AFIT/GCS/ENG/93M-05

(1)

DEVELOPMENT OF A VISUAL SYSTEM INTERFACE TO SUPPORT A DOMAIN-ORIENTED APPLICATION COMPOSITION SYSTEM

THESIS

Presented to the Faculty of the School of Engineering

of the Air Force Institute of Technology

Air University

In Partial Fulfillment of the

Requirements for the Degree of

Master of Science in Computer Engineering

Timothy Lee Weide, B.S.C.S.

Second Lieutenant, USAF

152pp

March 23, 1993

Approved for public release; distribution unlimited

20001026170

AFIT/GCS/ENG/93M-05

DEVELOPMENT OF A VISUAL SYSTEM INTERFACE TO SUPPORT A DOMAIN-ORIENTED APPLICATION COMPOSITION SYSTEM

THESIS

Timothy Lee Weide
Second Lieutenant, USAF

AFIT/GCS/ENG/93M-05

Acknowledgements

I wish to express my deep appreciation to my thesis advisor, Major Paul Bailor, who has given me invaluable guidance and encouragement over the last year-and-a-half. Also, I'd like to thank my thesis readers, Major David Luginbuhl and Lieutenant Colonel Elton P. Amburn, whose comments and suggestions helped improved the clarity of this thesis. I owe much gratitude to the other members of the KBSE group: Brad Mallare, Mary Boom, and especially Cindy Anderson and Mary Anne Randour, whose research provided a solid foundation for my own work.

I'd like to thank my family, whose unwaivering faith in me has meant so much: my parents, Bill and Carol Weide, for their prayerful support; my children, Alicia, Andrea, and Matthew, for helping me to keep things in perspective; and finally, my wife and best friend, Chris, for all of the personal sacrifices she has made. She has given me the best years of my life and I dedicate this thesis to her.

Timothy Lee Weide

Table of Contents

Page

Page

List of Figures

List of Tables

Abstract

This research designed and prototyped a visual system interface to generate, display, and modify domain-oriented application specifications. A visual system interface, called the Architect Visual System Interface (AVSI), supplements a text-based environment, called Architect, previously developed by two other students. Using canonical formal specifications of domain objects, Architect rapidly composes these specifications into a software application and executes a prototype of that application as a means to demonstrate its correctness before any programming language specific code is generated. This thesis investigates visual techniques for populating, manipulating, viewing, and composing these software application specifications within the formal object base scheme required by Architect. A Visual Specification Language (VSL) was developed to define the visual display characteristics of domain objects. AVSI provides automatic diagram layout, and also produces a textual display in a domain-specific language. The Software Refinery environment, including its graphical interface tool INTERVISTA, was used to develop techniques for visualizing application data and for manipulating the formal object base. AVSI was validated with a well-understood domain, digital logic, and was found to significantly enhance Architect's application composition process.

DEVELOPMENT OF A VISUAL SYSTEM INTERFACE TO SUPPORT A DOMAIN-ORIENTED APPLICATION COMPOSITION SYSTEM

I. Introduction

1.1 Background

The method used for programming computers is generally a function of the currently available technologies. We have progressed from programming with punched cards and setting hardware switches to programming with high-level languages within sophisticated interactive programming environments. While modern language technology has enabled these languages to provide an increasingly higher level of abstraction, the increasing size and complexity of the problems we are asked to solve with computers often results in programs being incomprehensible to any single individual. Therefore, the process of developing software applications grows more difficult to manage as these problems become larger and more complex (8:5–10).

The problem is often exacerbated by the difficulty of knowledge transfer between the user and the software engineer. Program specifications are traditionally formulated from informally stated system requirements. These requirements are usually stated in a natural language, such as English, and are frequently supplemented by informal diagrammatic information. The software engineer must transform this informal information into an unambiguous computer solution. Unfortunately, requirements are often incomplete, inconsistent, ambiguous, or poorly communicated, resulting in computer systems that do not meet the user's expectations or requirements (45:110).

A new approach to software development is currently being researched by the Knowledge Based Software Engineering (KBSE) research group at the Air Force Institute of Technology (AFIT). In this approach, a domain-oriented application composition system allows a sophisticated end-user, called an application specialist, to use the building blocks (or "components") of his domain to compose his own software applications. It is desirable to enable the application specialist, who knows the most about the problem to be solved, to encode the solution in a manner which he easily understands, i.e., the language of his domain. This language may take on various forms, both textual and visual, and is at a higher level of abstraction than conventional "languages of computation." A major goal is to allow the application specialist to develop his own application directly, without being burdened with the excessive details of how to model the problem for the computer, or by the need to communicate his requirements to a "middleman" (43:4-6).

In a domain-oriented application composition system, the general structure of a software architecture, along with the components of the domain to be modeled, are maintained in a formal object base. The domain components are developed by a domain engineer, who is an expert in modeling the real-world objects in his domain, and formalized by the software engineer. The application specialist builds a specification for an application by composing these domain components according to the domain model defined by the domain engineer and the software engineer (for a discussion of roles and interactions of these individuals, refer to appendix A). The end product of this process is an automatically generated software specification, from which target code such as Ada may be produced.

Figure 1.1 shows a simplified conceptual diagram of the prototype domain-oriented application generator, called *Architect*, being researched by the KBSE group at AFIT. The original implementation of Architect was developed by Capt Cynthia Anderson (2) and Capt Mary Anne Randour (39). Architect is implemented within the REFINE^{TM} wide-spectrum language environment. The domain model is defined by the body of existing knowledge within the problem domain, and is further defined by software architecture information from the software architecture model. The syntax of the domain-specific language is defined by a grammar, whose structure is determined by the domain model. Application-specific information encoded in this language is parsed by DI-

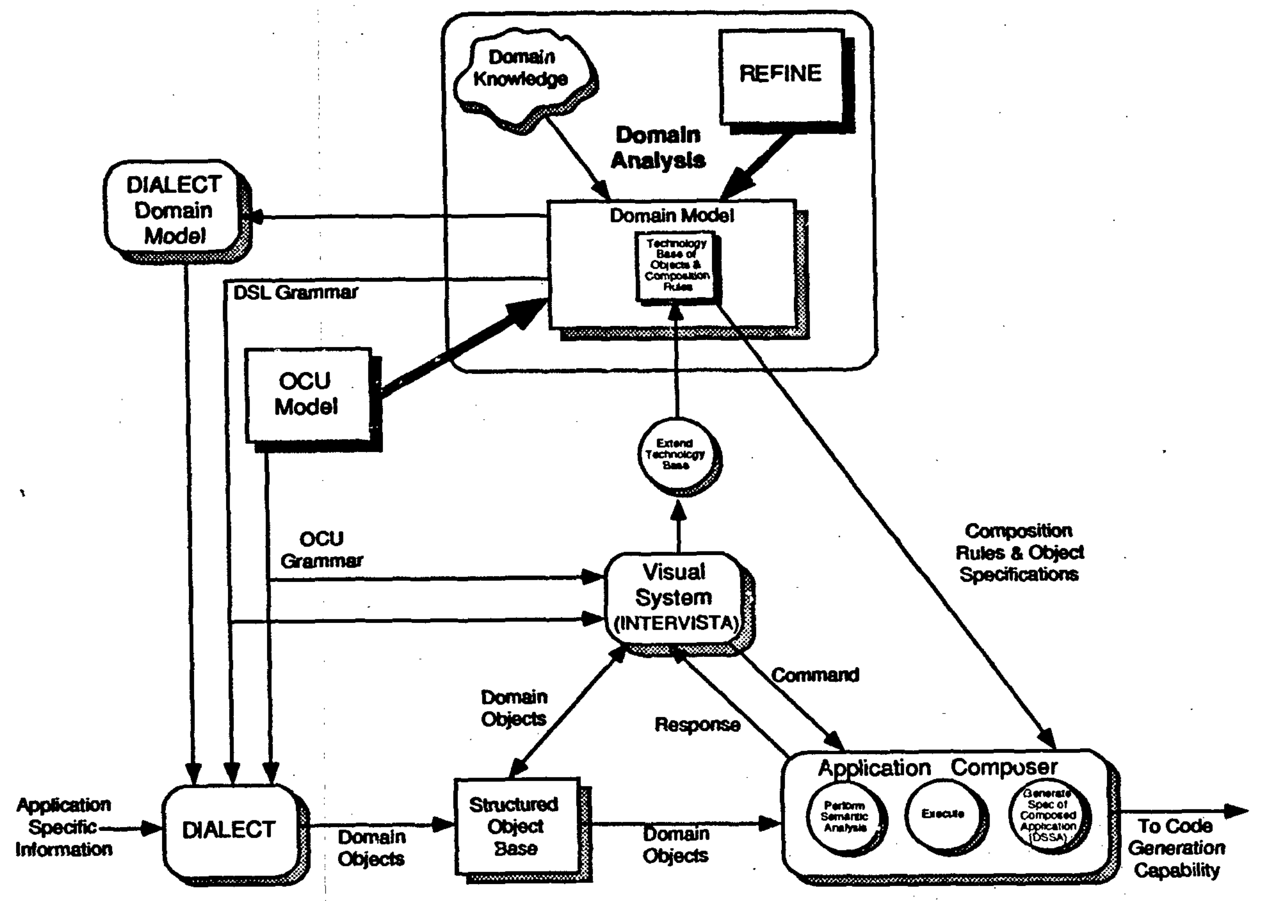

Domain Knowledge
REFINE
Domain Analysis
DIALECT Domain Model
Domain Model
Technology Base of Objects & Composition Rules
DSL Grammar
OCU Model
Extend Technology Base
OCU Grammar
Visual System (INTERVISTA)
Composition Rules & Object Specifications
Command
Domain Objects
Response
Application Specific Information
DIALECT
Domain Objects
Structured Object Base
Domain Objects
Application Composer
Perform Semantic Analysis
Execute
Generate Spec of Composed Application (DSSA)
To Code Generation Capability

Figure 1.1. "Architect System Overview"

ALECT, a tool that exists in the REFINE environment. DIALECT builds objects in the structured object base, structuring them according to the object hierarchy defined by the DIALECT domain model. This information contained within the structured object base is then used by the application composer to produce a final software specification. Appendix A gives a more detailed look at Architect.

A further enhancement to this development model is a visual system, which gives a graphical representation of the domain-specific language and allows the user to generate, view, and manipulate application specification information. A visual interface is also useful to the software engineer, who is given visual access to the information about the software architecture components within the domain. Shu (43) provides motivation behind providing a visual system:

1. Pictures are more powerful than words as a means of communication. They convey more meaning in a more concise unit of expression.
2. Pictures aid understanding and remembering.
3. Pictures may provide an incentive for learning to program.
4. Pictures do not have language barriers. When properly designed, the are understood by people regardless of what language they speak. (43:8-9)

The visual representation enhances understanding of the problem being modeled. Producing this visual interface was the goal of this thesis effort.

1.2 Problem Description

More precisely, the problem addressed was to provide a visual interface for generating, viewing, and manipulating domain knowledge and software architecture specifications in a formalized object base. The visual interface was built for creating, viewing, and modifying applications within the Architect domain-oriented application composition system. The name given to the visual interface was the Architect Visual System Interface, or AVSI. AVSI enhances the following basic capabilities provided by Architect:

1. Creating an application. With Architect, application specialists create applications by writing application specifications in a domain-specific language. The application is built by parsing the file containing the application specification. AVSI allows the application specialist to create an application by direct manipulation of visual objects while transparently generating the application specification in the domain-specific language.
2. Editing object base. Architect allows the application specialist to interactively edit objects that reside in the object base. AVSI gives visual support for editing the structure of the application as well as editing the internal attributes of individual objects in the object base.
3. Viewing an application. The only viewing capability the original implementation of Architect provides is textual. AVSI provides the application specialist with a graphical view of the various aspects of an application, such as structure, control flow, and communication paths between objects. Additionally, the visualization of the information in the object base is generated automatically, with little or no manual intervention.
4. Establishing Communication links between objects. In Architect, this step is called "preprocessing." Architect queries the user to provide communication paths between objects when it cannot resolve which objects communicate. AVSI provides visual support for establishing these connections.
5. Overall control of the Architect system. Architect is controlled by a command line interface. AVSI provides an integrated environment using windows and menus to allow the application specialist to perform any function. A general goal for AVSI was to provide an interface that is intuitive and easy to use.

1.3 Assumptions

Two basic assumptions were made during the research. The first assumption dealt with the sufficiency of the validation domain, digital logic, which was also used by Anderson and Randour for Architect. Though this domain is relatively small, it is well-defined and was assumed sufficient for demonstrating AVSI's functionality. The second assumption was that the graphics capabilities provided by the INTERVISTA tool would be powerful enough to develop the visual system.

1.4 Scope

AVSI's main emphasis was on the "front end" of the application composition process, which includes generating, viewing, and manipulating an application. Because of time constraints, the scope was limited:

- AVSI does not provide visual support for two stages of Architect's application composition process: semantic checks and application execution.
- AVSI only deals with objects' static properties; viewing and manipulating the dynamic behavior is beyond the scope of this project, but it will undoubtedly need to be researched in the future.

1.5 Approach and Sequence of presentation

The following approach was taken to achieve the objectives of this research:

1. A search of current literature provided information concerning visual user interfaces, visual programming, visual representation of knowledge base information, and screen layout algorithms. Chapter II presents an overview of current literature in these areas.
2. The next step was to assimilate the design of Anderson and Randour's Architect system, and develop a strategy for visualizing the structure of an application as defined by the Architect domain model. This strategy required traversing the abstract syntax tree structures in the object base to retrieve information about the objects and their structure. Additionally, it required creating visual objects (windows, icons, links, etc.) and associating these visual

objects to the objects they represent. INTERVISTA was used to create these visual objects in the REFINE object base. Chapter III gives the operational concept for AVSI.

3. A Visual Specification Language (VSL) was developed to define the domain-specific visual characteristics of domain objects. DIALECT was used to create the parser for VSL.

4. Techniques were developed for creating and modifying application data contained in the REFINE object base, and for establishing communication links between objects in an application. An integral part of this approach was the use of the Software Refinery environment. This environment provided all of the necessary tool support for the research. The REFINE wide spectrum specification language provided the ability to access the object base. DIALECT was used to create a parser for the Visual Specification Language. INTERVISTA provided visual support, including windows, icons, and mouse-sensitive text windows. Chapter IV presents the design and implementation of AVSI. Chapter V discusses the validation domain used, and presents an analysis of AVSI. Chapter VI discusses conclusions that may be drawn from this research and recommendations for future research.

Some additional information concerning Architect and AVSI is given in several appendices. Appendix A contains a basic description of Architect. Appendix B provides a sample session with AVSI, wherein an application in the digital circuits domain is built. Appendix C lists the files needed for AVSI. Appendix D gives the definition of VSL (its DIALECT domain model and grammar) and lists the visual specification for the digital circuits domain. Appendix E contains a listing of the REFINE source code for AVSI.

II. Literature Review

2.1 Introduction

The objective of the literature review was to examine current research in the areas of visual user interfaces, visual languages, visual programming systems (which include program visualization and visual programming), and algorithms for the automatic generation of diagram layouts. The review consisted of a "filtering" process, in which information, both general and specific, was examined to find information useful for developing the Architect Visual System Interface. Some of the literature deals with the overall goals of a visual interface. Different visualization techniques for visual programming and program visualization were surveyed, and several examples of visual interfaces for knowledge-based systems were examined.

2.2 Visual User Interfaces

2.2.1 Goals. Much emphasis has recently been given to human factors in interface design. The human-machine interaction needs to support clear and efficient communication and "support the user's tasks, plans, and goals" (31). A number of characteristics of a "good interface" are found in the literature. In "Visual Interface Design Systems," Huang (21) discusses the criteria of an effective user interface. He states a visual user interface should:

- Be intuitive
- Accommodate a wide range of skills
- Be customizable
- Be extensible
- Give plenty of feedback
- Be predictable
- Be consistent (most important)

These goals were kept in mind during the development of AVSI. Further goals are presented by Eisenstadt, who in "Visual Knowledge Engineering," gives criteria for a representation that "provides a good mapping to the way the programmers themselves tend to formulate solutions."

These criteria are mappability, manipulability, salience, scope, visibility, coupling, navigability, completeness, and convertibility (17).

2.2.2 Visual Interfaces for Knowledge-Based Systems. Since Architect is a knowledge-based system, examples of visual interfaces for similar systems provided insight pertinent for the development of AVSI. An interface can be considered "intelligent" if "intelligence is embodied in an underlying knowledge-base (12:404)." This section discusses some examples of systems which use this approach.

2.2.2.1 Graphical Knowledge-Based Model Editors. One source in particular describes a system that possesses the qualities desired for AVSI and appears to closely correspond to Architect. In "Graphical Knowledge-Based Model Editors," Cypher and Stelzner (12) discuss the graphical representation of information contained within knowledge-based systems. In what they call a graphical knowledge-based model, domain knowledge is embodied within an underlying knowledge base, and the graphical interface is a "dumb" interface to an intelligent system. The interface facilitates the visualization and manipulation of complex domains, and thus the objects and relationships contained within the domain are "evident and manipulable in the graphical display" (12:404). Cypher and Stelzner describe SimKit, which is a set of tools for building graphical model editors in a wide range of application domains. With such a graphical model editor, the user can build knowledge base information by creating icons, moving them on the screen, and linking them together with arrows.

The authors make a distinction between models, which describe a particular set of objects and the relations between those objects, and libraries, which contain the generic classes of objects and relationships. A single library can be used to produce several different models. A model is built with a domain-specific editor, which is built around the domain-specific information contained within the library. The information contained within the library constrains the information that may be contained within the domain-specific editor. Though library building and maintenance require programming experience, a model is generally built by non-programmers who are considered experts in their application domain.

A typical model-building session proceeds as follows: First, the user is presented with a display of icons, each of which represents a domain object contained within the library. The user then selects icons and places them on the screen. This action results in the creation of instances of the corresponding objects in the library. Finally, these icons are connected with lines, which represent relations between objects. The library descriptions of object classes also contain behavioral descriptions and initial values for object state information, and therefore, the model's behavior can be simulated once the model has been built.

SimKit has been used by over 100 users for building graphical model editors over a broad range of application domains. Such applications include factory control systems, battle management, and construction project management systems (12:408).

A significant correlation exists between SimKit and Architect, and this served as the basis for a considerable amount of the thought process behind the design of AVSI. SimKit and Architect are clearly parallel efforts, and it might be useful in future research efforts to consider interfacing SimKit to Architect. An advantage Architect has over SimKit is its development environment. The Software Refinery environment allows for very rapid development and provides integrated support for languages, graphics, and object-base manipulation. AVSI resides in this environment and retains Architect's capability to express domain objects in a domain-specific language, a notion that is missing in SimKit.

2.2.2.2 Visual Knowledge Engineering. Another visual "toolkit" for building visual knowledge-based models is the Knowledge Engineer's Assistant (KEATS) (17). Among the many tools in the suite are:

- FLIK (Frame Language in Keats), a frame-based knowledge representation language

- Rule interpreters
- Tables: a spreadsheet-like, table-based interface for data acquisition
- GIS (Graphical Interface System), a "direct manipulation" interface for knowledge bases.

