf(file_pointer,"\nBackground {");
fprintf(file_pointer,"\n    skyColor [");
fprintf(file_pointer,"\n       0.0 0.2 0.91,");
fprintf(file_pointer,"\n       0.0 0.3 1.0,");
fprintf(file_pointer,"\n       0.3 0.2 0.85");
fprintf(file_pointer,"\n    ]");
fprintf(file_pointer,"\nskyAngle [ 1.309, 1.571 ]");
fprintf(file_pointer,"\n}");

printf("\n Wrote Background... \n");

fprintf(file_pointer,"\nGroup {");
fprintf(file_pointer,"\nchildren [");

/* Print the View Title at Top of wrl file */
fprintf(file_pointer,"\n Transform { ");
fprintf(file_pointer,"\n    translation 0.0 4.00 0.0");
fprintf(file_pointer,"\n    children [");
fprintf(file_pointer,"\n      Shape {");
fprintf(file_pointer,"\n       appearance Appearance {");
fprintf(file_pointer,"\n        material
  Material { diffuseColor 1.0 1.0 0.0 }");
fprintf(file_pointer,"\n        }");
fprintf(file_pointer,"\n        geometry Text {");
fprintf(file_pointer,"\n          string \"%s\" ",inviewptr->View_ID);
fprintf(file_pointer,"\n          fontStyle FontStyle {");
fprintf(file_pointer,"\n             style \"BOLD\" ");
fprintf(file_pointer,"\n             justify \"MIDDLE\" ");
fprintf(file_pointer,"\n             size 0.94");
fprintf(file_pointer,"\n } } } ] },");

/* Generate the view legend to the right of the first arch */
fprintf(file_pointer,"\n Transform { ");
fprintf(file_pointer,"\n   translation 27.1 6.5 0.0");
fprintf(file_pointer,"\n    children[");
fprintf(file_pointer,"\n      Shape {");
fprintf(file_pointer,"\n       appearance Appearance {");
```

```
 fprintf(file_pointer,"\n        material
   Material { diffuseColor 1.0 1.0 0.0 }");
 fprintf(file_pointer,"\n        }");
 fprintf(file_pointer,"\n        geometry Text {");
 fprintf(file_pointer,"\n          string \" LEGEND \" ");
 fprintf(file_pointer,"\n          fontStyle FontStyle {");
 fprintf(file_pointer,"\n             style \"BOLD\" ");
 fprintf(file_pointer,"\n             justify \"BEGIN\" ");
 fprintf(file_pointer,"\n             size 0.94");
 fprintf(file_pointer,"\n } } } ] },");

 /* Generate the legend viewpoint for VRML view list */
 fprintf(file_pointer,"\nViewpoint {");
 fprintf(file_pointer,"\ndescription \"Legend\" ");
 fprintf(file_pointer,"\nposition 25.4 4.75 21.5 }");

for(temp_view_pointer=0;
 temp_view_pointer<= (curr_view_index-1); temp_view_pointer++)
 {
  printf("\n temp_view_pointer %d \n",temp_view_pointer);
  fprintf(file_pointer,"\n Transform { ");
  temp_y_legend = 4.0 - 1.95*temp_view_pointer;
  fprintf(file_pointer,
   "\n    translation 20.0 %6.3f 0.0 ",temp_y_legend);
  fprintf(file_pointer,"\n    children [");
  fprintf(file_pointer,"\n      Shape {");
  fprintf(file_pointer,"\n       appearance Appearance {");
  fprintf(file_pointer,
    "\n        material Material { diffuseColor 1.0 1.0 0.0 }");
  fprintf(file_pointer,"\n        }");
  fprintf(file_pointer,"\n        geometry Text {");
  fprintf(file_pointer,
     "\n          string \"%s\" ",view_names_array[temp_view_pointer]);
  fprintf(file_pointer,"\n          fontStyle FontStyle {");
  fprintf(file_pointer,"\n             style \"BOLD\" ");
  fprintf(file_pointer,"\n             justify \"BEGIN\" ");
  fprintf(file_pointer,"\n             size 0.94");
  fprintf(file_pointer,"\n } } } ] },");

  fprintf(file_pointer,"\n Transform { ");
  /* fprintf(file_pointer,"\n    translation 15.0 4.00 0.0");  */

  temp_y_legend = temp_y_legend + 0.35;
  fprintf(file_pointer,
   "\n    translation 29.2 %6.3f 0.0 ",temp_y_legend);
  fprintf(file_pointer,"\n    children [");


  if(strncmp(view_names_array[temp_view_pointer],inviewptr->View_ID,
     strlen(inviewptr->View_ID)) != 0)
     {
     strcpy(temp_view_name,view_names_array[temp_view_pointer]);
     strcat(temp_view_name,".wrl");
     fprintf(file_pointer,"\nAnchor {");
     fprintf(file_pointer,"\nurl \"%s\" ",temp_view_name);
     fprintf(file_pointer,"\n children [");
     fprintf(file_pointer,"\n    Shape  {");
```

```
      fprintf(file_pointer,"\n    appearance Appearance {");
      fprintf(file_pointer,"\n     material Material {");
      fprintf(file_pointer,"\n       diffuseColor 1.0 1.0 0.0");
      fprintf(file_pointer,"\n          }");
      fprintf(file_pointer,"\n    }");
      fprintf(file_pointer,"\n        geometry Sphere { ");
      fprintf(file_pointer,"\n        radius 0.62");
      fprintf(file_pointer,"\n       }");
      fprintf(file_pointer,"\n  }");
      fprintf(file_pointer,"\n  ]");
      fprintf(file_pointer,"\n  }");
      }
    fprintf(file_pointer,"\n ] },");

  }  /* end for */

  for(i=0; i<= (number_of_archs - 1); i++)
  {
    strcpy(temp_arch_id,inviewptr->arch_ID[i]);

    /* Get the architecture node from arch list */
    rtemp = inarchptr;

    /* search rtemp arch list node for arch ID. */
    if(rtemp == NULL)
    {
      fprintf(file_pointer,"\n Arch list empty!  ERROR...exiting... \n");
      exit(0);
    }

    if(rtemp != NULL)
     {
      item_located = 0;

      do
      {
      strcpy(arch_test_string,rtemp->Arch_ID);
      if(strncmp(arch_test_string,temp_arch_id,strlen(temp_arch_id))==0)
      {
      item_located = 1;

        /* Print the Architectural Heading */

        if(i > 0)
        {
         x_arch_displace = x_arch_displace - 25.5;
         z_arch_displace = z_arch_displace - 50.0;
         y_arch_displace = y_arch_displace + 15.0;

         fprintf(file_pointer,"\n Transform { ");
         fprintf(file_pointer,"\n   translation %6.3f %6.3f %6.3f ",
                 x_arch_displace,y_arch_displace,z_arch_displace);
         fprintf(file_pointer,"\n   children [");
         fprintf(file_pointer,"\n   Group {");
         fprintf(file_pointer,"\n   children [");

        }
```

```
 fprintf(file_pointer,"\n Transform { ");
 fprintf(file_pointer,"\n    translation 0.0 3.00 0.0");
 fprintf(file_pointer,"\n    children [");

 /* Generate the  viewport description for VRML view list */
 fprintf(file_pointer,"\nViewpoint {");
 fprintf(file_pointer,"\ndescription \"%s\" ",rtemp->Arch_ID);
 fprintf(file_pointer,"\nposition 0.0 -5.0 21.5 },");

 fprintf(file_pointer,"\n      Shape {");
 fprintf(file_pointer,"\n       appearance Appearance {");
 fprintf(file_pointer,"\n        material
  Material { diffuseColor 1.0 1.0 0.0 }");
 fprintf(file_pointer,"\n        }");
 fprintf(file_pointer,"\n        geometry Text {");
 fprintf(file_pointer,
  "\n          string \" Architecture %s\" ",rtemp->Arch_ID);
 fprintf(file_pointer,"\n          fontStyle FontStyle {");
 fprintf(file_pointer,"\n             style \"BOLD\" ");
 fprintf(file_pointer,"\n             justify \"MIDDLE\" ");
 fprintf(file_pointer,"\n             size 0.65");
 fprintf(file_pointer,"\n } } } ] },");

 /* Check for Layered Style */
 if(rtemp->Style == 2)
 {
  number_of_layers = rtemp->No_Comps;
  layer_height = (0.6)*(2.0*number_of_layers-1.0)*4.0 + 1.5;

  fprintf(file_pointer,"\nTransform {");
  fprintf(file_pointer,"\n  scale 0.5 0.5 0.5 ");
  fprintf(file_pointer,
   "\n  translation 0.0 %6.3f -6.0",-layer_height);
  fprintf(file_pointer,"\n    children [");
  fprintf(file_pointer,"\nGroup {");
  fprintf(file_pointer,"\n    children [");
 }

/* Traverse the nodes from root */

 curr_root = 0;   /* Start with the very first row */

 init_queue();    /* Initialize queue for node traversal */

 insert_queue_item(curr_root);   /* start with very first root */

 queue_empty_flag = is_queue_empty();

 /* Loop for non empty queue */

 while( queue_empty_flag != 1)
 {
  curr_root = remove_queue_item();   /* Get next root */

  /* Get x,y,z coordinate position of current node */
  root_x = rtemp->Arch_CompPosition[curr_root][0];
```

```
root_y = rtemp->Arch_CompPosition[curr_root][1];
root_z = rtemp->Arch_CompPosition[curr_root][2];
level = rtemp->Arch_CompPosition[curr_root][3];

temp_arch_style = rtemp->Style;
number_of_layers = rtemp->No_Comps;

/* Now Generate Comp Nodes using positions */

/* If arch style is Call and return */
if(temp_arch_style == 1)
 {
 /* Print Initial Transform Heading for Comp  */

 fprintf(file_pointer,"\nTransform {");
 fprintf(file_pointer,"\n  translation %6.3f %6.3f %6.3f",
         root_x,root_y,root_z);
 fprintf(file_pointer,"\n    children [");

if(level == 0)   /* If node is the root */
{
 Sphere_Radius = 0.60;

/* Check to see if Component is viewed at all... */
if((inviewptr->all_comps[i]==1) ||
   (inviewptr->all_comps[i]==0 &&
    inviewptr->view_comps[i][curr_root]==1))
{

 /* If viewed, then handle the case of hyperlink url */
 if(strncmp( (inviewptr->Hyperlinks_Comps[25*i
   + curr_root])," ",1) != 0)
  {
   fprintf(file_pointer,"\nAnchor {");
   fprintf(file_pointer,"\nurl \"%s\"  ",
    inviewptr->Hyperlinks_Comps[25*i + curr_root]);
   fprintf(file_pointer,"\n children [");
  }

 }  /* End if: check for comp to be viewed */

   fprintf(file_pointer,"\n    DEF Root_Node_Type Shape  {");
   fprintf(file_pointer,"\n    appearance Appearance {");
   fprintf(file_pointer,"\n     material Material {");
   fprintf(file_pointer,"\n       diffuseColor 0.1 0.99 0.99");

  /* Check if component is visible or not... */

   if((inviewptr->all_comps[i]==1) ||
     (inviewptr->all_comps[i]==0
     && inviewptr->view_comps[i][curr_root]==1))
   {
    fprintf(file_pointer,"\n      transparency 0.0 }");
   }
    else
   {
    fprintf(file_pointer,"\n      transparency 1.0 }");
```

```
 }
  fprintf(file_pointer,"\n    }");
  fprintf(file_pointer,"\n        geometry Sphere { ");
  fprintf(file_pointer,
   "\n        radius %6.3f ",Sphere_Radius);
  fprintf(file_pointer,"\n       }");
  /* fprintf(file_pointer,"\n    },"); */

  /* Check if Hyperlink was viewed for tail */
  if((inviewptr->all_comps[i]==1) ||
    (inviewptr->all_comps[i]==0
    && inviewptr->view_comps[i][curr_root]==1))
  {

  /* Handle alternate tails in case of hyperlink */
  if(strncmp( (inviewptr->Hyperlinks_Comps[25*i
  + curr_root])," ",1) != 0)
  {
   fprintf(file_pointer,"\n     }");
   fprintf(file_pointer,"\n ]");
   fprintf(file_pointer,"\n },");
   }
    else
   {
    fprintf(file_pointer,"\n     }, ");
   }

}

else  /* if hyperlink not viewed */
{
fprintf(file_pointer,"\n     }, ");
} /* End check if viewed hyperlink */

fprintf(file_pointer,"\n      Transform { ");
fprintf(file_pointer,"\n        translation -0.55 0.75 0.0");
fprintf(file_pointer,"\n        children [");
fprintf(file_pointer,"\n        Shape {");
fprintf(file_pointer,"\n       appearance Appearance {");
fprintf(file_pointer,"\n        material
 Material { diffuseColor 1.0 1.0 0.0 ");

/* Check if text is viewed */
if((inviewptr->all_comps[i]==1) ||
(inviewptr->all_comps[i]==0 &&
inviewptr->view_comps[i][curr_root]==1))
{
 fprintf(file_pointer,
 "\n                           transparency 0.0 }");
}
 else
{
 fprintf(file_pointer,
 "\n                            transparency 1.0 }");
}
```

```
fprintf(file_pointer,"\n        }");
fprintf(file_pointer,"\n      geometry Text {");
fprintf(file_pointer,"\n        string \"%s\" ",
 rtemp->Arch_CompID[curr_root]);
fprintf(file_pointer,"\n        fontStyle FontStyle {");
fprintf(file_pointer,"\n             style \"BOLD\" ");
fprintf(file_pointer,"\n             justify \"MIDDLE\" ");
fprintf(file_pointer,"\n             size 0.6");
fprintf(file_pointer,"\n } } } ] }");
fprintf(file_pointer,"\n ]");
fprintf(file_pointer,"\n },");
}


if(level == 1)  /* If node is first level */
{
 Sphere_Radius = 0.6;

/* Check to see if Component is viewed at all... */
if((inviewptr->all_comps[i]==1) ||
(inviewptr->all_comps[i]==0 &&
inviewptr->view_comps[i][curr_root]==1))
{

 /* If viewed, then handle the case of hyperlink url */
 if(strncmp( (inviewptr->Hyperlinks_Comps[25*i
 + curr_root])," ",1) != 0)
 {
 fprintf(file_pointer,"\nAnchor {");
 fprintf(file_pointer,"\nurl \"%s\"  ",
 inviewptr->Hyperlinks_Comps[25*i + curr_root]);
 fprintf(file_pointer,"\n children [");
 }

}  /* End if: check for comp to be viewed */

 fprintf(file_pointer,"\n    Shape  {");
 fprintf(file_pointer,"\n    appearance Appearance {");
 fprintf(file_pointer,"\n     material Material {");
 fprintf(file_pointer,"\n       diffuseColor 1.0 0.0 0.0");

 /* Check if component is visible or not... */
 if((inviewptr->all_comps[i]==1) ||
 (inviewptr->all_comps[i]==0 &&
 inviewptr->view_comps[i][curr_root]==1))
 {
  fprintf(file_pointer,"\n      transparency 0.0 }");
 }
  else
 {
  fprintf(file_pointer,"\n      transparency 1.0 }");
 }

  fprintf(file_pointer,"\n    }");   /* end Appearance */
  fprintf(file_pointer,"\n        geometry Sphere { ");
  fprintf(file_pointer,"\n        radius
   %6.3f ",Sphere_Radius);
```

```
 fprintf(file_pointer,"\n       }");
/* fprintf(file_pointer,"\n  },"); */

/* Check if Hyperlink was viewed for tail */
if((inviewptr->all_comps[i]==1) ||
(inviewptr->all_comps[i]==0 &&
 inviewptr->view_comps[i][curr_root]==1))
{

 /* Handle alternate tails in case of hyperlink */
 if(strncmp( (inviewptr->Hyperlinks_Comps[25*i
 + curr_root])," ",1) != 0)
 {
  fprintf(file_pointer,"\n     }");
  fprintf(file_pointer,"\n ]");
  fprintf(file_pointer,"\n },");
 }
  else
 {
  fprintf(file_pointer,"\n     }, ");
 }

}
else  /* if hyperlink not viewed */
{
fprintf(file_pointer,"\n     }, ");
} /* End check if viewed hyperlink */


fprintf(file_pointer,"\n      Transform { ");
fprintf(file_pointer,"\n        translation -0.55 0.65 0.6");
fprintf(file_pointer,"\n        children [");
fprintf(file_pointer,"\n        Shape {");
fprintf(file_pointer,"\n       appearance Appearance {");
fprintf(file_pointer,"\n        material
 Material { diffuseColor 1.0 1.0 0.0 ");

/* Check if text is viewed */
if((inviewptr->all_comps[i]==1) ||
(inviewptr->all_comps[i]==0 &&
 inviewptr->view_comps[i][curr_root]==1))
{
 fprintf(file_pointer,
  "\n                            transparency 0.0 }");
}
 else
{
 fprintf(file_pointer,
   "\n                            transparency 1.0 }");
}

 fprintf(file_pointer,"\n        }");
 fprintf(file_pointer,"\n        geometry Text {");
 fprintf(file_pointer,"\n          string \"%s\" ",
  rtemp->Arch_CompID[curr_root]);
 fprintf(file_pointer,"\n          fontStyle FontStyle {");
 fprintf(file_pointer,"\n             style \"BOLD\" ");
```

```
 fprintf(file_pointer,"\n             justify \"MIDDLE\" ");
 fprintf(file_pointer,"\n             size 0.43");
 fprintf(file_pointer,"\n } } } ] }");
 fprintf(file_pointer,"\n ]");
 fprintf(file_pointer,"\n },");
}


if(level == 2)  /* If node is second level */
{
Sphere_Radius = 0.5;

/* Check to see if Component is viewed at all... */
if((inviewptr->all_comps[i]==1) ||
 (inviewptr->all_comps[i]==0 &&
 inviewptr->view_comps[i][curr_root]==1))
{

/* If viewed, then handle the case of hyperlink url */
if(strncmp( (inviewptr->Hyperlinks_Comps[25*i
 + curr_root])," ",1) != 0)
{
 fprintf(file_pointer,"\nAnchor {");
 fprintf(file_pointer,"\nurl \"%s\"  ",
  inviewptr->Hyperlinks_Comps[25*i + curr_root]);
 fprintf(file_pointer,"\n children [");
}

}  /* End if check for comp to view */

 fprintf(file_pointer,"\n    Shape  {");
 fprintf(file_pointer,"\n    appearance Appearance {");
 fprintf(file_pointer,"\n     material Material {");
 fprintf(file_pointer,"\n       diffuseColor 0.0 1.0 0.0");

/* Check if component is visible or not... */
if((inviewptr->all_comps[i]==1) ||
 (inviewptr->all_comps[i]==0 &&
  inviewptr->view_comps[i][curr_root]==1))
{
 fprintf(file_pointer,"\n      transparency 0.0 }");
}
 else
{
 fprintf(file_pointer,"\n      transparency 1.0 }");
}

 fprintf(file_pointer,"\n    }");  /* end Appearance */
 fprintf(file_pointer,"\n        geometry Sphere { ");
 fprintf(file_pointer,"\n        radius
  %6.3f ",Sphere_Radius);
 fprintf(file_pointer,"\n       }");
 /* fprintf(file_pointer,"\n  },"); */

 /* Check if Hyperlink was viewed for tail */
 if((inviewptr->all_comps[i]==1) ||
 (inviewptr->all_comps[i]==0 &&
```

```
    inviewptr->view_comps[i][curr_root]==1))
   {

    /* Handle alternate tails in case of hyperlink */
    if(strncmp( (inviewptr->Hyperlinks_Comps[25*i
    + curr_root])," ",1) != 0)
    {
     fprintf(file_pointer,"\n     }");
     fprintf(file_pointer,"\n ]");
     fprintf(file_pointer,"\n },");
    }
     else
    {
     fprintf(file_pointer,"\n     }, ");
    }

  }

  else  /* if hyperlink not viewed */
{
  fprintf(file_pointer,"\n     }, ");
} /* End check if viewed hyperlink */


  fprintf(file_pointer,"\n      Transform { ");
  fprintf(file_pointer,"\n        translation -0.55 0.55 0.5");
  fprintf(file_pointer,"\n        children [");
  fprintf(file_pointer,"\n        Shape {");
  fprintf(file_pointer,"\n       appearance Appearance {");
  fprintf(file_pointer,"\n        material
   Material { diffuseColor 1.0 1.0 0.0");

/* Check if text is viewed */
if((inviewptr->all_comps[i]==1) ||
 (inviewptr->all_comps[i]==0 &&
  inviewptr->view_comps[i][curr_root]==1))
{
 fprintf(file_pointer,
 "\n                            transparency 0.0 }");
}
 else
{
 fprintf(file_pointer,
 "\n                            transparency 1.0 }");
}

fprintf(file_pointer,"\n        }");  /* end Appearance */
fprintf(file_pointer,"\n        geometry Text {");
fprintf(file_pointer,"\n          string \"%s\" ",
rtemp->Arch_CompID[curr_root]);
fprintf(file_pointer,"\n          fontStyle FontStyle {");
fprintf(file_pointer,"\n             style \"BOLD\" ");
fprintf(file_pointer,"\n             justify \"MIDDLE\" ");
fprintf(file_pointer,"\n             size 0.33");
fprintf(file_pointer,"\n } } } ] }");
fprintf(file_pointer,"\n ]");
fprintf(file_pointer,"\n },");
```

```
                }

/* Levels 3 to 9 were removed here; code follows similar pattern
/* to earlier code, with difference that Sphere_Radius and node
/* color are different for each level */
/* ..... */
/* ....  */
/* ....  */
/ *   We continue with level 10 */

                    if(level == 10)  /* If node is tenth level */
                     {
                      Sphere_Radius  = 0.25;

                     /* Check to see if Component is viewed at all... */
                     if((inviewptr->all_comps[i]==1) ||
                       (inviewptr->all_comps[i]==0 &&
                        inviewptr->view_comps[i][curr_root]==1))
                     {

                     /* If viewed, then handle the case of hyperlink url */
                     if(strncmp( (inviewptr->Hyperlinks_Comps[25*i
                      + curr_root])," ",1) != 0)
                     {
                     fprintf(file_pointer,"\nAnchor {");
                     fprintf(file_pointer,"\nurl \"%s\"  ",
                      inviewptr->Hyperlinks_Comps[25*i+curr_root]);
                     fprintf(file_pointer,"\n children [");
                     }

                   }  /* End if check for comp to view */

                   fprintf(file_pointer,"\n    Shape  {");
                   fprintf(file_pointer,"\n    appearance Appearance {");
                   fprintf(file_pointer,"\n     material Material {");
                   fprintf(file_pointer,"\n       diffuseColor 0.0 1.0 0.0");

                   /* Check if Sphere is viewed */
                   if((inviewptr->all_comps[i]==1) ||
                    (inviewptr->all_comps[i]==0 &&
                     inviewptr->view_comps[i][curr_root]==1))
                   {
                    fprintf(file_pointer,
                     "\n                            transparency 0.0 }");
                   }
                   else
                   {
                    fprintf(file_pointer,
                     "\n                            transparency 1.0 }");
                    }

                   fprintf(file_pointer,"\n    }"); /* end Appearance */
                   fprintf(file_pointer,"\n        geometry Sphere { ");
                   fprintf(file_pointer,"\n        radius
                   %6.3f ",Sphere_Radius);
                   fprintf(file_pointer,"\n       }");
                   /* fprintf(file_pointer,"\n  },"); */
```

```
 /* Check if Hyperlink was viewed for tail */
 if((inviewptr->all_comps[i]==1) ||
 (inviewptr->all_comps[i]==0 &&
  inviewptr->view_comps[i][curr_root]==1))
 {

 /* Handle alternate tails in case of hyperlink */
 if(strncmp( (inviewptr->Hyperlinks_Comps[25*i
  + curr_root])," ",1) != 0)
 {
  fprintf(file_pointer,"\n     }");
  fprintf(file_pointer,"\n ]");
  fprintf(file_pointer,"\n },");
 }
  else
 {
  fprintf(file_pointer,"\n     }, ");
  }

 }

 else  /* if hyperlink not viewed */
 {
 fprintf(file_pointer,"\n     }, ");
 } /* End check if viewed hyperlink */

 fprintf(file_pointer,"\n      Transform { ");
 fprintf(file_pointer,"\n        translation -0.5 0.35 0.25");
 fprintf(file_pointer,"\n        children [");
 fprintf(file_pointer,"\n        Shape {");
 fprintf(file_pointer,"\n       appearance Appearance {");
 fprintf(file_pointer,"\n       material
   Material { diffuseColor 1.0 1.0 0.0 ");

 /* Check if text is viewed */
if((inviewptr->all_comps[i]==1) ||
(inviewptr->all_comps[i]==0 &&
 inviewptr->view_comps[i][curr_root]==1))
 {
  fprintf(file_pointer,
  "\n                     transparency 0.0 }");
 }
 else
 {
  fprintf(file_pointer,
  "\n                           transparency 1.0 }");
  }

  fprintf(file_pointer,"\n        }");  /* end Appearance */
  fprintf(file_pointer,"\n        geometry Text {");
  fprintf(file_pointer,"\n          string \"%s\" ",
  rtemp->Arch_CompID[curr_root]);
  fprintf(file_pointer,"\n          fontStyle FontStyle {");
  fprintf(file_pointer,"\n             style \"BOLD\" ");
  fprintf(file_pointer,"\n             justify \"MIDDLE\" ");
  fprintf(file_pointer,"\n             size 0.23");
```

```
     fprintf(file_pointer,"\n } } } ] }");
     fprintf(file_pointer,"\n ]");
     fprintf(file_pointer,"\n },");
     }

    if(level > 10 && level < 25)  /* If node level > 10 */
    {
    Sphere_Radius = 0.25;

    /* Check to see if Component is viewed at all... */
   if((inviewptr->all_comps[i]==1) ||
   (inviewptr->all_comps[i]==0 &&
    inviewptr->view_comps[i][curr_root]==1))
    {

    /* If viewed, then handle the case of hyperlink url */
    if(strncmp( (inviewptr->Hyperlinks_Comps[25*i
    + curr_root])," ",1) != 0)
    {
    fprintf(file_pointer,"\nAnchor {");
    fprintf(file_pointer,"\nurl \"%s\"  ",
    inviewptr->Hyperlinks_Comps[25*i + curr_root]);
    fprintf(file_pointer,"\n children [");
    }

   }  /* End if check for comp to view */

    fprintf(file_pointer,"\n    Shape  {");
    fprintf(file_pointer,"\n    appearance Appearance {");
    fprintf(file_pointer,"\n     material Material {");
    fprintf(file_pointer,"\n       diffuseColor 0.0 0.0 1.0");

    /* Check if Sphere is viewed */
    if((inviewptr->all_comps[i]==1) ||
    (inviewptr->all_comps[i]==0 &&
     inviewptr->view_comps[i][curr_root]==1))
    {
    fprintf(file_pointer,
    "\n                            transparency 0.0 }");
    }
    else
    {
     fprintf(file_pointer,
     "\n                            transparency 1.0 }");
    }

    fprintf(file_pointer,"\n    }");  /* end Appearance */
    fprintf(file_pointer,"\n        geometry Sphere { ");
    fprintf(file_pointer,"\n        radius
    %6.3f ",Sphere_Radius);
    fprintf(file_pointer,"\n       }");
   /* fprintf(file_pointer,"\n  },"); */

  /* Check if Hyperlink was viewed for tail */
 if((inviewptr->all_comps[i]==1) ||
 (inviewptr->all_comps[i]==0 &&
  inviewptr->view_comps[i][curr_root]==1))
```

```
{
/* Handle alternate tails in case of hyperlink */
if(strncmp( (inviewptr->Hyperlinks_Comps[25*i +
curr_root])," ",1) != 0)
 {
  fprintf(file_pointer,"\n     }");
  fprintf(file_pointer,"\n ]");
  fprintf(file_pointer,"\n },");
 }
else
 {
  fprintf(file_pointer,"\n     }, ");
 }
}

else  /* if hyperlink not viewed */
{
fprintf(file_pointer,"\n     }, ");
} /* End check if viewed hyperlink */

fprintf(file_pointer,"\n      Transform { ");
fprintf(file_pointer,"\n        translation -0.5 0.35 0.25");
fprintf(file_pointer,"\n        children [");
fprintf(file_pointer,"\n        Shape {");
fprintf(file_pointer,"\n    appearance Appearance {");
fprintf(file_pointer,"\n   material
 Material { diffuseColor 1.0 1.0 0.0 ");

/* Check if text is viewed */
if((inviewptr->all_comps[i]==1) ||
(inviewptr->all_comps[i]==0 &&
 inviewptr->view_comps[i][curr_root]==1))
{
 fprintf(file_pointer,
 "\n                             transparency 0.0 }");
}
 else
 {
  fprintf(file_pointer,
  "\n                             transparency 1.0 }");
 }

  fprintf(file_pointer,"\n      }");  /* end Appearance */
  fprintf(file_pointer,"\n      geometry Text {");
  fprintf(file_pointer,"\n      string \"%s\" ",
  rtemp->Arch_CompID[curr_root]);
  fprintf(file_pointer,"\n          fontStyle FontStyle {");
  fprintf(file_pointer,"\n             style \"BOLD\" ");
  fprintf(file_pointer,"\n             justify \"MIDDLE\" ");
  fprintf(file_pointer,"\n             size 0.23");
  fprintf(file_pointer,"\n } } } ] }");
  fprintf(file_pointer,"\n ]");
  fprintf(file_pointer,"\n },");
}
```