The main emphasis of the KEATS system is to provide a "graphical representation of program behavior which provides a good mapping to the way the programmers themselves tend to formulate solutions" (17:1166). The GIS was designed to enable the representation of objects, relations, and dependencies in a manner similar to drawing on a blackboard. A certain amount of flexibility has been maintained by allowing the editing of the same knowledge base information by using either the GIS or a code editor. Variation of the granularity of information is provided by expanding and collapsing of visual information. Furthermore, simultaneous views are possible. A coarse-grained view, which shows a more global picture, may be viewed in conjunction with a fine-grained view, which gives a more detailed picture. This article provided a good overview of the types of information a developer needs to develop an application, and how this information should be presented.

2.2.2.3 Lockheed's Graphical Development Environment. In another knowledge-based system, Lockheed's Automatic Programming Technologies for Avionics Software (APTAS) System (29), an engineer constructs a tracking system by inputting specification data via a "dynamic forms interface." When the specifications are complete, an architecture generator automatically constructs an architecture from the "tracking taxonomy and coding design knowledge base" information. The architecture can then be viewed with the graphical interface which displays the architecture in a boxes and arrows format. The display supports the viewing of hierarchical information by "zooming in" when an icon is clicked on. The user can edit the architecture by making changes to its graphical representation.

Part of their Graphic Development Environment, Lockheed's Graphical System Description Language (48), maps system descriptions to their graphical representations. This language consists of three parts, the Type Sublanguage (TSL), the Declarative Sublanguage (DSL), and the Visual Sublanguage (VSL).

The Type sublanguage defines the primitive types, relations, and type classes contained in a particular domain. This provides templates for instantiations of objects, with the addition of default attributes.

The Declarative sublanguage is used for system description. Its basic components are scope objects, declarations, and relations. Scope objects group declarations and relations to define a type. Declarations are used to name instances of types. Finally, relations declare the named relations between these instances.

Both the TSL and DSL support embedded help information in their type descriptions. This help information is textual information to be used while developing applications, and aids the user by giving information about particular object types.

The Visual sublanguage is used to map DSL objects to their graphical representations. This mapping is based on the object's class, type, or relation type. This mapping assumes that all objects of the same type have the same basic look, differing in labeling characteristics. The VSL associates each type or type class with a specific icon object, which may be simple or complex. Other information, such as default position, grouping, display depth, and hierarchical information, is specified by this language.

The Graphical Development Environment, along with Graphical System Description Language, provided a good example of a system with some goals similar to those of Architect and AVSI. The Declarative and Type sublanguages provide an equivalent to Architect's domain model, and the role of the Visual sublanguage played an important role in defining AVSI's Visual Specification Language.

2.3 Visual Languages

Chang (11) presents a "formal specification of iconic systems using generalized icons." He provides the formal basis for a visual programming system in which each icon possesses both a physical and a logical representation. Icons can be one of several types: elementary, complex, composite, or structural. He further defines a set of operations for the iconic system. Iconic sentences can be both syntactically and semantically analyzed by a "visual language compiler."

A visual language requires a defining structure, a grammar. Huang (21) discusses the three most common forms of specifying grammar systems. These are Backus-Naur, state-transition diagrams, and object-oriented framework. Huang claims the object-oriented approach offers the most advantages for user interface design. He enumerates the advantages as follows:

1. It not only supports the separation of the interaction between user and application in a natural way, but it also enhances the development of a direct manipulation interface.
2. The interactive interface objects can be constructed quickly by modifying existing objects. This is a style of programming from example.
3. It can support the construction of multiple interfaces to a given application, which in turn can be used according to available I/O devices or user preferences.
4. It promotes adherence to interface standards by making it easy for interface designers to use code that has already been designed to meet those standards.
5. It enables natural partition of a task to be run on separate processors in distributed environment. (21:121-122)

Since the Software Refinery is an object-based environment, the object-oriented framework offers a further advantage in the development of AVSI, which directly uses the structure of Architect's objects in the REFINE objects base.

2.4 Visual Programming Systems

AVSI requires a two-way interaction with the programming system. A distinction can be made between two directions within the interaction. Program visualization is "the graphical display of program code or sys' m documentation," whose goal is to "help programmers form clear and correct mental images of a program's structure and function" (10). Visual programming, on the other hand, allows the user to interactively program using visual information. (22)

2.4.1 Program Visualization. AVSI's development required an answer to the question, "what kinds of information should be displayed, and how should this information be presented?" Program visualization can take on any number of forms. In systems such as Brown University's "program-development system," Pecan (1), several different "views" of a program are provided. In Pecan, an abstract syntax tree is used to produce multiple concurrent views, such as structured flow graphs, the program listing, the program's declarations, and symbol table information. This kind of "multi-media" approach supports a system design which is based on a "unified view of language," where language is viewed as an integration of communication modes (verbal, textual, etc.). Indeed, researchers in the artificial intelligence community have suggested advantages in including verbal, gestural, and tactile information within communication (33).

Pegasys (Programming Environment for the Graphical Analysis of SYStems) uses formal graphical information to represent the hierarchy of program "entities" such as subprograms, modules, and data objects. It performs consistency checks among the entities, and aids in the design of programs by graphically describing the relationships between data structures and algorithms (32).

In his thesis "Graph-Based Visualization of Formal Specification and Domain Specific Language," Langloss (27) implements a graph-based visual language called Visual Refine. Visual Refine provides a graphic view of the abstract syntax tree representation of a program written in the REFINE specification language. Each REFINE object and operation is represented by an icon, and the relationships between objects and operations are shown as connecting lines. Visual Refine is a very simple system, and requires a great deal of manual intervention. Displaying visual data requires the user to manually traverse the abstract syntax tree and find applicable REFINE rules to display each individual node. Once a rule has been applied to a node, an icon is created, which the user then manually places in the diagram. Though Visual Refine's functionality is very limited, it served as a good starting point for the development of AVSI. It also provided an example of a visual system that used the Software Refinery environment.

2.4.2 Visual Programming. Though systems such as Pegasys and Visual Refine aid the developer by allowing a visual representation of a program or program design, they lack the ability to create or modify a program. However, this ability was a necessary feature for AVSI. Several systems have been developed (or proposed) which provide this visual programming capability.

Arefi (3) proposes a visual, syntax-directed editor for visually editing a program. Such an editor requires a visual specification language, which is used to define the syntax and semantics of a visual programming language, to produce a syntax-directed editor. Similar to Visual Refine, Arefi's visual programs are represented as directed graphs. However, a graph-to-program transformation is also provided. The editor provides a means to edit the graph according to the syntax of the visual language. Each editing operation represents one or more graph transformation rules supplied by the language specification.

The notion of using a visual, syntax-directed editor is also discussed by El-Kassas in "Visual Languages: Their Definition and Applications in System Development (18)." El-Kassas discusses a model called an Attribute Icon-replacement Grammar, which "describes the rules for constructing well-formed graphs of interconnected icons." In this model, the editor is seen as a "derivation engine," which is used to create graphs consisting of terminals and non-terminals. The user can use the derivation engine to select a non-terminal symbol (a symbol to be replaced) and then select the production rule to apply to it. El-Kassas asserts that this method is well-suited for creating new graphs from a starting symbol, as well as modifying partial structures within the graph.

This concept proved to be useful for AVSI, which uses the syntax structure defined by Architect's domain model. This domain model explicitly defines how AVSI creates an application's object structure in the REFINE object base and provides a basis for directing the user through the object composition process.

2.4.2.1 Form-Based Dialogue. Though most of the systems examined in this research rely on a "direct manipulation" strategy, i.e., selecting, moving, and linking icons on a graphical display, another approach offers several advantages. "Form-based" dialogue interfaces have been used successfully in systems which require complex commands and data sets. Shu (43) cites several examples, such as QBE, a database query language, and FORMANAGER, a system which allows data definition, entry, updating, and query by completing forms. These systems are both built on an underlying relational table model (43:239-284). As mentioned previously, the APTAS system (29) relies heavily on a form-based dialogue, and AVSI was designed to take advantage of this capability to a limited degree.

2.5 Automatic Layout Algorithms

In Visual Refine, Langloss requires a large amount of manual intervention to correctly display the graphs corresponding to the abstract syntax tree representation of a REFINE program. Automating the process requires automatic graph drawing routines. To provide a user-friendly interface, it is desirable to minimize the amount of user intervention required to display visual information. In fact, an issue being examined in this thesis research is that of automatically generating the screen layout. The following sources provided insight into this issue.

Eades and Xuemin (16) examine the criteria and basic steps used to draw a directed graph. They state three aesthetic criteria. First, arcs pointing upward should be avoided. Second, nodes should be distributed evenly over the page. Last, there should be as few arc crossings as possible. The three steps for deriving the graph are: remove cycles from the graph, layer the acyclic graph, and finally position each node in each layer of the proper layered network. This algorithm is a generalization of several other graph-drawing algorithms.

Protsko, in "Toward the Automatic Generation of Software Diagrams (38)," discusses the criteria for drawing data flow diagrams, and spells out the placement and routing strategies in a system called MODRIAN. Other issues such as readability, shape, hierarchy, and compaction are examined in "Automatic Graph Drawing and Readability of Diagrams. (47)"

2.6 Conclusion

Current research provides an wide range of information on visual programming systems and visualization techniques. There seems to be a general lack of consensus on which techniques are the most valuable (43:9–10), but much was learned by examining the different approaches. AVSI was designed to take advantage of, and synthesize, the features of several example systems. Such features include direct manipulation, form-based interaction, syntax directed editing, and visual specification languages.

III. Operational Concept for Visual System

3.1 Overview

The Architect Visual System Interface (AVSI) is a visual system for the domain-oriented application composition system developed by Anderson (2) and Randour (39) called Architect. Appendix A contains a high-level overview of Architect. AVSI was developed along with, but a few months behind, Architect. AVSI directly uses the object structure defined by Architect's domain model but uses INTERVISTA's graphics capabilities to visualize and manipulate the formal object base. Anderson and Randour, in the early stages of Architect's development, used a very simple artificial domain that used simple primitives such as "widgets" and "gadgets," which have no counterparts in any "real" domain. This allowed them to avoid thinking in the terms of any specific domain. To validate their system, they moved to the digital logic (or "circuits") domain. AVSI's development and validation used these same domains.

The motivation behind providing a visual system for Architect was to give the application specialist, as well as the software engineer, an environment which is intuitive and as easy to use as possible. Instead of using a purely visual system, textual information was incorporated in some places for easier understanding of the underlying application. In this manner, textual and visual data are combined synergistically (33) to represent the information the user needs to develop an application. AVSI provides a considerable amount of functionality without using extremely sophisticated graphics. All of its visual functions rely exclusively on the capabilities provided by INTERVISTA. While lacking features such as complex icons and colors, INTERVISTA gives an easy way to access the objects in the REFINE object base. Since AVSI is intended as a proof-of-concept, rather than an industrial-strength tool, INTERVISTA was sufficiently powerful to provide the capability for program visualization as well as for visual programming. Much of INTERVISTA's power lies in its ability to map visual objects (icons, windows, etc.) to the logical objects in the REFINE object base. Moreover, it allows for mapping in the other direction, i.e. from the visual objects to the logical objects. Finally, it provides high-level graphical functions, thus avoiding many low-level programming issues.

AVSI serves as the interface between the user (the application specialist or software engineer) and Architect. The underlying system, as developed by Anderson and Randour, was designed according to the Software Engineering Institute's Object-Connection-Update (OCU) model (28).

According to the OCU model, an application is composed of, and ultimately implemented by, subsystems. The OCU representation of a subsystem is shown in Figure 3.1. A subsystem consists of four basic parts: the controller, the objects, the import area, and the export area. The controller, which is the locus of control for the subsystem is "connected to" the objects that it controls. The collection of these "controllees" may consist of primitive objects or other subsystems, which themselves control other objects. The import and export areas provide the communication links between subsystems.

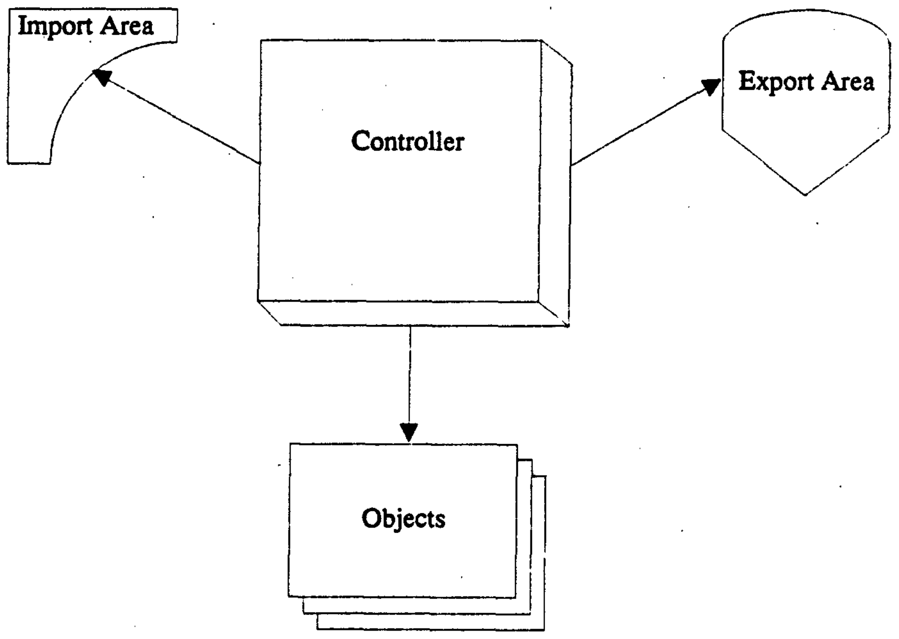


Figure 3.1. OCU Subsystem's Visual Representation

The OCU's conventions for visual symbols (icons) were also followed, where appropriate, in AVSI. The features, requirements, and the operational concept of the AVSI are described in the following sections.

3.2 Visual System Features

3.2.1 Visualization. AVSI furnishes the application specialist with a graphical view of the application as it is being developed:

1. AVSI provides a facility for visually composing formally defined, pre-existing components contained within the technology base, and it gives a graphical view of the structured object base as well as the application software architecture generated to satisfy the system requirements. For example, AVSI provides visual information about the hierarchical relationships between an application, its subsystems, and primitive objects.
2. Another view of the application shows the communication links between import and export objects in the import and export areas of the subsystems.
3. The user is provided with concurrent multiple views of system information wherever possible. For instance, graphical information about a subsystem in the form of diagrams, showing objects and relationships, may be displayed along with textual information about object attributes in tabular form. These displays may be further supplemented by text-based syntactical information, using the grammar which defines the domain objects.

3.2.2 Visual Programming. A further feature of AVSI is its visual programming capability. The application specialist and software engineer are able to manipulate the structured object base by direct manipulation of icons and links. Creating an icon according to predefined rules results in the creation of an instance of a corresponding domain object in the structured object base.

Connecting one icon to another by creating and placing a link between them results in the creation of a logical link between the corresponding objects in the structured object base.

3.2.3 Object Attribute Editor. Whenever an object is thus created, further information is usually required. In general, this information (attributes, state variables, and algorithms, etc.) already exists within the object because default values are assigned whenever an object instance is created. If it does not, the user may interactively enter this information with an object attribute editor. The object attribute editor allows the applicati specialist to view and modify the internal attributes of objects contained in the structured object base.

3.2.4 Visual specification language. The visual representation of each type of primitive domain object must be defined prior to creating an application within that domain. Information such as icon type, size, and shape must be declared before a graphical representation can be presented for a primitive object. To make this information easy to provide, a visual specification language capability such as Lockheed's Graphical System Description Language (48) affords a grammar-based specification method which is relatively easy to use, easy to modify, and which puts this information in a standard format for wide applicability across domains.

3.2.5 Menu system. The menu system provides the user a way to enter commands to the application generator. Depending on the context, possible choices of commands are enumerated in the menu format. Such commands invoke parsing of text files, editing objects, semantic checking, and execution. A main control panel window contains "buttons" for the major application composition functions.

3.2.6 Syntax-directed editing. The visual editing capability provided by AVSI augments Architect's previous method of parsing text files to build an application in the REFINE object base. AVSI's visual editing of the application follows the syntax of the same grammar used by the parser. Thus AVSI provides syntax-directed editing which guides the user in building correct applications.

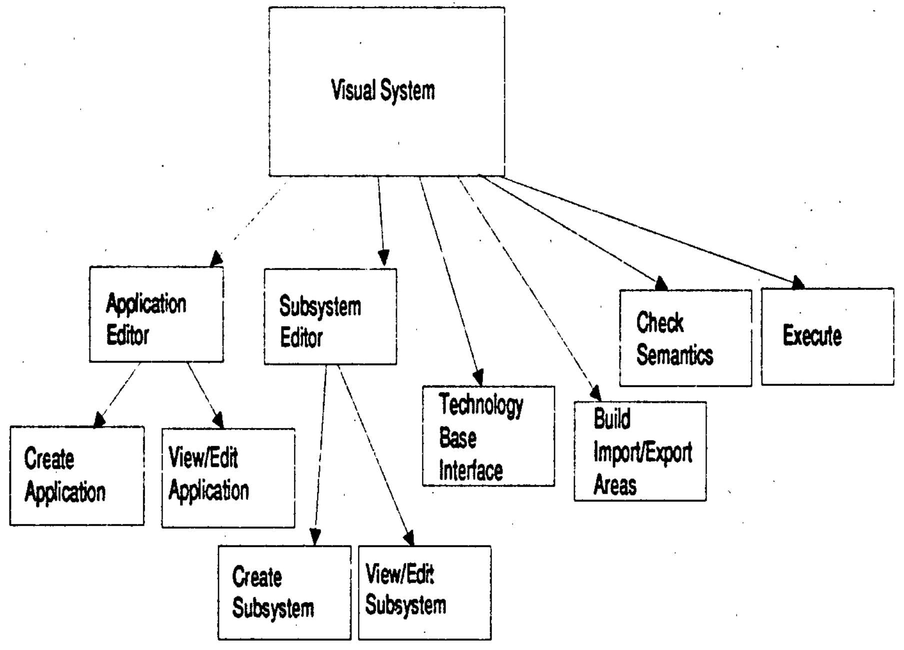


Figure 3.2. Tree Diagram of Requirements

3.3 Visualization Requirements

Since AVSI's purpose was to be an interface for Architect, its requirements centered around the basic capabilities provided by Architect. Figure 3.2 shows a tree diagram for its requirements. Each of the requirements is discussed in more detail below, and an operational concept of a system that meets these requirements is provided in section 3.4.

3.3.1 Application Editor. The application editor must provide the ability to view, create, and edit an application: Creating an application builds the basic structure required to generate an application specification. It is done either by parsing a text file or interactively with the visual system. Viewing and editing an application is done by displaying the application and allowing direct manipulation of the display.

3.3.2 Subsystem Editor. The subsystem editor must provide a mechanism for creating, viewing and editing subsystems, including all objects from which the subsystems are composed.

3.3.3 Technology Base Interface. The technology base interface must provide the ability to retrieve and store domain objects in the technology base. These objects include primitive objects, generics, and subsystems built and stored in previous sessions.

3.3.4 Build Import/Export Areas. AVSI must provide a method of viewing and editing subsystems' import and export areas, and the connections between import and export objects.

3.3.5 Check Semantics. Though no visual support is provided, an interface is provided to call Architect's semantic checking routines.

3.3.6 Execute. Though no visual support is provided, an interface is provided to call Architect's application execution function.