```
   for(col=0; col<= (rtemp->No_Comps - 1); col++)
   {
    conn_node = rtemp->Topology[curr_root][col];
    if(conn_node != -1)
    {
     conn_x = rtemp->Arch_ConnPosition[conn_node][0];
     conn_y = rtemp->Arch_ConnPosition[conn_node][1];
     conn_z = rtemp->Arch_ConnPosition[conn_node][2];
     delta_sum = rtemp->Arch_ConnPosition[conn_node][3];
     conn_level = rtemp->Arch_CompPosition[curr_root][3];
     strcpy(conn_name, rtemp->Arch_ConnID[conn_node]);
     conn_x = rtemp->Arch_ConnPosition[conn_node][0];
     conn_y = rtemp->Arch_ConnPosition[conn_node][1];
     conn_z = rtemp->Arch_ConnPosition[conn_node][2];

     if(conn_level == 0.0)
      Conn_Height = 15.0;

     if(conn_level == 1.0)
      Conn_Height = 7.5;

     if(conn_level == 2.0)
      Conn_Height = 3.75;

     if(conn_level == 3.0)
      Conn_Height = 2.9;

     if(conn_level == 4.0)
      Conn_Height = 1.5;

     if(conn_level == 5.0)
      Conn_Height = 1.5;

     if(conn_level >= 6.0)
      Conn_Height = 1.0;

/* Generate VRML Connector here */


fprintf(file_pointer,"\n");


if(delta_sum >= 0.0)
{
fprintf(file_pointer,"\n Transform {");
fprintf(file_pointer,"\n rotation 0.0 1.0 0.0 %6.3f ",delta_sum);
fprintf(file_pointer,"\n center
%6.3f %6.3f %6.3f ",root_x,root_y,root_z);
fprintf(file_pointer,"\n    children [");
/* Check to see if Connector is viewed at all... */
if((inviewptr->all_conns[i]==1) ||
(inviewptr->all_conns[i]==0 &&
 inviewptr->view_conns[i][conn_node]==1))
 {
/* If viewed, then handle the case of hyperlink url */
 if(strncmp( (inviewptr->Hyperlinks_Conns[25*i
 + conn_node])," ",1) != 0)
```

```
{
fprintf(file_pointer,"\nAnchor {");
fprintf(file_pointer,"\nurl \"%s\"  ",
inviewptr->Hyperlinks_Conns[25*i + conn_node]);
fprintf(file_pointer,"\n children [");
}

}  /* End if check for comp to view */



fprintf(file_pointer,"\n Transform {");
fprintf(file_pointer,"\n  translation
%6.3f %6.3f %6.3f",conn_x,conn_y,conn_z);
fprintf(file_pointer,"\n  rotation 0.0 0.0 1.0 -1.047");
fprintf(file_pointer,"\n    children [");
fprintf(file_pointer,"\n    Shape  {");
fprintf(file_pointer,"\n    appearance Appearance {");
fprintf(file_pointer,"\n     material Material {");
fprintf(file_pointer,"\n       diffuseColor 1.0 0.95 0.85");
/* Check if Cylinder is viewed */
if((inviewptr->all_conns[i]==1) ||
(inviewptr->all_conns[i]==0 &&
 inviewptr->view_conns[i][conn_node]==1))
{
fprintf(file_pointer,
"\n                            transparency 0.0 }");
}
else
{
fprintf(file_pointer,
"\n                            transparency 1.0 }");
}
fprintf(file_pointer,"\n    }"); /* end Appearance */
fprintf(file_pointer,"\n        geometry Cylinder { ");
fprintf(file_pointer,"\n        radius %6.3f ",Cylinder_Radius);
fprintf(file_pointer,"\n        height %6.3f ",Conn_Height);
fprintf(file_pointer,"\n       }");
fprintf(file_pointer,"\n  }");  /* end Shape */

fprintf(file_pointer,"\n ]");
/* fprintf(file_pointer,"\n }"); */

/* Check if Hyperlink was viewed for tail */
if((inviewptr->all_conns[i]==1) ||
(inviewptr->all_conns[i]==0 &&
inviewptr->view_conns[i][conn_node]==1))
{
  /* Handle alternate tails in case of hyperlink */
  if(strncmp( (inviewptr->Hyperlinks_Conns[25*i
  + conn_node])," ",1) != 0)
  {
   fprintf(file_pointer,"\n     }");
   fprintf(file_pointer,"\n ]");
   fprintf(file_pointer,"\n },");
   }
   else
```

```
 {
 fprintf(file_pointer,"\n     }, ");
 }

}
 else  /* if hyperlink not viewed */
{
 fprintf(file_pointer,"\n     }, ");
 } /* End check if viewed hyperlink */


 fprintf(file_pointer,"\n   Transform { ");
 fprintf(file_pointer,"\n   translation
 %6.3f %6.3f %6.3f",conn_x,conn_y,conn_z);
/* fprintf(file_pointer,"\n   rotation 0.0 0.0 1.0 -1.047"); */
 fprintf(file_pointer,"\n     children [");
 fprintf(file_pointer,"\n       Transform { ");

 if( (conn_level >= 0.0) && (conn_level <= 1.0) )
 { fprintf(file_pointer,"\n          translation -1.1 0.0 0.0");
   font_size = 0.33;
 }

    if( conn_level == 2.0 )
    {
    fprintf(file_pointer,"\n          translation -1.0 1.0 0.0");
    font_size = 0.3;
    }

    if( conn_level >= 3.0 )
    {
    fprintf(file_pointer,
    "\n          translation -0.80 0.75 0.0");
    font_size = 0.25;
    }

  fprintf(file_pointer,"\n          rotation 0.0 1.0 0.0
  %6.3f",-delta_sum);
  fprintf(file_pointer,"\n          children [");
  fprintf(file_pointer,"\n        Shape {");
  fprintf(file_pointer,"\n       appearance Appearance {");
  fprintf(file_pointer,"\n        material
   Material { diffuseColor 1.0 1.0 0.0 ");

  /* Check if text is viewed */
  if((inviewptr->all_conns[i]==1) ||
  (inviewptr->all_conns[i]==0 &&
   inviewptr->view_conns[i][conn_node]==1))
  {
   fprintf(file_pointer,
   "\n                            transparency 0.0 }");
  }
   else
  {
   fprintf(file_pointer,
   "\n                            transparency 1.0 }");
  }
```

```
        fprintf(file_pointer,"\n        }");  /* end Appearance */
        fprintf(file_pointer,"\n        geometry Text {");
        fprintf(file_pointer,"\n          string \"%s\" ",conn_name);
        fprintf(file_pointer,"\n          fontStyle FontStyle {");
        fprintf(file_pointer,"\n             style \"BOLD\" ");
        fprintf(file_pointer,"\n             justify \"MIDDLE\" ");
        fprintf(file_pointer,"\n             size %6.3f ",font_size);

        fprintf(file_pointer,"\n } } } ] } ] },");

        fprintf(file_pointer,"\n             ]");
        fprintf(file_pointer," \n }, ");

        }  /* end delta_sum >= 0 */



       /* Place component index, col, into queue */
       insert_queue_item(col);
       }
   }  /* end FOR loop */

} /* END if temp_arch_style == 1 Call and Return */


if(temp_arch_style == 2)   /* if the LAYERED style */
{
 /* Render Layer Here */
scale_value = (4*curr_root) + (curr_root*(curr_root-1)/2.0);
position_value = scale_value;
title_value = (4*(curr_root+1)) + (((curr_root+1)*curr_root)/2.0);


/* Translation for next layer component */
fprintf(file_pointer,"\n");
fprintf(file_pointer,"Transform  { \n");
fprintf(file_pointer,"   scale 0.70 0.70 0.70 \n");
fprintf(file_pointer,"   translation 0.0 ");
fprintf(file_pointer,"%4.2f",position_value);
fprintf(file_pointer," 0.0 \n");
fprintf(file_pointer,"   children  [ \n");
fprintf(file_pointer,"  Transform { \n");
fprintf(file_pointer,"    translation ");
fprintf(file_pointer,"                %4.2f 2.0 0.0 \n",title_value);
fprintf(file_pointer,"    children [ \n");
fprintf(file_pointer,"      Shape { \n");
fprintf(file_pointer,"      appearance Appearance { \n");
fprintf(file_pointer,"        material
 Material  {diffuseColor 1.0 1.0 0.0 \n");

/* Check if Layer-Title is viewed */
if((inviewptr->all_comps[i]==1) ||
(inviewptr->all_comps[i]==0 &&
 inviewptr->view_comps[i][curr_root]==1))
 {
  fprintf(file_pointer,
```

```
 "\n                        transparency 0.0 }");
}
else
{
fprintf(file_pointer,
 "\n                        transparency 1.0 }");
}

fprintf(file_pointer,"           } \n");  /* end Appearance */
fprintf(file_pointer,"        geometry Text  {  \n");
fprintf(file_pointer,"        string [%c",quotemark);
fprintf(file_pointer,"%s",rtemp->Arch_CompID[curr_root]);
fprintf(file_pointer,"%c",quotemark);
fprintf(file_pointer,"] \n");
fprintf(file_pointer,"         fontStyle
 FontStyle { size 2.0 } \n");
fprintf(file_pointer,"        } \n");
fprintf(file_pointer,"       } \n");
fprintf(file_pointer,"      ] \n");
fprintf(file_pointer,"      },");
fprintf(file_pointer," \n");

/* Check to see if Layer-Component is viewed at all... */
if((inviewptr->all_comps[i]==1) ||
  (inviewptr->all_comps[i]==0 &&
   inviewptr->view_comps[i][curr_root]==1))
{
/* If viewed, then handle the case of hyperlink url */
   if(strncmp( (inviewptr->Hyperlinks_Comps[25*i
   + curr_root])," ",1) != 0)
 {
   fprintf(file_pointer,"\nAnchor {");
   fprintf(file_pointer,"\nurl \"%s\"  ",
   inviewptr->Hyperlinks_Comps[25*i + curr_root]);
   fprintf(file_pointer,"\n children [");
 }

}  /* End if check for layer to view */

/* Generate extrusion node for layer */
   fprintf(file_pointer,"Shape { \n");
   fprintf(file_pointer,"   appearance Appearance { \n");
   fprintf(file_pointer,"      material Material { \n");

   /* If even layer, use separate color than odd layer */

   /* Also check for visibility: yes or no */

   if((inviewptr->all_comps[i]==1) ||
     (inviewptr->all_comps[i]==0 &&
      inviewptr->view_comps[i][curr_root]==1))
   {
   /* Handle the case where visible */
   if( (curr_root>0) && (curr_root%2)!=0)
     fprintf(file_pointer,"      diffuseColor 1.0 0.0 1.0 \n");
   else
     fprintf(file_pointer,"      diffuseColor 1.0 1.0 0.0 \n");
```

```
}
 else
{
 fprintf(file_pointer,"      diffuseColor 0.0 0.0 0.0 \n");
 fprintf(file_pointer,
 "\n                              transparency 1.0");
}

 fprintf(file_pointer,"      } \n");
 fprintf(file_pointer,"   } \n");
 fprintf(file_pointer,"   geometry Extrusion { \n");
 fprintf(file_pointer,"       creaseAngle 1.57 \n");
 fprintf(file_pointer,"       endCap TRUE \n");
 fprintf(file_pointer,"       solid TRUE  \n");
 fprintf(file_pointer,"       crossSection [ \n");
 fprintf(file_pointer,"       # Circle \n");
 fprintf(file_pointer,"        1.00  0.00,   0.90 -0.38,  \n");
 fprintf(file_pointer,"        0.70 -0.70,   0.38 -0.90,  \n");
 fprintf(file_pointer,"        0.00 -1.00,  -0.38 -0.90,  \n");
 fprintf(file_pointer,"       -0.70 -0.70,  -0.91 -0.38,  \n");
 fprintf(file_pointer,"       -1.00 -0.00,  -0.91  0.38,  \n");
 fprintf(file_pointer,"       -0.70  0.70,  -0.38  0.90,  \n");
 fprintf(file_pointer,"        0.00  1.00,   0.38  0.90,  \n");
 fprintf(file_pointer,"        0.70  0.70,   0.90  0.38,  \n");
 fprintf(file_pointer,"        1.00  0.00  \n");
 fprintf(file_pointer,"        ] \n");
 fprintf(file_pointer,"        spine [ \n");
 fprintf(file_pointer,"        # Straight-line \n");
 fprintf(file_pointer,
 "          0.0 0.0 0.0,  0.0 0.4 0.0,  \n");
 fprintf(file_pointer,
 "          0.0 0.8 0.0,  0.0 1.2 0.0,  \n");
 fprintf(file_pointer,
 "          0.0 1.6 0.0,  0.0 2.0 0.0,  \n");
 fprintf(file_pointer,
 "          0.0 2.4 0.0,  0.0 2.8 0.0,  \n");
 fprintf(file_pointer,
 "          0.0 3.2 0.0,  0.0 3.6 0.0   \n");
 fprintf(file_pointer,"          0.0 4.0 0.0  \n");
 fprintf(file_pointer,"        ]  \n");
 fprintf(file_pointer,"        scale [ \n");
 fprintf(file_pointer,"        %4.2f",scale_value +0.0);
 fprintf(file_pointer," %4.2f",scale_value + 0.0);
 fprintf(file_pointer,", ");
 fprintf(file_pointer," %4.2f",scale_value + 0.40);
 fprintf(file_pointer," %4.2f",scale_value + 0.40);
 fprintf(file_pointer,", \n");
 fprintf(file_pointer,"        %4.2f",scale_value + 0.80);
 fprintf(file_pointer," %4.2f",scale_value + 0.80);
 fprintf(file_pointer,", ");
 fprintf(file_pointer," %4.2f",scale_value + 1.2);
 fprintf(file_pointer," %4.2f",scale_value + 1.2);
 fprintf(file_pointer,", \n");
 fprintf(file_pointer,"        %4.2f",scale_value + 1.6);
 fprintf(file_pointer," %4.2f",scale_value + 1.6);
 fprintf(file_pointer,", ");
 fprintf(file_pointer," %4.2f",scale_value + 2.0);
```

```
 fprintf(file_pointer," %4.2f",scale_value + 2.0);
 fprintf(file_pointer,", \n");
 fprintf(file_pointer,"        %4.2f",scale_value + 2.4);
 fprintf(file_pointer," %4.2f",scale_value + 2.4);
 fprintf(file_pointer,", ");
 fprintf(file_pointer," %4.2f",scale_value + 2.8);
 fprintf(file_pointer," %4.2f",scale_value + 2.8);
 fprintf(file_pointer,", \n");
 fprintf(file_pointer,"        %4.2f",scale_value + 3.2);
 fprintf(file_pointer," %4.2f",scale_value + 3.2);
 fprintf(file_pointer,", ");
 fprintf(file_pointer," %4.2f",scale_value + 3.6);
 fprintf(file_pointer," %4.2f",scale_value + 3.6);
 fprintf(file_pointer,", \n");
 fprintf(file_pointer,"        %4.2f",scale_value + 4.0);
 fprintf(file_pointer," %4.2f",scale_value + 4.0);
 fprintf(file_pointer," \n");
 fprintf(file_pointer,"        ] \n");
 fprintf(file_pointer,"        } \n");
 fprintf(file_pointer,"} \n");
 fprintf(file_pointer,"] \n");
 /* fprintf(file_pointer,"},");    */

/* Check if Hyperlink was viewed for tail */
if((inviewptr->all_comps[i]==1) ||
(inviewptr->all_comps[i]==0
  && inviewptr->view_comps[i][curr_root]==1))
{

 /* Handle alternate tails in case of hyperlink */
 if(strncmp( (inviewptr->Hyperlinks_Comps[25*i
 + curr_root])," ",1) != 0)
 {
  fprintf(file_pointer,"\n     }");
  fprintf(file_pointer,"\n ]");
  fprintf(file_pointer,"\n },");
  }
  else
  {
  fprintf(file_pointer,"\n     }, ");
  }
  }
  else  /* if hyperlink not viewed */
 {
  fprintf(file_pointer,"\n     }, ");
 } /* End check if viewed hyperlink */

 /* Get the next layer component, store in queue */
 for(col=0; col<= (rtemp->No_Comps - 1); col++)
  {
  conn_node = rtemp->Topology[curr_root][col];
  if(conn_node != -1)
  {

  conn_level = rtemp->Arch_CompPosition[curr_root][3];
  strcpy(conn_name, rtemp->Arch_ConnID[conn_node]);
  insert_queue_item(col);  /* insert next component */
```

```
                }

                }    /* end for */

              }   /* end if LAYERED style */

        queue_empty_flag = is_queue_empty();

     }  /* end WHILE */

     if(rtemp->Style == 2)
     {
     fprintf(file_pointer,"\n ]");  /* end of Group */
     fprintf(file_pointer,"\n }");
     fprintf(file_pointer,"\n ] }"); /* end of Transform */
     }

     /* Print Footer for architecture */
     fprintf(file_pointer,"\n ]");
     fprintf(file_pointer,"\n }");
     fprintf(file_pointer,"\n");

     if(i > 0)
     {
      fprintf(file_pointer,"\n ]");
      fprintf(file_pointer,"\n }");
     }
    }
     else
     {
      rtemp = rtemp->archlink;
     }
  }    while( (rtemp != NULL) && (item_located != 1) );

  }      /* end if rtemp not NULL */

       }  /* end for i loop for arch_ID of view */

       fclose(file_pointer);
       inviewptr = inviewptr->viewlink;
       } /* End while inviewptr not NULL */
}
```

Appendix G

Test Cases

Test Cases

Numerous implementation tests were conducted on the VTADL-to-VRML compiler. A representative sample of two tests will be used to demonstrate that the compiler is working according to requirements.

The Case Study Report Template will be used for each test case.

Test Case One

Case Study Report

1. **Name of Case Study:** Test Case One

2. **On-Line Posting (if any):** None. Testing purposes only.

3. **Brief Description of the Purpose and Background of the Case Study:**

This test was designed to show that multiple architectures (of both call-and-return and layered styles) could be rendered within a view. The test includes different versions of the same layered architecture, selectively hidden or displayed components, and hyperlinks to other views and external files.

The VTADL source file, "Example.txt," defines two architectures: "ExampleProgram" and "ExampleLayer." ExampleProgram is a hierarchical structure in the call-and-return style. ExampleProgram has a root and three children (four components and three connectors). ExampleLayer has four components in the layered architectural style.

The Main view contains both architectures, using all components and connections in each architecture. A hyperlink exists from the root in ExampleProgram to the source VTADL file (Example.txt). Another hyperlink is established from the top layer of ExampleLayer to a view named "SecondView."

SecondView contains three versions of ExampleLayer. Version one uses the first and second layers only, hiding the other layers; version two uses the first, third and fourth layers; and version three uses all four layers. The first layer of version one contains a hyperlink to the Main view; the second layer of version one contains a hyperlink to the source VTADL file, Example.txt.

4. **Name of VTADL Source File(s):**

Source File	**View Files**	**Hyperlinked Files (Referenced)**
Example.txt	Main.wrl	Example.txt
	SecondView.wrl	Main.wrl
		SecondView.wrl

5. VTADL Source Code Listing ("Example.txt")

The VTADL source code for Example.txt was listed in full in Appendix E.

6. VRML Target Code: Main.wrl

The generated VRML code for Main.wrl is provided as a sample. The VRML code for SecondView.wrl is not provided for sake of brevity.

Generated VRML Code: Main.wrl

```
#VRML V2.0 utf8
#Viewpoint of Architectures
#Generated by VTADL Version 1.0
Background {
    skyColor [
       0.0 0.2 0.91,
       0.0 0.3 1.0,
       0.3 0.2 0.85
     ]
skyAngle [ 1.309, 1.571 ]
}
Group {
children [
 Transform {
    translation 0.0 4.00 0.0
    children [
      Shape {
       appearance Appearance {
        material Material { diffuseColor 1.0 1.0 0.0 }
        }
        geometry Text {
          string "Main"
          fontStyle FontStyle {
             style "BOLD"
             justify "MIDDLE"
             size 0.94
 } } } ] },
 Transform {
   translation 27.1 6.5 0.0
    children[
      Shape {
       appearance Appearance {
        material Material { diffuseColor 1.0 1.0 0.0 }
        }
        geometry Text {
          string " LEGEND "
          fontStyle FontStyle {
             style "BOLD"
             justify "BEGIN"
```

```
             size 0.94
 } } } ] },
Viewpoint {
description "Legend"
position 25.4 4.75 21.5 }
 Transform {
    translation 20.0  4.000 0.0
    children [
      Shape {
       appearance Appearance {
        material Material { diffuseColor 1.0 1.0 0.0 }
        }
        geometry Text {
          string "Main"
          fontStyle FontStyle {
             style "BOLD"
             justify "BEGIN"
             size 0.94
 } } } ] },
 Transform {
    translation 29.2  4.350 0.0
    children [
 ] },
 Transform {
    translation 20.0  2.050 0.0
    children [
      Shape {
       appearance Appearance {
        material Material { diffuseColor 1.0 1.0 0.0 }
        }
        geometry Text {
          string "SecondView"
          fontStyle FontStyle {
             style "BOLD"
             justify "BEGIN"
             size 0.94
 } } } ] },
 Transform {
    translation 29.2  2.400 0.0
    children [
Anchor {
url "SecondView.wrl"
 children [
    Shape  {
    appearance Appearance {
     material Material {
       diffuseColor 1.0 1.0 0.0
          }
    }
        geometry Sphere {
        radius 0.62
       }
  }
  ]
  }
 ] },
 Transform {
```

```
    translation 0.0 3.00 0.0
    children [
Viewpoint {
description "ExampleProgram"
position 0.0 -5.0 21.5 },
      Shape {
       appearance Appearance {
        material Material { diffuseColor 1.0 1.0 0.0 }
        }
        geometry Text {
          string " Architecture ExampleProgram"
          fontStyle FontStyle {
             style "BOLD"
             justify "MIDDLE"
             size 0.65
 } } } ] },
Transform {
  translation  0.000  0.000  0.000
    children [
Anchor {
url "Example.txt"
 children [
    DEF Root_Node_Type Shape  {
    appearance Appearance {
     material Material {
       diffuseColor 0.1 0.99 0.99
      transparency 0.0 }
    }
        geometry Sphere {
        radius  0.600
       }
     }
 ]
 },
      Transform {
        translation -0.55 0.75 0.0
        children [
        Shape {
       appearance Appearance {
        material Material { diffuseColor 1.0 1.0 0.0
                            transparency 0.0 }
        }
      geometry Text {
        string "Alpha"
        fontStyle FontStyle {
             style "BOLD"
             justify "MIDDLE"
             size 0.6
 } } } ] }
 ]
 },

 Transform {
 rotation 0.0 1.0 0.0  0.000
 center  0.000  0.000  0.000
    children [
 Transform {
```

```
 translation -6.495 -3.750  0.000
 rotation 0.0 0.0 1.0 -1.047
   children [
   Shape  {
   appearance Appearance {
    material Material {
      diffuseColor 1.0 0.95 0.85
                           transparency 0.0 }
   }
       geometry Cylinder {
       radius  0.100
       height 15.000
      }
 }
]
    },
  Transform {
  translation -6.495 -3.750  0.000
    children [
      Transform {
         translation -1.1 0.0 0.0
         rotation 0.0 1.0 0.0 -0.000
         children [
       Shape {
      appearance Appearance {
       material Material { diffuseColor 1.0 1.0 0.0
                           transparency 0.0 }
       }
       geometry Text {
         string "Call1"
         fontStyle FontStyle {
            style "BOLD"
            justify "MIDDLE"
            size  0.330
} } } ] } ] },
            ]
},

Transform {
rotation 0.0 1.0 0.0  2.094
center  0.000  0.000  0.000
   children [
Transform {
 translation -6.495 -3.750  0.000
 rotation 0.0 0.0 1.0 -1.047
   children [
   Shape  {
   appearance Appearance {
    material Material {
      diffuseColor 1.0 0.95 0.85
                           transparency 0.0 }
   }
       geometry Cylinder {
       radius  0.100
       height 15.000
      }
 }
```

```
]
    },
  Transform {
  translation -6.495 -3.750  0.000
    children [
      Transform {
         translation -1.1 0.0 0.0
         rotation 0.0 1.0 0.0 -2.094
         children [
       Shape {
      appearance Appearance {
       material Material { diffuseColor 1.0 1.0 0.0
                           transparency 0.0 }
       }
       geometry Text {
         string "Call2"
         fontStyle FontStyle {
            style "BOLD"
            justify "MIDDLE"
            size  0.330
} } } ] } ] },
            ]
},

Transform {
rotation 0.0 1.0 0.0  4.189
center  0.000  0.000  0.000
   children [
Transform {
 translation -6.495 -3.750  0.000
 rotation 0.0 0.0 1.0 -1.047
   children [
   Shape  {
   appearance Appearance {
    material Material {
      diffuseColor 1.0 0.95 0.85
                           transparency 0.0 }
   }
       geometry Cylinder {
       radius  0.100
       height 15.000
      }
 }
]
    },
  Transform {
  translation -6.495 -3.750  0.000
    children [
      Transform {
         translation -1.1 0.0 0.0
         rotation 0.0 1.0 0.0 -4.189
         children [
       Shape {
      appearance Appearance {
       material Material { diffuseColor 1.0 1.0 0.0
                           transparency 0.0 }
       }
```

```
        geometry Text {
          string "Call3"
          fontStyle FontStyle {
             style "BOLD"
             justify "MIDDLE"
             size  0.330
 } } } ] } ] },
             ]
 },
Transform {
  translation -12.990 -7.500  0.000
    children [
    Shape  {
    appearance Appearance {
     material Material {
       diffuseColor 1.0 0.0 0.0
      transparency 0.0 }
    }
        geometry Sphere {
        radius  0.600
       }
     },
      Transform {
        translation -0.55 0.65 0.6
        children [
        Shape {
       appearance Appearance {
        material Material { diffuseColor 1.0 1.0 0.0
                            transparency 0.0 }
        }
        geometry Text {
          string "Beta"
          fontStyle FontStyle {
             style "BOLD"
             justify "MIDDLE"
             size 0.43
 } } } ] }
 ]
 },
Transform {
  translation  6.495 -7.500 11.250
    children [
    Shape  {
    appearance Appearance {
     material Material {
       diffuseColor 1.0 0.0 0.0
      transparency 0.0 }
    }
        geometry Sphere {
        radius  0.600
       }
     },
      Transform {
        translation -0.55 0.65 0.6
        children [
        Shape {
       appearance Appearance {
```

```
        material Material { diffuseColor 1.0 1.0 0.0
                            transparency 0.0 }
        }
        geometry Text {
          string "Gamma"
          fontStyle FontStyle {
             style "BOLD"
             justify "MIDDLE"
             size 0.43
 } } } ] }
 ]
 },
Transform {
  translation  6.495 -7.500 -11.250
    children [
    Shape  {
    appearance Appearance {
     material Material {
       diffuseColor 1.0 0.0 0.0
      transparency 0.0 }
    }
        geometry Sphere {
        radius  0.600
       }
      },
       Transform {
         translation -0.55 0.65 0.6
         children [
         Shape {
       appearance Appearance {
        material Material { diffuseColor 1.0 1.0 0.0
                            transparency 0.0 }
        }
        geometry Text {
          string "Delta"
          fontStyle FontStyle {
             style "BOLD"
             justify "MIDDLE"
             size 0.43
 } } } ] }
 ]
 },
 ]
 }

 Transform {
   translation -25.500 15.000 -50.000
   children [
   Group {
   children [
 Transform {
    translation 0.0 3.00 0.0
    children [
Viewpoint {
description "ExampleLayer"
position 0.0 -5.0 21.5 },
      Shape {
```

```
       appearance Appearance {
        material Material { diffuseColor 1.0 1.0 0.0 }
        }
        geometry Text {
          string " Architecture ExampleLayer"
          fontStyle FontStyle {
             style "BOLD"
             justify "MIDDLE"
             size 0.65
 } } } ] },
Transform {
  scale 0.5 0.5 0.5
  translation 0.0 -18.300 -6.0
    children [
Group {
    children [
Transform  {
   scale 0.70 0.70 0.70
   translation 0.0 0.00 0.0
   children  [
  Transform {
    translation                 4.00 2.0 0.0
    children [
      Shape {
      appearance Appearance {
        material Material  {diffuseColor 1.0 1.0 0.0

                       transparency 0.0 }           }
        geometry Text  {
        string ["LevelOne"]
         fontStyle FontStyle { size 2.0 }
        }
       }
      ]
      },
Shape {
   appearance Appearance {
      material Material {
      diffuseColor 1.0 1.0 0.0
      }
   }
   geometry Extrusion {
       creaseAngle 1.57
       endCap TRUE
       solid TRUE
       crossSection [
       # Circle
        1.00  0.00,   0.90 -0.38,
        0.70 -0.70,   0.38 -0.90,
        0.00 -1.00,  -0.38 -0.90,
       -0.70 -0.70,  -0.91 -0.38,
       -1.00 -0.00,  -0.91  0.38,
       -0.70  0.70,  -0.38  0.90,
        0.00  1.00,   0.38  0.90,
        0.70  0.70,   0.90  0.38,
        1.00  0.00
        ]
```

```
        spine [
        # Straight-line
          0.0 0.0 0.0,  0.0 0.4 0.0,
          0.0 0.8 0.0,  0.0 1.2 0.0,
          0.0 1.6 0.0,  0.0 2.0 0.0,
          0.0 2.4 0.0,  0.0 2.8 0.0,
          0.0 3.2 0.0,  0.0 3.6 0.0
          0.0 4.0 0.0
        ]
        scale [
        0.00 0.00,  0.40 0.40,
        0.80 0.80,  1.20 1.20,
        1.60 1.60,  2.00 2.00,
        2.40 2.40,  2.80 2.80,
        3.20 3.20,  3.60 3.60,
        4.00 4.00
        ]
        }
}
]

     },
Transform  {
   scale 0.70 0.70 0.70
   translation 0.0 4.00 0.0
   children  [
  Transform {
    translation                 9.00 2.0 0.0
    children [
      Shape {
      appearance Appearance {
        material Material  {diffuseColor 1.0 1.0 0.0

                       transparency 0.0 }           }
        geometry Text  {
        string ["LevelTwo"]
         fontStyle FontStyle { size 2.0 }
        }
       }
      ]
      },
Shape {
   appearance Appearance {
      material Material {
      diffuseColor 1.0 0.0 1.0
      }
   }
   geometry Extrusion {
       creaseAngle 1.57
       endCap TRUE
       solid TRUE
       crossSection [
       # Circle
        1.00  0.00,   0.90 -0.38,
        0.70 -0.70,   0.38 -0.90,
        0.00 -1.00,  -0.38 -0.90,
       -0.70 -0.70,  -0.91 -0.38,
```

```
       -1.00 -0.00,  -0.91  0.38,
       -0.70  0.70,  -0.38  0.90,
        0.00  1.00,   0.38  0.90,
        0.70  0.70,   0.90  0.38,
        1.00  0.00
        ]
        spine [
        # Straight-line
          0.0 0.0 0.0,  0.0 0.4 0.0,
          0.0 0.8 0.0,  0.0 1.2 0.0,
          0.0 1.6 0.0,  0.0 2.0 0.0,
          0.0 2.4 0.0,  0.0 2.8 0.0,
          0.0 3.2 0.0,  0.0 3.6 0.0
          0.0 4.0 0.0
        ]
        scale [
        4.00 4.00,  4.40 4.40,
        4.80 4.80,  5.20 5.20,
        5.60 5.60,  6.00 6.00,
        6.40 6.40,  6.80 6.80,
        7.20 7.20,  7.60 7.60,
        8.00 8.00
        ]
        }
}
]

     },
Transform  {
   scale 0.70 0.70 0.70
   translation 0.0 9.00 0.0
   children  [
  Transform {
    translation                 15.00 2.0 0.0
    children [
      Shape {
      appearance Appearance {
        material Material  {diffuseColor 1.0 1.0 0.0

                       transparency 0.0 }           }
        geometry Text  {
        string ["LevelThree"]
         fontStyle FontStyle { size 2.0 }
        }
       }
      ]
      },
Shape {
   appearance Appearance {
      material Material {
      diffuseColor 1.0 1.0 0.0
      }
   }
   geometry Extrusion {
       creaseAngle 1.57
       endCap TRUE
       solid TRUE
```

```
       crossSection [
       # Circle
        1.00  0.00,   0.90 -0.38,
        0.70 -0.70,   0.38 -0.90,
        0.00 -1.00,  -0.38 -0.90,
       -0.70 -0.70,  -0.91 -0.38,
       -1.00 -0.00,  -0.91  0.38,
       -0.70  0.70,  -0.38  0.90,
        0.00  1.00,   0.38  0.90,
        0.70  0.70,   0.90  0.38,
        1.00  0.00
        ]
        spine [
        # Straight-line
          0.0 0.0 0.0,  0.0 0.4 0.0,
          0.0 0.8 0.0,  0.0 1.2 0.0,
          0.0 1.6 0.0,  0.0 2.0 0.0,
          0.0 2.4 0.0,  0.0 2.8 0.0,
          0.0 3.2 0.0,  0.0 3.6 0.0
          0.0 4.0 0.0
        ]
        scale [
        9.00 9.00,  9.40 9.40,
        9.80 9.80,  10.20 10.20,
        10.60 10.60,  11.00 11.00,
        11.40 11.40,  11.80 11.80,
        12.20 12.20,  12.60 12.60,
        13.00 13.00
        ]
        }
}
]

     },
Transform  {
   scale 0.70 0.70 0.70
   translation 0.0 15.00 0.0
   children  [
  Transform {
    translation                 22.00 2.0 0.0
    children [
      Shape {
      appearance Appearance {
        material Material  {diffuseColor 1.0 1.0 0.0

                       transparency 0.0 }           }
        geometry Text  {
        string ["LevelFour"]
         fontStyle FontStyle { size 2.0 }
        }
       }
      ]
      },

Anchor {
url "SecondView.wrl"
 children [Shape {
```

```
    appearance Appearance {
       material Material {
       diffuseColor 1.0 0.0 1.0
       }
    }
    geometry Extrusion {
        creaseAngle 1.57
        endCap TRUE
        solid TRUE
        crossSection [
        # Circle
         1.00  0.00,   0.90 -0.38,
         0.70 -0.70,   0.38 -0.90,
         0.00 -1.00,  -0.38 -0.90,
        -0.70 -0.70,  -0.91 -0.38,
        -1.00 -0.00,  -0.91  0.38,
        -0.70  0.70,  -0.38  0.90,
         0.00  1.00,   0.38  0.90,
         0.70  0.70,   0.90  0.38,
         1.00  0.00
         ]
         spine [
         # Straight-line
           0.0 0.0 0.0,  0.0 0.4 0.0,
           0.0 0.8 0.0,  0.0 1.2 0.0,
           0.0 1.6 0.0,  0.0 2.0 0.0,
           0.0 2.4 0.0,  0.0 2.8 0.0,
           0.0 3.2 0.0,  0.0 3.6 0.0
           0.0 4.0 0.0
         ]
         scale [
         15.00 15.00,  15.40 15.40,
         15.80 15.80,  16.20 16.20,
         16.60 16.60,  17.00 17.00,
         17.40 17.40,  17.80 17.80,
         18.20 18.20,  18.60 18.60,
         19.00 19.00
         ]
         }
}
]

     }
 ]
 },
 ]
 }
 ] }
 ]
 }

 ]
 }
```

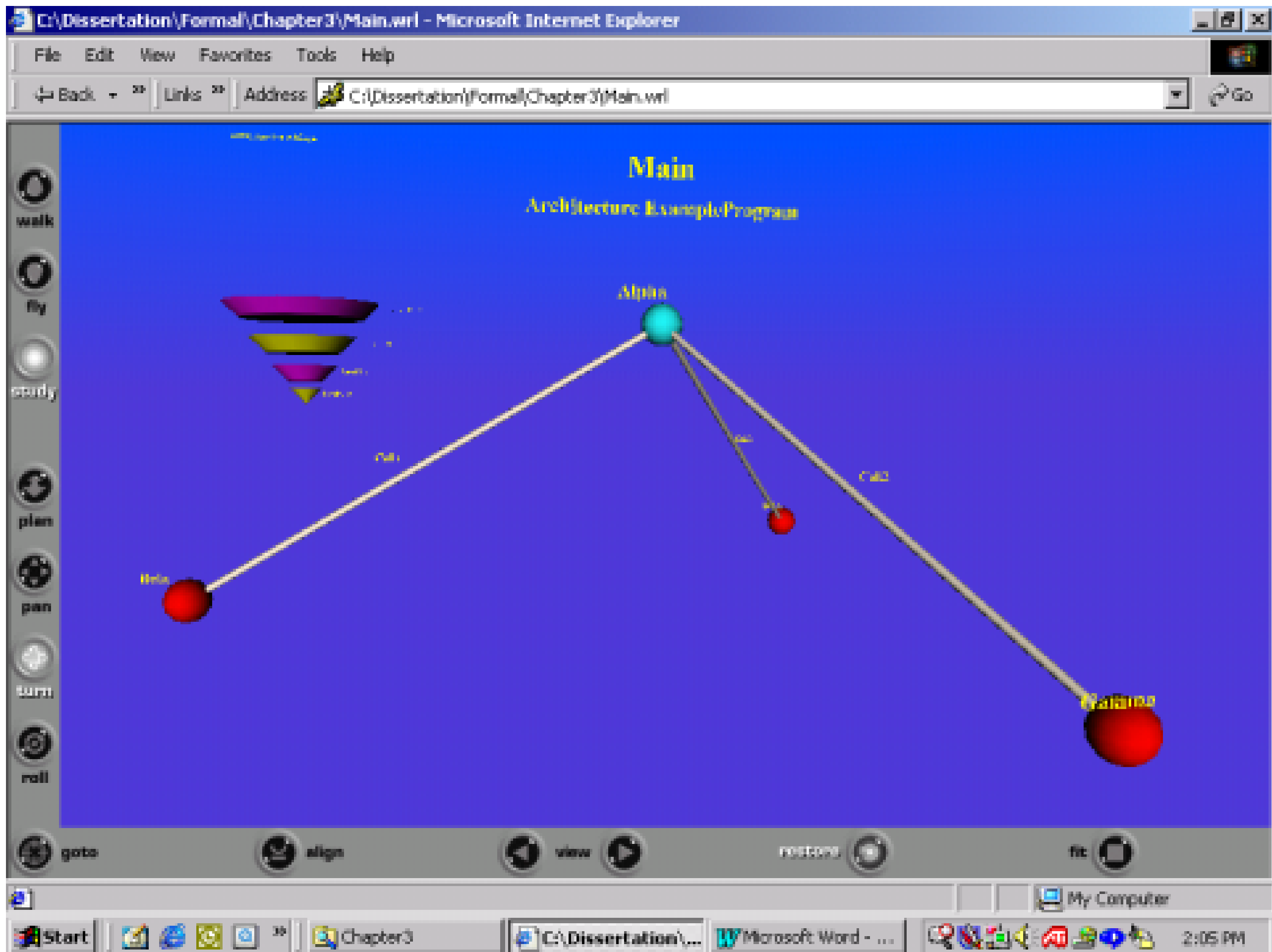
Generated VRML Code: SecondView.wrl

(Note: The generated VRML code for SecondView.wrl has been excluded from this listing for sake of brevity). It is hoped that the VRML code for Main.wrl will give the reader a representative glimpse of the VRML target code.

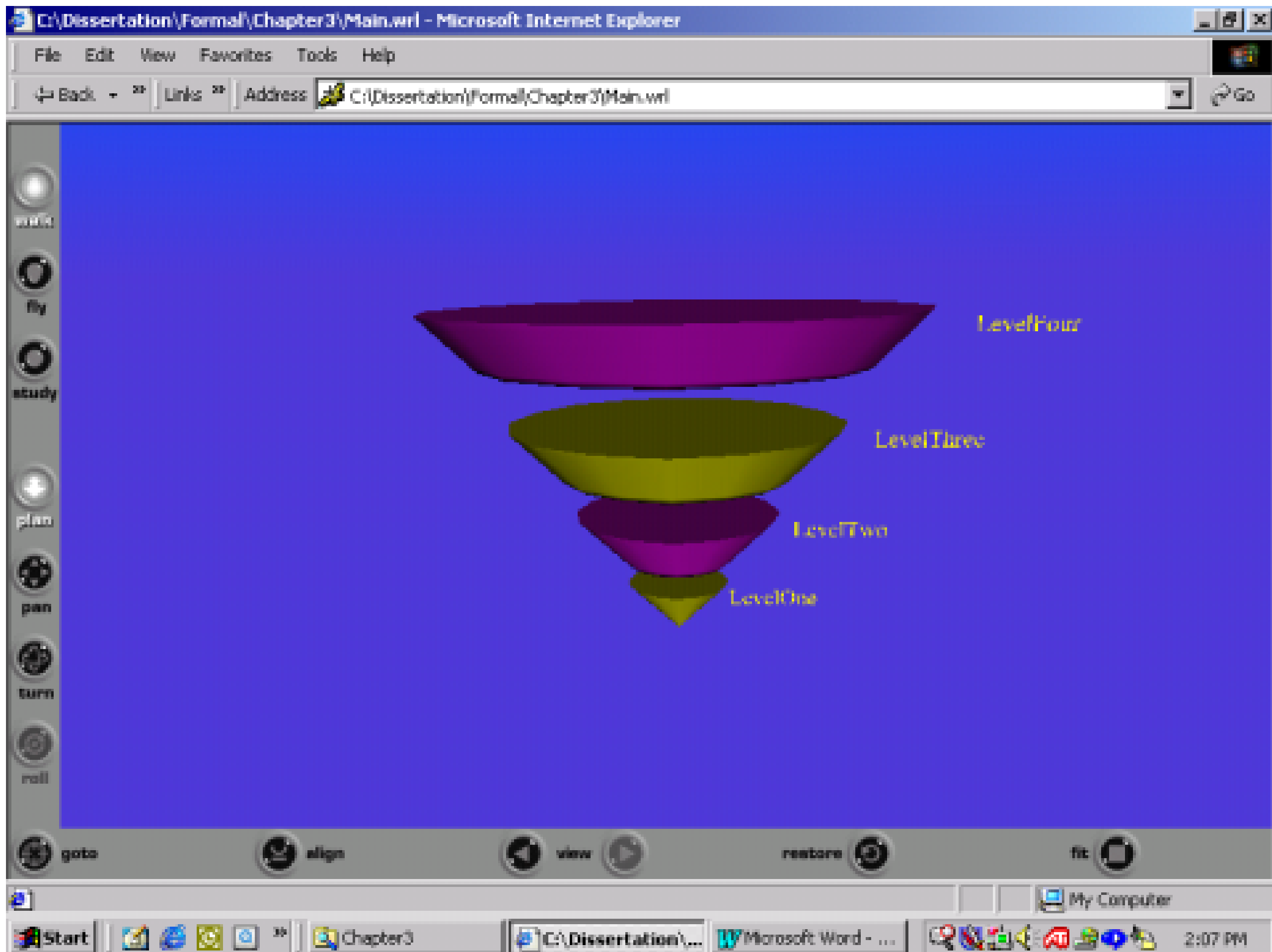
7. Screen Snapshots of VRML Images

The VTADL source file, Example.txt, generates two views: Main.wrl and SecondView.wrl. The following screen is the legend portion of the Main view, implemented in file Main.wrl.

The following screen shows the first architecture in the Main view, ExampleProgram, rendered in the call-and-return style. There are four components and three connectors, with a root named “Alpha.” Alpha contains a hyperlink to the VTADL source file, Example.txt. Visible in the background is the second layered architecture, ExampleLayer.

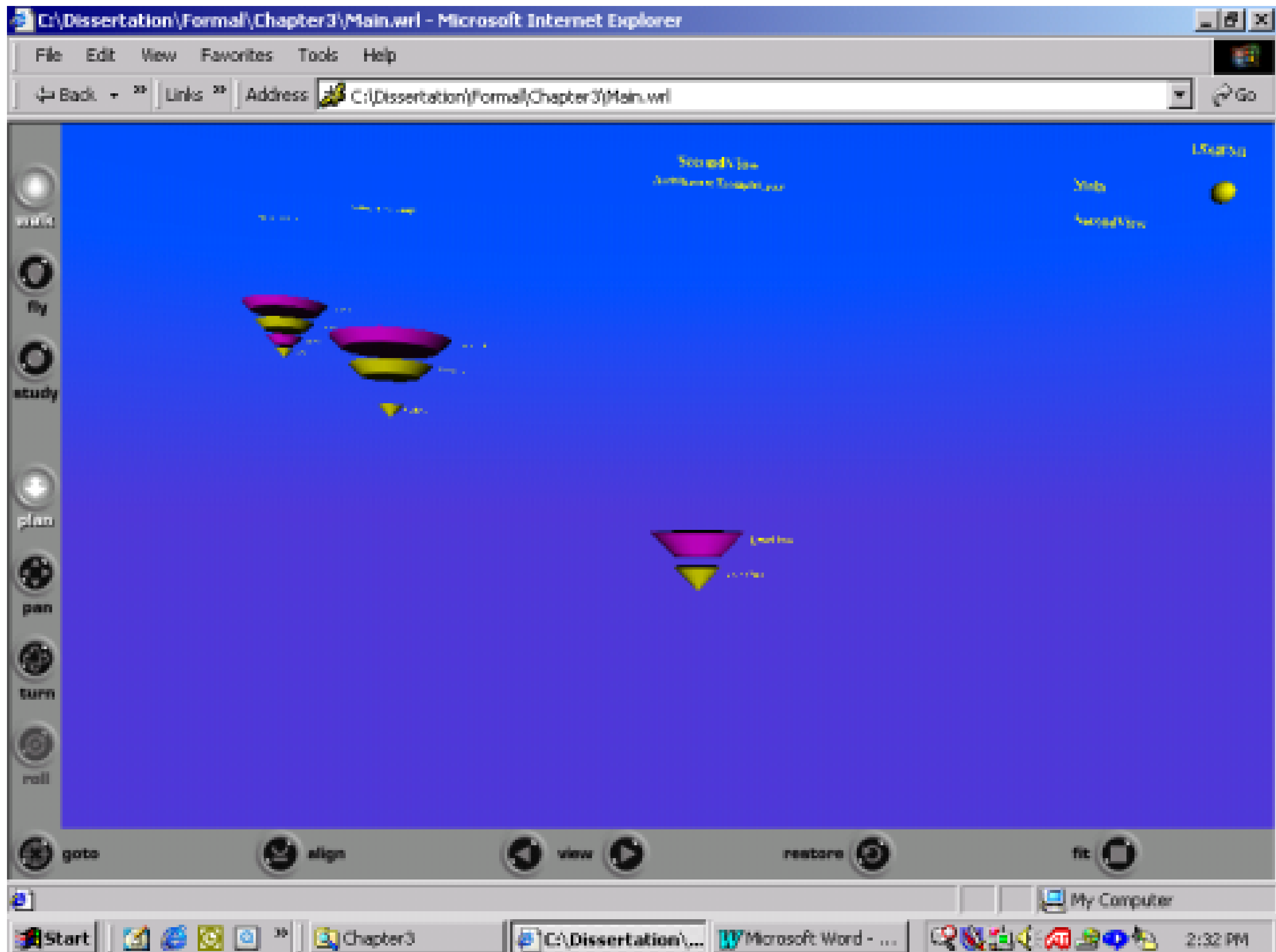


The next screen focuses on the second architecture (ExampleLayer) of the Main view. The fourth layer, labeled "LevelFour," contains a hyperlink to the VRML file, SecondView.



In the second view (SecondView), the same layered architecture is used in three different versions. The first version uses only the first two layers; the second version omits the second layer, while the third version uses all four layers.

The first version contains a hyperlink from the first layer to the Main view, and a hyperlink from the second layer to the source VTADL file, Example.txt.



Test Case Two

Case Study Report

1. **Name of Case Study:** Test Case Two

2. **On-Line Posting (if any):** None. Testing purposes only.

3. **Brief Description of the Purpose and Background of the Case Study:**

VTADL source file, "Test2.txt," contains two architectures, but with more intricate components and connections than the first test. We aim, in this test, to selectively show or hide components and connectors, to insure that single or multiple architectures can be displayed in a view, that multiple versions of the same architecture can be established in a view, and that hyperlinks from elements of an architecture can be established either to other views or to external files.

The Main view contains two architectures, "TestArch1" (a call-and-return style with thirteen components) and "TestArch2" (a layered style with six layers). TestArch1 contains a hyperlink to the source file, "Test2.txt," from the root. TestArch1 also contains a hyperlink from one of the children (component "D") to a view, "TestView1." TestArch2 contains a hyperlink to the view, "TestView2," from the fourth layer.

The second view, TestView2, contains only one architecture (TestArch1) but with selected components shown and others hidden. TestView2 contains a hyperlink from the root to "TestView3."

TestView1 contains only one architecture, TestArch2, with the third and fourth layers hidden from view. A hyperlink exists from the top layer to the view, TestView2.

TestView3 contains four architectural structures: three versions of TestArch1 (with different components and connectors kept hidden in each version), and one version of TestArch2. The version of TestArch2 has the second and fourth layers hidden, and the other layers visible.

4. **Name of VTADL Source File(s):**

Source File	**View Files**	**Hyperlinked Files (Referenced)**
Test2.txt	Main.wrl	Test2.txt
	TestView1.wrl	TestView1.wrl
	TestView2.wrl	TestView2.wrl
	TestView3.wrl	TestView3.wrl

5. VTADL Source Code Listing

The VTADL source file, Test2.txt, is listed below:

```
Architecture TestArch1
Style Program
  {
     ComponentList
        {
             Component A
               ComponentType Cprogram;
               Properties:
                 CompRole:  CompConsumer;
                 ChildOf: ;
                 Layer: ;
                 Process: ;
               InterfaceList:
                Interface Bottom ASocket;
                  { InterfaceRole: Consumer;  }

             Component B
               ComponentType Cprogram;
               Properties:
                   CompRole: CompProducer;
                   ChildOf: ;
                   Layer: ;
                   Process:  ;
                   InterfaceList:
                   Interface Top BSocket;
                   { InterfaceRole: Producer;  }

              Component C
               ComponentType Cprogram;
               Properties:
                   CompRole: CompProducer;
                   ChildOf: ;
                   Layer: ;
                   Process: ;
                   InterfaceList:
                   Interface Top CSocket;
                   { InterfaceRole: Producer; }
```

```
Component D
 ComponentType Cprogram;
 Properties:
      CompRole: CompProducer;
      ChildOf: ;
      Layer: ;
      Process: ;
      InterfaceList:
      Interface Top DSocket;
      { InterfaceRole: Producer; }


Component E
 ComponentType Cprogram;
 Properties:
      CompRole: CompProducer;
      ChildOf: ;
      Layer: ;
      Process: ;
      InterfaceList:
      Interface Top ESocket;
      { InterfaceRole: Producer; }


Component F
 ComponentType Cprogram;
 Properties:
      CompRole: CompProducer;
      ChildOf: ;
      Layer: ;
      Process: ;
      InterfaceList:
      Interface Top FSocket;
      { InterfaceRole: Producer; }

Component G
 ComponentType Cprogram;
 Properties:
      CompRole: CompProducer;
      ChildOf: ;
      Layer: ;
      Process: ;
      InterfaceList:
      Interface Top GSocket;
      { InterfaceRole: Producer; }
```

```
Component H
 ComponentType Cprogram;
 Properties:
     CompRole: CompProducer;
     ChildOf: ;
     Layer: ;
     Process: ;
     InterfaceList:
     Interface Top HSocket;
     { InterfaceRole: Producer; }

Component I
 ComponentType Cprogram;
 Properties:
     CompRole: CompProducer;
     ChildOf: ;
     Layer: ;
     Process: ;
     InterfaceList:
     Interface Top ISocket;
     { InterfaceRole: Producer; }

 Component J
 ComponentType Cprogram;
 Properties:
     CompRole: CompProducer;
     ChildOf: ;
     Layer: ;
     Process: ;
     InterfaceList:
     Interface Top JSocket;
     { InterfaceRole: Producer; }

Component K
 ComponentType Cprogram;
 Properties:
     CompRole: CompProducer;
     ChildOf: ;
     Layer: ;
     Process: ;
     InterfaceList:
     Interface Top KSocket;
     { InterfaceRole: Producer; }

Component L
 ComponentType Cprogram;
```

```
        Properties:
            CompRole: CompProducer;
            ChildOf: ;
            Layer: ;
            Process: ;
            InterfaceList:
            Interface Top LSocket;
            { InterfaceRole: Producer; }

      Component M
        ComponentType Cprogram;
        Properties:
            CompRole: CompProducer;
            ChildOf: ;
            Layer: ;
            Process: ;
            InterfaceList:
            Interface Top MSocket;
            { InterfaceRole: Producer; }
    }

ConnectionList
    {
      Connector BCall
        ConnectType Controlflow Unidirect;
        Connect(ASocket, BSocket);
      Connector CCall
        ConnectType Controlflow Unidirect;
        Connect(ASocket, CSocket);
      Connector DCall
        ConnectType Controlflow Unidirect;
        Connect(ASocket, DSocket);
            Connector ECall
        ConnectType Controlflow Unidirect;
        Connect(BSocket, ESocket);
      Connector FCall
        ConnectType Controlflow Unidirect;
        Connect(BSocket, FSocket);
      Connector GCall
        ConnectType Controlflow Unidirect;
        Connect(BSocket, GSocket);
            Connector HCall
        ConnectType Controlflow Unidirect;
        Connect(DSocket, HSocket);
      Connector ICall
        ConnectType Controlflow Unidirect;
```

```
                Connect(DSocket, ISocket);
             Connector JCall
                ConnectType Controlflow Unidirect;
                Connect(DSocket, JSocket);
            Connector KCall
                ConnectType Controlflow Unidirect;
                Connect(DSocket, KSocket);
             Connector LCall
                ConnectType Controlflow Unidirect;
                Connect(KSocket, LSocket);
             Connector MCall
                ConnectType Controlflow Unidirect;
                Connect(KSocket, MSocket);
        }

 }

Architecture TestArch2
Style Layer
 {
   ComponentList
     {
      Component One
        ComponentType Cprogram;
        Properties:
         CompRole:  CompProducer;
         ChildOf: ;
         Layer: L1;
         Process: ;
        InterfaceList:
          Interface Bottom Level1Socket;
            { InterfaceRole: Producer; }

       Component Two
        ComponentType Cprogram;
        Properties:
        CompRole: CompConsumer;
        ChildOf: ;
        Layer: L2;
        Process: ;
        InterfaceList:
          Interface Top Level2Socket;
            { InterfaceRole: Consumer; }

       Component Three
        ComponentType Cprogram;
```

```
    Properties:
    CompRole: CompConsumer;
    ChildOf: ;
    Layer: L3;
    Process: ;
    InterfaceList:
      Interface Top Level3Socket;
         { InterfaceRole: Consumer; }

    Component Four
     ComponentType Cprogram;
     Properties:
      CompRole: CompConsumer;
      ChildOf: ;
      Layer: L4;
      Process: ;
      InterfaceList:
        Interface Top Level4Socket;
           { InterfaceRole: Consumer; }

   Component Five
     ComponentType Cprogram;
     Properties:
      CompRole: CompConsumer;
      ChildOf: ;
      Layer: L4;
      Process: ;
      InterfaceList:
        Interface Top Level5Socket;
           { InterfaceRole: Consumer; }

   Component Six
     ComponentType Cprogram;
     Properties:
      CompRole: CompConsumer;
      ChildOf: ;
      Layer: L4;
      Process: ;
      InterfaceList:
        Interface Top Level6Socket;
           { InterfaceRole: Consumer; }
}
ConnectionList
 {
   Connector Service1
     ConnectType Dataflow Unidirect;
```

```
        Connect(Level1Socket,Level2Socket);

      Connector Service2
        ConnectType Dataflow Unidirect;
        Connect(Level2Socket,Level3Socket);

      Connector Service3
        ConnectType Dataflow Unidirect;
        Connect(Level3Socket,Level4Socket);

        Connector Service4
          ConnectType Dataflow Unidirect;
          Connect(Level4Socket,Level5Socket);

        Connector Service5
          ConnectType Dataflow Unidirect;
          Connect(Level5Socket,Level6Socket);

       }
}

   ViewList
      {
       ViewMain  {  { UsingArch TestArch1;
                   { Components All;
                     Connections All;
                     HyperLinkOn A
                              ToFile "Test2.txt";
                     HyperLinkOn D
                              ToFile TestView1; }}
                        { UsingArch TestArch2;
                              {  Components All;
                                 Connections All;
                                 HyperLinkOn Four
                                 ToFile TestView2; }}
                   }


          View TestView1 {  { UsingArch TestArch2;
                   { Components One Two Five Six;
                     Connections All;
                              HyperLinkOn Six ToFile TestView2; }}
                 }


        View TestView2 { { UsingArch TestArch1;
```

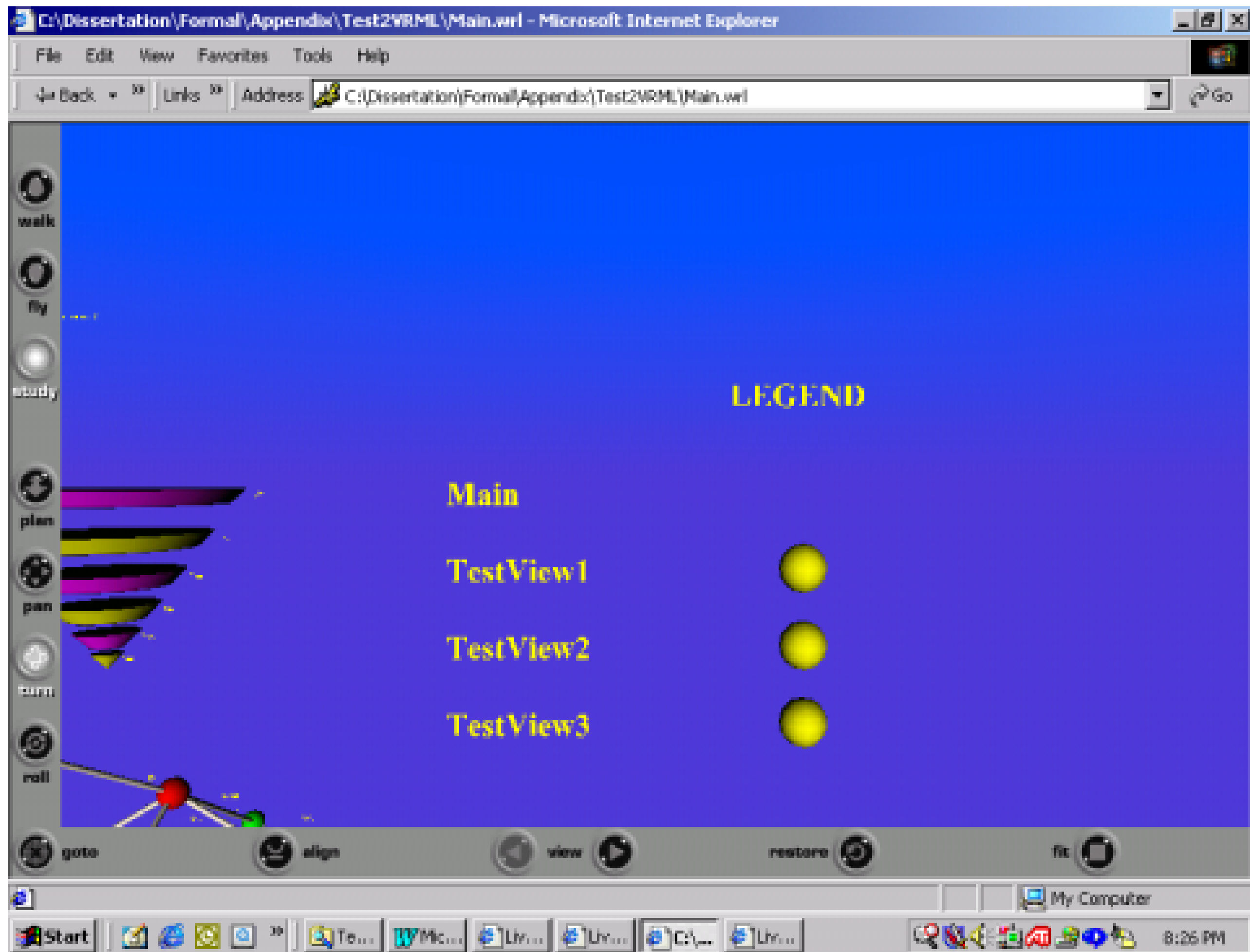
```
            { Components A B C D E F K L M;
              Connections BCall CCall DCall ECall
                FCall KCall LCall MCall;
              HyperLinkOn A
                ToFile TestView3; }}
        }