3.4 Operational Concept

Applications are generated by populating the structured object base with instances of domain objects and composing them according to predefined system composition rules. The application specialist begins this process by creating a new application which is either built "from scratch" or is parsed from a text file containing a description of an application. The user adds new subsystem objects or primitive-objects to the application in the same manner, either by creating new object instances or by using "saved" objects from the technology base. The saved objects were previously saved as text files and are parsed into the object base.

The subsystem editor allows the user to further define a subsystem by adding controllees, which may either be subsystems or primitives. The structure of the subsystem and its controllee objects are viewed and edited with the subsystem editor. The attributes of any object may be viewed and edited with the object attribute editor.

The communication links between the subsystem-objects must also be defined. Once the subsystems are all created and fully defined, the application specialist issues a command to the application generator to ascert. 'n the correctness of the specification by performing semantic checks on the structured object base. Should any semantic errors exist in the specification, the application specialist may correct these errors by editing the objects in the structured object base. When the specification is error-free, an "execute" command may be issued to observe system behavior. Finally, a command may be issued to generate a formal specification suitable for code synthesis.

The following sections provide an operational concept for the Architect Visual System Interface. Where there are alternative approaches to a step in the process, the different approaches are discussed separately.

3.4.1 Creating an Application. The first step in creating a new application is to select the application domain. AVSI simply lists the possible domains in a multiple-item menu and prompts the user to make a choice.

There are two basic ways to create a new application:

1. *Parse Text File* Based on the chosen domain, the user chooses from existing application definitions written in the appropriate domain-specific language. Upon selection, the file is parsed into the structured object base. This text file may either be a hand-coded text file, containing an application definition, or a saved application, which is a system-generated text file containing an application definition previously contained within the structured object base. A saved application contains additional inf rmation about the communication links between the application's subsystems. These two file types are essentially the same, and the steps required for parsing are identical.

2. *Create Application* To create an application "from scratch" the user is only asked to provide an name for the new application. AVSI creates the basic application structure to which new subsystems and primitives can be subsequently added.

3.4.2 Subsystems. AVSI provides the ability to view, build, and edit subsystems.

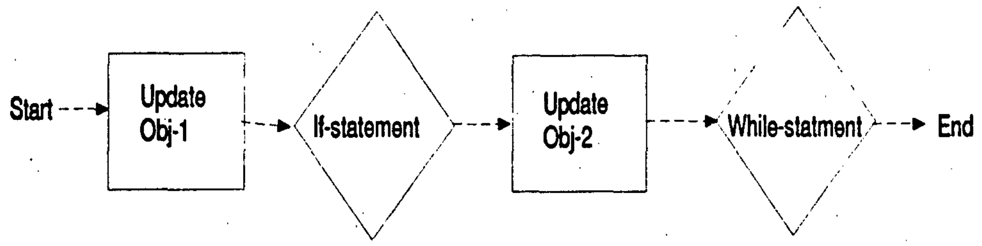


Figure 3.3. "Top Level View of Controller's Update Algorithm"

3.4.2.1 Viewing a Subsystem. A subsystem may be viewed either textually or graphically. The textual representation is merely a "pretty-printed" view of the subsystem in the domain-specific language for the selected domain. The graphical view of the subsystem consists of a diagrammatic view, consisting of icons and links.

The OCU representation of a subsystem as shown in Figure 3.1 is the basic "top-level" visual representation of a single subsystem (28). The subsystem editor displays a similar diagram for a subsystem. This top-level representation may be expanded by examining any of its components. The user examines these components by clicking on the appropriate icon. By expanding the view of the controller, the user may view the controller's update algorithm. This algorithm can be viewed in its textual form, as specified by the grammar for the specific domain, for example,

```
Update(Obj-1);

If <condition>
   then Update(Obj-3);
Else Update(Obj-2);

Update(Obj-2);

While <Condition>
   Update(Obj-1);
```

Alternatively, a graphical representation may be viewed, as in Figure 3.3. The algorithm is represented as a directed graph, which shows the control structure of the algorithm. Different configurations of icons and directed arrows are used to show various patterns of control. For example, graphs depicting sequential, iteration, and branch on condition patterns of control are

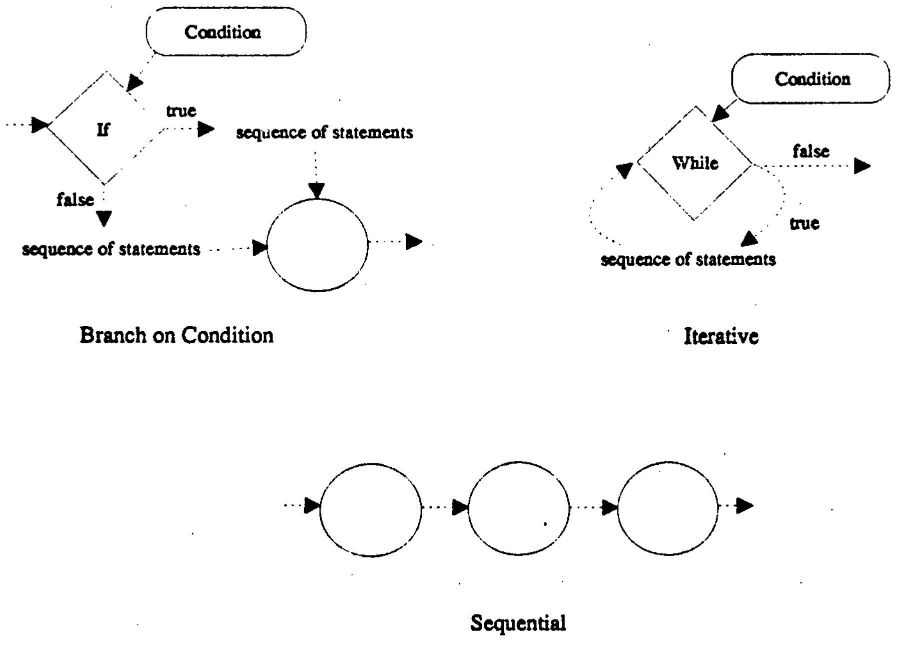


Figure 3.4. Patterns of Control

shown in Figure 3.4. An If statement is represented as a single icon, which may be expanded to show its components: the condition, the then-part, and the optional else-part. The condition may then be expanded to show an expression tree. The then-part and else-part are statement sequences which may also be expanded. Similarly, a while statement's icon may be expanded to show its components: the condition and the while part.

A new statement is added to a statement sequence by inserting an icon which represents the statement type, which may be an update-call statement, an if-statement, or a while-statement. Additionally, a statement is removed from a statement sequence by deleting its icon.

The subsystem's objects, which are represented as a single icon in the top-level view of the subsystem, can be expanded to show a hierarchical view of the entire collection of objects. Since some of a subsystem's controllees may be subsystems themselves, this hierarchy of objects is represented visually as a tree structure with the top-level subsystem as its root, as in Figure 3.5. Each icon is labeled with the name of its object class along with the name of the object instance.

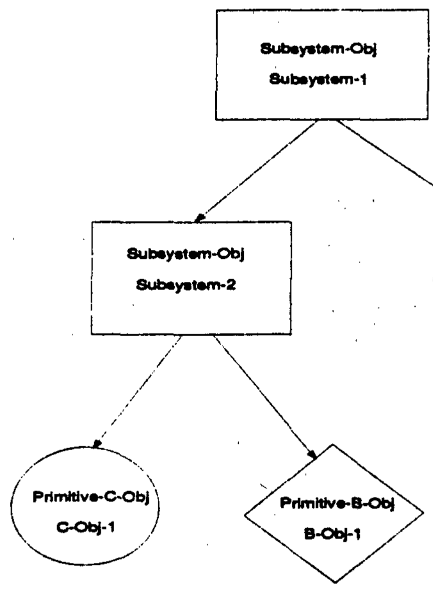


Figure 3.5. Hierarchy of Objects

The visual attributes of the icon used to represent each primitive object are defined by the Visual Specification Language for that object's domain.

The import and export areas may also be expanded to view their import and export objects. AVSI displays this information in a read-only text window.

3.4.2.2 Building a Subsystem. Three different methods can be used to put a subsystem in the structured object base. These methods are: selecting a pre-defined subsystem from the technology base, creating a generic instance, and building a new subsystem "from scratch" by direct-manipulation of icons and links.

A subsystem which has been previously built and saved in the technology base may be selected from the technology base for inclusion in an application. This action is performed by selecting a subsystem from the "Technology Base Window." If the saved subsystem contains controllee objects (primitive-objects or other subsystem-objects) these objects and their icons are also created and added to the application and its visual display. AVSI creates an icon for the retrieved subsystem and allows the user to place the icon in the diagram window and link it to the icon that represents its controlling subsystem or application.

A new subsystem-object may also be created from any generic object template existing in the technology base. The generic template contains a basic subsystem definition, with "placeholders" for which primitive objects must be provided. An icon representing the generic instance is placed in the current edit-subsystem-objects or edit-application-objects window. The user links this icon to its controlling subsystem (or application object). Finally, the user must create a primitive-object and supply a name for each of the "placeholders" defined in the generic template. Finally, the icon for each of these primitives must be linked to the generic-instance's icon. Architect then converts the generic instance to an actual subsystem-object, whereupon the generic-instance-icon is replaced by a subsystem icon.

A new subsystem may be built by directly manipulating icons and links which represent its parts. To build a new subsystem, the user follows these steps:

1. Create a subsystem-icon within either the edit subsystem-objects window or the edit application-objects window. A new subsystem-object is created and added to the application.
2. Link the subsystem-icon to its controller. The subsystem's controller may be either the application-object or another subsystem-object.
3. Choose and place controllee objects for the subsystem. These objects are either primitive objects or other subsystems. Primitive objects are selected from the Technology Base Window. The primitive objects contained within the Technology Base Window are the objects which are legal for the currently chosen domain. Each object selected is placed in the edit subsystem objects window or the edit application-objects window by dragging its icon to an appropriate position.

4. Fill in information required to complete definitions of the objects. Primitive objects' attributes and their default values are displayed in a text window. These values may be modified using the attribute editor.
5. Create the update algorithm for each controller. For controllers, update algorithms are given by either writing in the surface syntax of the domain-specific language, or by constructing directed graphs via a direct manipulation scheme similar to that used to construct subsystems, i.e. placing icons and links.
6. Place a link between each icon and its parent controller's icon. This link is represented as an arrow, pointing from the controller to controllee.

3.4.2.3 Editing a Subsystem. Once a subsystem has been loaded or built, AVSI allows the user to modify it. The methods for editing a subsystem are as follows:

1. Changing the structure or configuration of the subsystem. This is done by direct-manipulation of icons and links. Icons and links may be created, deleted, or moved. Creating an icon requires the same steps as those outlined above in "building subsystems."
2. Editing an object's attributes. An attribute editor presents a list of an object's attributes and allows the user to interactively change their values.

3.4.3 Building Import and Export Areas. According to the OCU model, the communication links between objects exist via the import and export objects contained within subsystems' import and export areas. Architect may be able to infer what some of the links are, using the data type of the import and export objects. However, when ambiguity exists, for example when there exists more than one possible source for a given import object, the application specialist must tell Architect which source is to be used. This is done by manually connecting import and export areas. To connect the import and export areas, the application specialist begins by making the connections between the objects within each subsystem. These links are internal to the subsystem, and

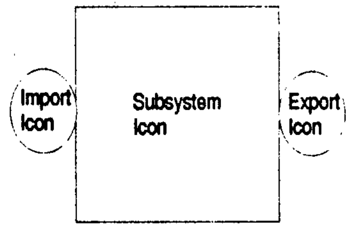


Figure 3.6. Subsystem Icon Group

are established by connecting objects within the subsystem with arrows, pointing from export area to import area. Finally, the external links, those between subsystems, are established. Similarly, these links are made by connecting subsystem icons with arrows.

To display and build the application's import and export areas, the user clicks on the "Build Import and Export Areas" button on the main control panel window. A new window, entitled "Imports/Exports" will open. This window displays an icon group for each of the application's subsystems. The subsystem icon group used to display and connect subsystems' import and export areas is composed of three sub-icons, as shown in Figure 3.6.

1. *Subsystem-icon* This is the main icon of the icon group, and is represented as a rectangle. This icon is labeled with the Subsystem's name. If this is a "nested" subsystem, the parent subsystem is included (in parentheses, beneath the subsystem name) in the subsystem's label. Clicking on this icon will allow the user to either interactively move the icon group or to invoke the Subsystem Editor.

2. *Import-icon* This is represented as a circle, attached to the left hand side of the Subsystem-icon. This icon is labeled, "Imp." Clicking on this icon results in one of two actions, depending on the particular sequence of events in the editing process. These actions are discussed in the next section.

3. *Export-icon* This is represented as a circle, attached to the right hand side of the Subsystem-icon. This icon is labeled, "Exp." This icon is similar in function to the import-icon.

Architect's preprocessing facility builds the import and export areas for each subsystem and establishes connections between whatever import and export areas it can. If an import object can receive data from more than one export object, the user is required to make the connection manually to one or more of the export objects. If an import object is connected to multiple export objects, Architect arbitrarily selects one of the export objects from which to receive its data.

If each of the import objects within a subsystem are connected to at least one export object, then the subsystem icon's import-icon will be displayed in reverse-video. Similarly, if each of the export objects within a subsystem are connected to at least one import object, then the subsystem icon's export-icon is shown in reverse-video. No connections are shown on the display initially. Figure 3.7 illustrates the initial display.

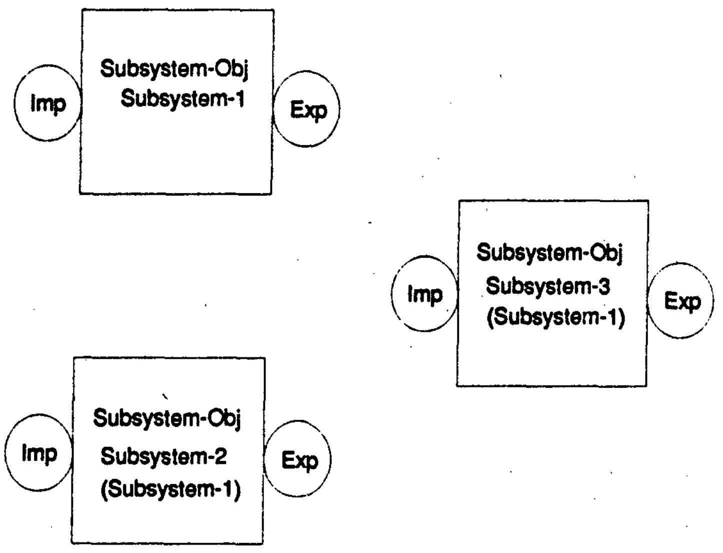


Figure 3.7. Initial Display of Import/Export Diagram

3.4.4 Connecting Import and Export Areas. The connection between import and export objects may be made in either direction. If a subsystem's import area is chosen first, then for each import object in the import area, export objects may be chosen. Conversely, if the export area is chosen first, then for each export object in the export area, import objects may be chosen. Each method is described in the following sections.

3.4.4.1 Selecting Export Objects for an Import Area. Clicking on a subsystem's import-icon at this point will result in two display actions. First, a window will open containing textual information about the subsystem's import area. This information is comprised of the following information:

- *Import Name* This is the name of the import object, and is not necessarily unique. Thus this name may be duplicated in the list of import objects.
- *Import Consumer* This is the name of the consumer object for the import data. Including this name in the list is necessary for distinguishing between import objects which share the same import name.
- *Import Category* This identifies the data type of the import object.
- *Source* This shows information about the export object(s) connected to this import object.

A second display action is performed in conjunction with the above. If one or more of the import objects within the subsystem are connected to an export object, a solid arrow is displayed between this subsystem's import-icon, and the export-icon of the subsystem which contains the export object (Refer to Figure 3.8).

An import object is selected from the import area by clicking on its entry in the import-area text window. The selected import object's entry in the text window is displayed in reverse video. Once an import object is thus selected, the solid arrows disappear. Dashed arrows are now displayed, showing the subsystem export areas which contain candidate export objects (Refer to Figure 3.9). Clicking on a subsystem's export-icon will open a textual window for the corresponding export area. This text window is similar to the import area's text window, with minor differences in the field names. Each export object in the export area is listed in the window, and clicking on an

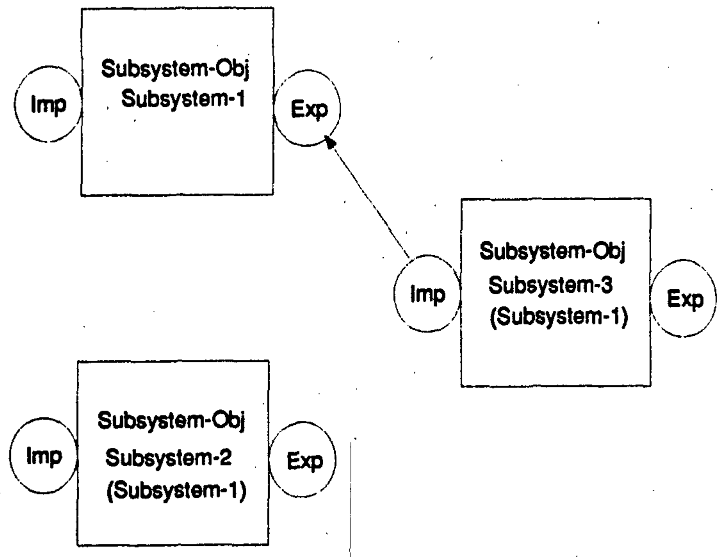


Figure 3.8. A solid arrow represents an existing connection

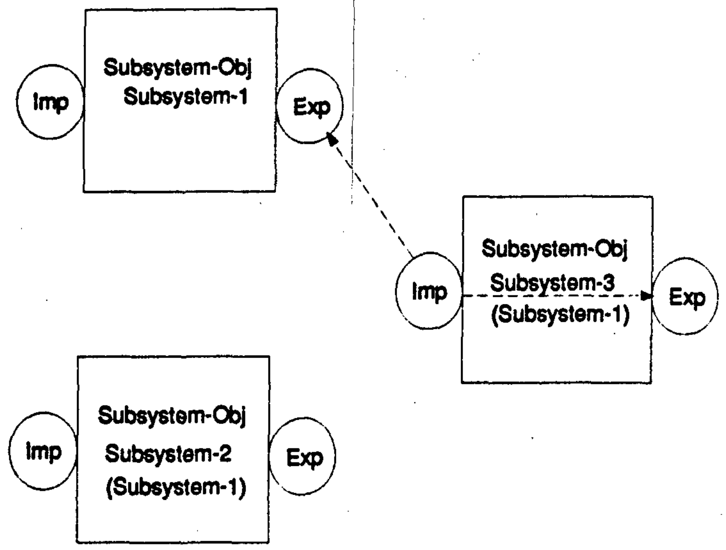


Figure 3.9. Dashed arrows indicate potential connections

entry will select the corresponding export area to be used as the source for the previously selected import area. Both the import-area text window and the export-area text window will be updated to reflect the change.

With each connection made, if the change results in the completion of any import or export area's connections, the corresponding icon will be displayed in reverse-video. The user is not required to define the connection for all of the import-objects in the import area at this time. If no import area is currently selected in the import-area text window, a new import area may be chosen by clicking on any subsystem icon's import-icon. This will cause the previous import-area text window to disappear, and a new one will be opened for the selected import-area. Additionally, the dashed-lines will disappear, with new ones appearing for the new import-area.