    View TestView3 {
          { UsingArch TestArch1;
            { Components A C D K L M;
              Connections CCall DCall
               KCall LCall MCall; }}
          { UsingArch TestArch1;
            { Components A B C D;
              Connections BCall CCall; }}
          { UsingArch TestArch1;
            { Components A B D G H I J;
              Connections BCall GCall HCall
               ICall JCall; }}
          { UsingArch TestArch2;
             { Components One Three Five Six;
               Connections All; }}
        }
    }
$
```

6. VRML Target Code

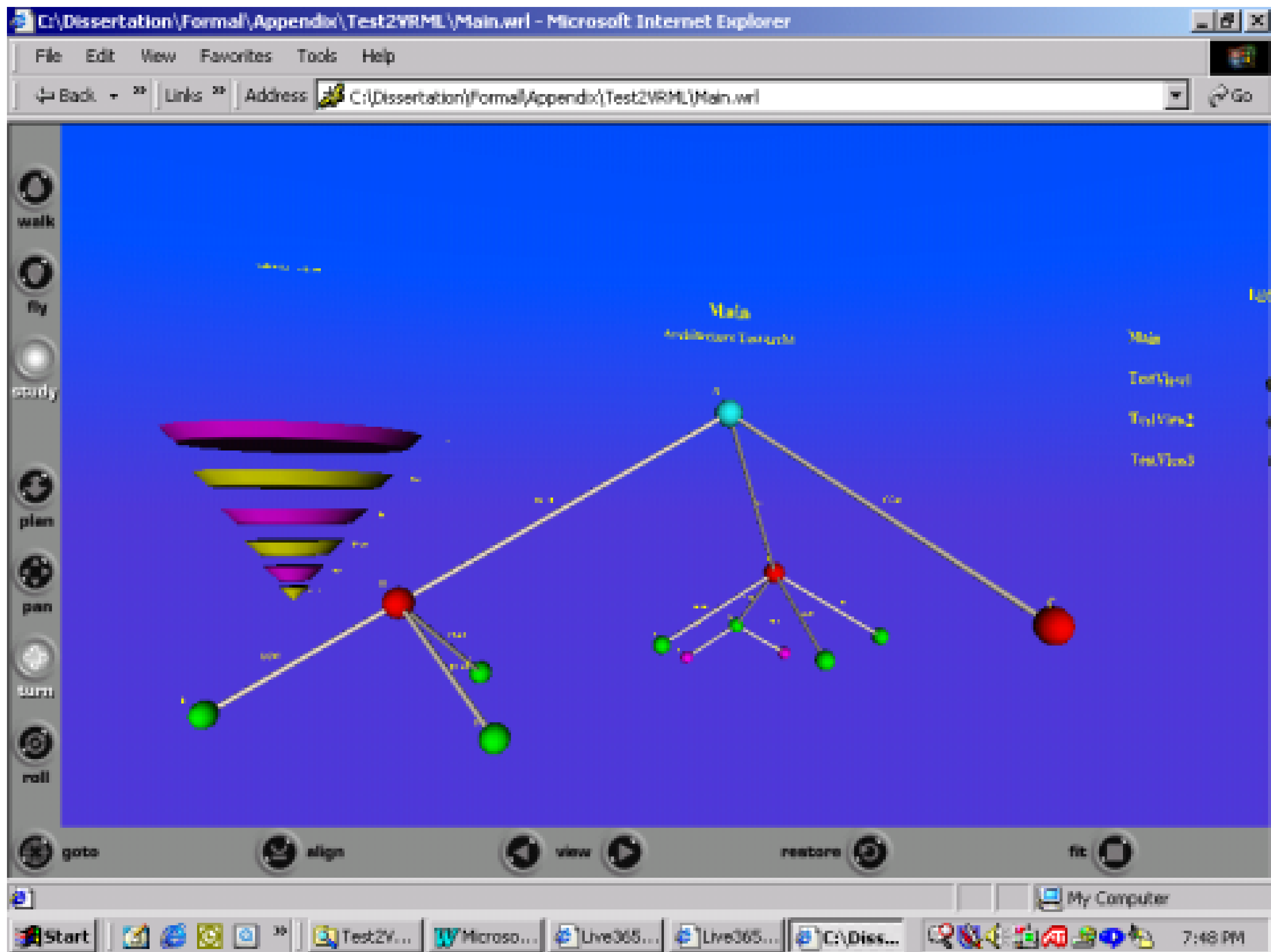
(Not included for sake of brevity).

7. Screen Snapshots of VRML Images

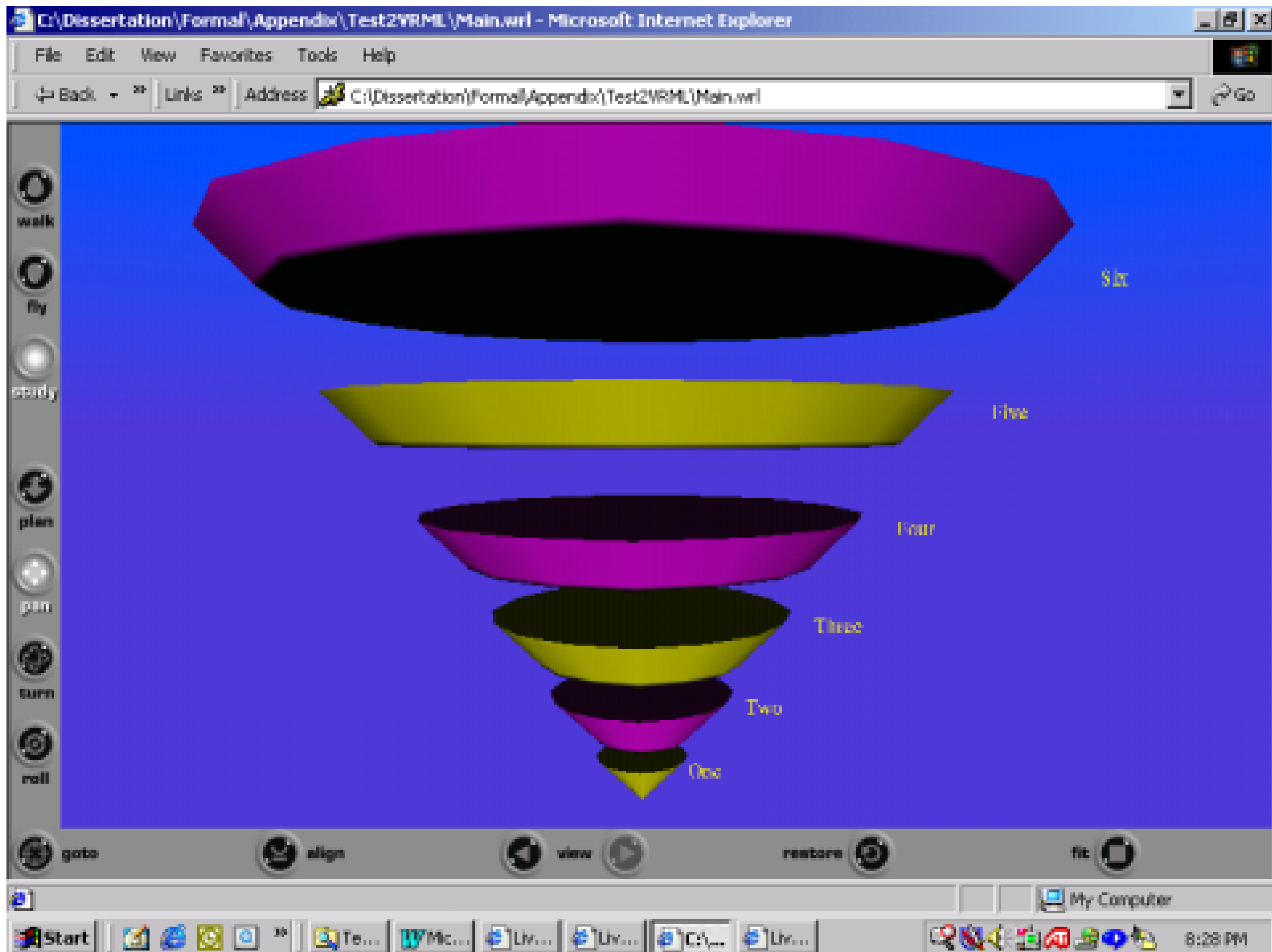
Test2 contains four views. The legend from the Main view is given below:



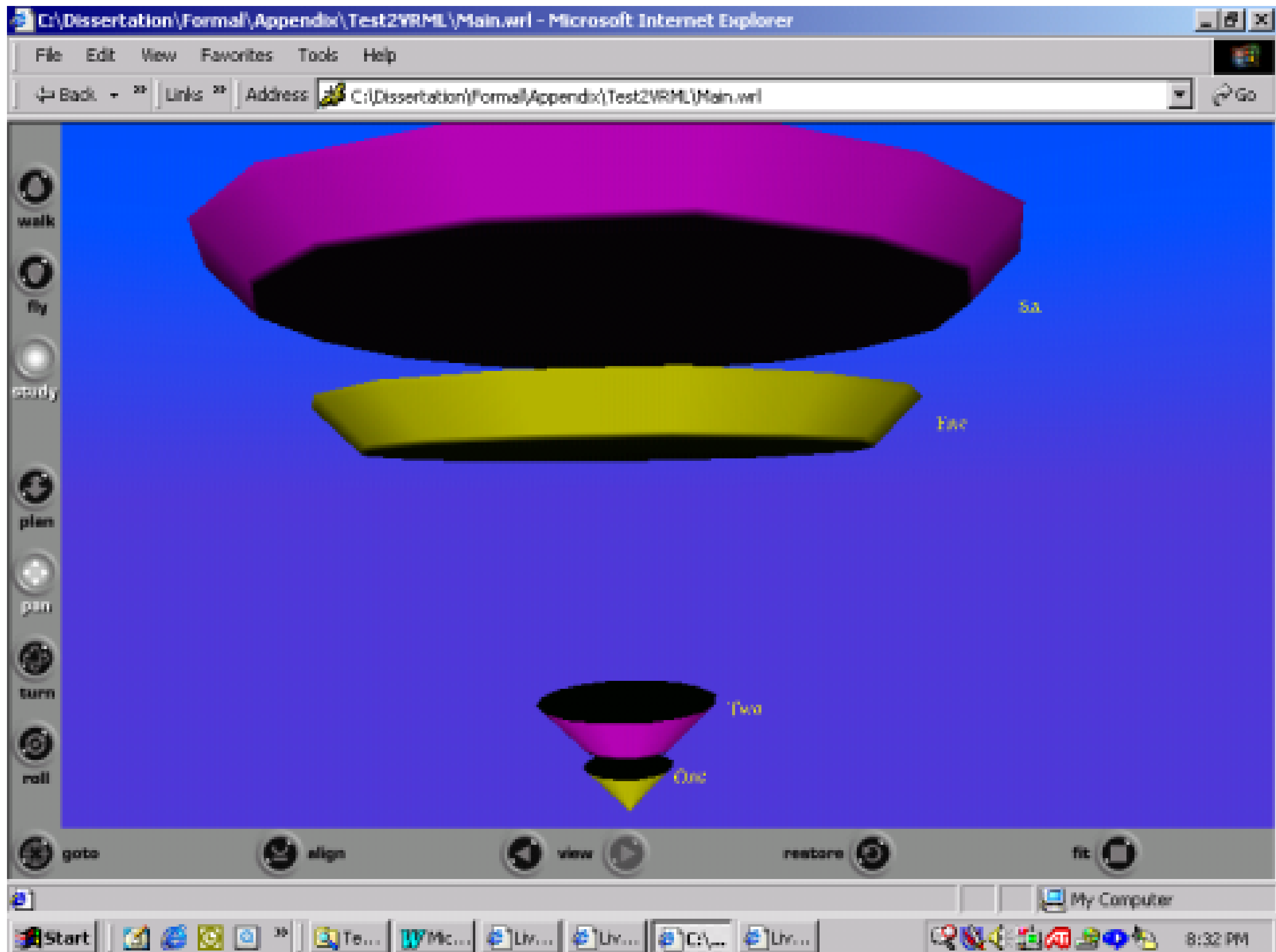
The following screen shows the Main view, with architecture TestArch1 in the foreground, and the six-layered architecture, TestArch2, in the background. The root of TestArch1 contains a hyperlink to the source VTADL file, Test2.txt. A hyperlink is also established from the component D to the view, TestView1.



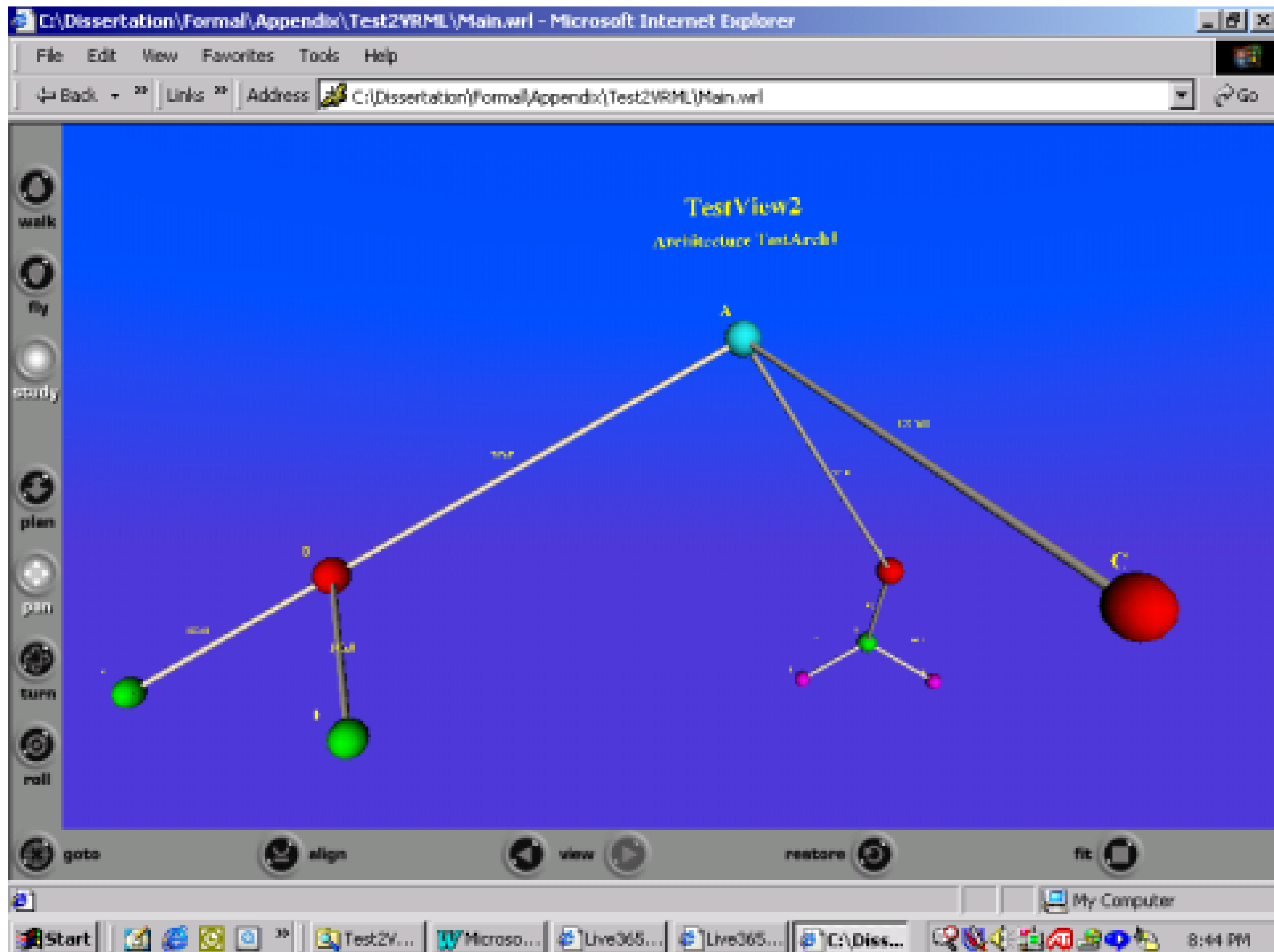
The focus of the following screen is the second architecture in the Main view, TestArch2. The fourth layer of the architecture contains a hyperlink to TestView2.



The next view, TestView1, contains the layered architecture, TestArch2. However, layers three and four are hidden. A hyperlink is established from layer six to TestView2.

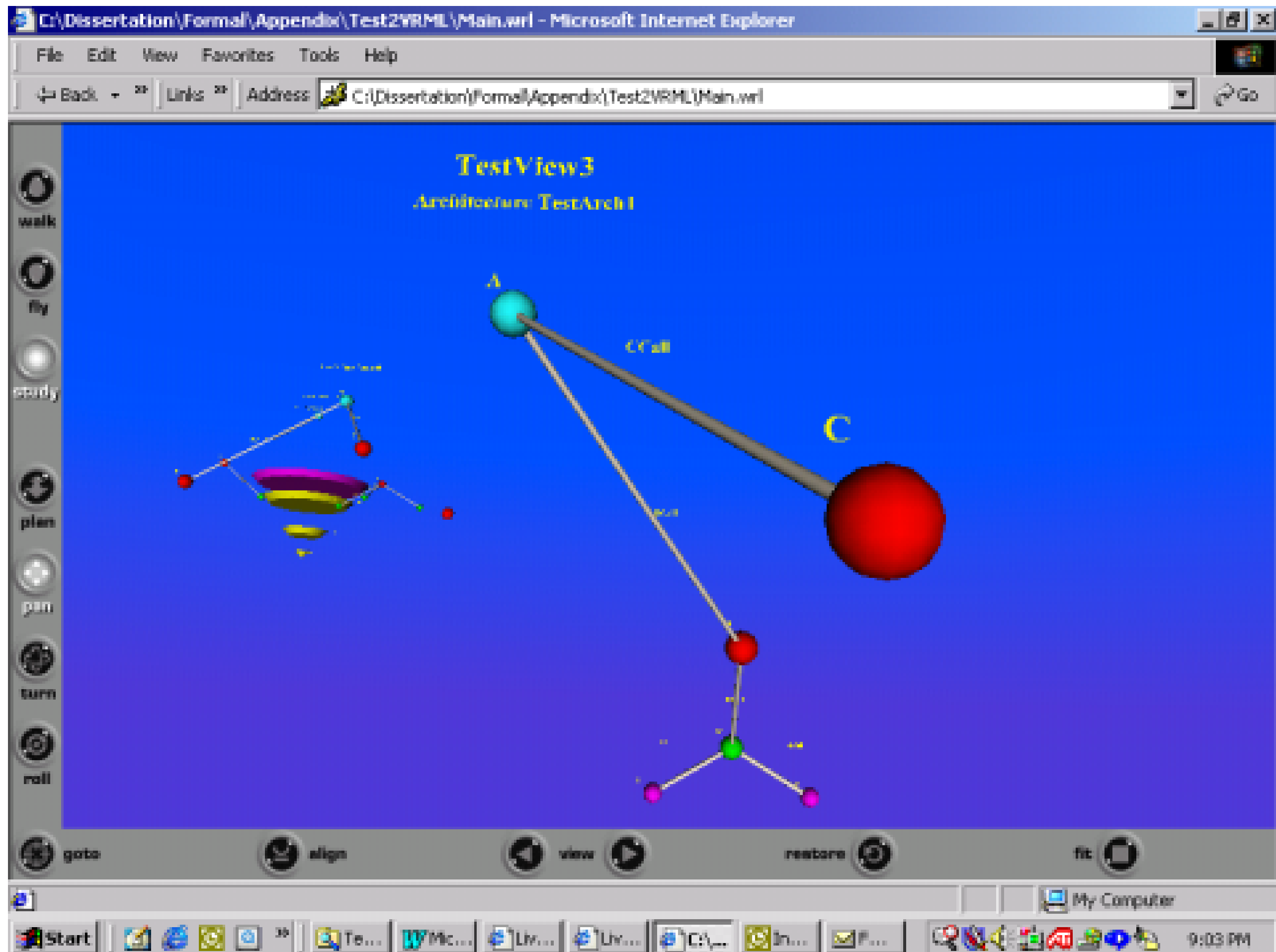


The view, TestView2, contains a modified version of TestArch1. Several nodes on the left and right subtrees of architecture TestArch1 are omitted. Only nodes A, B, C, D, E, F, K, L, and M are shown. A hyperlink is established from the root (node A) to TestView3.

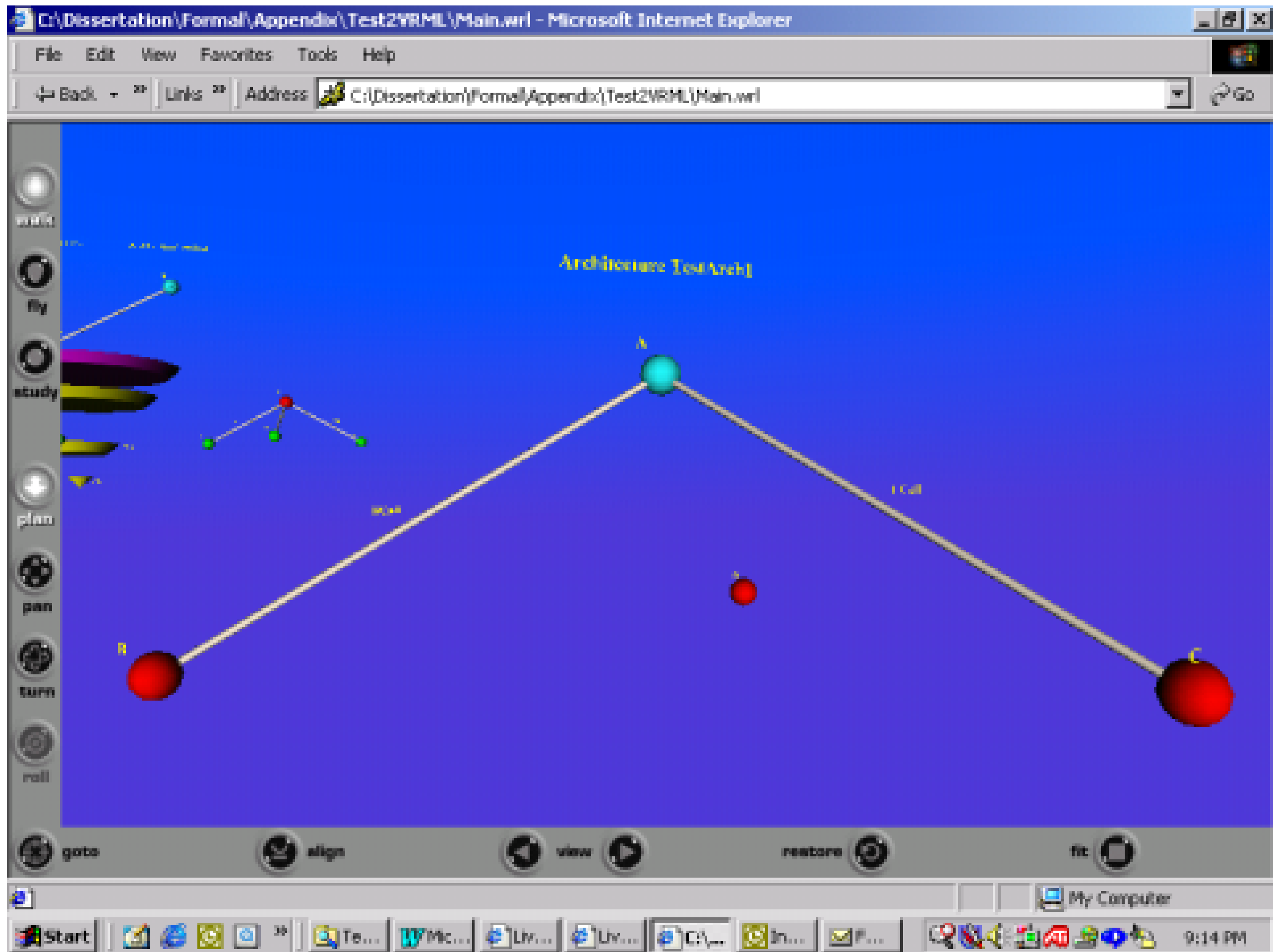


TestView3 is a view that contains three versions of TestArch1 (the call-and-return architecture), and a single version of TestArch2 (the layered architecture). This view demonstrates multiple versions of the same architecture with both components and connections kept hidden.

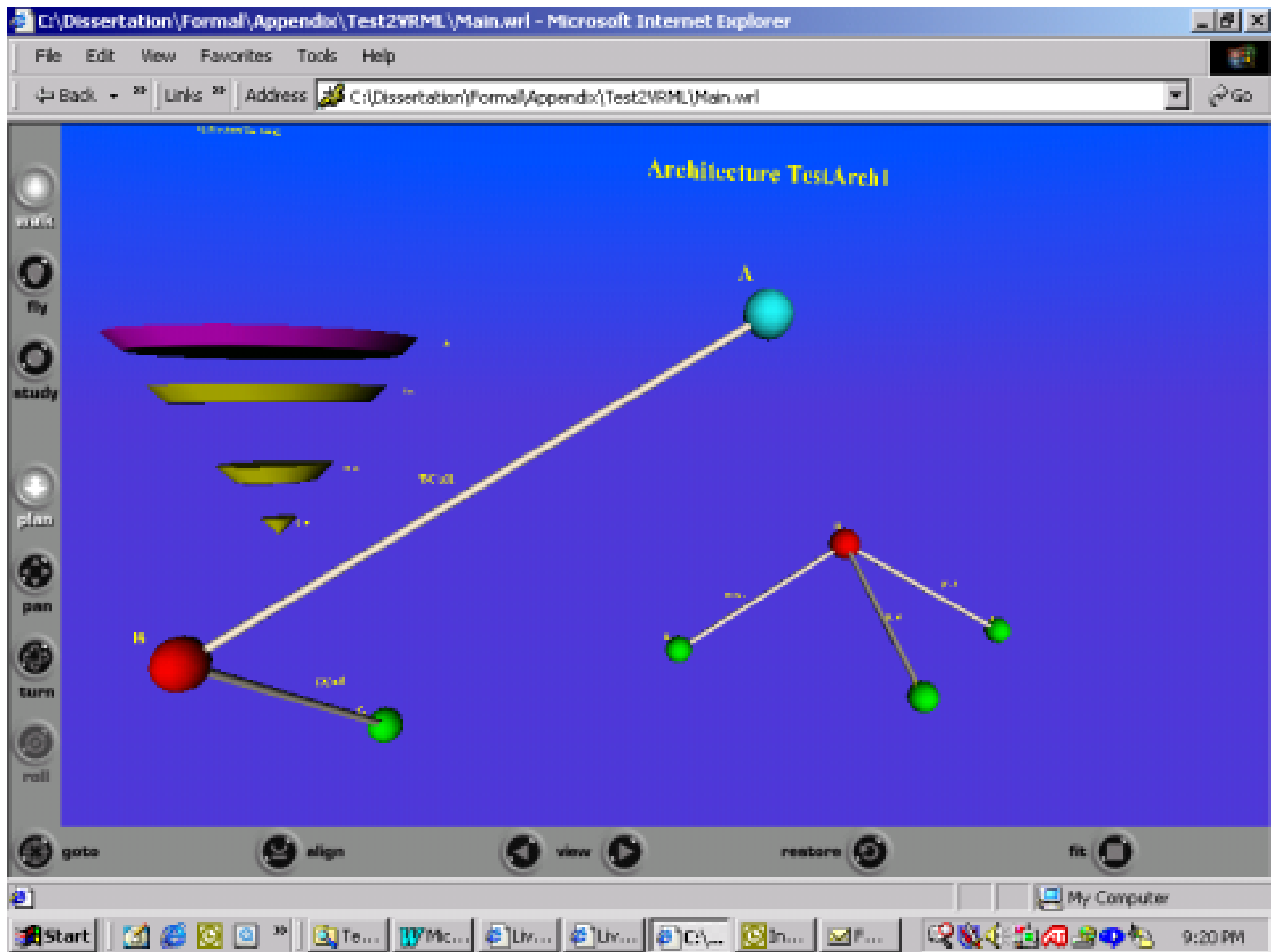
The following screen snapshot displays a version of TestArch1 in the foreground, with the three other structures in the background. The version of TestArch1 in the foreground uses only components A, C, D, K, L, and M, but with the pertinent connectors displayed.



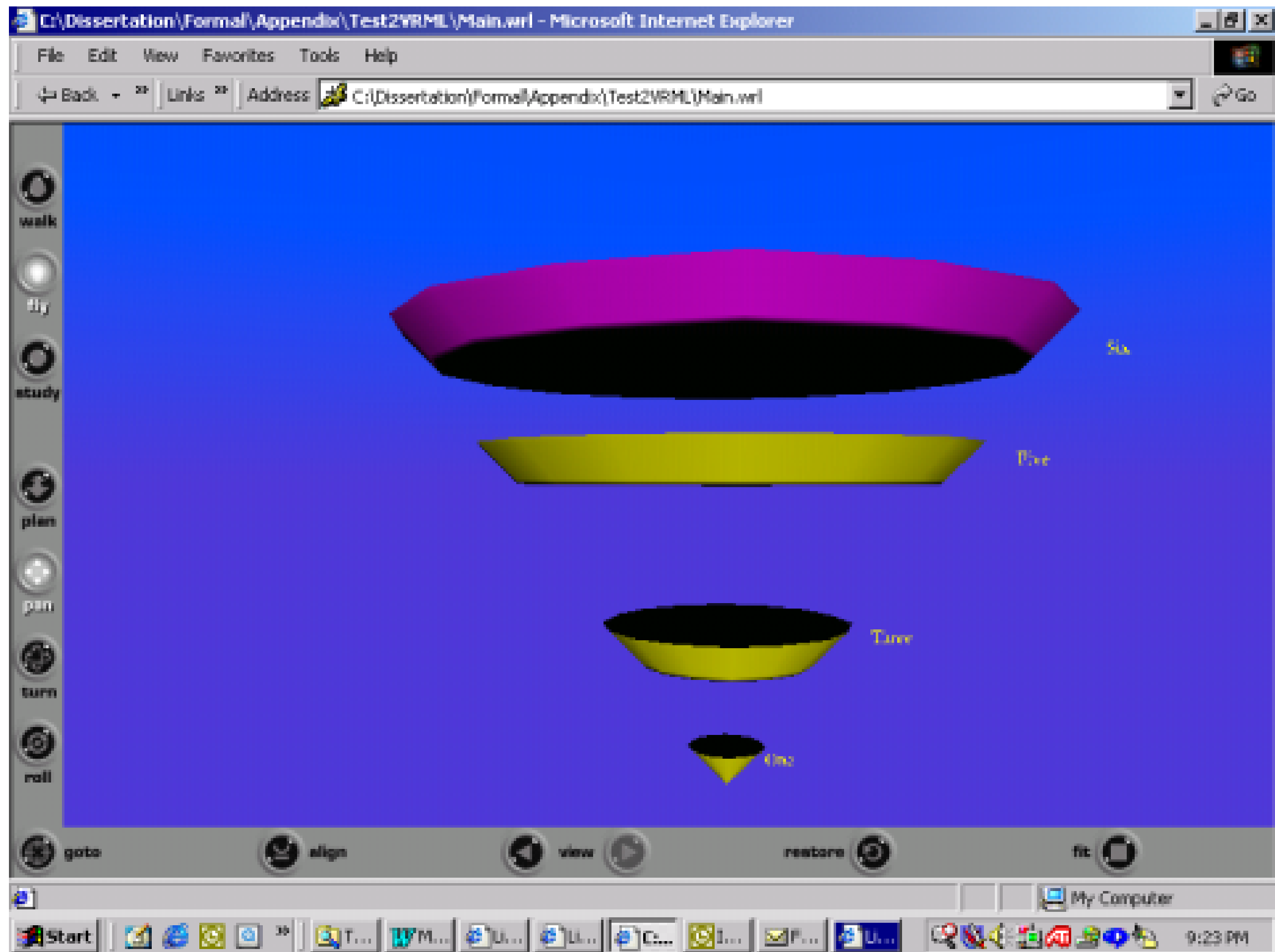
For the second version of TestArch1, nodes A, B, C, and D are shown, but the connector between A and D is hidden. Node D is isolated as an "orphan" component in the distance.



The third version of architecture TestArch1 consists of nodes A, B, G, D, H, I, and J. However, the connector between A and D is eliminated, resulting in two sub-architectures. The first sub-architecture consists of nodes A, B, and G; the second sub-architecture consists of nodes D, H, I, and J, shown as follows:



The layered architecture of TestArch2 is shown. The second and fourth layers are hidden:



Appendix H

Case Study Report Template

Case Study Report Template

1. **Name of Case Study:** (Supply title of case study here)

2. **On-Line Posting (if any):** (Indicate url of any posting)

3. **Brief Description of the Purpose and Background of the Case Study:**

   (Describe what the case study was about, any previous history in two-dimensional form, etc.)

4. **Name of VTADL Source File(s):**

Source File	**View Files**	**Hyperlinked Files (Referenced)**
(VTADL)	(VRML)	(if any)

5. **VTADL Source Code Listing**

   (Attach when applicable).

6. **VRML Target Code**

   (Optional; used when necessary)

7. **Screen Snapshots of VRML Images**

   (Provide representative screen shots).

Appendix I

Case Studies

Case Study One: Views on a Mobile Robot Architecture

Case Study Report

1. **Name of Case Study:** Views on a Mobile Robot Architecture

2. **On-Line Posting (if any):**
http://www.nova.edu/~inouyej/Dissertation/CaseStudies.

3. **Brief Description of the Purpose and Background of the Case Study:**

This case study re-represents three different solutions to the problem of modeling the software architecture to control a mobile robot. Each solution is represented in VTADL, then translated to the views in VRML.

The requirements for a mobile robot were described on page 43 in *Software Architecture: Perspectives on an Emerging Discipline* [Shaw & Garlan, 1996]. We encapsulate the requirements here. The architecture of the mobile robot must be designed so that the robot can successfully respond to stimuli from the environment while yet taking actions towards a goal. The architecture must allow for uncertainty in information, danger to the system (e.g., lowered power supply), and design flexibility for future requirements.

Solution 1, as described by Shaw and Garlan, used a control loop architecture. Feedback from the environment was fed to a sensor component, which supplied data to a controller. Based on the sensor data, the controller would direct action of the mobile robot by means of an actuator component; the actuator component would direct action of the mobile robot within the environment (thus completing the feedback loop). The fundamental benefit of Solution 1 was its simplicity, capturing the essence of the robot's interaction with the environment. The disadvantage to the control loop was the failure to model tasks requiring complex decomposition.

Solution 2 used a layered architecture. The architecture, in the words of the authors, "nicely organizes the components needed to coordinate the robot's operation." The disadvantage was that information often did not truly pass between adjacent layers as the layered model would imply.

Solution 3 used an implicit invocation architecture called TCA (for Task-Control Architecture). By "implicit invocation" was meant that a process could be invoked by the occurrence of an event, with the constraint that the invoked processes do not interact with one another. TCA instantiated a hierarchy of tasks called a task tree. TCA had the capability to reconfigure task trees during execution in response to changing robot states and environment. The advantage to this model was that the model provided a clear-cut separation of action; replacement of components in such a modularized architecture was straightforward. However, the drawback to this architecture was that the architecture did not model how the robot would handle uncertain conditions in the environment.

Solution 4 used a blackboard architecture, but we did not model this architecture since the blackboard architectural style was not implemented in the compiler.

The three solutions (Solution 1, 2, and 3) were first represented using VTADL prior to visualization. The VTADL representation was then translated into one or more VRML files using the compiler tool developed by this dissertation.

4. Name of VTADL Source File(s):

Source File	**View Files**	**Hyperlinked Files (Referenced)**
MobileRobot.txt	Main.wrl	ActUponEnviron.html
	Solution1.wrl	Main.wrl
	Solution2.wrl	MobileRobot.txt
	Solution3.wrl	MobileRobotHTML.html
		RunActuator.html
		SenseData.html
		Solution1.wrl
		Solution2.wrl
		Solution3.wrl

5. VTADL Source Code Listing ("MobileRobot.txt")

The VTADL source code for the Mobile Robot case study is provided below.

```
Architecture LayerSolutionOne
Style Layer
  {
      ComponentList
          {
              Component Environment
                ComponentType Cprogram;
                Properties:
                  CompRole:  CompProducer;
                  ChildOf: ;
                  Layer: e1;
                  Process: ;
                InterfaceList:
                  Interface Top EnviroData;
                    { InterfaceRole: Producer;  }

              Component ActiveComponent
                ComponentType Cprogram;
                Properties:
                    CompRole: CompProducer;
                    ChildOf: ;
                    Layer: e2;
                    Process:  ;
                    InterfaceList:
                    Interface Bottom LoopFromEnviron;
                    { InterfaceRole: Consumer;  }

              Component Controller
                ComponentType Cprogram;
                Properties:
                    CompRole: CompProducer;
                    ChildOf: ;
                    Layer: e3;
                    Process: ;
                    InterfaceList:
                    Interface Bottom ControlData;
                    { InterfaceRole: Consumer; }
          }
      ConnectionList
          {
             Connector SensorData
```

```
          ConnectType Dataflow Bidirect;
          Connect(EnviroData, LoopFromEnviron);
        Connector DataControl
          ConnectType Dataflow Bidirect;
          Connect(LoopFromEnviron, ControlData);
      }
  }

Architecture SenseSolutionOne
Style Program
  {
     ComponentList
       {
          Component ActiveComponent
           ComponentType Cprogram;
           Properties:
             CompRole:  CompProducer;
             ChildOf:  ;
             Layer: x1;
             Process: ;
           InterfaceList:
            Interface Bottom ActiveSocket;
              { InterfaceRole: Producer;  }

          Component Actuators
           ComponentType Cprogram;
           Properties:
               CompRole: CompProducer;
               ChildOf: ;
               Layer: x2;
               Process:  ;
               InterfaceList:
               Interface Top ConnActuator;
               { InterfaceRole: Producer;  }

          Component Sensors
           ComponentType Cprogram;
           Properties:
               CompRole: CompConsumer;
               ChildOf: ;
               Layer: x3;
               Process: ;
               InterfaceList:
               Interface Top SocketSense;
               { InterfaceRole: Consumer; }
```

```
                Component Environment
                 ComponentType Cprogram;
                 Properties:
                     CompRole: CompConsumer;
                     ChildOf: ;
                     Layer: x4;
                     Process: ;
                     InterfaceList:
                     Interface Top EnvConn;
                     { InterfaceRole: Consumer; }
             }

        ConnectionList
             {
                Connector RunActuator
                 ConnectType Dataflow Unidirect;
                 Connect(ActiveSocket, ConnActuator);
                Connector SenseData
                 ConnectType Dataflow Unidirect;
                 Connect(ActiveSocket, SocketSense);
                Connector DetectEnv
                 ConnectType Dataflow Unidirect;
                 Connect(SocketSense, EnvConn);
             }
   }

Architecture ActSolutionOne
Style Program
   {
        ComponentList
             {
                Component ActiveComponent
                 ComponentType Cprogram;
                 Properties:
                   CompRole:  CompProducer;
                   ChildOf:  ;
                   Layer: p1;
                   Process: ;
                 InterfaceList:
                   Interface Bottom ActiveSocket;
                     { InterfaceRole: Producer;  }

                Component Actuators
                 ComponentType Cprogram;
                 Properties:
                     CompRole: CompProducer;
```

```
                ChildOf: ;
                Layer: p2;
                Process:  ;
                InterfaceList:
                Interface Top ConnActuator;
                { InterfaceRole: Producer;  }

            Component Sensors
             ComponentType Cprogram;
             Properties:
                CompRole: CompConsumer;
                ChildOf: ;
                Layer: p3;
                Process: ;
                InterfaceList:
                Interface Top SocketSense;
                { InterfaceRole: Consumer; }

           Component Environment
             ComponentType Cprogram;
             Properties:
                CompRole: CompConsumer;
                ChildOf:  ;
                Layer: p4;
                Process: ;
                InterfaceList:
                Interface Top EnvConn;
                { InterfaceRole: Consumer; }
        }

    ConnectionList
        {
           Connector RunActuator
             ConnectType Dataflow Unidirect;
             Connect(ActiveSocket, ConnActuator);
           Connector SenseData
             ConnectType Dataflow Unidirect;
             Connect(ActiveSocket, SocketSense);
           Connector ActUponEnviron
             ConnectType Dataflow Unidirect;
             Connect(ConnActuator, EnvConn);
        }
 }

Architecture SolutionTwo
Style Layer
```

```
{
  ComponentList
     {
       Component Environment
        ComponentType Cprogram;
        Properties:
          CompRole:  CompProducer;
          ChildOf: ;
          Layer: e1;
          Process: ;
        InterfaceList:
         Interface Top EN1;
           { InterfaceRole: Producer;  }

       Component RobotControl
        ComponentType Cprogram;
        Properties:
           CompRole: CompProducer;
           ChildOf: ;
           Layer: e2;
           Process:  ;
           InterfaceList:
           Interface Bottom RC2;
           { InterfaceRole: Consumer;  }

       Component SensorInterpret
        ComponentType Cprogram;
        Properties:
           CompRole: CompProducer;
           ChildOf: ;
           Layer: e3;
           Process: ;
           InterfaceList:
           Interface Bottom SR3;
           { InterfaceRole: Consumer; }

       Component SenseIntegrate
        ComponentType Cprogram;
        Properties:
           CompRole: CompProducer;
           ChildOf: ;
           Layer: e3;
           Process: ;
           InterfaceList:
           Interface Bottom SI4;
           { InterfaceRole: Consumer; }
```

```
        Component RealWorldModel
         ComponentType Cprogram;
         Properties:
             CompRole: CompProducer;
             ChildOf: ;
             Layer: e3;
             Process: ;
             InterfaceList:
             Interface Bottom Real5;
             { InterfaceRole: Consumer; }

        Component Navigation
         ComponentType Cprogram;
         Properties:
             CompRole: CompProducer;
             ChildOf: ;
             Layer: e3;
             Process: ;
             InterfaceList:
             Interface Bottom Nav6;
             { InterfaceRole: Consumer; }

        Component GlobalPlan
         ComponentType Cprogram;
         Properties:
             CompRole: CompProducer;
             ChildOf: ;
             Layer: e3;
             Process: ;
             InterfaceList:
             Interface Bottom GlobalSocket;
             { InterfaceRole: Consumer; }

        Component Supervisor
         ComponentType Cprogram;
         Properties:
             CompRole: CompProducer;
             ChildOf: ;
             Layer: e3;
             Process: ;
             InterfaceList:
             Interface Bottom SupSocket;
             { InterfaceRole: Consumer; }
    }
ConnectionList
```

```
        {
          Connector OneConn
            ConnectType Dataflow Bidirect;
            Connect(EN1,RC2);
          Connector TwoConn
            ConnectType Dataflow Bidirect;
            Connect(RC2,SR3);
           Connector ThreeConn
            ConnectType Dataflow Bidirect;
            Connect(SR3,SI4);
          Connector FourConn
            ConnectType Dataflow Bidirect;
            Connect(SI4,Real5);
          Connector FiveConn
            ConnectType Dataflow Bidirect;
            Connect(Real5, Nav6);
          Connector SixConn
            ConnectType Dataflow Bidirect;
            Connect(Nav6,GlobalSocket);
          Connector SevenConn
            ConnectType Dataflow Bidirect;
            Connect(GlobalSocket,SupSock);
        }
  }

Architecture ImplicitInvoke
Style Program
  {
     ComponentList
        {
           Component Task
            ComponentType Cprogram;
            Properties:
              CompRole:  CompProducer;
              ChildOf:  ;
              Layer: p1;
              Process: ;
            InterfaceList:
             Interface Bottom TaskSocket;
               { InterfaceRole: Producer;  }

           Component ExceptTask
            ComponentType Cprogram;
            Properties:
               CompRole: CompProducer;
               ChildOf: ;
```

```
      Layer: p2;
      Process:  ;
      InterfaceList:
      Interface Top ExceptSocket;
      { InterfaceRole: Producer;  }

 Component Ether
  ComponentType Cprogram;
  Properties:
      CompRole: CompConsumer;
      ChildOf: ;
      Layer: p3;
      Process: ;
      InterfaceList:
      Interface Top EthSocket;
      { InterfaceRole: Consumer; }

Component DispatchTask
  ComponentType Cprogram;
  Properties:
      CompRole: CompConsumer;
      ChildOf:  ;
      Layer: p4;
      Process: ;
      InterfaceList:
      Interface Top DispSocket;
      { InterfaceRole: Consumer; }

Component WiredTask
  ComponentType Cprogram;
  Properties:
      CompRole: CompConsumer;
      ChildOf:  ;
      Layer: p4;
      Process: ;
      InterfaceList:
      Interface Top WiretapSocket;
      { InterfaceRole: Consumer; }

Component TappedTask
  ComponentType Cprogram;
  Properties:
      CompRole: CompConsumer;
      ChildOf:  ;
      Layer: p4;
      Process: ;
```

```
                    InterfaceList:
                    Interface Top TapSocket;
                    { InterfaceRole: Consumer; }
            }
        ConnectionList
            {
               Connector Exception
                 ConnectType Dataflow Unidirect;
                 Connect(TaskSocket, ExceptSocket);
               Connector Message
                 ConnectType Dataflow Unidirect;
                 Connect(TaskSocket, EthSocket);
               Connector Dispatched
                 ConnectType Dataflow Unidirect;
                 Connect(EthSocket, DispSocket);
                Connector Tapped
                 ConnectType Dataflow Unidirect;
                 Connect(EthSocket, TapSocket);
                Connector WireTapped
                 ConnectType Dataflow Unidirect;
                 Connect(EthSocket, WiretapSocket);
            }
  }

Architecture TaskTree
Style Program
  {
        ComponentList
            {
                Component GatherRock
                 ComponentType Cprogram;
                 Properties:
                   CompRole:  CompProducer;
                   ChildOf:  ;
                   Layer: p1;
                   Process: ;
                 InterfaceList:
                  Interface Bottom GatherSocket;
                    { InterfaceRole: Producer;  }

                Component GotoPosition
                 ComponentType Cprogram;
                 Properties:
                     CompRole: CompProducer;
                     ChildOf: ;
                     Layer: p2;
```

```
        Process:  ;
        InterfaceList:
        Interface Top PositionSocket;
        { InterfaceRole: Producer;  }

 Component GrabRock
  ComponentType Cprogram;
  Properties:
        CompRole: CompConsumer;
        ChildOf: ;
        Layer: p3;
        Process: ;
        InterfaceList:
        Interface Top GrbSocket;
        { InterfaceRole: Consumer; }

Component LiftRock
  ComponentType Cprogram;
  Properties:
        CompRole: CompConsumer;
        ChildOf:  ;
        Layer: p4;
        Process: ;
        InterfaceList:
        Interface Top LiftSocket;
        { InterfaceRole: Consumer; }

Component MoveLeft
  ComponentType Cprogram;
  Properties:
        CompRole: CompConsumer;
        ChildOf:  ;
        Layer: p4;
        Process: ;
        InterfaceList:
        Interface Top LeftMoveSocket;
        { InterfaceRole: Consumer; }

Component MoveForward
  ComponentType Cprogram;
  Properties:
        CompRole: CompConsumer;
        ChildOf:  ;
        Layer: p4;
        Process: ;
        InterfaceList:
```

```
                Interface Top FwdMoveSocket;
                { InterfaceRole: Consumer; }
        }
    ConnectionList
        {
          Connector PositCall
            ConnectType Dataflow Unidirect;
            Connect(GatherSocket, PositionSocket);
          Connector GrabCall
            ConnectType Dataflow Unidirect;
            Connect(GatherSocket, GrbSocket);
          Connector LiftCall
            ConnectType Dataflow Unidirect;
            Connect(GatherSocket, LiftSocket);
           Connector LeftCallMove
            ConnectType Dataflow Unidirect;
            Connect(PositionSocket, LeftMoveSocket);
           Connector FwdMoveSocket
            ConnectType Dataflow Unidirect;
            Connect(PositionSocket, FwdMoveSocket);
        }
}


  ViewList
     {
      ViewMain  {  { UsingArch LayerSolutionOne;
                     { Components All;
                       Connections All;
                       HyperLinkOn Controller ToFile "MobileRobotHTML.html";
                       HyperLinkOn ActiveComponent ToFile Solution1; }}
                  }

      View Solution1
                 {   { UsingArch SenseSolutionOne;
                        { Components ActiveComponent Actuators Sensors;
                          Connections RunActuator SenseData;
                          HyperLinkOn ActiveComponent ToFile "MobileRobot.txt"; }}

                      { UsingArch SenseSolutionOne;
                       { Components All;
                         Connections All; }}

                     { UsingArch ActSolutionOne;
                      { Components All;
                        Connections All;
```

```
                HyperLinkOn ActiveComponent ToFile "MobileRobot.txt";
                HyperLinkOn RunActuator ToFile "RunActuator.html";
                HyperLinkOn SenseData ToFile "SenseData.html";
                HyperLinkOn ActUponEnviron ToFile "ActUponEnviron.html";  }}
                  }

    View Solution2
                 {  { UsingArch SolutionTwo;
                      { Components Navigation GlobalPlan Supervisor;
                        Connections All;
                        HyperLinkOn Supervisor ToFile "MobileRobotHTML.html"; }}

                    { UsingArch SolutionTwo;
                      { Components Environment RobotControl SensorInterpret
                        SenseIntegrate;
                        Connections All;  }}

                { UsingArch SolutionTwo;
                      { Components SenseIntegrate RealWorldModel Navigation;
                        Connections All; }}

                    { UsingArch SolutionTwo;
                      { Components All;
                        Connections All; }}
                  }

    View Solution3
                 {  { UsingArch ImplicitInvoke;
                      { Components All;
                        Connections All; }}
                    { UsingArch TaskTree;
                      { Components All;
                        Connections All; }}
                  }
   }