3.4.4.2 Selecting Import Objects for an Export Area. The process of selecting import objects for an export area is similarly described. If no import or export object is currently selected, clicking on a subsystem's export-icon will cause a window to open which contains textual information about the subsystem's export area. This information is comprised of the following information:

- *Export Name* This is the name of the export object, and is not necessarily unique. Thus this name may be duplicated in the list of export objects.
- *Export Producer* This is the name of the producer object for the export data. Including this name in the list is necessary for distinguishing between import objects which share a common name.
- *Export Category* This identifies the data type of the export object.
- *Target* This identifies the import object(s) connected to this export object.

The user's interaction is the same as in the above section, except that the connections are made in the other direction. The arrows, both solid and dashed, now point from a single export-area to multiple import areas. An export-object is first selected, the export-object is highlighted in the export-area textual window, and dashed arrows point to the potential targets. An import area is selected by clicking on a subsystem's import-icon. An import-area text window opens and the user selects an import object.

3.5 Conclusion

AVSI is required to provide visual support for the application composition activities of the Architect system. This visual support provides methods for viewing and manipulating the various parts of an application, making the application composition process easier and more intuitive. The operational concept presented in this chapter demonstrates how AVSI meets the stated requirements. AVSI generates application specifications by populating the structured object base with instances of domain objects using the visual techniques supplied by the application editor and the subsystem editor. The technology base interface provides access to existing components within a domain, and the Visual Specification Language defines the domain-specific visual information required to display domain objects. AVSI allows the user to edit domain objects, make communication links between subsystems, and provides a command interface for semantic checking and application execution. The following chapters provide information on the design, implementation, and validation of AVSI. Additionally, Appendix B provides a sample session of creating an application.

IV. Design and Implementation of the Architect Visual System

4.1 Introduction

This chapter describes the design and implementation of the Architect Visual System Interface (AVSI). The first section discusses the Visual Specification Language (VSL). Subsequent sections present the design of AVSI. Finally, details are provided concerning the AVSI implementation in the Software Refinery environment. As mentioned in the previous chapter, AVSI provides a visual environment for Architect. devised by Capt Cynthia Anderson (2) and Capt Mary Anne Randour (39). A high-level overview of Architect is provided in appendix A. The functionality of the system devised by Anderson and Randour has been preserved in AVSI.

4.2 Visual Specification Language (VSL)

A central goal throughout the development of Architect was to maintain domain-independence. The system should require no code modification in switching from one domain to another. Rather, it is desirable to allow domain-specific information to be represented in a standard format which can be "plugged into" the system for each domain without requiring any subsequent changes to the basic system. The Visual Specification Language (VSL) provides a means of encapsulating the information required by the visual system for each individual domain. VSL defines the visual representation for each primitive, and provides object attribute information for use by the attribute editor. This domain-specific information, written in VSL, is easily modified; the software engineer or application specialist may make changes to the domain objects' visual representation or attribute information without modifying the Architect software system or AVSI.

The grammar and domain model for the Visual Specification Language (VSL) were written in the REFINE language. This grammar is used by the DIALECT tool to create a parser to read VSL descriptions for the primitive-objects contained within a domain. DIALECT's default lexical analyzer was used for this parser since it provided the functionality required by VSL. VSL currently provides two basic types of information for the visual system: icon-attributes and edit-attributes.

4.2.1 Icon Attributes. The iconic representation of each instance of an domain primitive's object class is defined using the Icon-Attribute clause of VSL. INTERVISTA, the tool used for creating the visual system, provides a very limited capability for defining icons. Only four basic icon shapes are allowed. Icons using these basic shapes may be "customized" by changing their size or distorting their basic shape by changing the height-width ratio. If an icon attribute is omitted in the icon-attribute sequence, the default value (defined in the domain model) for that attribute is used. The attributes may be entered in any order. If an attribute is listed more than once, the last occurrence will be used. The allowable icon attributes for inclusion in an Icon-Attribute clause are:

- *Active?* If true, then the icon will be displayed. If false, the icon and all its links still exist in the system, but are not displayed in any window. The default value for Active? is true.
- *Icon-Type* Legal values are BOX, DIAMOND, ELLIPSE, and TEXT. An icon of type text has only its label displayed. The default value of Icon-Type is BOX.
- *Size-Factor* The size-factor is a positive real number which specifies the size of the icon. The default value is 1.0. Using a smaller number will decrease the size of the icon, and using a larger number will increase the size of the icon.
- *Height-Width-Ratio* The Height-Width-Ratio is a positive real number which controls the proportions of the icon. The default value is 1.0, which results in the width being equal to the height. Using a smaller number decreases the icon's width, while using a larger number increases its width. The icon's height is not affected.
- *Label-Function* The Label-Function is a symbol which specifies the name of the labeling function used to create the icon's label. The default value is a make-object-label, which displays the name of the object class followed by the name of the object instance.

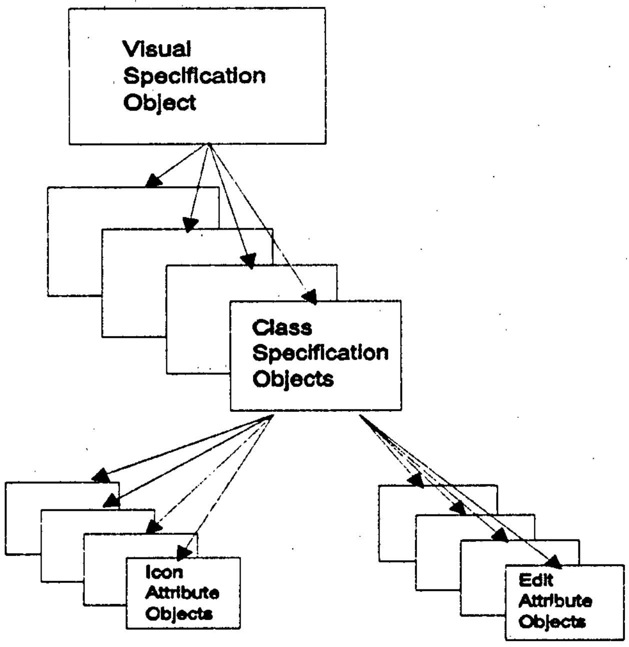


Figure 4.1. Visual Specification Object

- *Mouse-Sensitive?* Controls the mouse sensitivity of the icon. If true, then the icon is mouse sensitive; the icon will be highlighted when the mouse cursor passes over it and a mouse-handler will be invoked when the icon is "clicked on." The default value is true.

4.2.2 Edit-Attributes. Each primitive-object class' definition includes attributes which the user may want to (or need to) edit. The domain model for each object class contains these definitions and assigns default values for these attributes whenever an object instance is created. The edit-attribute clause in VSL specifies which of these attributes are in fact editable, and tells the visual system the data type of each of these attributes.

4.2.3 Structure of Visual Specification Objects. For each domain, a visual specification file is parsed by DIALECT. This creates a Visual Specification Object (VSO) abstract syntax tree in the REFINE object base. The structure of the VSO is shown in figure 4.1. Each VSO has multiple Class Specification Objects, one for each object class type within the domain. Each Class Specification

Object has two parts: its Icon Attributes, which is a sequence of Icon Attribute Objects, and its Edit Attributes. The visual system extracts information from the VSO whenever an icon for a primitive-object is created, or when the attribute editor is invoked. Figure 4.2 shows an example Visual Specification Object definition in VSL

VSL's grammar is relatively simple, partially due to the limited availability of icon definition provided by INTERVISTA. The grammar may be extended however, to incorporate complex icons and nested icons as in Lockheed's Graphical Specification Design Language (48).

4.3 Design

The design of AVSI is directly tied to Anderson's and Randour's domain model, which defines the structure of an application's abstract syntax tree in the REFINE object base. The syntax is defined by their OCU-grammar, which is inherited by all domain-grammars. Therefore the same basic structure is used across all domains. AVSI builds applications according to this structure, directly manipulating the application's abstract syntax tree in the structured object base. Therefore, the resultant abstract syntax tree is identical to that created by parsing an application definition from a text file.

4.3.1 System Structure. The structure of the visual system is shown in figure 4.3. The top level function, AVSI, ties together all of the subordinate functions necessary to compose a software application. It simply provides a button panel in the main window which gives choices for the major application-building activities specified by the requirements discussed in the previous chapter. The main window also contains a text window to output messages from the system. These messages are generally error messages or instructions to the user.

AVSI is a windows-based system. Several windows may be active simultaneously, and thus the user may be engaged in multiple concurrent stages of the application composition process. For example, an application is composed of subsystems, which in turn are composed of primitive-

```
Visual Specs for TEST-DOMAIN are

   attributes for PRIMITIVE-1 are
      Icon :
         icon-type = ellipse;
         active?   = true;
         size-factor = 1.1;
         height-width-ratio = 1.0;
         label = class-and-name;
         clip-icon-label? = false;
         mouse-sensitive? = true
      Edit :
         name : symbol;
         attribute-1 : integer;
         attribute-2 : symbol;
         attribute-3 : boolean
   end;

   attributes for PRIMITIVE-2 are
      Icon :
         icon-type = box;
         active?   = false;
         size-factor = 1.1;
         height-width-ratio = 0.95;
         label = class-and-name;
         clip-icon-label? = false;
         mouse-sensitive? = true
      Edit :
         name : symbol
         attribute-11 : real;
         attribute-12 : symbol;
   end;

end
```

Figure 4.2. Example Visual Specification Object Description in VSL

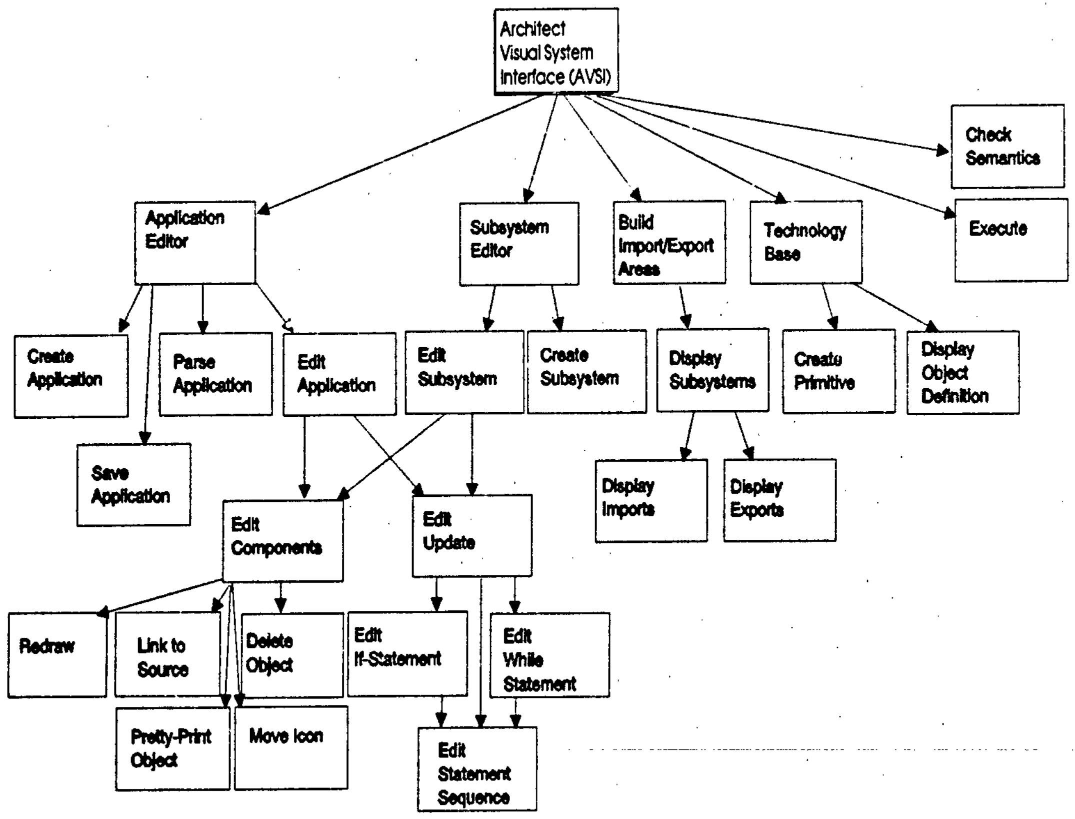


Figure 4.3. Visual System Structure

objects, or possibly even other subsystems. It is possible for the user to switch from editing a subsystem to editing the application at a higher level by simply switching to a different window. In this sense, the system is event-driven. By clicking the mouse on an object within a window, the user invokes that window's mouse handler, which in turn invokes some function that has been previously defined for the object that was clicked on. This provides a certain amount of flexibility in that the user is not forced to follow a completely rigid sequence of steps to complete an application definition. However, AVSI maintains consistency because the user is constrained by the syntax imposed by Architect's domain model and grammar. The user is not allowed to construct an application definition that is not syntactically correct.

4.3.2 Application Editor. The application editor allows the user to take multiple approaches to building an application. First, an application may be built "from scratch." Alternatively, an application may be parsed from a text file which contains an application definition written in the domain-specific language for the given domain. This file is either hand-coded or automatically generated by the system during a previous session. An application may either be built "from scratch," or by parsing a text file containing an application definition. Either method builds the same type of application definition structure in the structured object base.

4.3.2.1 Create New Application. Create-Application first sets the context of the system by prompting the user to choose a domain. The user is given a choice of domains based on the domains it "knows" about, i.e. those domains for which Visual Specification files are currently loaded into the system. AVSI creates and names the instances of the objects required for a basic application. The user will further define the application in later steps. Creating a new application is composed of three steps:

1. *Create application-definition-object* The application-definition object is a structure that "holds together" the entire application. It is the topmost ancestor of every object that exists in

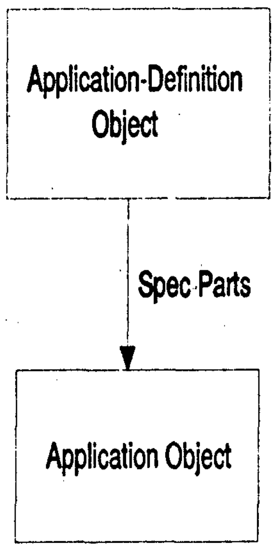


Figure 4.4. Abstract Syntax Tree for Application

the application. This provides the ability to save (and later restore) an application using DIALECT.

2. *Create application-object* The application-object is the application executive according to the current implementation of Architect. It may be thought of as the highest-level subsystem in the application. The application-object is placed under the application-definition-object in the application definition. Figure 4.4 shows the abstract syntax tree created by this process.

3. *Set domain context* The user chooses, from a menu listing all available domains, the domain for the application. The list of available domains is the set of domains for which Visual Specification Objects exist in the structured object base; thus, the VSL file for the desired domain must be parsed before building the application. Additionally, the grammar for the chosen domain must be loaded prior to this step.

4.3.2.2 Parse Application. The Parse Application function uses the DIALECT tool to parse a text file containing an application definition into the REFINE object base using the domain-specific grammar already defined for that application's domain. The resultant abstract syntax tree representation of the application is identical to one created by AVSI's direct manipulation techniques.

Parsing an application definition from a text file is straightforward. First, the user sets the domain context; this selects the grammar used by DIALECT to parse the text file. Next, the user interactively selects the file to be parsed. If the parse is successful, DIALECT builds the abstract syntax tree for the application definition. If the parse is unsuccessful, the user is notified in the message window.

4.3.2.3 Load Saved Application. A variant of parsing an application definition is loading a saved application, and the difference between the two is subtle. A "saved application" is a system-generated application definition text file (written in the domain-specific language) written during a previous session. It generally contains extra information about the import/export areas of subsystems which does not normally exist in a regular application definition text file, and it requires some extra processing. This extra processing is a result of a feature of the OCU grammar, as defined by Anderson and Randour. All information about the connections between import and export areas is specified by listing which export objects are connected to an import object. This information is an attribute of an import object. On the other hand, an export object has no direct knowledge about which import objects it is connected to. The visual system needs this information however, to allow the user to establish the connections from either direction. AVSI adds an attribute to the export objects which lists the import areas it is connected to. This is discussed in greater detail in section 4.3.5.1.

4.3.2.4 Save Application. The user may choose to save an application at any point in the application composition process, as long as the current state of the application is "correct," i.e. the abstract syntax tree must be *syntactically correct* according to the grammar rules of the domain-specific language. However, this poses no problems, since both DIALECT and AVSI build applications according to the same syntax rules. The Save Application function saves the abstract syntax tree which contains the current state of the application definition in the form of a "pretty-print" to a text file.

4.3.2.5 Edit Application. The application editor provides two subfunctions, Edit Application Components, and Edit Application Update Algorithm. These editing functions modify the application's object structure as well as modifying the application's main control algorithm. Once a new application definition has been created, the user must complete the definition by adding and defining components, and providing an application update algorithm. If, as in the case of a parsed application definition, these items already exist, the user may modify them using the application editor.

4.3.2.6 Edit Application Components. This function displays the application object (which may be thought of as the top-level subsystem in the application) as an icon to which subsystems may be created and connected. Several functions are available within the Edit-Application-Components Window. New subsystems may be created and linked to the application or to other subsystems. In keeping with the current Architect model, an application may only control subsystems, therefore no domain primitives may be introduced at this point. The Application Component Editor additionally allows the user to view a pretty-print of the application definition in a text window. The viewed text is presented in the domain-specific language defined for the application's domain.

As in each diagram window in AVSI, a mouse handler is defined for within the Edit-Application-Components Window. Clicking a mouse button while the mouse cursor is in the window's screen region will invoke the window's mouse handler. The mouse handler will invoke a function, based on the object (icon, link, or diagram-surface) beneath the mouse cursor when the button is clicked. The following functions are defined for this window:

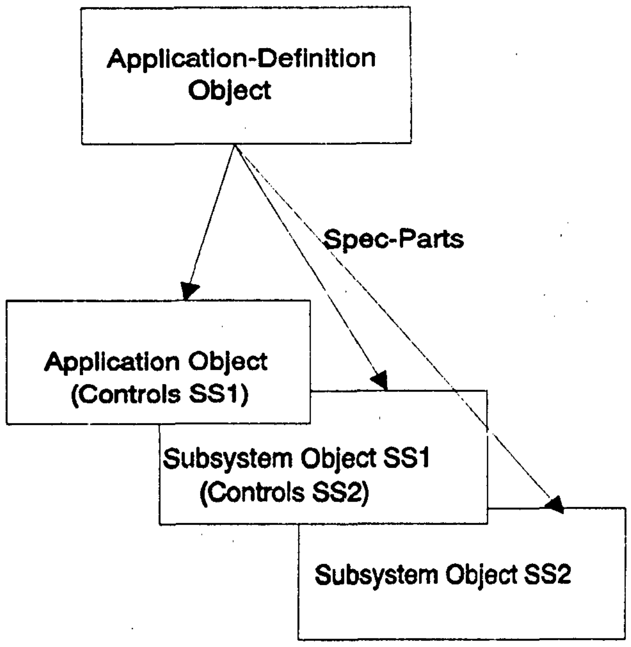

Application-Definition Object
Spec-Parts
Application Object (Controls SS1)
Subsystem Object SS1 (Controls SS2)
Subsystem Object SS2

Figure 4.5. Application Definition with Subsystems

1. *Create New Subsystem* Th: user interactively creates a subsystem and places its icon in the diagram window. AVSI adds the subsystem to the application definition's abstract syntax tree.