$
```

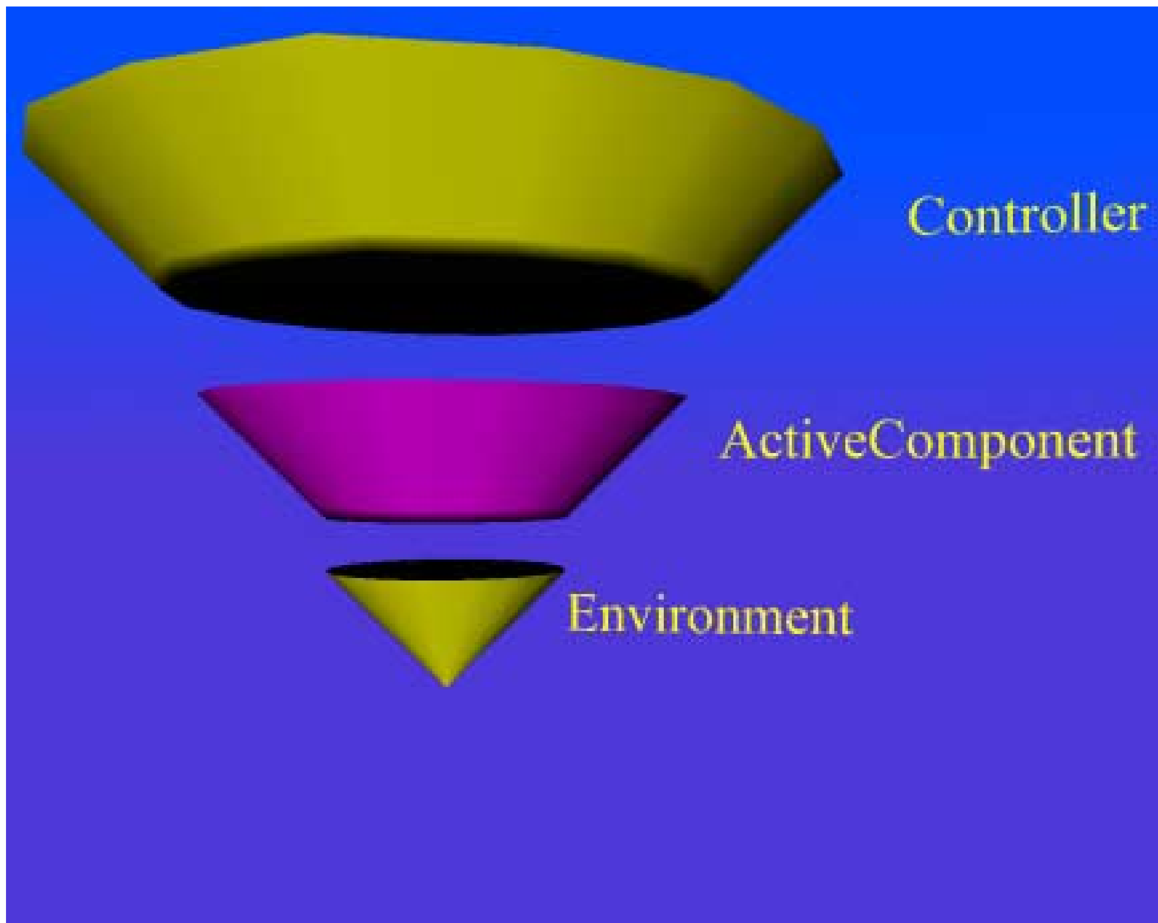
6. VRML Target Code: Main.wrl

The generated VRML code is not included here for sake of brevity.

7. Screen Snapshots of VRML Images:

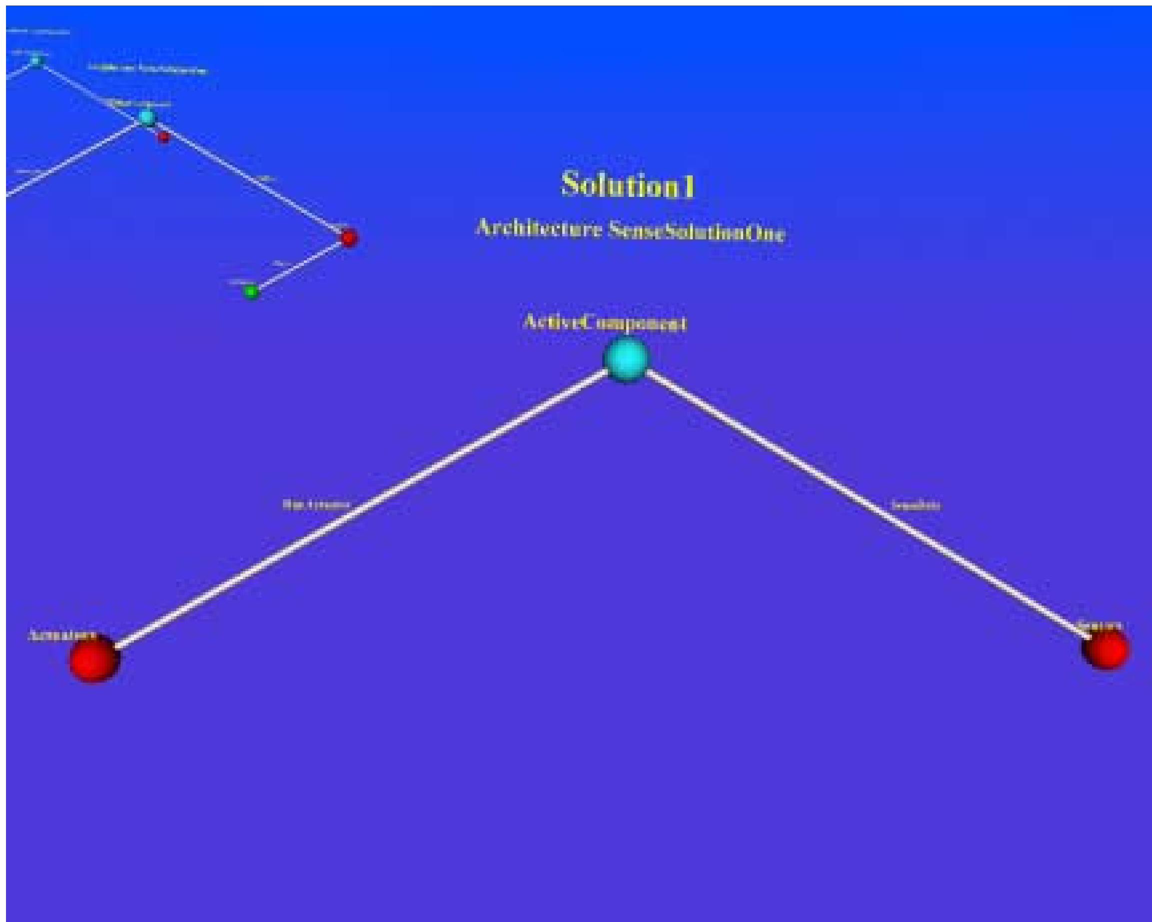
The legend within the main view is provided below. In addition to the main view, three other views are provided: Solution1, Solution2, and Solution3.

The layered architecture, "LayerSolutionOne," is also contained within the main view. LayerSolutionOne has three layers which represent the environment, a generic active component of the robot architecture, and the controller. A hyperlink exists from the controller layer to an html file (an html file that describes the architecture, "MobileRobotHTML.html"). A hyperlink from ActiveComponent serves as a portal to the next view, "Solution1."

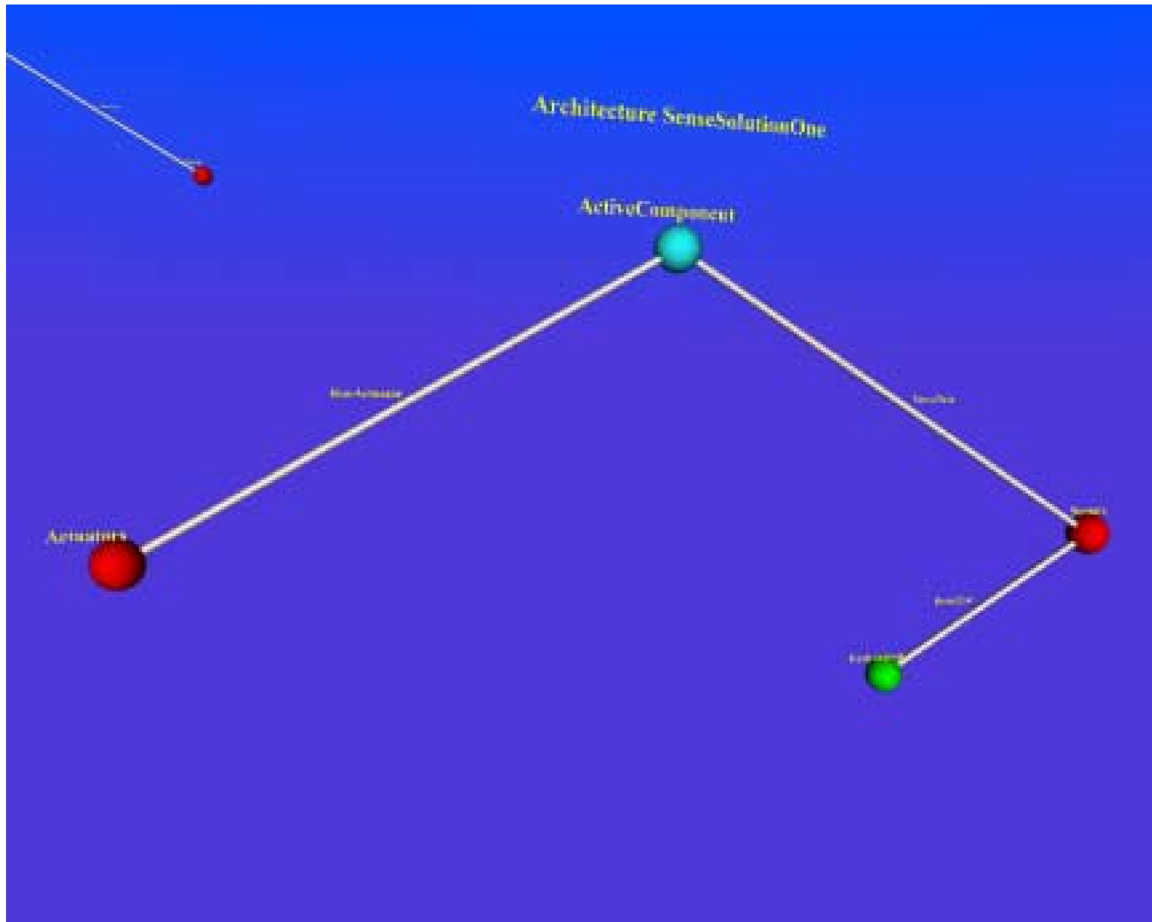

Controller
ActiveComponent
Environment

The next view, "Solution1," contains two architectures, "SenseSolutionOne" and "ActSolutionOne." Two renditions are made of SenseSolutionOne. The architecture SenseSolutionOne is pictured below as a call-and-return architecture representing a generic active component's control over actuators and sensors in the mobile robot (the other rendition can be seen in the background, along with ActSolutionOne). In the snapshot, only three components and two connectors are visualized, with other architectural elements hidden.

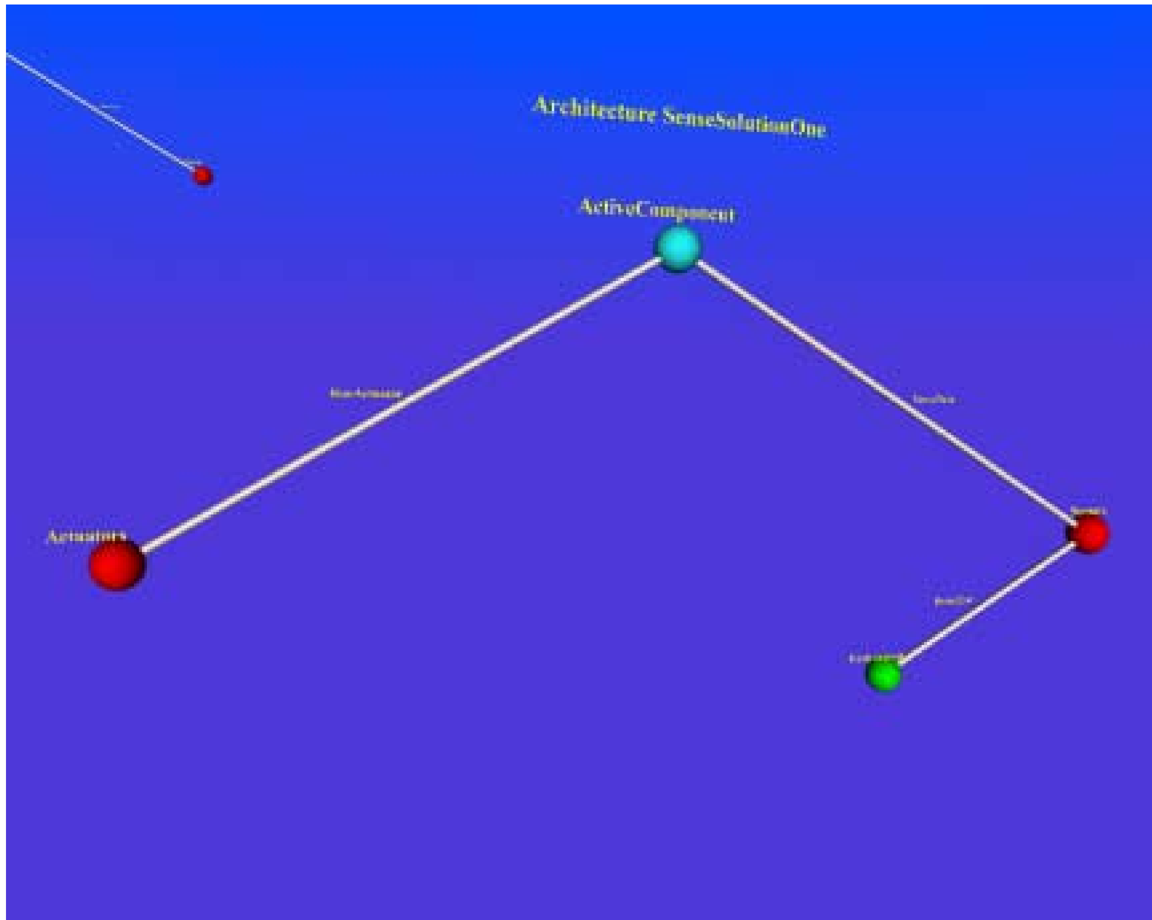
A hyperlink exists from the root, ActiveComponent, to the source VTADL file, "MobileRobot.txt."



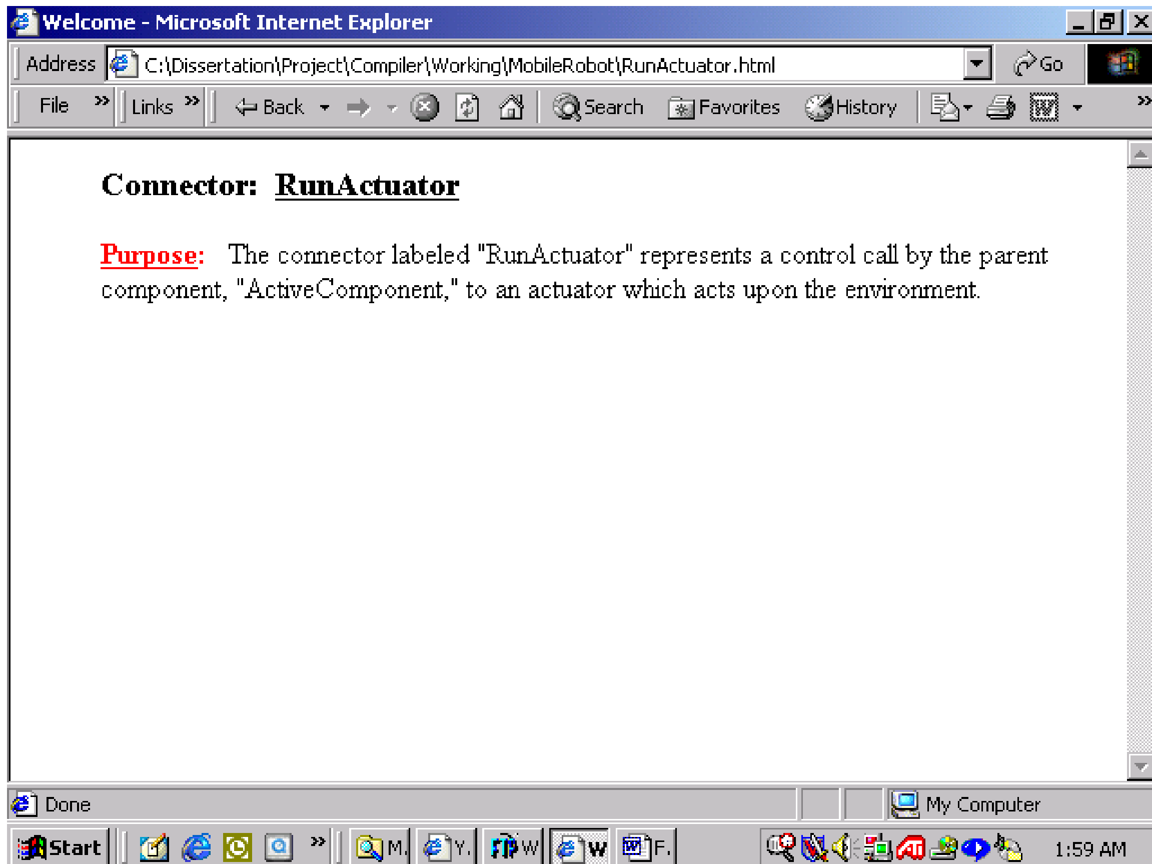
The second rendition of SenseSolutionOne (also contained within view Solution1) displays all components and connections, as seen below:



Architecture “ActSolutionOne” is the second architecture within view file Solution1. All components and connectors are shown. Several hyperlinks were established: from “ActiveComponent” to the source VTADL file, MobileRobot.txt; from the connector, “RunActuator” to an html file, “RunActuator.html”; from the connector, “SenseData” to the file, “SenseData.html”; and from connector “ActUponEnviron” to the html file, “ActUponEnviron.html.”

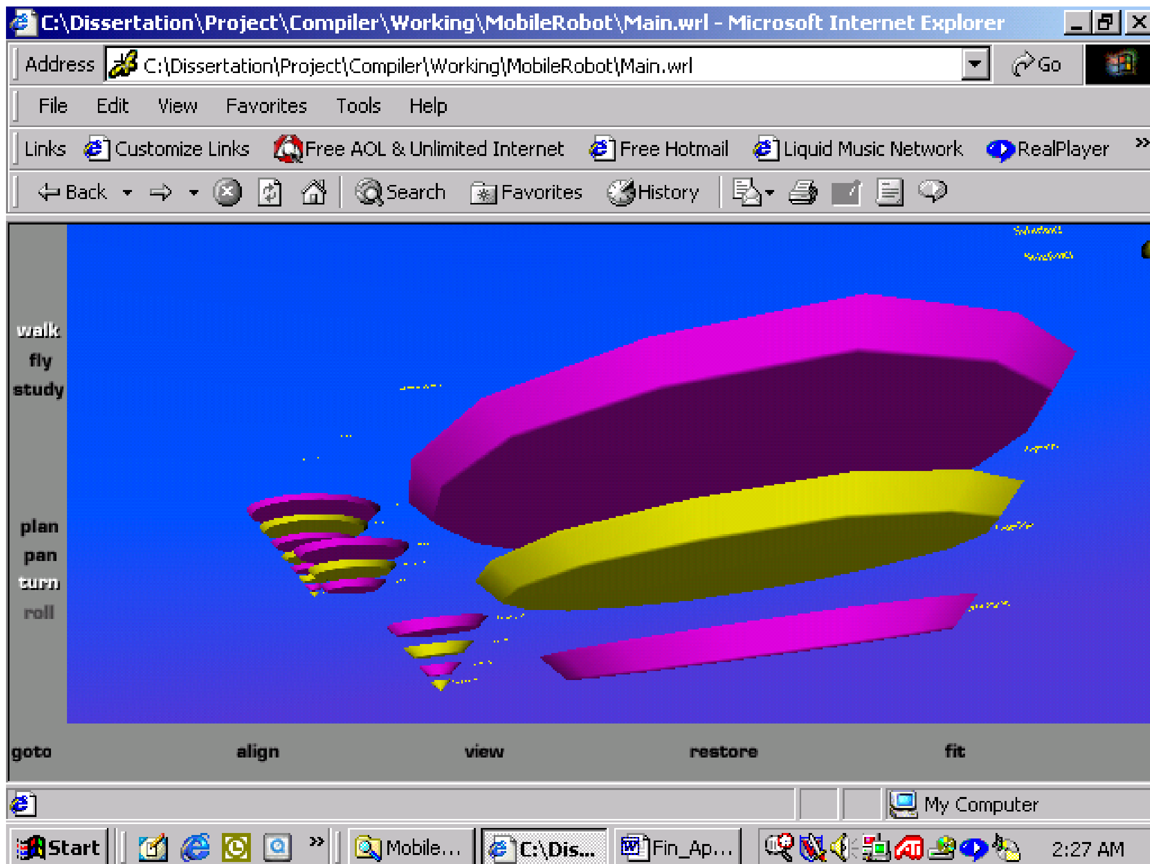


As a demonstration of the hyperlink, the html file referenced by connector RunActuator (“RunActuator.html”) is provided below as a screen snapshot:



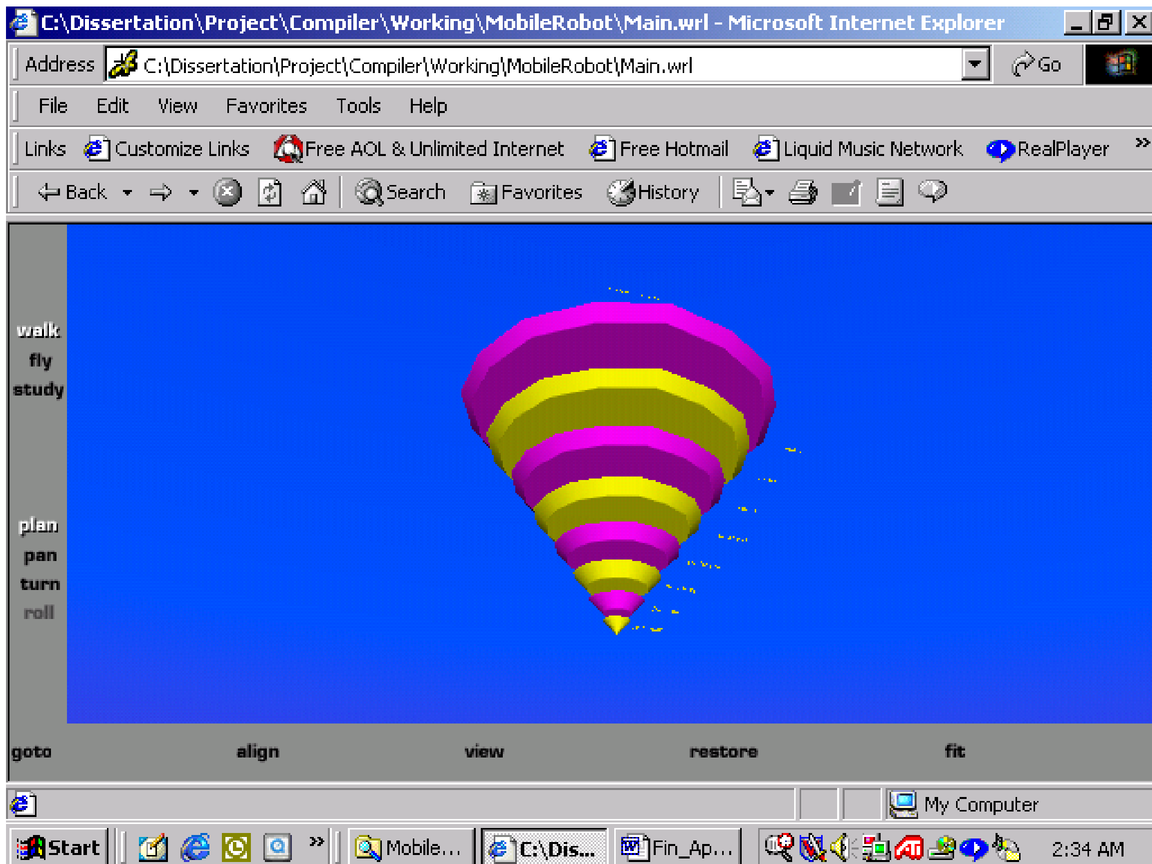
The view, “Solution2,” contains four different renditions of the same layered architecture, “SolutionTwo.”

In the snapshot below, the first rendition of SolutionTwo is seen in the foreground. The top three layers of the architecture are shown, and the remaining five layers are kept hidden. A hyperlink is established from the Supervisor layer to the file, MobileRobotHTML.html. The other renditions of the architecture are visible in the background:



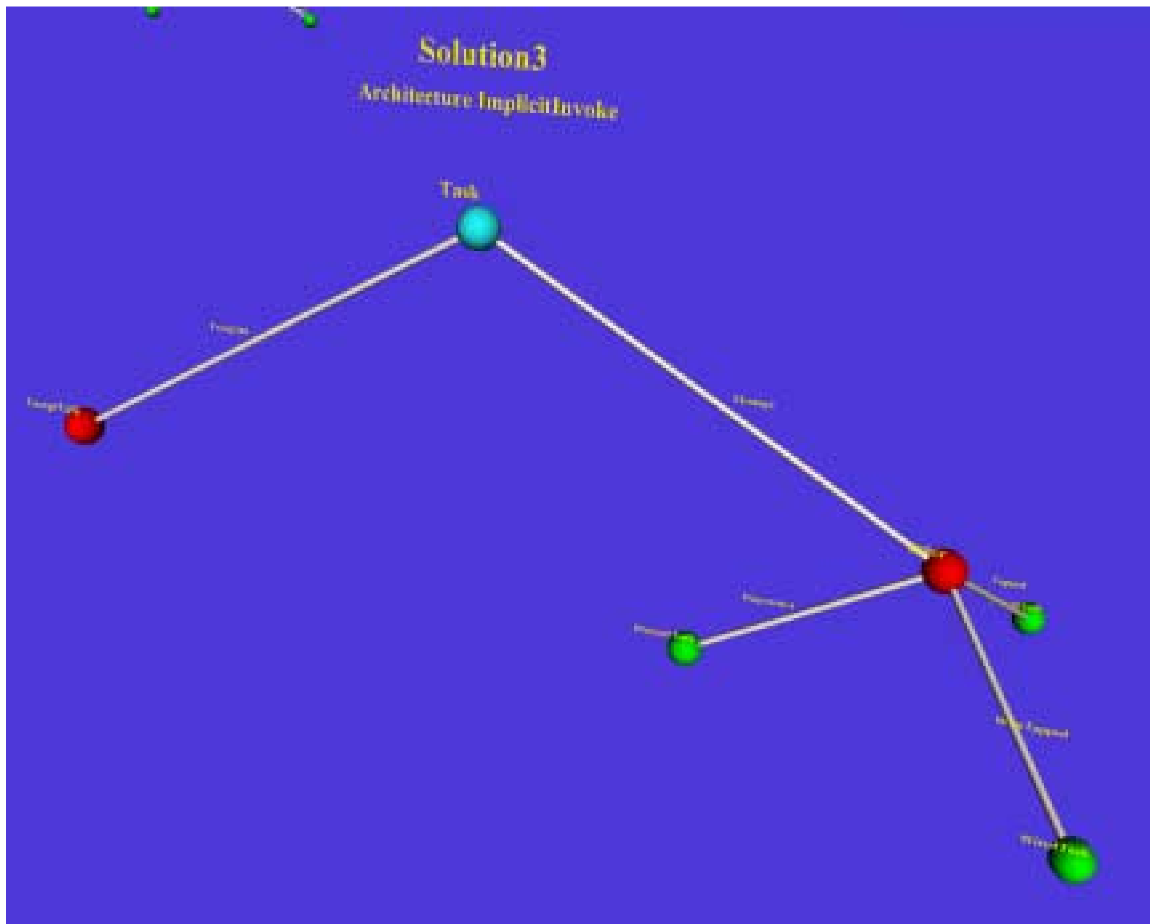
A second rendition of architecture SolutionTwo shows only the lower three layers, while the third rendition shows the middle three layers. The fourth rendition displays all layers.

We show only one other rendition of architecture SolutionTwo. The fourth rendition below shows all eight layers of the architecture:



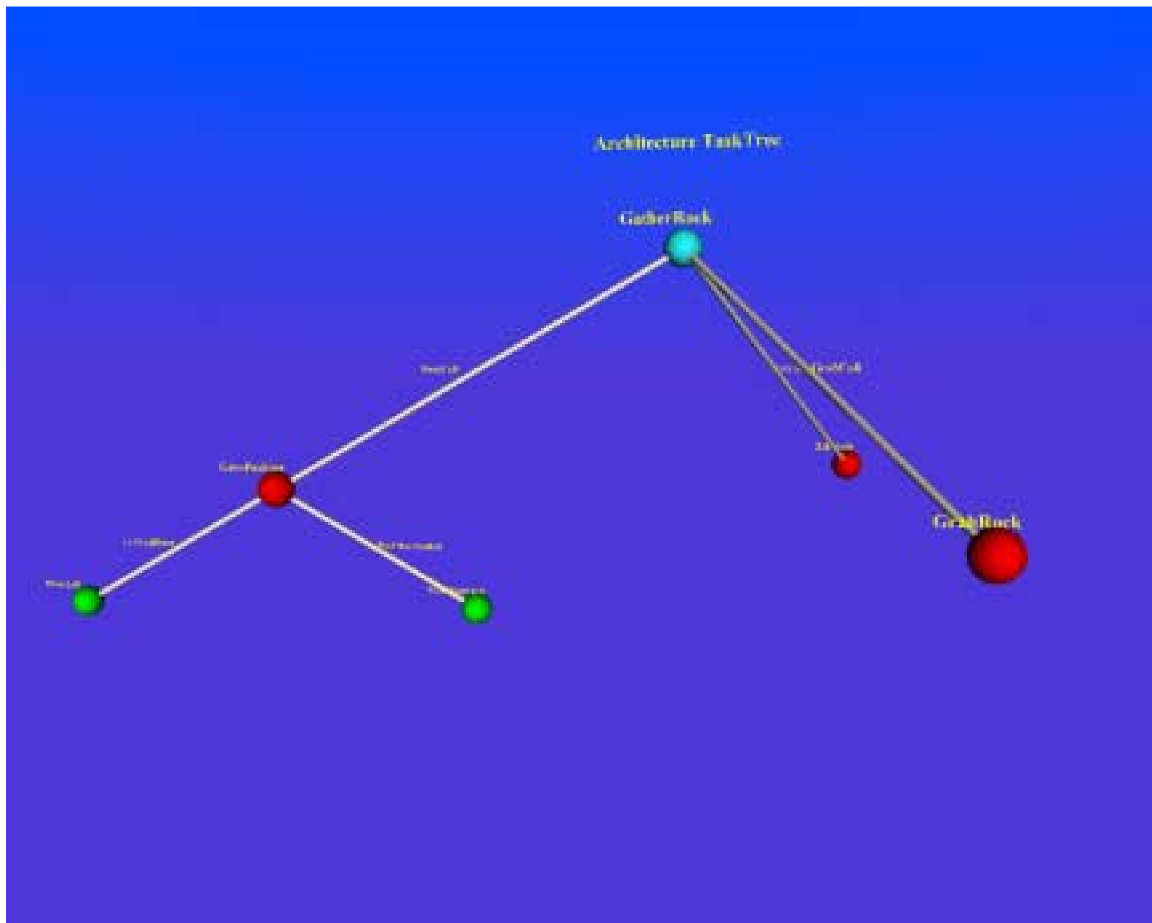
The view, “Solution3,” contains two architectures. The first, “ImplicitInvoke,” implements the implicit invocation solution to the mobile robot architecture (consult the text describing the original case study [Shaw and Garlan, 1996] for full details). The second architecture, “TaskTree,” represents an instantiation for the task of moving to a rock and picking it up.

The architecture ImplicitInvoke is shown in the foreground in the following screen snapshot:



In the ImplicitInvoke architecture, a parent task generates an exception event, resulting in a child exception task (“ExceptTask”). A parent task may also generate a message through the ether by various methods (dispatch, wiretapping, or tapping), resulting in the actualization of a child task.

Architecture TaskTree shows how the task of gathering a rock is instantiated:



The parent task of gathering a rock consists of the mobile robot first moving to the rock (represented by the component, "GoToPosition," and the children of GoToPosition); then grabbing the rock (represented by "Grab Rock"); and finally lifting the rock (represented by component "LiftRock"). The connectors to these subtasks represent the calls to the child tasks.

Case Study Two: Visualization of the Linux Conceptual and Concrete Architectures

Case Study Report

1. **Name of Case Study:** Visualization of Linux Conceptual and Concrete Architectures.

2. **On-Line Posting (if any):** http://www.nova.edu/~inouyej/Dissertation/CaseStudies.

3. **Brief Description of the Purpose and Background of the Case Study:**

This case study modifies the original visualization in the plane of the Linux operating system architecture [Bowman et al., 1998]. Viewpoints on the conceptual and concrete architectures are re-modeled in VTADL, then compiled into several visualized, three-dimensional worlds (viewpoints) in VRML.

4. **Name of VTADL Source File(s):**

Source File	View Files	Hyperlinked Files (Referenced)
ArchConcrete.txt	Main.wrl	ArchConcept.txt
ArchConcept.txt	IPCSubsystem.wrl	ConceptVirtualFile.wrl
	ConceptVirtualFile.wrl	ConceptIPC.wrl
	ConceptNetwork.wrl	IPCSubsystem.wrl
	ConceptMemoryMgr.wrl	
	ConceptIPC.wrl	

5. VTADL Source Code Listing

Two source code files were used instead of one. The first VTADL file was "ArchConcrete.txt," containing architectures and viewpoints on the concrete architecture of the Linux IPC (Interprocess Controller) subsystem. The second VTADL file was "ArchConcept.txt," containing architectures and viewpoints on the high-level, conceptual architecture of the Linux operating system. The viewpoints (VRML files) were integrated through hyperlinks within the architectural structures.

The file, ArchConcrete.txt, generated the VRML files "Main.wrl" and "IPCSubsystem.wrl." The main file generated by ArchConcrete.txt was not used in the final visualization. Instead, Main.wrl was overwritten by the Main.wrl file generated by VTADL file, ArchConcept.txt.

Thus, ArchConcrete.txt was compiled first. ArchConcept.txt was compiled second, overwriting the main file generated by ArchConcrete.txt.

File ArchConcept.txt generated the final "main.wrl" file used, along with VRML files "ConceptVirtualFile" (representing the conceptual architecture of the Virtual File subsystem), "ConceptNetwork" (representing the conceptual architecture of the Network subsystem), "ConceptMemoryMgr" (representing the conceptual architecture of the Memory Manager), and "ConceptIPC" (representing the conceptual architecture of the Interprocess Controller, which references via hyperlink the concrete architecture, "IPCSubsystem").

The VTADL source code for ArchConcrete.txt and ArchConcept.txt are provided below.

(The following code is from ArchConcept.txt and represents the conceptual architecture of the Linux operating system kernel).

```
Architecture ConceptualProcess
Style Program
  {
     ComponentList
          {

              Component ProcessSched
                ComponentType Cprogram;
                Properties:
                  CompRole:  CompConsumer;
                  ChildOf: ;
                  Layer: ;
                  Process: ;
                InterfaceList:
                  Interface Bottom ProcessSocket;
                    { InterfaceRole: Consumer;  }

              Component VirtualFileSystem
```

```
    ComponentType Cprogram;
    Properties:
        CompRole: CompProducer;
        ChildOf: ;
        Layer: ;
        Process:  ;
        InterfaceList:
        Interface Top FSSocket;
        { InterfaceRole: Producer;  }

   Component Network
    ComponentType Cprogram;
    Properties:
        CompRole: CompProducer;
        ChildOf: ;
        Layer: ;
        Process: ;
        InterfaceList:
        Interface Top NetSocket;
        { InterfaceRole: Producer; }

   Component MemoryMgr
    ComponentType Cprogram;
    Properties:
        CompRole: CompProducer;
        ChildOf: ;
        Layer: ;
        Process: ;
        InterfaceList:
        Interface Top MMSocket;
        { InterfaceRole: Producer; }

  Component IPC
    ComponentType Cprogram;
    Properties:
        CompRole: CompProducer;
        ChildOf: ;
        Layer: ;
        Process: ;
        InterfaceList:
        Interface Top IPCSocket;
        { InterfaceRole: Producer; }

}

ConnectionList
```

```
        {
          Connector CallVfs
            ConnectType Dataflow Unidirect;
            Connect(ProcessSocket, FSSocket);
          Connector CallNet
            ConnectType Dataflow Unidirect;
            Connect(ProcessSocket, NetSocket);
          Connector CallMemMgr
            ConnectType Dataflow Unidirect;
            Connect(ProcessSocket, MMSocket);
          Connector CallIPC
            ConnectType Dataflow Unidirect;
            Connect(ProcessSocket, IPCSocket);
        }

  }

Architecture ConcreteProcess
Style Program
  {
     ComponentList
        {

          Component sched
            ComponentType Cprogram;
            Properties:
              CompRole:  CompConsumer;
              ChildOf: ;
              Layer: ;
              Process: ;
            InterfaceList:
             Interface Bottom ProcessSocket;
               { InterfaceRole: Consumer;  }

          Component fs
            ComponentType Cprogram;
            Properties:
                CompRole: CompProducer;
                ChildOf: ;
                Layer: ;
                Process:  ;
                InterfaceList:
                Interface Top FSSocket;
                { InterfaceRole: Producer;  }

           Component net
```

```
        ComponentType Cprogram;
        Properties:
            CompRole: CompProducer;
            ChildOf: ;
            Layer: ;
            Process: ;
            InterfaceList:
            Interface Top NetSocket;
            { InterfaceRole: Producer; }

       Component mm
        ComponentType Cprogram;
        Properties:
            CompRole: CompProducer;
            ChildOf: ;
            Layer: ;
            Process: ;
            InterfaceList:
            Interface Top MMSocket;
            { InterfaceRole: Producer; }

      Component ipc
        ComponentType Cprogram;
        Properties:
            CompRole: CompProducer;
            ChildOf: ;
            Layer: ;
            Process: ;
            InterfaceList:
            Interface Top IPCSocket;
            { InterfaceRole: Producer; }

  }

ConnectionList
  {
      Connector Callfs
        ConnectType Dataflow Unidirect;
        Connect(ProcessSocket, FSSocket);
      Connector Callnet
        ConnectType Dataflow Unidirect;
        Connect(ProcessSocket, NetSocket);
      Connector Callmm
        ConnectType Dataflow Unidirect;
        Connect(ProcessSocket, MMSocket);
      Connector Callipc
```

```
                ConnectType Dataflow Unidirect;
                Connect(ProcessSocket, IPCSocket);
            }

    }

Architecture NetLayers
Style Layer
 {
    ComponentList
      {
       Component HWDrivers
         ComponentType Cprogram;
         Properties:
          CompRole:  CompProducer;
          ChildOf: ;
          Layer: hw1;
          Process: ;
         InterfaceList:
           Interface Top HWService1;
              { InterfaceRole: Producer; }

        Component NetProtocols
         ComponentType Cprogram;
         Properties:
         CompRole: CompConsumer;
         ChildOf: ;
         Layer: proto1;
         Process: ;
         InterfaceList:
           Interface Bottom ReceiveHW1;
              { InterfaceRole: Consumer; }

        Component Network
          ComponentType Cprogram;
          Properties:
          CompRole: CompConsumer;
          ChildOf: ;
          Layer: Network;
          Process: ;
          InterfaceList:
            Interface Bottom ReceiveNet1;
               { InterfaceRole: Consumer; }

      }
      ConnectionList
```

```
      {
       Connector NetService
         ConnectType Dataflow Unidirect;
         Connect(HWService1,ReceiveHW1);
       Connector NetProto
         ConnectType Dataflow Unidirect;
         Connect(ReceiveHW1,ReceiveNet1);
      }
 }

Architecture VirtualFileLayers
Style Layer
 {
   ComponentList
     {
      Component HWDrivers
        ComponentType Cprogram;
        Properties:
         CompRole:  CompProducer;
         ChildOf: ;
         Layer: hw1;
         Process: ;
        InterfaceList:
          Interface Top HWService1;
             { InterfaceRole: Producer; }

       Component LogicFileSystem
        ComponentType Cprogram;
        Properties:
        CompRole: CompConsumer;
        ChildOf: ;
        Layer: LogicFile;
        Process: ;
        InterfaceList:
          Interface Bottom ReceiveHW1;
             { InterfaceRole: Consumer; }

       Component VirtualFileSystem
         ComponentType Cprogram;
         Properties:
         CompRole: CompConsumer;
         ChildOf: ;
         Layer: VF;
         Process: ;
         InterfaceList:
           Interface Bottom ReceiveVF1;
```

```
            { InterfaceRole: Consumer; }

     }
     ConnectionList
      {
        Connector FileService
          ConnectType Dataflow Unidirect;
          Connect(HWService1,ReceiveHW1);
        Connector FileSystem
          ConnectType Dataflow Unidirect;
          Connect(ReceiveHW1,ReceiveVF1);
      }
 }

Architecture MemoryLayers
Style Layer
 {
   ComponentList
     {
       Component HWDependent
         ComponentType Cprogram;
         Properties:
          CompRole:  CompProducer;
          ChildOf: ;
          Layer: HWDep;
          Process: ;
         InterfaceList:
           Interface Top HWService1;
             { InterfaceRole: Producer; }

         Component HWIndependent
         ComponentType Cprogram;
         Properties:
         CompRole: CompConsumer;
         ChildOf: ;
         Layer: HWIndep;
         Process: ;
         InterfaceList:
           Interface Bottom ReceiveHW1;
             { InterfaceRole: Consumer; }

        Component MemoryMgr
          ComponentType Cprogram;
          Properties:
          CompRole: CompConsumer;
          ChildOf: ;
```

```
        Layer: MMMgr;
        Process: ;
        InterfaceList:
          Interface Bottom ReceiveMem1;
            { InterfaceRole: Consumer; }

    }
    ConnectionList
     {
       Connector HWDepService
        ConnectType Dataflow Unidirect;
        Connect(HWService1,ReceiveHW1);
       Connector HWIndepService
        ConnectType Dataflow Unidirect;
        Connect(ReceiveHW1,ReceiveMem1);
     }
 }

Architecture IPC
Style Layer
 {
   ComponentList
    {
      Component IPCCallInterface
        ComponentType Cprogram;
        Properties:
         CompRole:  CompProducer;
         ChildOf: ;
         Layer: ;
         Process: ;
        InterfaceList:
          Interface Bottom SystemCall;
            { InterfaceRole: Consumer; }

       Component SystemVIPC
        ComponentType Cprogram;
        Properties:
        CompRole: CompConsumer;
        ChildOf: ;
        Layer: ;
        Process: ;
        InterfaceList:
          Interface Top SystemVSocket;
            { InterfaceRole: Producer; }

    }
```

```
 ConnectionList
  {
   Connector SystemCallInterface
     ConnectType Dataflow Unidirect;
     Connect(SystemCall,SystemVSocket);
  }
}


 ViewList
   {
    ViewMain  {  { UsingArch ConceptualProcess;
                   { Components All;
                     Connections All;
                     HyperLinkOn VirtualFileSystem ToFile ConceptVirtualFile;
                     HyperLinkOn CallIPC ToFile ConceptIPC;
                     HyperLinkOn ProcessSched ToFile "archConcept.txt"; }}
              { UsingArch IPC;
                   { Components All;
                     Connections All; }}
              { UsingArch ConcreteProcess;
                   { Components All;
                     Connections All; }}
             }

    View ConceptVirtualFile
             {  { UsingArch VirtualFileLayers;
                   { Components All;
                     Connections All; }}
             }

    View ConceptNetwork
             {  { UsingArch NetLayers;
                   { Components All;
                     Connections All; }}

             }

    View ConceptMemoryMgr
             {  { UsingArch MemoryLayers;
                   { Components All;
                     Connections All; }}
             }

    View ConceptIPC
             {  { UsingArch IPC;
```

```
                { Components All;
                  Connections All;
                  HyperLinkOn IPCCallInterface ToFile IPCSubsystem; }}
            }


    }

$
```

(The following code is from ArchConcrete.txt and represents the concrete architecture of the Linux kernel):

```
Architecture IPCSubsystem
Style Program
  {
     ComponentList
        {

           Component SystemCallInterface
            ComponentType Cprogram;
            Properties:
              CompRole:  CompConsumer;
              ChildOf: ;
              Layer: ;
              Process: ;
            InterfaceList:
             Interface Bottom SysCallSocket;
               { InterfaceRole: Consumer;  }

           Component NetIPC
            ComponentType Cprogram;
            Properties:
                CompRole: CompProducer;
                ChildOf: ;
                Layer: ;
                Process:  ;
                InterfaceList:
                Interface Top NetIPCSocket;
                { InterfaceRole: Producer;  }
                Interface Bottom NetDomainSocket;
                { InterfaceRole: Consumer; }

            Component SystemVIPC
            ComponentType Cprogram;
            Properties:
```

```
    CompRole: CompProducer;
    ChildOf: ;
    Layer: ;
    Process:  ;
    InterfaceList:
    Interface Top SystemVSocket;
    { InterfaceRole: Producer;  }
    Interface Bottom SystemVLower;
    { InterfaceRole: Consumer; }

Component FileIPC
 ComponentType Cprogram;
 Properties:
    CompRole: CompProducer;
    ChildOf: ;
    Layer: ;
    Process:  ;
    InterfaceList:
    Interface Top FileSocket;
    { InterfaceRole: Producer;  }
    Interface Bottom FileLower;
    { InterfaceRole: Consumer; }

Component DomainSockets
 ComponentType Cprogram;
 Properties:
    CompRole: CompProducer;
    ChildOf: ;
    Layer: ;
    Process: ;
    InterfaceList:
    Interface Top DomainPlug;
    { InterfaceRole: Producer; }

Component MessageQueues
 ComponentType Cprogram;
 Properties:
    CompRole: CompProducer;
    ChildOf: ;
    Layer: ;
    Process: ;
    InterfaceList:
    Interface Top MsgSocket;
    { InterfaceRole: Producer; }

Component SharedMemory
```

```
 ComponentType Cprogram;
 Properties:
     CompRole: CompProducer;
     ChildOf: ;
     Layer: ;
     Process:  ;
     InterfaceList:
     Interface Top SharedSocket;
     { InterfaceRole: Producer;  }

Component KernelIPC
 ComponentType Cprogram;
 Properties:
     CompRole: CompProducer;
     ChildOf: ;
     Layer: ;
     Process: ;
     InterfaceList:
     Interface Top KernelSocket;
     { InterfaceRole: Producer; }

 Component Semaphores
 ComponentType Cprogram;
 Properties:
     CompRole: CompProducer;
     ChildOf: ;
     Layer: ;
     Process:  ;
     InterfaceList:
     Interface Top SemaSocket;
     { InterfaceRole: Producer;  }

 Component fifo
 ComponentType Cprogram;
 Properties:
     CompRole: CompProducer;
     ChildOf: ;
     Layer: ;
     Process:  ;
     InterfaceList:
     Interface Top fifoSocket;
     { InterfaceRole: Producer;  }

 Component pipes
 ComponentType Cprogram;
 Properties:
```

```
            CompRole: CompProducer;
            ChildOf: ;
            Layer: ;
            Process:  ;
            InterfaceList:
            Interface Top pipesSocket;
            { InterfaceRole: Producer;  }

        Component WaitQueues
         ComponentType Cprogram;
         Properties:
            CompRole: CompProducer;
            ChildOf: ;
            Layer: ;
            Process:  ;
            InterfaceList:
            Interface Top WaitQSocket;
            { InterfaceRole: Producer;  }

        Component Signals
         ComponentType Cprogram;
         Properties:
            CompRole: CompProducer;
            ChildOf: ;
            Layer: ;
            Process:  ;
            InterfaceList:
            Interface Top SignalsSocket;
            { InterfaceRole: Producer;  }
    }

ConnectionList
    {
      Connector CallNetIPC
        ConnectType Dataflow Unidirect;
        Connect(SysCallSocket, NetIPCSocket);
      Connector CallSysVIPC
        ConnectType Dataflow Unidirect;
        Connect(SysCallSocket, SystemVSocket);
      Connector CallFileIPC
        ConnectType Dataflow Unidirect;
        Connect(SysCallSocket, FileSocket);
      Connector CallDomain
        ConnectType Dataflow Unidirect;
        Connect(NetDomainSocket, DomainPlug);
      Connector CallMessage
```

```
              ConnectType Dataflow Unidirect;
              Connect(SystemVLower,MsgSocket);
            Connector CallSharedMem
              ConnectType Dataflow Unidirect;
              Connect(SystemVLower, SharedSocket);
            Connector CallKernelIPC
              ConnectType Dataflow Unidirect;
              Connect(SystemVLower, KernelSocket);
            Connector CallSemaphores
              ConnectType Dataflow Unidirect;
              Connect(SystemVLower, SemaSocket);
            Connector Callfifo
              ConnectType Dataflow Unidirect;
              Connect(FileLower,fifoSocket);
            Connector Callpipes
              ConnectType Dataflow Unidirect;
              Connect(FileLower,pipesSocket);
            Connector CallWaitQ
              ConnectType Dataflow Unidirect;
              Connect(KernelSocket, WaitQSocket);
            Connector CallSignals
              ConnectType Dataflow Unidirect;
              Connect(KernelSocket, SignalsSocket);
          }

  }


Architecture IPCDepends
Style Program
  {
     ComponentList
          {

            Component IPC
              ComponentType Cprogram;
              Properties:
                CompRole:  CompConsumer;
                ChildOf: ;
                Layer: ;
                Process: ;
              InterfaceList:
                Interface Bottom SysDepends;
                  { InterfaceRole: Consumer;  }

            Component ProcessScheduler
```

```
            ComponentType Cprogram;
            Properties:
                CompRole: CompProducer;
                ChildOf: ;
                Layer: ;
                Process:  ;
                InterfaceList:
                Interface Top ProcSocket;
                { InterfaceRole: Producer;  }

           Component MemoryMgr
            ComponentType Cprogram;
            Properties:
                CompRole: CompProducer;
                ChildOf: ;
                Layer: ;
                Process:  ;
                InterfaceList:
                Interface Top MemSocket;
                { InterfaceRole: Producer;  }

           Component FileSystem
            ComponentType Cprogram;
            Properties:
                CompRole: CompProducer;
                ChildOf: ;
                Layer: ;
                Process:  ;
                InterfaceList:
                Interface Top FileSysSocket;
                { InterfaceRole: Producer;  }

       }

ConnectionList
     {
        Connector CallProcessSched
          ConnectType Dataflow Unidirect;
          Connect(SysDepends, ProcSocket);
        Connector CallMemMgr
          ConnectType Dataflow Unidirect;
          Connect(SysDepends, MemSocket);
        Connector CallFileSys
          ConnectType Dataflow Unidirect;
          Connect(SysDepends, FileSysSocket);
     }
```

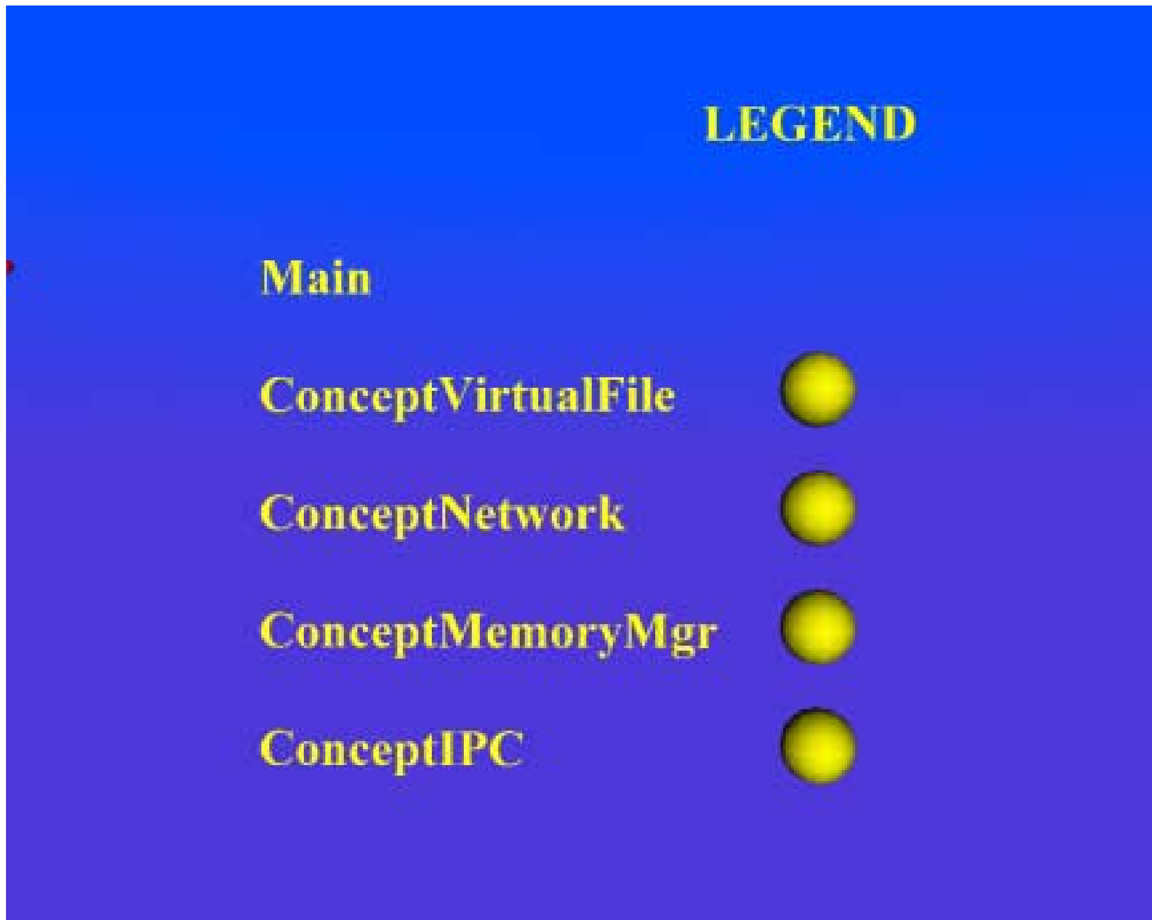
```
 }

   ViewList
      {
       ViewMain  {  { UsingArch IPCSubsystem;
                   { Components All;
                    Connections All;
                    HyperLinkOn NetIPC ToFile "Testref.txt"; }}
               }

       View IPCSubsystem {  { UsingArch IPCSubsystem;
                      { Components All;
                       Connections All; }}
                    { UsingArch IPCDepends;
                      { Components All;
                       Connections All; }}
                   }