2. *Link to Source* When the user links the subsystem-object's icon to the controller's icon, AVSI updates the controller's list of controllees. Note that the user may only add subsystems in the edit-application-components window; primitive-objects may be created only in the subsystem editor. AVSI represents the application as a tree-like graph, with the application-object icon as its root. Arrows show the direction of control, from the controller to the controllee. Figure 4.5 shows an example application with subsystems. Note that subsystems may be nested.

3. *Redraw Screen* This forces a screen redraw, which invokes the automatic layout algorithm for the application. This function redraws the application as a tree-like directed graph, with the application-icon as the root. If the application definition contains any primitives at this

point, these are displayed beneath their controlling subsystems. The function scales the icons to fit the entire diagram on the viewed surface.

4. *Zoom-in/Zoom-out* If the system contains a very large number of subsystems, the above-mentioned layout function, with its scaling, may shrink the icons down to a very small size, making it difficult to view individual icons or sections of the diagram. The zoom-in function allows the user to enlarge a section of the diagram. Zoom-in's complementary function, zoom-out enables the user to get more of an overall look at the diagram.

5. *Move Icon* The user may occasionally want to change the placement of certain icons in the diagram window. Move-icon allows him to drag the icon to another location. Because AVSI uses dynamic links, any links connected to the icon are automatically updated when the icon is moved.

6. *Delete Object* Any subsystem in the diagram window may be deleted at any time. Delete-object erases the object from the object base, deletes all references to the object in the application definition, deletes the object's icon, and removes all links connected to the icon. A subsystem's controllees are not deleted when the subsystem is deleted. However they are isolated until they are relinked to another controlling subsystem or to the application object itself. An application object may not be erased in the edit-application-objects window. Once an object is erased, it cannot be recovered without re-creating and redefining it.

7. *Pretty-print Object* The description of an object's abstract syntax tree is displayed in a separate text window. The pretty-printer prints according to the rules of the current grammar. Any object (application-object, subsystem-object, or primitive-object) may be pretty-printed in this window.

4.3.3 Edit Application Update. The Application Update Algorithm Editor, like the Application Component Editor allows the user to build the application's update algorithm by direct-manipulation of visual objects (icons and links). The update algorithm is composed of a sequence of

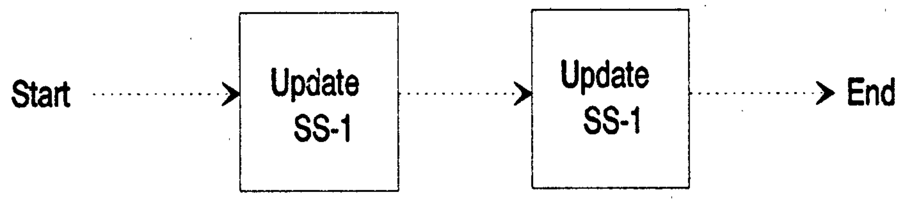


Figure 4.6. Application Update Sequence

statements, and is visually represented as a directed graph with the nodes representing statements and the edges showing flow of control. According to the current model of Architect, if-statements and while-statements are not allowed in the application update algorithm.

Edit-application-update provides a dual view of the application's update algorithm in two separate windows. The text window merely shows a pretty-print of the update algorithm. This text window is updated whenever any change has been made. The only legal statements in an application's update algorithm are call statements, thus the control structure is strictly sequential (if-statements and while-statements *are* allowed in a subsystem's update algorithm: see section 4.3.4.2). The diagram-window displays the algorithm in diagrammatic form, as a linear list, left to right (see Figure 4.6). Note than two icons in the diagram, the "start-icon" and "end-icon," do not have corresponding statements in the actual update algorithm's actual statement sequence. These two special icons may not be modified or deleted. A null sequence then consists of an arrow, from the start to the end icon. The user modifies the algorithm by adding, deleting, or editing nodes in the list:

1. ***Add Statement*** Adding a statement to the update algorithm inserts a statement object in the statement sequence. AVSI places the statement in the sequence according to its position in the diagram. A newly-created call statement is incomplete, since the user has not yet specified the operand name.

2. ***Edit Statement*** Editing a call statement is simply supplying an operand name to the call statement object.

3. *Delete Statement* Deleting a statement removes the statement object from the statement sequence, erases the statement object and its corresponding icon, and links the two icons which were adjacent to the deleted icon.

4.3.4 Editing a Subsystem. Editing a subsystem is very similar to editing an application. Indeed the application-editor and the subsystem-editor share many of the same functions The three major components of the subsystem editor are Edit-Subsystem-Components, Edit Subsystem Update Algorithm, and Display Import/Export Areas.

4.3.4.1 Edit Subsystem. Editing a subsystem, like editing an application, comprises editing its components and editing its update algorithm. The Subsystem Editor's similarity to the application editor is a result of the way an application object is viewed and represented in the current implementation of Architect: an application object is essentially the top-level subsystem. The two functions involved are Edit-Subsystem-Components and Edit-Update.

Edit-Subsystem-Components is similar in function to Edit-Application-Components. The subsystem object is represented as a subsystem icon in the Edit-Subsystem-Components diagram window, with directed arrows pointing to the icons representing its controllees (which may include other subsystems). Adding a primitive to the subsystem is done by selecting a primitive-object icon from the technology base window, dragging it into the Edit-Subsystem-Components diagram window, and then linking the primitive-object icon to the subsystem which controls it. The system provides an alternate, textual view of any object in the window by pretty printing the abstract syntax tree for that object in a text window.

AVSI displays a subsystem as a tree-like directed-graph, with the icon representing the edited subsystem having no ancestor node. Figure 4.7 shows an example subsystem. The same layout algorithm used for displaying applications is used for displaying subsystems. The user may add subsystems or primitives to the subsystem being edited:

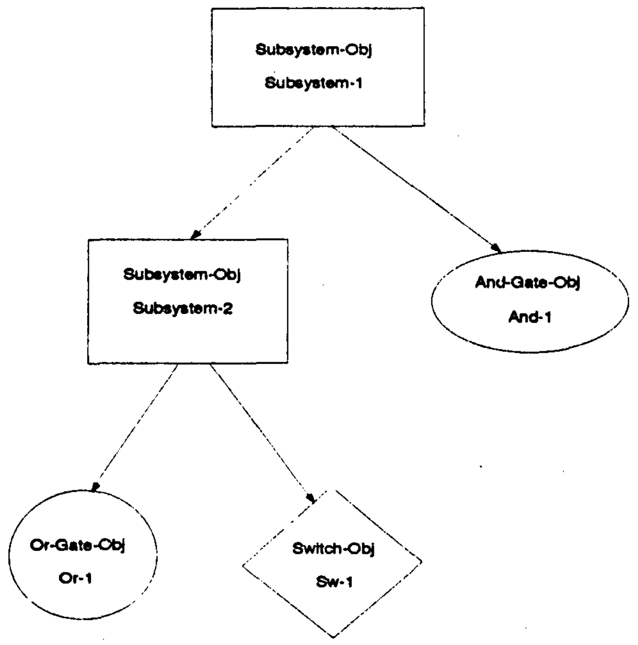


Figure 4.7. Example Subsystem

1. *Creating a new subsystem* The function Create-Subsystem creates a subsystem object, and prompts the user for its name. The new subsystem is represented by a subsystem icon in the diagram window. The subsystem must be linked to its controlling subsystem (or application object) by linking its icon to the icon of its controller. The new subsystem must be controlled by some subsystem already existing in the application definition (a primitive may not control a subsystem).

2. *Creating a new primitive-object* The user creates a new primitive-object by dragging the desired primitive-object's icon into the window from the technology base window (which contains the primitive-object classes for the current domain). AVSI creates the primitive-object instance, and inserts it in the application-definition's abstract syntax tree.

3. *Link to Source* The user must link both subsystem objects and primitive-objects to their controlling subsystem. AVSI adds the primitive-object's import and export data to the import and export areas of the object's controlling subsystem.

4. *Pretty-Print Object* Either subsystem-objects or primitive-objects may be pretty-printed. A separate text window displays the object's pretty-printed code.

5. *Move-Icon, Redraw, Zoom-In/Zoom-Out* These functions are described in section 4.3.2.6.

6. *View/Edit Primitive-object Attributes* Any attributes listed among the edit-attributes, as defined by the VSL description of a primitive-object class, may be edited. AVSI lists the attribute names and current values in a text window. Clicking on an attribute name brings up a small pop-up window into which a new value may be entered. AVSI uses the LISP read-from-string function to extract the value, and the REFINE set-attribute function to update the object's attribute.

7. *Delete Object* The topmost subsystem in the window may not be deleted (the user may use the application editor to delete this or *any* subsystem, should he want to do this). Any nested subsystems in this window may be deleted as described in section 4.3.2.6. Deleting a primitive-object erases the object instance and removes all references to the object from the application's abstract syntax tree. These references include entries in the subsystem's import and export areas.

4.3.4.2 Edit Subsystem Update Algorithm. Editing a subsystem's update algorithm is similar to editing an application's update algorithm; however, since a subsystem's update algorithm may contain conditional or loop constructs, two other statement types are allowed: if-statements and while statements. Editing an if-statement is done by editing the conditional-part, the then-part, and an optional else-part. Similarly, editing a while statement is done by editing the conditional part and the while-part. Then-parts, else-parts, and while-parts are simply statement sequences.

1. *If-Statements* The user inserts an if-statement into a statement sequence in the same manner as inserting a call-statement. AVSI represents the if-statement as a single icon in the statement

sequence. The user edits the individual parts of the if-statement separately, by clicking on the if-statement icon in the statement sequ~nce. AVSI then displays an expanded view of the if-statement in a new window which it "stacks" on top of the current window. This expanded view is a generalized view of an if-statement, and serves as a menu to edit the three parts of the if-statement.

The first part of the if-statement is the if-condition. The if-condition is an expression-object which the user may build in two different ways: typing the expression, and building it. If the user chooses to type the expression, AVSI parses the expression from the user's input string and inserts the resultant abstract syntax tree in the if-statement object's conditional part. This expression tree corresponds to the expression's abstract syntax tree in the structured object base. AVSI allows the user to edit the expression tree by adding and deleting nodes. Any identifiers in the expression need to be linked to some import-object or export-object within its controlling subsystem. If AVSI is unable to determine the correct import/export object, it prompts the user to choose from among the currently available import/export objects. To provide a certain amount of flexibility, the user is allowed to defer this action to a later time. This flexibility is necessary because the subsystem's components may not have been completely defined at this point, and thus the desired import/export object may not exist yet. AVSI now displays the expression as an expression tree in a new window. If the user chooses to build the expression, rather than type it in, he does this by building the expression tree within this window.

The other parts of the if-statement, the then-part and the optional else-part, are statement sequences. The user edits these statement sequences in exactly the same manner as editing the main statement sequence. All statement types are legal in these statement sequences.

2. *While-Statements* There is little difference between editing an if-statement and a while statement. A while statement contains the same conditional-part and contains a do-part, which is a statement-sequence.

4.3.5 Building Import/Export Areas. As AVSI adds each primitive-object to a subsystem, it automatically builds that part of the subsystem's import and export areas which pertain to the primitive-object. This action is invisible to the user, and the user is not allowed to modify the subsystem's import or export areas; however, the user is allowed to view the subsystem's import/export areas in a text window. AVSI displays this information by pretty-printing to read-only text windows which the user may view at any time.

Though AVSI automatically builds the application's subsystems' import and export areas as the user adds primitive-objects to each subsystems, or when an application is parsed, the connections between all of the import and export objects must still be established. This section discusses the process of making the connections.

4.3.5.1 Logical Representation of Import and Export Areas. Each subsystem's import area contains all import objects for all of the primitives it controls. Each of these import objects may have one or more sources, which are export-objects controlled by any subsystem within the application (including its own controlling subsystem).

Similarly, a subsystem's export area contains the export objects for all its primitives. Each export object may have one or more targets, which are import-objects within any subsystem. Consequently, there is a many-to-many relationship between import and export objects.

In Anderson's implementation of Architect, the connection is specified by storing the source as an attribute of each import-object. The user makes all connections by specifying the sources of all the import objects. This establishes a one-way connection between export and import objects, and such a representation is adequate for her implementation. AVSI however provides more flexibility

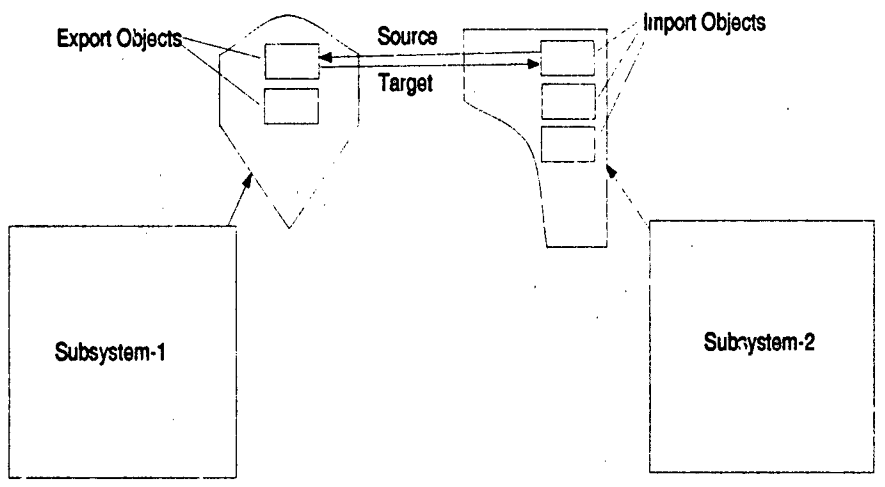


Figure 4.8. Import and Export Objects with Source and Target Attributes

by allowing the user to specify the connection from either the import-object's or the export-object's perspective. Figure 4.8 shows the representation of these connections. The new attribute *target* is the functional converse of the already-existing attribute *source*; thus, the existence of a source implies the existence of a target and vice-versa.

4.3.5.2 Visual Representation of Import and Export Areas. AVSI uses a somewhat simple scheme to display and manipulate the connections between import and export objects. The operational concept was discussed in chapter III. Graphical and tabular information is combined to show the connections, and the number of links shown on the display at any given time is minimized, thus avoiding a cluttered visual display. At the same time, the complete connectivity of each import or export area is readily visible.

4.3.6 Technology Base. Architect's usefulness is in part due to the reuse it promotes by providing a technology base of reusable components. AVSI gives the user an interface in which to access the components contained in the technology base. Three basic types of reusable components are primitives, generics, and "saved" objects.

4.3.6.1 Primitives. AVSI fully implements the retrieval of primitive-objects for inclusion in an application's subsystems. A technology-base window displays an icon for each primitive-object class defined for the current domain. Each icon is drawn according to the icon definition contained in the VSL description of the corresponding object class. These icons are arranged in a simple lattice in the technology-base window, and two different functions are available for primitives:

4.3.6.2 Create Primitive-Object. Creating a primitive-object is discussed in section 4.3.4.1. When an object is created, its icon is created and attached to the mouse cursor for placement in the edit-subsystem-components window.

4.3.6.3 Display Primitive-Object Definition. AVSI displays the Primitive-Object Definition for a primitive-object class in a read-only text window. This definition is in the form of REFINE source code, and contains the structural and behavioral description of the primitive object.

4.3.6.4 Generics. AVSI provides visual support for Architect's capability to create subsystems from generic templates. After the user chooses a generic template by selecting its file from a menu, AVSI creates a generic instance, and a corresponding icon. The generic template contains one or more "placeholders," for which controllees must be provided. For each one of these placeholders, the user creates, and provides a name for, an object. AVSI uses the functions provided by Architect to convert this generic instance to a subsystem, which may then be included in the application.

4.3.6.5 Saved Objects. AVSI allows for the reuse of previously saved objects. These objects may be subsystems or primitives, and are represented as text files containing pretty-prints of objects from previous sessions. The application specialist retrieves one of these objects by selecting the filename from a menu. Architect parses the file, creating the object. AVSI creates the corresponding icon and allows the user to interactively place it in the diagram window.

4.3.7 Semantic Checks. Semantic checks are performed as part of the preprocessing step. AVSI provides no visual support for semantic checks. Anderson's code is used directly, and any messages generated by the process are output to the Emacs window.

4.3.8 Execute. Providing a visual display of the application's execution was beyond the scope of this thesis. As with the semantic checks, Anderson's code is used for execution, and the output is displayed on the Emacs screen.

4.4 Implementation

4.4.1 REFINE *Impact and Influence.* INTERVISTA contains the basic facilities to provide a visual-based user interface. It supplies a windowing system with mouse handling, pop-up menus, mouse-sensitive text windows, diagram drawing methods, and direct access to the REFINE object base (40). INTERVISTA programs are written in the REFINE language and use the REFINE object base. Graphical objects, such as icons, links, diagram surfaces, menus, and windows can be directly mapped to other objects in the object base. INTERVISTA is rich in high-level functions for maintaining and manipulating the graphical objects but is at the same time somewhat restrictive at the lower levels. For example, only four basic icon shapes are provided. These basic shapes may be given different sizes and height/width ratios. Creating complex icons, however, requires calling lower-level functions from the underlying windowing systems, and was beyond the scope of this thesis. Though somewhat limited in terms of icon definition (without using the underlying CommonWindows and XWindows systems), INTERVISTA provided an excellent platform for developing a prototype for the visual system.

Following an object-based approach, visual data in INTERVISTA consists of a set of icon objects, which map to the set of logical objects (for example, the subsystem-objects and primitive-objects used in Architect) in the underlying system. The icon and link objects' display attributes can be interactively modified by the user. For example, if the user "clicks" on an icon and moves

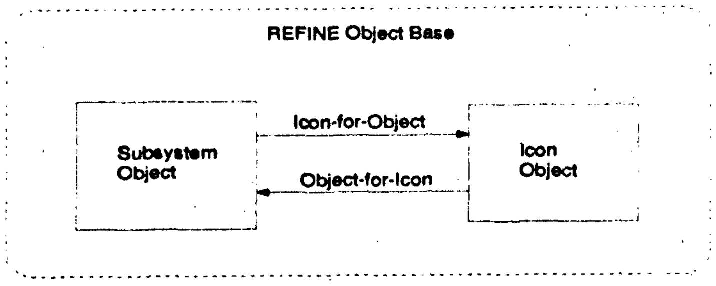


Figure 4.9. Object/Icon Relationship

to another location within a window, the icon's display coordinates are updated automatically, and all of its links are automatically redrawn to reflect their new position. The display objects can also be modified by the screen layout algorithms which are used for displaying already-existing diagrams. Relationships between the objects will be visually represented by link objects (lines with arrowheads) which connect the icons. The set of icons and links are grouped together into a single view (or diagram surface) which is itself an object in the REFINE object base, and may be manipulated by certain functions, such as repositioning, scaling, and zooming in and out.

The DIALECT tool provided the ability to easily implement the Visual Specification Language (VSL). VSL's grammar is defined by and parsed by a parser created by DIALECT. Thus the visual specification information is maintained in the REFINE object base, and AVSI uses this information to display and edit domain-specific information. The Visual Specification Language is loosely based on Lockheed's Graphical System Description Language's Visual Sublanguage (48).

4.4.2 Representing Objects.

4.4.2.1 Object-Icon Relation. An object in the REFINE object base is related to its visual representation by a two-way mapping between the object and the icon. For each object, an attribute, *icon-for-object*, represents its associated icon. An inverse attribute, *object-for-icon*, represents the object for a given icon. Figure 4.9 shows this relationship. Using the computed-using clause in the definition of an icon, the mere reference to an object's icon will dynamically create an icon, if the icon does not already exist. For example, the following defines the attribute Icon-for-Object.

```
var Icon-For-Object: map(object, icon)
  computed-using icon-for-object(obj) = make-icon-for-object(obj)
```

The computed-using clause calls a function which creates an instance of an icon object in the object base, and sets its attributes (position, icon-type, size, etc.). By including the following form, this attribute is cached; an icon will be created only on the first reference to the attribute. Subsequent references will use the icon created on the first reference.

```
form Cache-Icon-Object
    cache('icon-for-object, true)
```

The inverse attribute, object-for-icon, is defined to be initially an empty map:

```
var Object-For-Icon:
    map(icon, object) = {| |}
```

Defining the icon-for-object and object-for-icon maps to be functional converses results in the automatic definition of the object-for-icon attribute whenever an icon-for-object mapping is created:

```
form Object-Icon-Converses
    define-fun-converses('icon-for-object, 'object-for-icon, true)
```

This method of representing the relationship between an object and its icon provides an efficient, yet powerful, two-way access between the logical and visual representations of an object. An abstract syntax tree in the object base may be traversed and its visual counterpart consisting of icons and links constructed along the way. Conversely, as icons are created and linked, the corresponding objects and relationships may be created. Moreover, since icons exist as objects within the REFINE object base, they may be reused; new icons are not recreated if an object's icon has been previously viewed, even when the window containing that icon has been closed and then reopened.

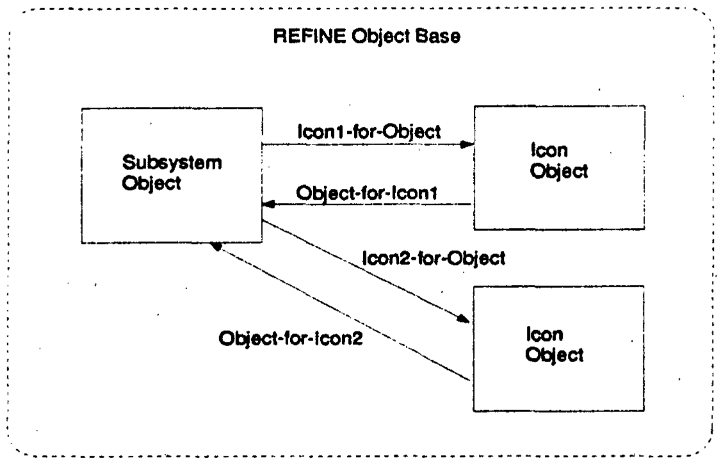


Figure 4.10. Multiple Object/Icon Relationships

4.4.2.2 Multiple Representations of an Object. Depending on the context in which the user is working, an object within an application may have more than one visual representation. For example, a subsystem-object, in the subsystem edit-components window, is represented as a simple box icon. In the build-imports/exports window, a subsystem is represented as a group of three icons. The subsystem will thus have more than one icon. Figure 4.10 shows this relationship. Note that the attribute names for the different icons are named differently.

4.4.3 Object-Window Relation. As icons may be mapped to objects (as in the previous discussion), so also may windows be mapped to objects. The following definition shows this mapping.

```
var Window-For-Object: map(object, diagram-window)
  computed-using window-for-object(d1) = make-window-for-object(d1)


var Object-For-Window: map(diagram-window, object) = {||}


form Cache-Window-For-Object
    cache('window-for-object, true)
```

```
form Object-Window-Converses

    define-fun-converses('window-for-object, 'object-for-window, true)
```

The first time an object's window is referred to, a diagram window is created and mapped to the object. Subsequent references to the window will return the window created on the first reference, since this attribute is cached. Additionally, since the attributes object-for-window and window-for-object are defined to be functional converses, both attributes are automatically defined upon the creation of the window.

This technique is useful because it allows each object to have its own window or windows. For example, AVSI creates a window for each subsystem for editing subsystem components. Thus several windows may be open simultaneously, providing the ability to edit multiple objects. This technique also results in the reuse of windows, avoiding the slow process of creating a new window each time the window is viewed. Furthermore, since the mapping is a one-to-one mapping, only one window may be open for an object, eliminating the possibility of obsolete information existing in an outdated window or conflicting information in multiple windows.

An object may be mapped to more than one type of window however, since the mapping may use more than one attribute name. For example, AVSI displays two possible windows for a subsystem: either an edit-components window or an import/export window (or both may be open simultaneously).

4.4.4 Mapping Object Sequences to Visual Objects. A problem exists when mapping an object sequence to a visual object: though REFINE allows such a mapping, the result of such a mapping does not have the desired effect. The problem is illustrated in the following code. The attributes are defined as follows:

```
var icon-for-seq : map(seq(object), icon)
  computed-using icon-for-seq(obj-seq) = make-icon-for-seq(obj-seq)
```

```
var seq-for-icon : map(icon, seq(object)) = {||}


form icon-seq-fun-converses

define-fun-converses('icon-for-seq, 'seq-for-icon, true)
```

This definition is essentially the same as that for the object-icon relation. However, the map is between a *particular* sequence and an icon. If the sequence changes by having objects added or deleted, then the map does not apply to the modified sequence. This problem occurred in the development of edit-update. Each statement sequence in the update algorithm requires its own window. The statement sequence is simply a sequence of statement objects, and the sequence is an attribute of its parent object. For example, the top-level statement sequence may be an attribute of a subsystem-object; the then-part and else-part of an if statement are attributes of an if-statement-object. To implement this correctly, these sequences are viewed as attributes of objects, rather than objects themselves. Rather than mapping windows to the sequences, the windows are mapped to the parent objects themselves. The sequence is accessed from the window, and the window is accessed from the sequence, indirectly through the parent object. Figure 4.11 illustrates this mapping for the then-part of an if-statement object. Note there is no direct relationship between the window and the statement sequence.

4.5 Summary

This chapter gave an overview of the design and implementation of the Architect Visual System Interface and the Visual Specification Language (VSL). The goals of the design were to preserve the functionality of Architect and to maintain domain independence. AVSI is built on Anderson and Randour's Architect and builds application specifications according to their domain model. AVSI provides the same functionality as Architect, and VSL allows for the specification

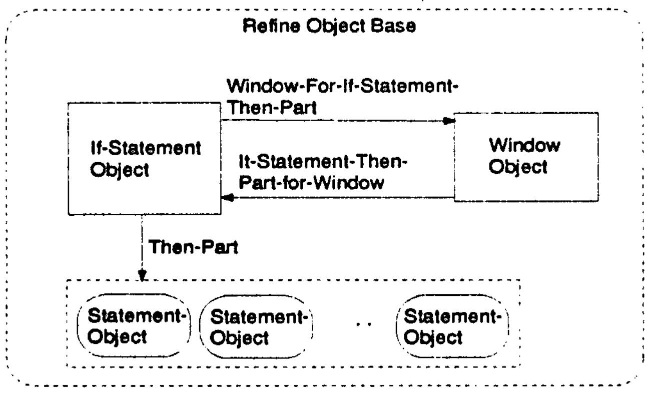


Figure 4.11. Window/Object Sequence Relationship

of domain-specific objects' visual information. The next chapter discusses the validation and an analysis of AVSI, including a discussion of the problems encountered and how they were solved.

V. *Validation and Analysis of the Architect Visual System*

The validation domain used for AVSI was the same domain used by Anderson and Randour for Architect: digital circuits. This chapter discusses the validation using this domain, and presents an assessment of AVSI.

5.1 Validation Domain

The main validation domain used for AVSI was the digital circuits domain. The "artificial" domain mentioned earlier was also examined to demonstrate the domain-independence of AVSI, but was not employed extensively due to its limited usefulness.

5.1.1 Circuits Domain. The primitive objects that are defined for the domain are:

- 2-Input And gate
- 2-Input Or gate
- 2-Input Nand gate
- 2-Input Nor gate
- Not gate
- JK flip flop
- Counter
- Switch
- LED
- Half Adder
- 3 x 8 Decoder
- 4 x 1 Multiplexer

This set of primitives was sufficient for building a large number of circuits of varying complexity. Examples of circuits composed of these primitives included decoders, a full adder, a binary array multiplier, and a universal shift register. These applications were also created using the previous methods provided by Anderson and Randour. Part of the validation process was to compare the domain-specific code generated by AVSI to the domain-specific code generated by Architect for identical applications. Moreover, the applications were executed to observe correct behavior. The switches and LEDs provided I/O capability, and allowed the operation of circuits to be observed.

One limitation imposed by this domain was that only one communication data type, "signal" was provided for communication between subsystems' import and export areas. Thus when import and export areas are being connected, AVSI generally shows all subsystems as containing potential objects for connection. If the domain contained more than one communication data type, connecting imports and exports would be easier, since the set of potential connections would be smaller.

5.1.1.1 Visual Specifications of Circuits Domain.

The only requirement in preparing a domain for use with AVSI is to specify the visual characteristics of the domain objects using the Visual Specification Language (VSL). The icon-attributes clause for each primitive object defines the physical appearance of that object's icon. Since INTERVISTA allows only a limited number of basic shapes, the conventional shapes for the logic gates were not used. Each primitive object was given a unique shape and size combination, however, to demonstrate the usefulness of VSL description. The edit-attributes clause for each primitive object specifies those attributes that are made available for AVSI's object attribute editor. The entire visual specification of the circuits domain is given in appendix D.

5.2 Using AVSI

Appendix C lists the files required to run AVSI and provides instructions on how to load and initiate the system. A description of the process of building an application is provided in the previous two chapters, and appendix B gives a sample session, in which a binary array multiplier is built.

5.3 Analysis

The development of AVSI was largely influenced by the REFINE environment in which it was created, as well as by the previous development of Architect. This section examines these influences and their effect on AVSI. This discussion is followed by an overview of problems encountered during AVSI's development and a discussion of AVSI's present weaknesses.

5.3.1 The REFINE *Environment.* Being an object-based system, REFINE provided an ideal environment for AVSI. It has a wide range of high-level operations for manipulating the structures in the object base.

The compiler for the Visual Specification Language was very easy to develop with the DIALECT tool. DIALECT's default lexical analyzer provided the functionality required by the language. VSL's language definition was defined with very little code. Appendix D contains the source listing for the domain model and grammar.

The visual portion of AVSI was built exclusively with INTERVISTA. Though its icon definition capabilities are somewhat limited, its usefulness is clear. INTERVISTA provides easy access to the REFINE object base, and as discussed earlier, its data objects (icons, links, windows, etc.) exist within the object base itself. As for its limitations, Intervista didn't provide very good support for creating interactive forms, complex icons, and multiple colors. These features, though desirable, were not necessary for demonstrating the usefulness of the visual system. Lower-level features such as these may be incorporated into the system by accessing the CommonWindows system, the underlying window system, or by using the graphics capabilities provided by graphics systems such as Motif.

5.3.2 AVSI and Architect. AVSI was primarily developed as a visual interface for the Architect system. The previous implementation of Architect relied on text files for building an application, and on text-based LISP input/output functions for further application definition. Anderson and Randour developed a complete working system, and AVSI was built around the "core" of their system. The lessons learned from the transition from Architect to AVSI are useful in developing similar systems.

The first step in this transition was to devise a transformation from a general application's abstract syntax tree to the application's visual representation. Additionally, the basic components of an application and its subsystems needed to be represented individually. Rather than just providing a diagram representing the abstract syntax tree itself (which *is* possible, and actually quite simple), only the information essential for visualizing the application's structure is represented at the topmost level. Other information contained within the object's abstract syntax tree, such as object attributes are not represented in diagrammatic, but in tabular, textual form.

Aside from "pruning" information from the abstract syntax tree, some information in the diagram comes from sources not actually in the abstract syntax tree. An example of this is in the Architect domain model's definition of a subsystem object and its controllees. A subsystem is "linked" to its controllees, not by being directly linked to them, but indirectly by their symbolic names. The visual representation, however, shows the icons for the actual objects, and thus AVSI must use a call to the REFINE *find-object* command, using the object's name. This, of course, is a further departure from a visualization of the actual abstract syntax tree, but is by no means a significant problem.

The second step was to establish techniques for building the abstract syntax tree based on the user's input. In terms of how the visual information would be "processed," there were at least two basic approaches possible. One approach was to allow the user to complete a diagram and then as a second step, to parse the diagram. This approach would be particularly useful in systems where the spatial relationships of the icons represented some meaningful information about the application (11). A second approach, the one chosen for AVSI, was to translate the user's actions as each individual action is performed. This method provides a sort of syntax-directed visual editing capability. Certain actions are only possible within the context of previously performed actions. For example, an application object must first be created before adding any subsystems, at least one subsystem must have already been created before adding a primitive object, and so forth. This is the simpler of the two approaches. Icon spacing is not considered, and some information, such as object attributes, may be modified at any time during the process.

5.3.2.1 Changes to Architect. The transition from Architect to AVSI was, with a few exceptions, direct and required little modification to the original domain model and REFINE source code. The basic structure of an application essentially remained the same. The only change made to the domain model was adding an additional attribute to export-objects. Adding this attribute consisted of simply adding a few lines of code to the OCU domain model, as discussed in Chapter 4.

A potential problem exists with the way Architect saves "saved objects" in the technology base. Architect uses the pretty-printer to write the object to a text file, and thus all information contained in the object's abstract syntax tree is saved, including its import and export object connections to other subsystems. When a saved subsystem object is retrieved, it may now contain erroneous references to non-existing subsystems. AVSI deals with this problem by deleting all connections when a saved subsystem is retrieved.

Another area requiring some rework was Architect's I/O. Architect relies entirely on basic LISP I/O functions, input from, and output to the EMACS window. However, INTERVISTA provides only one way of inputting textual data. The INTERVISTA function *get-string* presents a small window, in which the user is prompted to type input, and the input is returned as a string. The REFINE function PARSE-FROM-STRING provides a way to parse the input from this string, according to the grammar of the current domain.

5.3.3 Range of Applicability. Though AVSI was specifically developed as a visual interface for Architect, its usefulness is not limited to this particular application. AVSI's methods may be applied to other systems which require a transformation from a diagram to a REFINE formal specification. The techniques used in AVSI may be adapted for any domain model if there is an unambiguous mapping from a diagram to an abstract syntax tree in the REFINE object base.

An example of an application, to which these techniques may be applied, is a thesis effort by Capt Mary Boom and Capt Brad Mallare, entitled *Formalization and Transformation of Informal Analysis Models Into Executable* REFINE *Specifications* (9). Boom and Mallare developed a method of translating informal specifications (based on information extracted from Entity Relationship, State Transition, and Data Flow Models) into an object-based representation in REFINE. A "Unified Abstract Model" (UAM) combines the information from these informal models and "forms the basis for defining a formal language, the Object Modeling Language (OML), used to capture the information contained in the UAM" (9). Once the informal specifications have been translated, the OML description is parsed into an abstract syntax tree in the REFINE object base. The information in the abstract syntax tree forms the basis for an executable specification. What their transformation system lacks is an front-end elicitation tool which allows the end-user to construct the OML specification. The elicitation tool would allow the user to enter visual information by drawing commonly-used informal diagrams. Such an elicitation tool may be built, using the techniques developed in AVSI for visualizing and manipulating the object base. To build such a tool, it would be necessary to define the process of converting the diagrams to the abstract syntax tree representation that would normally be created by the OML parser created by DIALECT. Since Mallare and Boom have already defined a manual process for converting the diagrams to the OML description, this process may be extended to bypass the creation of an explicit OML textual description. This is essentially the process AVSI uses to create the abstract syntax tree representation of Architect applications.

5.3.4 Problems Encountered. Several problems were encountered during the development of AVSI. Some of these problems were conceptual, but most might be considered technical problems. Fortunately, all of the problems encountered were surmountable:

1. I/O was a constant source of frustration; REFINE provides minimal support for I/O, and the rich set of functions provided by underlying LISP system was used extensively.

2. Developing a technique to transform the abstract syntax trees into a diagram was not entirely straightforward; there was not a direct correspondence between the structures in the object base, as defined by Architect's domain model, and the diagram AVSI creates to represent it. Because of the way Anderson and Randour defined the domain model, some objects relationships were defined directly, as a mapping from one object to another; other relationships were defined indirectly, by referencing object names. For example, the statements in a subsystem's update algorithm are mapped directly to the subsystem, but the subsystem's controllees are simply listed as a sequence of object names. Creating the diagram to represent the hierarchical structure of the subsystem and its controllees required a thorough knowledge of the structure of the domain model's definitions of the objects' structure. Some attributes had to be pruned, others required finding the indirectly referenced objects using the REFINE *find-object* function.

3. Developing a method of displaying and connecting communication links presented conceptual problems. The major problem was to devise a way to represent the information in a simple, meaningful manner. Some restrictions were imposed by the use of INTERVISTA, which does not allow using colors, and does not provide much control over how icons are laid out on the screen. With the goal of minimizing the number of icons and links, the method of combining textual information with graphics allowed for a very readable display, since the number of links is kept to an absolute minimum.

4. Allowing the user to connect imports and exports from both directions initially presented difficulties. The difficulty stemmed from the way the connection was represented in the original domain model: An import object contained an attribute that stored information about which export object(s) it was connected to, but export objects did not have direct

access to this information. Though it would have been possible to write a function to find this information, it was easier to make a simple modification to the domain model, adding a new attribute to the export-object definition.

5. Implementing the object attribute editor presented major difficulties. Though REFINE provides a way to find the attributes of an object class, there is no easy way to find the data types of the attributes. Additionally, the function provided to find the attributes returns a large list of attributes, most of which are system-related, and it would not be prudent to allow the user to modify these. The easiest way to give the object attribute editor the information it needed was to include the "editable" attribute names, along with their data types in the Visual Specification Language description for each domain object. This allows for absolute control over which attributes may be edited, and provides all information required by the attribute editor.

5.3.5 AVSI's Shortcomings. Of course, there is always room for improvement. Fortunately, as Architect evolves, so will its interface. Major areas of needed improvement are:

1. A more domain-specific interface. AVSI applications look essentially the same, even though the application domains may be different. The limitations imposed by INTERVISTA may be overcome by using more sophisticated graphics. AVSI provides the basic framework for building an application across domains, but more domain-specific visual information must be added to this framework. This information should not be limited only to icons, but should also define different display functions for the various domains. The Visual Specification Language developed for AVSI should be extended to accommodate this additional information.

2. Visual support for semantic analysis. AVSI currently uses Architect's semantic analysis reporting, which simply reports semantic errors in the Emacs window in textual form. Though this is adequate for the current implementation, using visualization for semantic error reporting would provide a more user-friendly system. Such a use of visualization would bring up

one or more of AVSI's editors in the "problem area." For example, if there was a problem with the definition of a subsystem's structure, the subsystem editor would be invoked, and would contain the subsystem in question, with the problem area highlighted. If there was a problem with one of an object's attributes, the object attribute editor would be invoked, with the attribute highlighted.

3. Visual support for application execution. As with semantic analysis, AVSI relies on Architect's textual output for displaying the results of application execution. A good feature to add to AVSI would be a visual display of objects' state information during execution. For example, this information may be expressed by changing the color of icons and links to show information about objects and relationships between objects.