      }

$
```

6. VRML Target Code: Main.wrl

The generated VRML code for this visualization is not provided for sake of brevity.

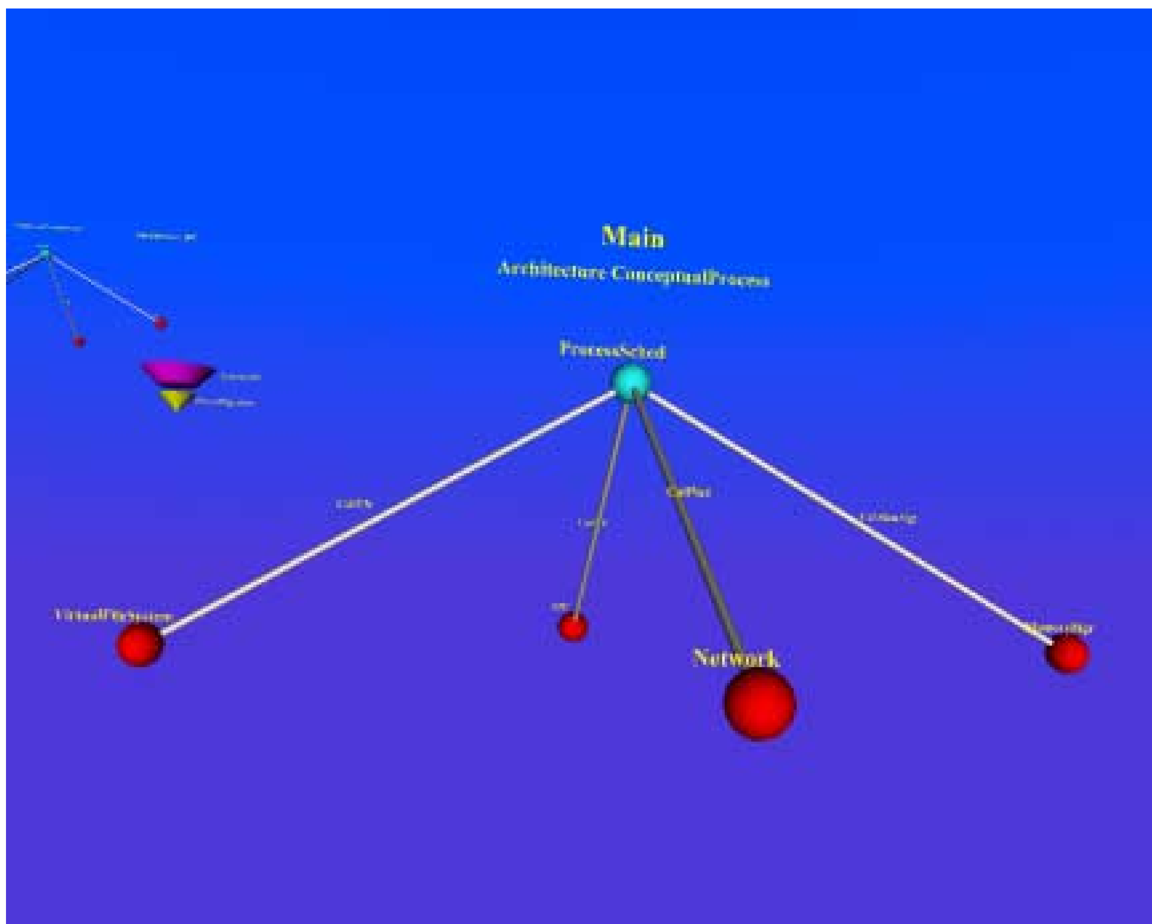
7. Screen Snapshots of VRML Images:

The legend in the main view of the Linux visualization is provided in the screen snapshot below.

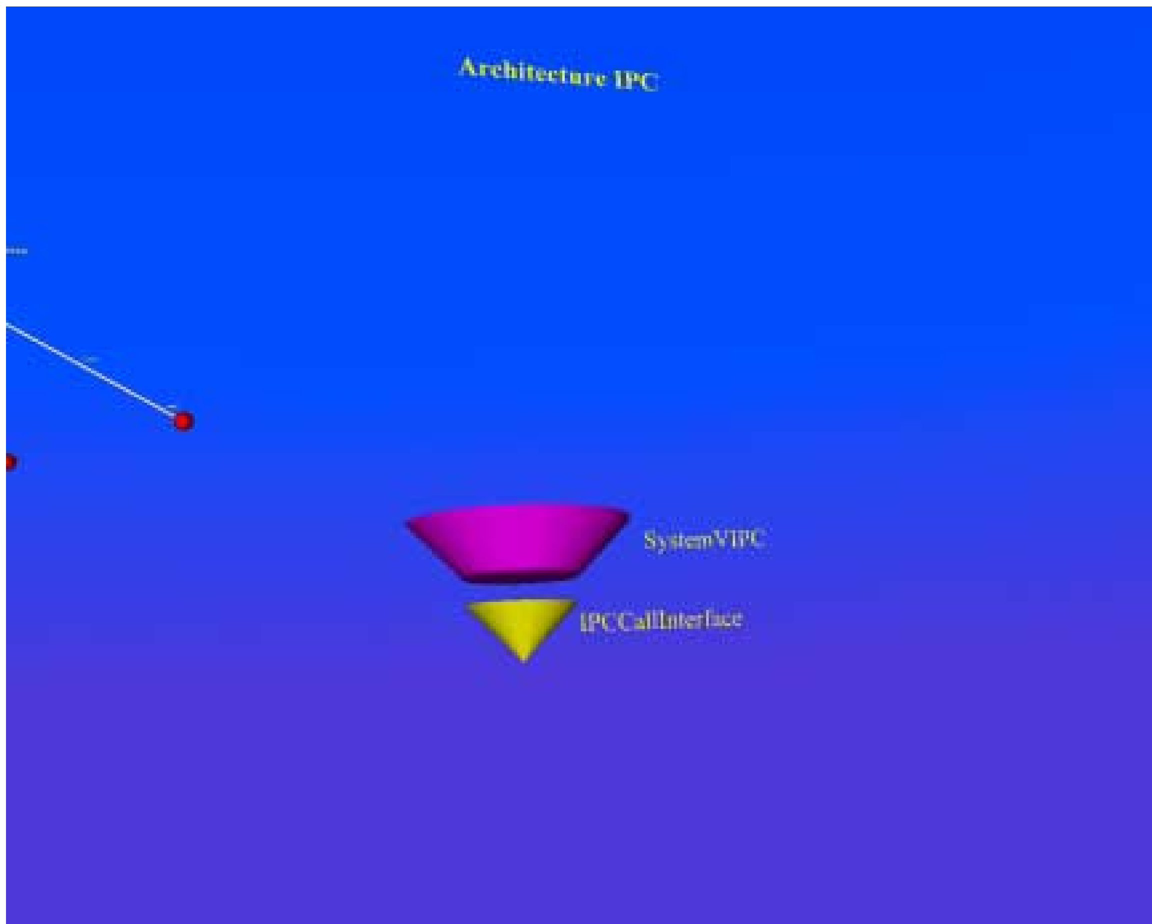


In the screen shot below, the first architecture (of three architectures within the main view) is visualized. The architecture in the foreground within the main view is “ConceptualProcess,” representing the highest level of the conceptual architecture of the Linux kernel. The view of ConceptualProcess is from the vantage point of the Process Scheduler (component “ProcessSched” at the root). The children of ProcessSched represent the conceptual modules called by the Process Scheduler. The Process Scheduler may call the Virtual File System, Memory Manager, Network, or IPC.

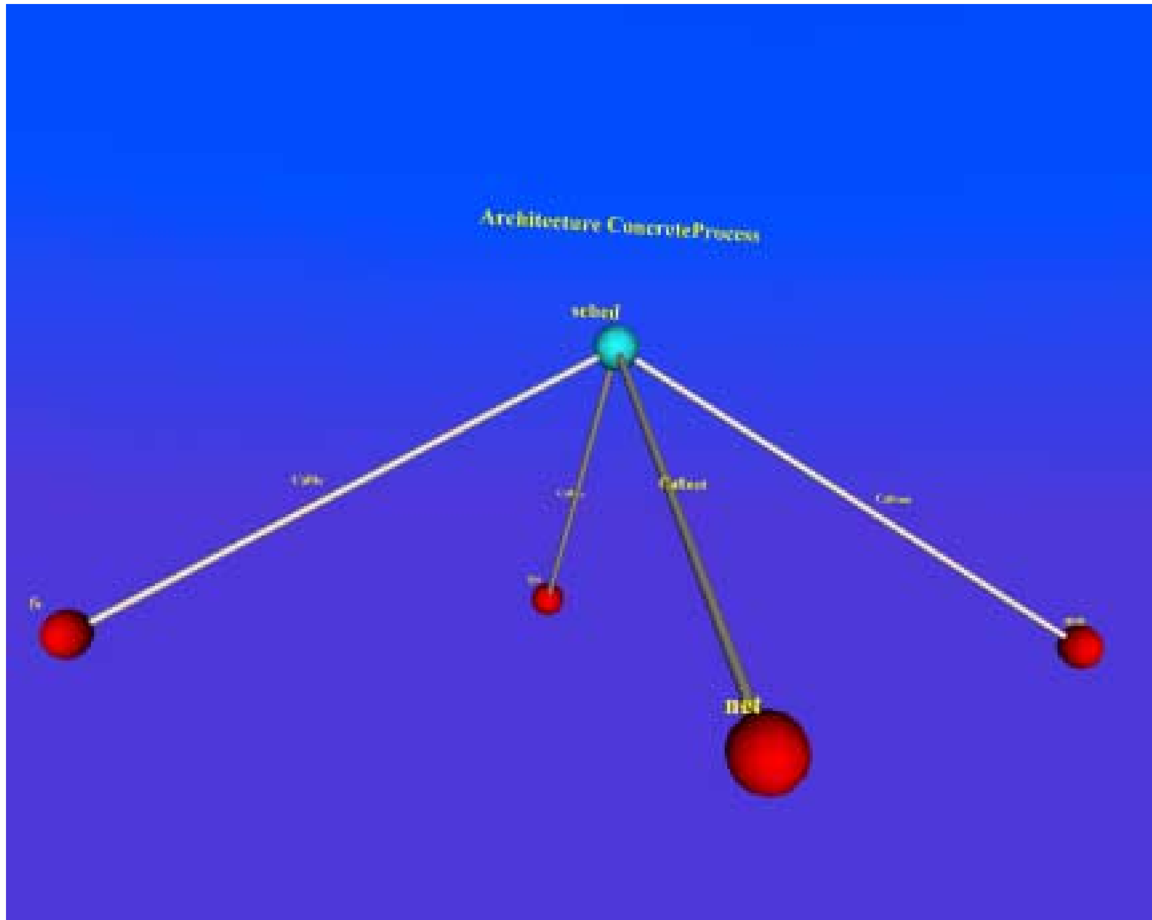
A hyperlink is established on the component VirtualFileSystem to the ConceptVirtualFile view. A hyperlink is also established on component CallIPC to the view file, ConceptIPC. Finally, a hyperlink exists from the root (ProcessSched) to the VTADL source file, “ArchConcept.txt.”



The second architecture within the main view is the conceptual IPC architecture, represented by a layered structure with two layers: the IPC Call Interface and the System V IPC controller.

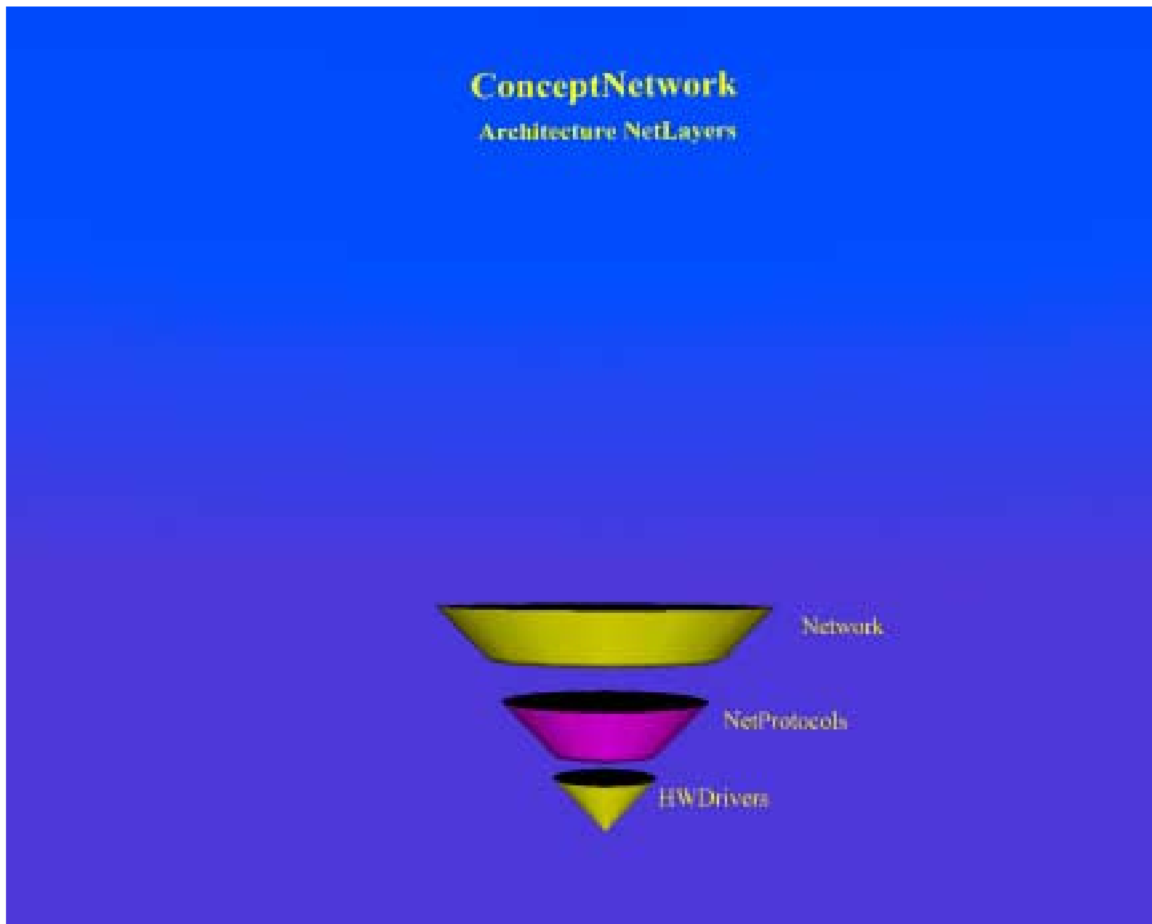


The snapshot below provides the third architecture in the main view, a representation of the Concrete Process of the Linux system architecture. The scheduler is represented as a root ("sched") of a three-dimensional tree, with children labeled "fs" (file system module), "net" (the network drivers), "mm" (the memory management routine), and "ipc" (for the interprocess control module).



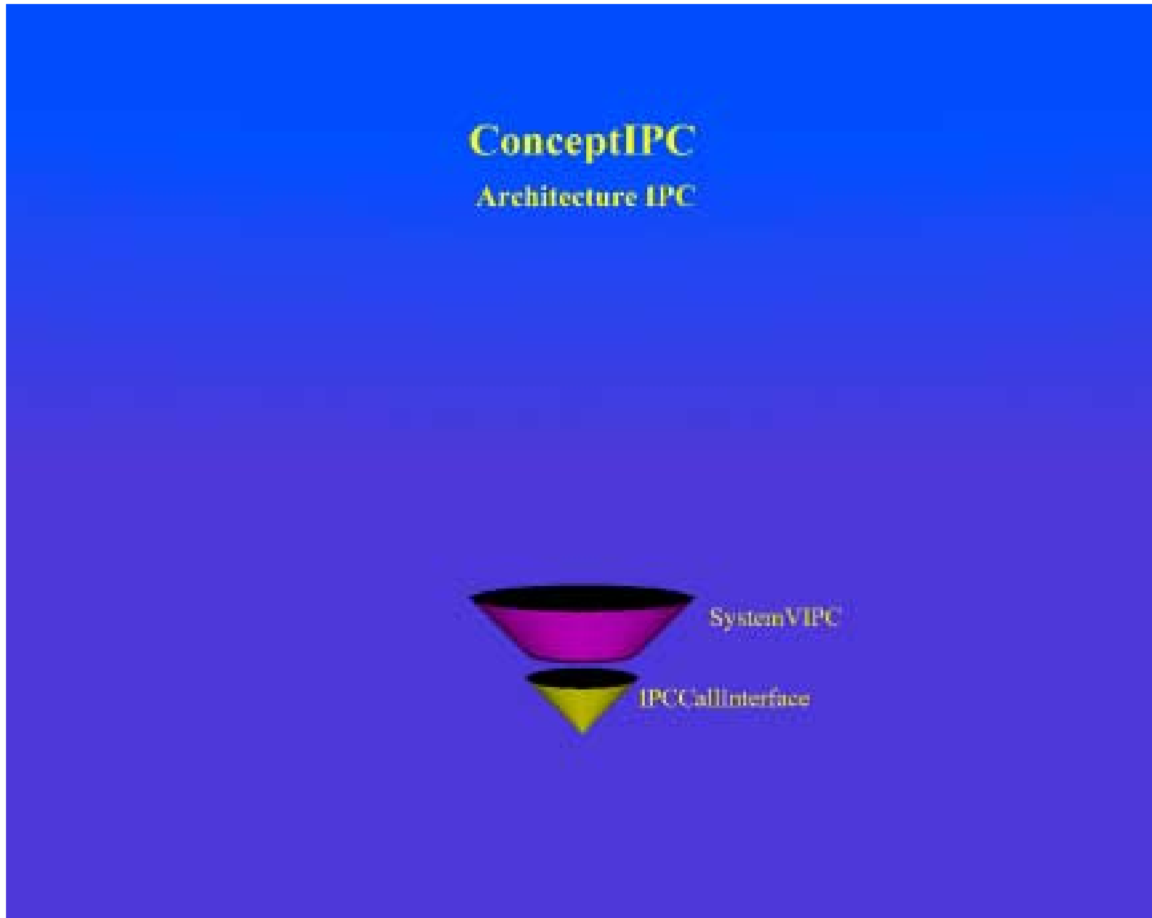
In the next view, “ConceptVirtualFile,” emphasis is placed on the conceptual virtual file manager architecture. A single architecture, “VirtualFileLayers,” is represented within ConceptVirtualFile. This single architecture consists of three layers: “HW Drivers” (or the layer representing hardware drivers), “LogicFileSystem” (the logical file system layer), and “VirtualFileSystem” (the layer representing the virtual file system of the Linux kernel).

The view, "ConceptNetwork," represents the conceptual architecture of the network system. The network architecture is visualized as a layered cone, with hardware drivers at the base, network protocols in the center, and the network layer itself at the top:



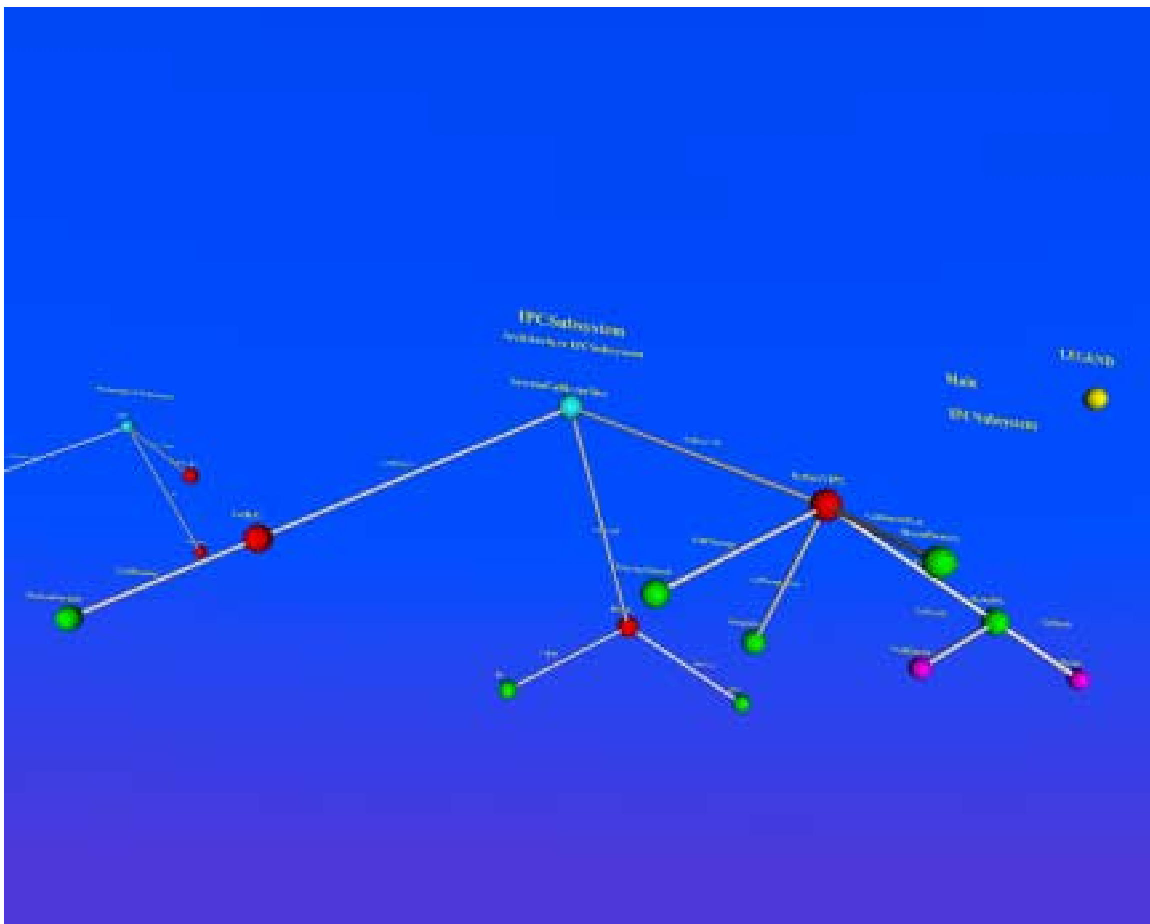
The view of the conceptual memory manager, “ConceptMemoryMgr,” consists of a single layered architecture, “MemoryLayers,” as seen below:

The view, “ConceptIPC,” is comprised of a single layered architecture called “IPC.” The layered architecture (shown below) represents the conceptual architecture of the Linux kernel Interprocess Controller. The top layer represents the System V IPC, while the bottom layer represents the IPC Call Interface. The bottom layer contains a very important hyperlink to the detailed view of the concrete IPC Subsystem. The view file referenced is “IPCSubsystem.wrl.”

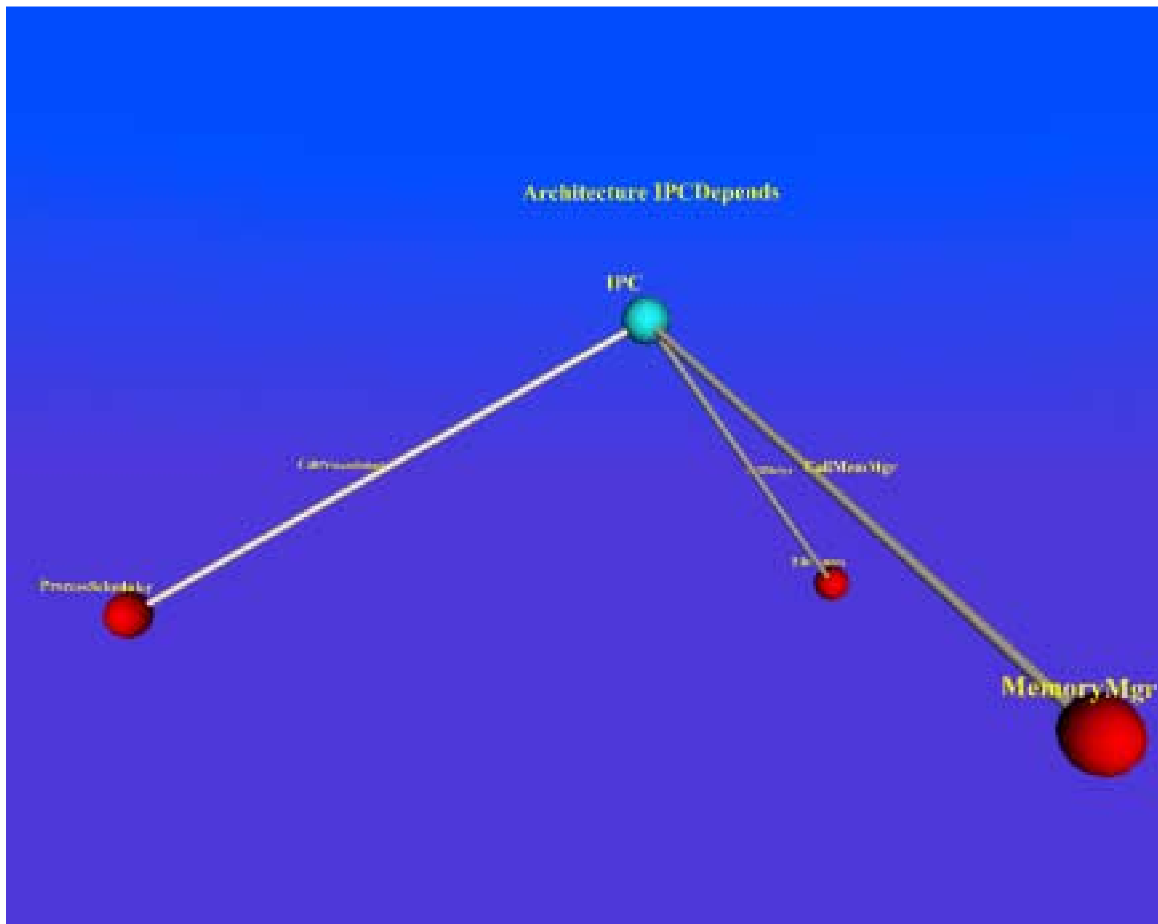


The view, “IPCSubsystem,” describes the concrete architecture of the IPC subsystem to the lowest level of detail. IPCSubsystem consists of two call-and-return style architectures, “IPCSubsystem” and “IPCDepends.” IPCSubsystem is in the foreground in the screen snapshot provided below (the IPCDepends architecture is visible in the left background).

At the root of the IPCSubsystem architecture is a node representing the system call interface of the interprocess controller. The system call interface may make calls to the network IPC, the file IPC, or the System V IPC. The network IPC may call domain sockets; the file IPC may, in turn, call fifo (first-in, first-out) data structures or pipes. The System V IPC may call message queues to support interprocess messaging. The System V IPC may also call semaphores, shared memory, or data structures available to the kernel IPC (waitqueues or signals). Thus, the hierarchy of control within the IPC architecture is visualized in the VRML medium.



In our final snapshot, the architecture, “IPCDepends,” is brought to the foreground. This architecture describes the dependencies between a generic IPC process and the process scheduler, memory manager, and file system.



Reference List

Abowd, G.D., Allen, R., & Garlan, D. (1995). Formalizing Style to Understand Descriptions of Software Architecture. *ACM Transactions on Software Engineering and Methodology*, 4(4), 319-364.

Abowd, G., Bass, L., Clements, P., Kazman, R., Northrop, L., & Zaremski, A. (1997). Recommended Best Industrial Practice for Software Architecture Evaluation. Technical Report, Software Engineering Institute, Carnegie Mellon University. CMU/SEI-96-TR-025, January, 1997.

Aho, A.V., Sethi, R., & Ullman, J.D. (1986). *Compilers: Principles, Techniques, and Tools*. Reading: Addison-Wesley.

Alexander, C. (1979). *The Timeless Way of Building*. New York: Oxford University Press.

Alexander, C., Ishikawa, S., & Silverstein, M. (1977). *A Pattern Language*. New York: Oxford University Press.

Allen, R.J. (1997). *A Formal Approach to Software Architecture*. PhD Dissertation, CMU-CS-97-144. Pittsburgh: Carnegie Mellon University.

Ames, A.L., Nadeau, D.R., & Moreland, J.L. (1997). *VRML 2.0 Sourcebook.* New York: John Wiley & Sons, Inc.

Antonakos, J.L. & Mansfield, K.C. Jr. (1999). *Practical Data Structures Using C/C+ +*. Upper Saddle River: Prentice Hall.

Bass, L., Clements, P., & Kazman, R. (1998). *Software Architecture in Practice*. Reading: Addison-Wesley.

Biggerstaff, T.J., Mitbander, B.G., & Webster, D. (1993). The Concept Assignment Problem in Program Understanding. In *Proceedings of the 15th International Conference on Software Engineering*, ICSE '93 (pp. 483-498). Los Alamitos: IEEE Computer Society Press.

Boasson, M. (1995). The artistry of software architecture. *IEEE Software*, 12(6), 13-16.

Booch, G. (1994). *Object-Oriented Analysis and Design with Applications*. Reading: Addison-Wesley.

Booch, G., Rumbaugh, J., & Jacobson, I. (1999). *The Unified Modeling Language User Guide*. Boston: Addison-Wesley.

Bowman, I.T. (1998). *Conceptual Architecture of the Linux Kernel*. http://plg.uwaterloo.ca/~itbowman/CS746G/a2. Published under the auspices of the University of Waterloo, Canada. Online publication: February 12, 1998. Date of last access: February 10, 2001.

Bowman, I.T., Siddiqi, S., & Tanuah, M.C. (1998). *Concrete Architecture of the Linux Kernel*. http://plg.uwaterloo.ca/~itbowman/CS746G/a1. Published under the auspices of the University of Waterloo, Canada. Online publication: January, 1998. Date of last access: February 10, 2001.

Buhr, R.J.A. & Casselman, R.S. (1992). Architectures With Pictures. In *Conference Proceedings, OOPSLA '92 (Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pp. 466-483). New York: ACM Press.

De Michelis, G., Dubois, E., Jarke, M., Matthes, F., Mylopoulos, J., Schmidt, J.W., Woo, C., & Yu, E. (1998). A Three-Faceted View of Information Systems. *Communications of the ACM*, 41(12), 64-70.

Dincel, E., Roshandel, R., & Medvidovic, N. (2000). *ADL Independent Architectural Representation in XML.* Technical Report, Center for Software Engineering, University of Southern California. USC-CSE-2000-519.

Dijkstra, E. (1968). The Structure of the "THE"-Multiprogramming System. *Communications of the ACM*, 11(5), 341-346.

Egyed, A. (1999). *Integrating Architectural Views in UML.* Technical Report, Center for Software Engineering, University of Southern California. USC-CSE-99-TR-514.

Egyed, A. (2000). *Using Patterns to Integrate UML Views*. Technical Report, Center for Software Engineering, University of Southern California. USC-CSE-2000-519.

Egyed, A. & Medvidovic, N. (1999). *Extending Architectural Representation in UML with View Integration*. Technical Report, Center for Software Engineering, University of Southern California. USC-CSE-99-519.

Egyed, A., Nikunj, M., & Medvidovic, N. (1999). *Software Connectors and Refinement in Family Architectures*. Technical Report, Center for Software Engineering, University of Southern California. USC-CSE-99-527.

Eixelsberger, W. & Gall, H. (1998). Describing Software Architectures by System Structure and Properties. In *Proceedings of the 22nd Computer Software and Applications Conference, COMSAC '98* (pp. 106-11). Los Alamitos: IEEE Computer Society Press.

Ellis, W.J., Hilliard, R.F., Poon, P.T., Rayford, D., Saunders, T.F., Sherlund, B., & Wade, R.L. (1996). Toward a Recommended Practice for Architectural Description. In *Proceedings of the Second IEEE International Conference on Engineering of Complex Computer Systems, Montreal*. Los Alamitos: IEEE Computer Society Press.

Feijs, L. & de Jong, R. (1998). 3D Visualization of Software Architectures. *Communications of the ACM*, 41(12), 73-78.

Feijs, L., Krikhaar, R., & van Ommering, R. (1998). A Relational Approach to Support Software Architecture Analysis. *Software Practice and Experience*, 28(4), 371-400.

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading: Addison-Wesley.

Garlan, D., Allen, R., & Ockerbloom, J. (1994). Architectural Mismatch: Why Reuse is So Hard. *IEEE Software*, 12(6), 17-26.

Garlan, D., Allen, R., & Ockerbloom, J. (1994). Exploiting Style in Architectural Design Environments. *Software Engineering Notes*, 19(5): 175-88.

Garlan, D., Ockerbloom, J., & Wile, D. (1998). *Towards an ADL Toolkit*, EDCS Architecture and Generation Cluster, http://www.cs.cmu.edu/~spok/adl/index.html. Date Posted: December, 1998. Date of Last Access: November 13, 2001.

Garlan, D. & Perry, D.E. (1995). Introduction to the Special Issue on Software Architecture. *IEEE Transactions on Software Engineering*, 21(4), 269-274.

Garlan, D. & Shaw, M. (1993). An Introduction to Software Architecture. In Ambriola, V. and Tortora, G. (Eds.), *Advances in Software Engineering and Knowledge Engineering, Vol. I.* Singapore: World Scientific Publishing Company.

Gomaa, H. & Wijesekera, D. (2001). The Role of UML, OCL and ADLs in Software Architecture. In *International Conference on Software Engineering (ICSE 2001): First Workshop on Describing Software Architecture with UML, Toronto, Canada.* Available from Rational Corporation Site: http://www.rational.com/events/ICSE2001/ICSEwkshop/index.jsp.

Harel, D. (1988). On Visual Formalisms. *Communications of the ACM*, 31(5), 514-530.

Hearn, D. & Baker, M.P. (1986). *Computer Graphics*. Englewood Cliffs: Prentice-Hall.

Hilliard, R. (2001). Viewpoint Modeling. In *International Conference on Software Engineering (ICSE 2001): First Workshop on Describing Software Architecture with UML, Toronto, Canada.* Available from Rational Corporation Site: http://www.rational.com/events/ICSE2001/ICSEwkshop/index.jsp.

Hilliard, R. (2001). IEEE Std. 1471 and Beyond. *Position Paper, Software Engineering Institute Workshop on Software Architecture Representation*, Carnegie Mellon University Special Report, CMU/SEI-2001-SR-010, pp. 27-32.

Hofmeister, C., Nord, R., & Soni, D. (2000). *Applied Software Architecture*. Reading: Addison Wesley Longman, Inc.

Holt, R.C. (1996). *Binary Relational Algebra Applied to Software Architecture*, CSRI Technical Report 345, University of Toronto, Canada.

Inouye, J.M. (2001). An Architecture Description Language for Visualizing Neural Network Designs. *Proceedings of the Fifth Multiworld Conference on Systemics, Cybernetics, and Infomatics* (SCI 2001), Orlando, Florida.

Institute of Electrical and Electronic Engineers, IEEE. (2000). *Recommended Practice for Architectural Description of Software-Intensive Systems, IEEE Std. 1471-2000.* New York: Institute of Electrical and Electronics Engineers, Inc.

Inverardi, P. & Wolf, A.L. (1995). Formal Specification and Analysis of Software Architectures Using the Chemical Abstract Machine Model. *IEEE Transactions on Software Engineering*, 21(4), 373-386.

Jacobson, I., Booch, G., & Rumbaugh, J. (1999). *The United Software Development Process*. Reading: Addison-Wesley.

Jazayeri, M., Ran, A., & van der Linden, F. (2000*). Software Architecture for Product Families: Principles and Practice*. Boston: Addison-Wesley.

Kruchten, P. (1995). The 4+1 View Model of Architecture. *IEEE Software*, 12(6): 42-50.

Levine, J.R., Mason, T., & Brown, D. (1992). *lex & yacc*. Beijing: O'Reilly & Associates, Inc.

Luckham, D.C., Kenney, J.J., Augustin, L.M., Vera, J., Bryan, D., & Mann, W. (1995). Specification and Analysis of System Architecture Using Rapide. *IEEE Transactions on Software Engineering*, 21(4), 336-355.

Maier, M., Emery, D., & Hilliard, R. (2001). *IEEE Std. 1471 Frequently Asked Questions (FAQ). Version 3.0.* http://www.pithecanthropus.com/~awg/ieee-1471-faq.html. Online publication: February 28, 2001. Date of last access: July 15, 2001.

Medvidovic, N. & Rosenblum, D.S. (1997). Domains of concern in Software Architectures and Architecture Description Languages. In *Proceedings of the USENIX Conference on Domain Specific Languages, Santa Barbara, California*, pp. 199-212.

Medvidovic, N., Rosenblum, D.S., Robbins, J.E., & Redmiles, D.F. (2000). *Modeling Software Architectures in the Unified Modeling Language*. Technical Report, Center for Software Engineering, University of Southern California, USC-CSE-2000-512.

Medvidovic, N. & Taylor, R.N. (2000). A Classification and Comparison Framework for Software Architecture Description Languages. *IEEE Transactions on Software Engineering*, 26(1), pp. 70-93.

Miller, G. (1956). The Magical Number Seven, Plus or Minus Two: Some Limits on our Capacity for Processing Information. *The Psychological Review,* 63(2), 81-97.

Monroe, R.T., Kompanek, A., Melton, R., & Garlan, D. (1997). Architectural Styles, Design Patterns, and Objects. *IEEE Software*, 14(1): 43-52.

Muller, P. (1997). *Instant UML.* Acocks Green (United Kingdom): Wrox Press Ltd.

Papakostas, A. & Tollis, I.G. (1999). Algorithms for Incremental Orthogonal Graph Drawing in Three Dimensions. *Journal of Graph Algorithms and Applications*, 3(4), 81-115.

Parker, G., Franck, G., & Ware, C. (2000). Visualization of Large Nested Graphs in 3D: Navigation and Interaction. *Journal of Visual Language and Computing, Special Issue on Visual Navigation: Methods and Tools*, 9(3), 299-317.

Parnas, D. (1972). On the Criteria to be Used in Decomposing Systems into Modules. *Communications of the ACM*, 15(12), 1053-1058.

Perry, D.E. & Wolf, A.L. (1992). Foundations for the study of software architecture. *ACM SIGSOFT Software Engineering Notes*, 17(4), 40-52.

Prieto-Diaz, R. & Neighbors, J.M. (1986). Module Interconnection Languages. *The Journal of Systems and Software*, 6(4), 307-334.

Quatrani, T. (2000). *Visual Modeling with Rational Rose 2000 and UML.* Boston: Addison-Wesley.

Selic, B. (2001). On Modeling Architectural Structures with UML. In *International Conference on Software Engineering (ICSE 2001): First Workshop on Describing Software Architecture with UML*, *Toronto, Canada.* Available from Rational Corporation Site: http://www.rational.com/events/ICSE2001/ICSEwkshop/index.jsp.

Shaw, M. (1995). Comparing Architectural Design Styles. *IEEE Software*, 12(6), 27-41.

Shaw, M. (2000). Sufficient Correctness and Homeostasis in Open Resource Coalitions: How Much Can You Trust Your Software System? In *Proceedings of the Fourth International Software Architecture Workshop (ISAW-4)*, affiliated with the Twenty-Second International Conference on Software Engineering (ICSE 2000), Limerick, Ireland, June, 2000, pp. 46-50. Also available from Composable Software Systems Research Group, Carnegie Mellon University, http://www-2.cs.cmu.edu/afs/cs.cmu.edu/project/vit/www/paper_abstracts/isaw4-fin.html.

Shaw, M. & Clements, P. (1997). A Field Guide to Boxology: Preliminary Classification of Architectural Styles for Software Systems. In *Proceedings of the 21st Annual International Computer Software and Applications Conference*, COMPSAC '97 (pp. 6-13). Los Alamitos: IEEE Computer Society Press.

Shaw, M., DeLine, R., Klein, D.V., Ross, T.L., Young, D.M., & Zelesnik, G. (1995). Abstractions for Software Architecture and Tools to Support Them. *IEEE Transactions on Software Engineeering*, 21(4), 314-335.

Shaw, M. & Garlan, D. (1996). *Software Architecture: Perspectives on an Emerging Discipline*. Upper Saddle River: Prentice Hall.

Soni, D., Nord, R., & Hofmeister, C. (1995). Software Architecture in Industrial Applications. In *Proceedings of the 17th International Conference on Software Engineering* (pp. 196-207). Los Alamitos: IEEE Computer Society Press.

Sugiyama, K., Tagawa, S., & Toda, M. (1981). Methods for Visual Understanding of Hierarchical System Structures. *IEEE Transactions on Systems, Man and Cybernetics*, SMC-11(2), 108-125.

Van der Hoek, A., Rakic, M., Roshandel, R., & Medvidovic, N. (2000). Taming Architectural Evolution. Technical Report, Center for Software Engineering, University of Southern California, USC-CSE-2000-523.

Ware, C. & Franck, G. (2000). Evaluating Stereo and Motion Cues for Visualizing Information Nets in Three Dimensions. *ACM Transactions on Graphics*, 15(2), 121-139.

Ware, C., Hui, D., & Franck, G. (1993). Visualizing Object Oriented Software in Three Dimensions. In *Proceedings of the Centre for Advanced Studies Conference, CASCON '93* (pp. 612-620). Toronto: IBM Canada Ltd.

Web3D Consortium. (2001). *The Web3D Consortium Home Page. http://www.web3d.org*. Date last modified: December 14, 2001. Date of last access: December 14, 2001.