5.4 Summary

AVSI was validated with the digital circuits domain, and proved to successfully compose the same applications used to validate the original Architect system. Few changes to Architect were required, and the Software Refinery environment provided very good support for the development of AVSI. The techniques used in AVSI for visualizing and manipulating the abstract syntax tree in the Architect domain model are adaptable for other systems which require a transformation from a diagram to an abstract syntax tree in a REFINE object base. This chapter discussed some of the problems encountered during the development process, and how they were solved. Finally, some of AVSI's weakness were mentioned: the need for a more domain-specific visual display, the need for visual support for semantic analysis, and the need for visual support for application execution.

VI. Conclusion and Recommendations

The purpose of this research was to provide a visual interface for viewing and manipulating domain knowledge and software architecture specifications in a formalized object base. In particular, the main goal was to build a visual-oriented interface for Anderson's (2) and Randour's (39) Architect system. The system was to provide an automated means of visualizing an application that exists within the REFINE object base. Additionally, the system was to give the user a means for creating and editing an application.

6.1 Results of This Research

The Architect Visual System Interface successfully met all original goals. The following is a summary of the results:

1. *Provided visualization of domain-knowledge and software architecture* AVSI provides a visual medium for displaying all information about an application contained within the structured object base. The graphical view of an application is divided into several parts:

   (a) The hierarchical structure of the application and each of its subsystems is given graphically by a tree-structured graph.

   (b) Flow of control, which is contained within objects' update algorithms, is represented in diagrammatic form.

   (c) Communication links between subsystems are shown using a combination of graphics and text.

   (d) AVSI allows the user to view the internal attributes of any object and provides a way to examine the REFINE description of each primitive object within a given domain.

   (e) The domain-specific "code" for an application, or any of its components, can be viewed in a text window.

2. *Developed an interface to serve as a "front-end" to the Architect System* Besides providing program visualization and visual programming capabilities, AVSI draws together the various activities involved in composing an application with Architect. Though AVSI provides no visual support for semantic checks or program execution, these activities can be performed from within AVSI.

3. *Developed a Visual Specification Language (VSL) to define the visual characteristics of domain-specific objects* Though VSL is rather simple, its successful development and implementation demonstrates its usefulness. It enables AVSI to remain domain-independent, isolating domain-specific information in an easy-to-modify text file according to a standard format defined by the grammar of VSL.

4. *Developed techniques for manipulating an object-based representation as an abstract syntax tree by using direct manipulation of visual objects* These techniques provide AVSI with a significant visual programming capabi v. Logical objects within an application are directly linked to their representative visual objects. The manipulation of icons and links is translated to a corresponding manipulation of the objects they represent, and the structure of the application of which they are a part. Manipulation of textual information is used in some places where iconic representation wouldn't mal sense, or would be too cumbersome.

6.2 Conclusions

Several conclusions may be drawn from this research:

1. AVSI successfully demonstrates techniques for transforming visual information into executable specifications in domain-specific languages. Fairly simple actions performed by the user composes modules containing previously stored technology into application structures which may be formally evaluated for correctness and executed to observe if the application achieves the

desired behavior. The techniques developed in AVSI may be applied to other systems which are oriented towards transforming diagrammatic information to REFINE formal specifications.

2. AVSI provides a syntax-directed editing capability. This guides the user through the application composition process, and enforces syntactic correctness of the application structure.

3. A thorough understanding of the domain model is essential. The structure of an application's abstract syntax tree is defined by the DIALECT domain model, and AVSI's visualization capability relies on the way the abstract syntax tree is traversed. AVSI's visual programming capability relies on the way the abstract syntax tree is manipulated. The importance of a good domain model definition cannot be overstated.

4. The original implementation of Architect, including its domain model, was relatively easy to create a visual system for. Few changes to the domain model were required; those changes actually made were minor in nature and had no real impact on the original system.

5. The Software Refinery environment provides an excellent platform for developing systems such as Architect and AVSI. Moreover, it consists of a set of integrated tools that allow for quick development of system prototypes. The REFINE language contains many high-level operations for manipulating the object base and traversing its structures. The language supports set theory, logic, transformation rules, pattern-matching, and procedure (41). Given a language's grammar, the DIALECT tool creates a lexical analyzer, parser, and pretty-printer for the language. The parser builds object structures in the object base from text files which may then be manipulated by REFINE programs. Finally, INTERVISTA provides tools for creating diagrams, interactive menus, and windows. The visual objects (icons, windows, etc.) created and used by INTERVISTA coexist with all other REFINE objects in the objec' base. Using the Software Refinery environment allowed AVSI to be developed at a fairly high level; this was especially useful, as it obviated the need to get bogged down with low-level

graphics. The use of these tools significantly decreased the development time of the system. The environment is very flexible, allowing very rapid prototyping of system features.

6.3 Recommendations for Further Research

1. *Develop a more fully domain-oriented visual display methodology* AVSI provides the desired visual features with one exception. The set of icons currently provided by the INTERVISTA tool is limited to three basic shapes: box, ellipse, and diamond. Therefore, an application in one domain essentially has the same "look and feel" as an application in any other domain. For instance, a digital circuit built with AVSI does not look like a digital circuit. A better visual representation would look like a circuit schematic diagram that uses standard logic symbols. Each domain most likely has its own specific way of visually representing its objects and their interrelationships. For example, modeling an assembly line would require displaying objects such as conveyer belts and work stations, and might display a "pipeline" configuration of these objects. Devising an effective visual display should be included in the analysis of the domain being modeled.

2. *Incorporate more complicated domains* AVSI was designed to be applicable across a broad range of domains, and the next step is to apply its techniques to a more substantial domain. The digital logic domain was a good domain to start with because it is well-understood, relatively simple, and it allows applications with multiple instantiations of primitive objects. It was not necessary to spend a great deal of time establishing the domain primitives and devising circuits that would be composed of these primitives. However, partly because of its single communication data type, the digital circuits domain chosen as a validation domain did not fully demonstrate the usefulness of AVSI.

   The Joint Modeling and Simulation System (J-MASS) (4) requires a rich set of domains for constructing simulation models. One such example is signal processing, which consists of primitives such as:

- Coupler
- Gaussian Number Generator
- Variable Attenuator
- Amplitude Detector
- Mixer
- Amplifier
- Comparator
- Filter
- Oscillator

These and other primitives can be used to compose higher level subsystems such as signal generators and receivers. Correspondingly, these can be used to build other, more complex, subsystems such as jammers and trackers. Such a domain would prove useful for demonstrating AVSI's (and Architect's) power to compose sophisticated application systems.

3. *Add extensions to the Visual Specification Language* The goal of providing a domain-independent system does not necessarily conflict with the goal of providing a domain-oriented display for an application. The Visual Specification Language provides a vehicle for specifying domain-specific visual display functions that may be plugged into the visual system. A standard way of defining these display functions should be developed. The visual specification language should also be extended to include information concerning color and display depth once a more sophisticated graphics capability is introduced into the system, i.e., a capability beyond what INTERVISTA provides.

4. *Use lower-level graphics systems for more sophisticated visual display* INTERVISTA does not provide an easy way to implement certain desirable features, and in some cases provides no support at all. For instance, a form-based window, such as the attribute editor would be more user-friendly if values could be typed directly on the form, instead of in a separate window. Moreover, the meaningfulness of visual information would be greatly enhanced by the addition of color and more sophisticated icon shapes. An illustration of this is the display used in the

APTAS system (29). A component's icon contains a nested display of icons representing the components it contains. A green box represents a bottom-level component.

Despite its limitations, INTERVISTA is still a very useful tool, and should not be casually disregarded. It may be possible to augment its functionality by using features of the underlying CommonWindows system. An alternative approach might be a graphical system that is based on Common LISP and CLOS.

5. *Provide visual support for semantic checks* A more complete and user-friendly visual system should guide the user through finding mistakes when they occur. It would be helpful, for example, when a semantic error occurs, that an appropriate window is opened, and the problem icons or links are visually highlighted.

6. *Provide visual support for execution* This is actually a separate (and large) area of research. At a minimum however, state information could be displayed in an icon itself. A good use of this might be changing the appearance of a switch icon to simulate its on or off configuration, or changing the color of an LED icon to simulate its being lit (in the digital circuits domain).

6.4 Concluding Remarks

Coupled with Architect, AVSI enables an application specialist to compose a fairly complex application with relative ease of effort. Though it is not as sophisticated as it might be (and will be in the future), it represents a step towards bringing computer application development to the non-computer-specialist. As Shu (43) points out, it is somewhat ironic that the world's oldest form of written communication (picture drawing) is now becoming an important form of communicating with new and powerful computing devices.

Appendix A. Overview of Architect

This appendix provides a high-level view of the Architect domain-oriented application composition system, upon which AVSI was built. It was jointly-written by Capt Mary Anne Randour (39) and Capt Cynthia Anderson (2). It appears in their individual theses and in AFIT Technical Report AFIT/EN/TR-92-5.

A.1 Introduction

The wide availability of powerful, relatively low-cost computer hardware has led to an explosion in the demand for computer software products to automate a multitude of new tasks. Using traditional methods, computer scientists and programming professionals have been unable to meet, in a timely manner, this demand for the sophisticated, large-scale, reliable software systems required for these new applications. Clearly, a new approach to software design and construction is needed.

> Software engineering will evolve into a radically changed discipline. Software will become adaptive and self-configuring, enabling end users to specify, modify and maintain their own software within restricted contexts. Software engineers will deliver knowledge-based application generators rather than unmodifiable application programs. These generators will enable an end user to interactively specify requirements in domain-oriented terms.... and then automatically generate efficient code that implements these requirements. In essence, software engineers will deliver the knowledge for generating software rather than the software itself.
>
> Although end users will communicate with these software generators in domain-oriented terms, the foundation for the technology will be formal representations... Formal languages will become the lingua franca, enabling knowledge-based components to be composed into larger systems. Formal specifications will be the interface between interactive problem acquisition components and automatic program synthesis components.
>
> Software development will evolve from an art to a true engineering discipline. Software systems will no longer be developed by handcrafting large bodies of code. Rather, as in other engineering disciplines, components will be combined and specialized through a chain of value-added enhancements. The final specializations will be done by the end user. KBSE (Knowledge Based Software Engineering) will not replace the human software engineer; rather, it will provide the means for leveraging human expertise and knowledge through automated reuse. New subdisciplines, such as domain analysis and design analysis, will emerge to formalize knowledge for use in KBSE components. (30:629-630)

Perhaps this vision can become a reality for selected domains, not just within the next century as Michael Lowry predicts, but within the next few years. Research is currently underway at the Air Force Institute of Technology (AFIT) to achieve such a reality. Developing a full-scale application generation system, which is capable of automatically producing efficient code to satisfy user-specified requirements presented in domain-oriented terms, is a considerable task which will require several man-years of effort. However, one element of application generation, the combining or composing of required components into the proper framework or architecture, is attainable in the near term. This chapter explores the issues involved in developing such an end-user application composer and describes one possible methodology for accomplishing it.

A.2 Operational Concept

Several roles are discussed in describing this new approach to software development, an approach where the end-user generates a software application to satisfy his requirements using the software professional's knowledge about how to generate such applications. Some of these roles are new, others are relatively unchanged from those in traditional software system development.

1. System Analyst – Specifies new systems in a domain (25:4). Responsible for developing the concept of operations (defining policy, strategy, and use of application) and defining training requirements (13).
2. System Engineer – Works with the system analyst to partition the system into subsystems and assigns the tasks to software or hardware development, as appropriate (5).
3. Domain Engineer – Possesses detailed knowledge about the domain and gathers all the information pertinent to solving problems in that domain (25:4). Models the real-world entities required to satisfy the policy, strategy, and use of an application as defined by the system analyst. Determines how, if possible, these entities can be modeled within the constraints specified by the software engineer (13).
4. Software Engineer – Designs new software systems in the domain (25:4). Responsible for defining a formalized structure for the domain knowledge and providing the translation from the domain-specific terms to executable software (13).
5. Application Specialist – Uses systems in the domain (25:4). Familiar with the overall domain and understands what the new application must do to meet the requirements (a sophisticated "user"). Provides the application-specific information needed to specify an application.

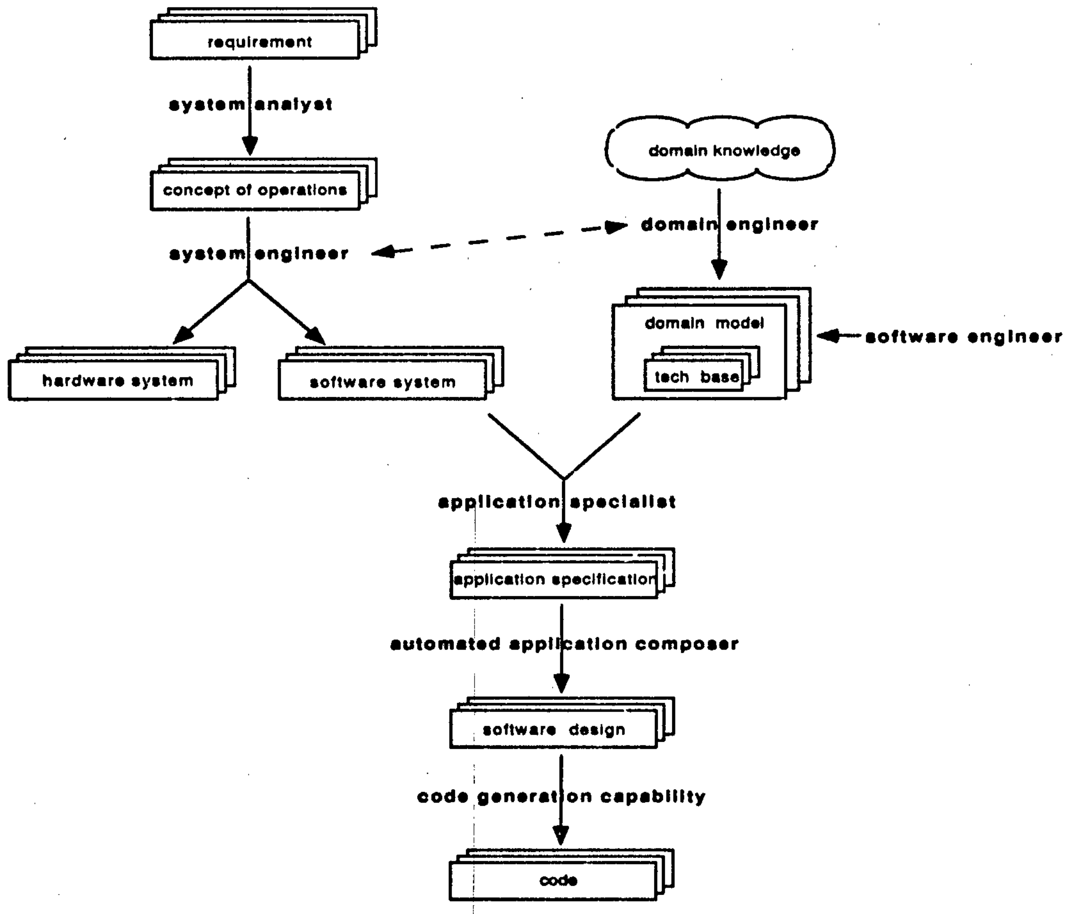


Figure A.1. Roles

The relationships among these roles are shown in Figure A.1. Usually, a new system begins with the identification of a new requirement. This requirement, if valid, is forwarded to a system analyst who develops a concept of operations. The system analyst works closely with the system engineer who partitions the system into software and hardware subsystems. The system engineer consults the appropriate domain engineer to define which components of his domain will be needed for software applications in the domain. The domain engineer and the software engineer decide on which components are needed to model the domain. The software engineer formalizes the domain knowledge provided by the domain engineer into a domain model and its technology base. The application specialist, using the domain model established by the software and domain engineers, creates a specification for an application. From this specification, an automated application composer generates a software design which is then input to a code generation capability.

A.3 General System Concept

A.3.1 Overview. An overview of the application composition system's components and their relationships to each other appears in Figure A.2. First, domain analysis is performed, which consists of gathering appropriate domain knowledge, formalizing it via a domain modeling language, and storing it in a domain model. The structure of the domain model is determined, in part, by the domain modeling language (DML) chosen. The software architecture model, like the DML, imposes a specific structure on the domain model, on the grammar used by the application specialist, and, ultimately, on the final application specification. The domain model is used to develop a domain-specific grammar. Although it may be transparent to the application specialist, he actually uses two grammars: one to identify domain-specific information and one to specify the architecture of the application. The architecture grammar remains the same for different domains; only the domain-specific grammar changes. Application-specific data is written using these two grammars and is converted into objects in the structured object base by the parser.

The populated structured object base and information from the technology base are combined to build an executable prototype. First, the application specialist performs semantic checking on the structured object base to ensure all constraints on the system have been met. He then executes the prototype to demonstrate the behavior of the proposed application. If the prototype does not behave as required, the application specialist can change the original input and re-parse it into the structured object base. Using the knowledge encoded in the domain model and the software architecture model, the structured object base is manipulated into a formal specification for a domain-specific software architecture (DSSA). The DSSA is the system design and becomes the

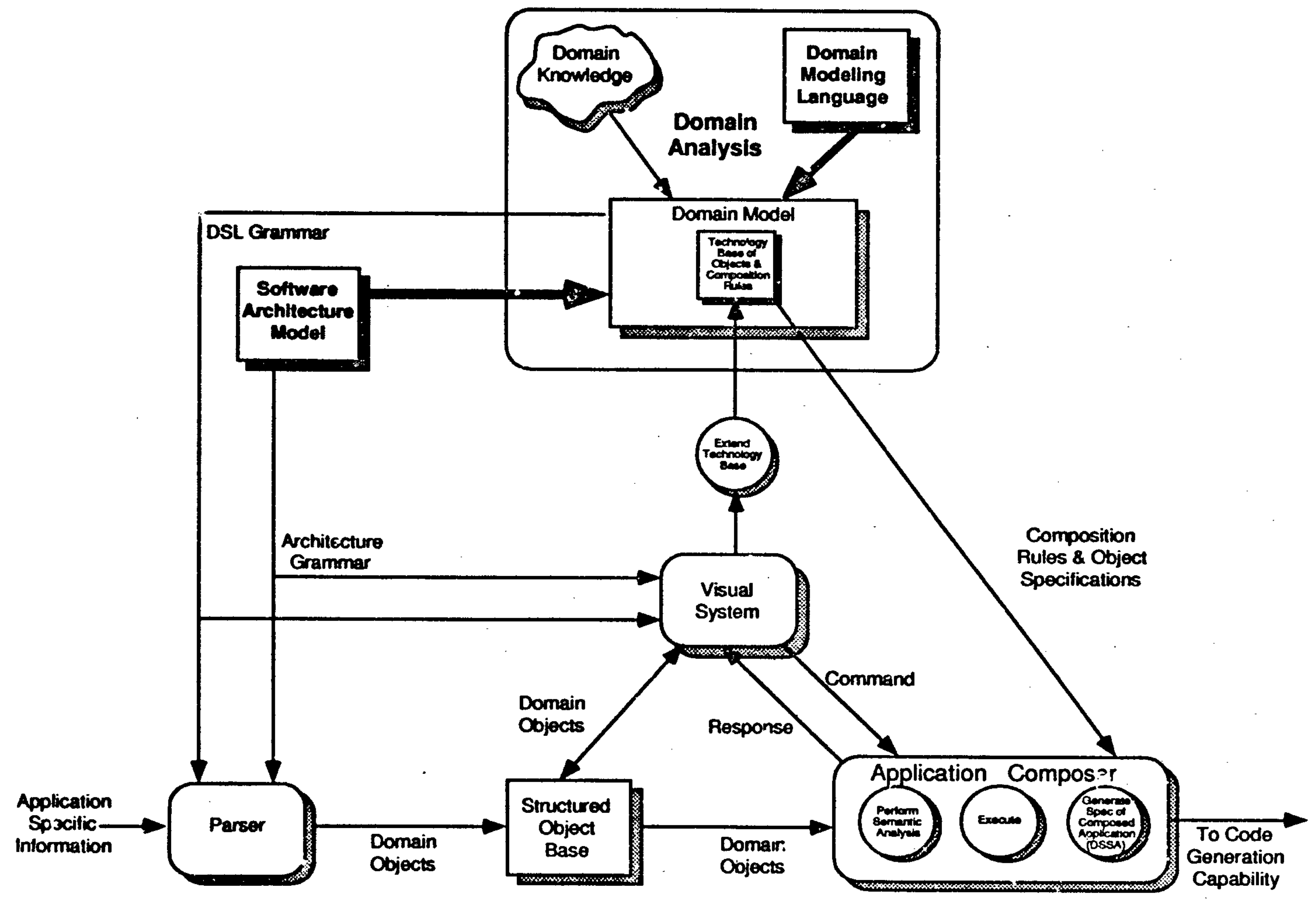


Figure A.2. General System Overview

basis from which code is generated. A visual system provides a graphical representation of the structured object base and the DSSA, as well as a means to add to or modify them.

The remainder of this section desc.ibes the above concepts and activities in more detail.

A.3.2 Developing a Formalized Domain Model. Before any applications can be composed using this proposed system, the domain must be analyzed and modeled. In the software engineering context, a domain is commonly defined as "an application area, a field for which software systems are developed" (36:50) or "a set of current and future applications which share a set of common capabilities and data" (25:2). Identifying the boundaries of the domain, as well as "identifying, collecting, organizing, and representing the relevant information in a domain based on the study of existing systems and their development histories, knowledge captured from domain experts, underlying theory, and emerging technology within the domain" (25:2-3), constitutes domain analysis. Domain analysis is currently the subject of several other research efforts and is not directly addressed in this project. However, it is important to gather the basic data, formalize it, and store it in a standard format.

A.3.2.1 Domain Knowledge. Domain knowledge is the "relevant knowledge" that results from a thorough domain analysis and later evolves naturally as more experience is gained solving problems in the domain (36:47). More specifically, domain knowledge consists of: basic facts and relationships, problem-solving heuristics, domain-specific data types, and descriptions of processes to app.'y the knowledge (6). In the context of this project, domain knowledge includes: descriptions of domain-specific objects (including their attributes and operations), data types, composition rules, and templates for commonly used architectural fragments.

A.3.2.2 Domain Modeling Language. An analogy to a domain modeling language (DML) can be found in the more familiar data definition language of a database management system. A data definition language describes the logical structure and access methods of a database

(26), just as our DML describes the logical structure of a domain model and defines how the objects can be accessed. A DML used to encode domain knowledge into a domain model must be able to formally describe:

1. Object Classes: Abstractions of real-world entities of interest in the domain.
2. Operations: Behavior of the objects in the domain.
3. Object Relationships and Constraints: Rules for relating objects (and sets of objects) to other objects, as well as the constraints on these relationships. Examples include:
   (a) Communication Structure: Message passing between/among domain classes and operations.
   (b) Composition Structure: Rules for combining domain object classes into higher-level application classes and operations into higher-level application operations.
4. Exception Handling: What to do when an error is encountered.

To be useful in an automated system, the domain knowledge must be encoded into a format that the software system can manipulate. This problem is analogous to encoding knowledge in an expert system, where human knowledge is gathered and represented as rules that allow a computer program to utilize the information. Neil Iscoe describes a method for encoding domain knowledge into a domain model (see (24) for details). He proposes using a domain modeling language or a meta-model as the basic framework to instantiate a domain model based on some operational goal(s) (reasons for which the knowledge will be used) (see Figure A.3). Our operational goal is to "use the domain model, software architecture model, and structured object base to generate a software architecture for the application problem to be solved to generate a domain-specific software architecture" (5).

A.3.2.3 Domain Model. A domain model is a "specific representation of appropriate aspects of an application domain" (23:302) including functions, objects, data, and relationships (35). It is a result of expressing appropriate domain knowledge (identified by the domain engineer) in a domain modeling language with respect to certain operational goals (23:301-2).

Several researchers (7, 14, 15, 28) have indicated that software engineering must become more of an engineering discipline if we are ever to reap the benefits of design reuse (increased pro-

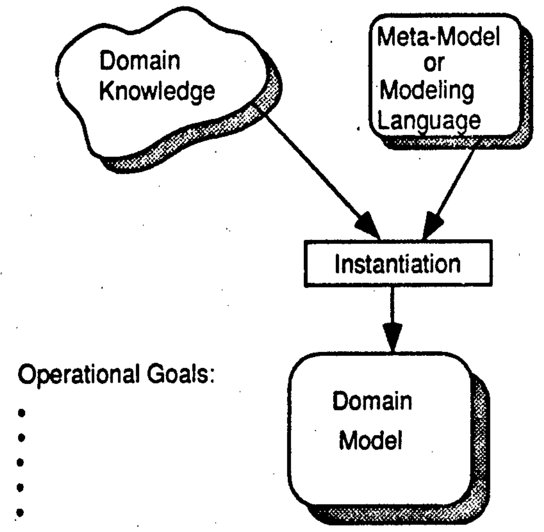


Figure A.3. Domain Model Instantiation

ductivity, improved reliability, certifiability, etc.). When designing specific applications, engineers use models, "codified bodies of scientific knowledge and technology presented in (re)usable forms" (14:256) which are available to all practioners in various technology bases. Reuse of these validated, commonly-used models, which are readily available in various technology bases, allows the engineer to construct a practical, reliable solution to the problem at hand.

Contained within our domain model is such a technology base which acts as a repository for our reusable models. In our system, these models are often referred to as components. Using an object-based perspective, a component can represent a real-world entity, concept or abstraction and encompasses all descriptive and state information for that entity/concept/abstraction as well as its behavior (what operations or functions it performs and/or what transformations it undergoes). Components can be primitive domain objects as described above or a "packaging" of these objects whose structure is determined by the software architecture model. These packaged components will be referred to as architectural fragments since they can be used to build an application architecture. The technology base contains templates for generic components, rules for component composition, and descriptions of primitive object behavior. The parameters required to instantiate these generic templates will be specified by the application specialist.

Domain analysis reveals common features of the software architectures that can be used to implement various specific applications within the domain. In addition, common constraints are identified and codified into rules used to determine how software components can be legally combined. Using rules allows additional flexibility; any specific architecture can be built as long as it meets the criteria specified by the rules.

A.3.3 Building A Structured Object Base. Several steps must be taken to build the structured object base. The following system components are essential to this phase.

A.3.3.1 Domain-Specific Language. As with our domain modeling language, an analogy to a domain-specific language (DSL) can be found in a data manipulation language from the realm of database management systems. In the database context, a data manipulation language allows the user of a database to retrieve, insert, delete, and modify data stored in the database (26:13). In our context, a DSL is a language with syntax and semantics which represents all valid objects and operations in a particular domain, allowing modeling and specification of systems within that domain (37). According to James Neighbors, a domain language is a machine-processable language derived from a domain model. It is used to define components and to describe programs in each different problem area (i.e., domain). The objects and operations represent analysis information about a problem domain (34). In our research, a domain-specific language is defined as a formal language used to define instances of objects and operations specific to a domain.

The objective of our DSL is to generate the structured object base needed to specify an application architecture within a specific domain. To do so, it must be able to:

1. Instantiate objects
2. Instantiate generic objects
3. Instantiate generic architectural fragments
4. Compose the instantiated objects and architectural fragments in some meaningful way

The object classes defined in the domain model are merely templates or patterns to be used when constructing objects; they do not refer to specific, individual objects. The first sentence type listed above creates specific instances of the objects in the object base. These objects are used in building architectural fragments or as parameters for generics. Default values can be used for attributes so these values need not be entered through the DSL every time they are used.

Generics, stored in the technology base, provide templates for commonly used objects and components; thus, the application specialist need not start from scratch each time he wants to include one of these commonly used components. Generics must be instantiated before they can be used. Instantiation is done by specifying which model is to be used and providing specific instances and/or other data, as required. For example, a generic architectural fragment may use three objects of a certain class. When this generic is instantiated, three specific object instances of the required class must be given.

A.3.3.2 Software Architecture Model. In addition to identifying the objects to be used in generating a particular application, the application specialist must indicate what is to be done with those objects; i.e., he must identify the application operations. Domain primitive operations, associated with primitive objects, are available in the technology base. But how can these primitive operations be assembled (composed) into application-specific operations? What are the rules for composing these primitive operations into application operations? How can these rules be represented and implemented?

Software architectures provide insight into software system composition. In its most fundamental sense, an architecture is a recognizable style or method of design and construction. A software architecture has been defined as "a template for solving problems within an application domain" (46:2-2) or "the high level packaging structure of functions and data, their interfaces and controls, to support the implementation of applications in a domain" (25:3). It provides a mechanism for separating "the design of (domain) models from the design of the software" (13). This separation of domain knowledge from software engineering knowledge allows each type of engineer to concentrate on the issues relevant to his own area of experience, without becoming an expert in the other discipline. By focusing only on the design of the software, the software engineer is able to develop simplified packaging and control structures which can be reused across a wide variety of domains.

Because a software architecture serves as a structural framework for software development, we can expect it to provide a consistent representation of system components as well as the interfaces between those components. A standard representation ensures that each component is developed in the same manner, eliminating many implementation choices and simplifying the development process. This standardization also results in consistent interfaces between all components, enabling them to be easily combined. This consistency of component representation and interfaces should provide a suitable and flexible framework for composing primitive operations into application-specific ones.

A.3.3.3 Architecture Grammar. Certain portions of the application specialist's input are not dependent on any particular domain; rather, they depend on the software architecture model. These architectural aspects of the application can be specified using a grammar common to all domains, an architecture grammar. This grammar enforces the structure imposed by the software architecture model by defining valid sentences for packaging the primitive domain objects into architectural fragments to define an application architecture. These sentences will compose application operations using domain-specific components described by the domain-specific grammar and other application operations.

A.3.3.4 Parser. After the application specialist specifies the application components using the domain-specific language and architecture language, the input must be parsed into objects in the structured object base. The parser generates specific object instances whose initial states are determined by the application specialist's input.

A.3.3.5 Structured Object Base. The structured object base contains application specific information: specific instances of domain object classes with all appropriate attribute values for determining the object's state, as well as relationships for both domain objects and operations. The kinds of objects that might populate the object base and the overall structural framework of those objects (the shape of the abstract syntax trees) are established by the domain and software architecture models. The specific object instances and the actual structure of the object base are determined by the application-specific information provided by the application specialist using the DSL and architecture grammars.

A.3.4 Composing Applications. The application composer generates the application architecture specified by the application specialist. This is accomplished by combining the appropriate instantiated domain objects from the structured object base in accordance with the domain composition rules. After the architecture is generated, its behavior can be simulated to demonstrate its suitability and correctness. It should be noted that the operations associated with each object in the technology base are certifiably correct; that is, individual objects are guaranteed to behave as required. However, the specific objects which are composed into the application may have been combined in such a way that the composed application may not behave as expected or required. When the application specialist is satisfied that the composed architecture is actually the one desired, he can generate a formal specification for the architecture which can later be used to develop a fully coded system.

A.3.4.1 Semantic Analysis. After an application is identified, the next step is to ensure that the specified composition is appropriate; i.e., that it makes sense and meets the constraints imposed by the composition rules. This step is accomplished via a semantic analysis phase. As in programming language compilers, one aspect of semantic analysis is to verify that a syntactically correct construct, which satisfies the restrictions of the grammar in which it was written, is "legal and meaningful" (19:10). To be legal and meaningful, the proposed application must meet certain other composition restrictions: e.g., components must already exist before they can be used, an input to one component must be produced as an output from another component, etc. Another aspect of semantic analysis is to use knowledge about domain objects and typical system constructions to assist the application specialist in choosing the components needed and in combining them appropriately to create applications which behave as desired. Errors identified during the semantic analysis phase must be corrected before the composition process can proceed.

A.3.4.2 Execute. A composed application architecture that passes all semantic analysis checks is legal and meaningful, but does it do what the application specialist wants it to do? The execute component of the application composer simulates the behavior of the architecture, using object operations which specify each component's behavior. This behavior simulation may not be efficient or robust enough to serve as a full-scale operational system, but it provides the application specialist timely feedback on the correctness of the specified architecture. If the application is incorrect (i.e., it does not behave as required/expected), the application specialist reassesses the components which were used in the application and how they were combined, creating a new or editted application to satisfy his requirements. This ability to simulate execution behavior in this rapid-prototype manner assures the application specialist that the proposed application actually behaves correctly before a formal specification and fully-coded system are generated.

A.3.4.3 Generate Specification. A legal, meaningful, and correctly composed application provides a software architecture which satisfies the application specialist's requirements for a particular application. The software architecture can be used as a blueprint, template, or specification from which to design and implement a full-scale, operational version of the application. The generated specification is intended to be in a formal, machine-processable format which can be used directly by a code generation tool to produce a fully-coded application. However, the specification format could be tailored to provide whatever form is appropriate for the using organization: graphical, textual, etc.

A.3.5 Extend Technology Base. Eventually, the technology base, which formalizes the knowledge about domain objects, will become outdated as understanding of the domain evolves and as the domain itself adapts to accommodate a changing technological environment. Although the technology base may appear to be static, it must be dynamic enough to accommodate this additional information as well as higher-level object classes and operations, generic components and architectural fragments that are developed. These additional elements give added flexibility to the application specialist because more predefined components are available for future applications

A specialized set of tools allows the technology base to be modified or extended to include this additional or revised domain knowledge. The extender must enforce the structure dictated by the domain modeling language and the software architecture model.

A.3.6 Visualization. "A picture is worth a thousand words." This old adage is still true today, especially when dealing with complex and abstract concepts. The visual system provides the application specialist with a graphical view of the structured object base, as well as the application software architecture generated to satisfy his requirements. By reviewing these "pictures," the application specialist can more fully understand the components available for composition and the application just composed. Moreover, the visual system will also be capable of inserting new instances of domain objects into the structured object base, editing domain objects already in the object base, and executing the application composer. It also provides the capability to extend the technology base, enabling the application specialist and/or the software engineer to add/modify domain object classes, add/modify generic components, and add/modify architectural fragments.

A.4 Related Research

Several other research efforts have addressed various aspects of the system we are attempting to develop. This section summarizes this related work and analyzes the similarities to and differences from our project.

A.4.1 Hierarchical Software Systems With Reusable Components. Don Batory and Sean O'Malley are working to incorporate an engineering culture into software engineering. The traditional engineering mindset dictates that new systems are created by fitting well-tested, well-defined, and readily available building blocks into a well-understood blueprint or architecture, which, if properly used, is guaranteed to produce the desired system. To this end, they have developed a "domain-independent model of hierarchical software design and construction that is based on interchangeable software components and large-scale reuse" (7:2).

In Batory and O'Malley's view, each interchangeable component consists of an interface (everything externally visible) and an implementation (everything else). Different components with the same interface belong to a realm. All the components in a realm are considered to be interchangeable or "plug-compatible" (7:3) because they have identical interfaces. Symmetric components have at least one parameter from their own realm and can be combined in "virtually arbitrary ways" (7:2) (also see Figure A.4). Conceptually, components are seen as layers or building blocks for an application; a system is seen as a stacking of components, i.e., a composition of components. Constraints on stacking components (i.e., rules of composition) are derived from the compatibility of their interfaces.

Hierarchical software system design recognizes that constructing large software systems is a matter of addressing only two issues: which components should be used in a construction and how those components are to be combined together (7:16). It employs an open software architecture, which is limited only by the inherent ability of the components to be combined, i.e., by their interfaces. Symmetric components have no inherent composition restrictions; thus, composition rules are simplified while ensuring maximum design flexibility and potential reusability of components.

Batory and O'Malley use an interesting analogy, equating their concepts to a grammar, as shown in Table A.1 (7:5). Using this analogy, a domain is a language. Consider the following example (7:5):

Given the following plug-compatible components:

A[x:R], B[x:R], C[x:R]

Some of the valid compositions include:

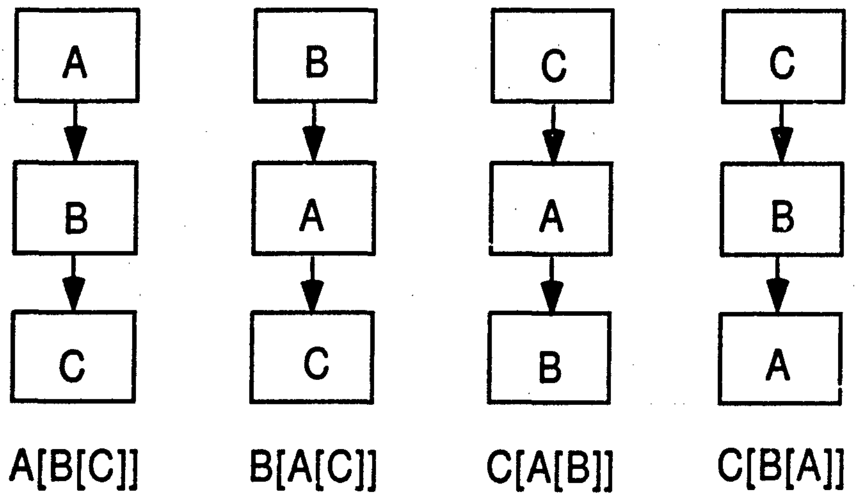


Figure A.4. Combining Plug-Compatible Components

S = {a, b, c} $S \rightarrow a \mid b \mid c$

R = { g[x:S], h[x:S], i[y:R]} $R \rightarrow gS \mid hS \mid iR$

A realm S, having a set of components (a, b, and c), corresponds to a production where the non-terminal S can be replaced by either a, b, or c. Whenever a component from realm S is needed, a, b, or c could be used, depending on the behavior and level of detail needed. A realm R, whose components g, h, and i require parameters from realms S, S, and R, respectively, can be represented by a production where a non-terminal can be replaced by both a terminal and a non-terminal. The non-terminals on the right-hand side are the realms from which the parameters are provided. The complete analogy is summarized in Table A.1.

Concept	Grammar
Parameterized Components	Productions with non-terminals on right
Parameterless Components	Productions that only reference terminals
Symmetric Components	Recursive production
Component Interface	Left side of a production
Implementation	Right side of a production
Realm	Set of all productions with the same head
Software System	Sentence
Rules of Composition	Semantic error checking

Table A.1. Analogy to Grammar

Batory and O'Malley's work provides support for our research. It confirms the underlying principle of an application generator: building software systems from reusable components is "simply" a matter of selecting which components to use and deciding how to compose them together. It reinforces our intention to use an object-oriented approach in designing our system. It also illustrates the role of component interfaces in system composition and demonstrates the importance of consistent interfaces and composition styles in developing rules for combining components.

On the other hand, the Batory/O'Malley work falls short, in some ways, of what we are attempting. It does not incorporate a mechanism for an application specialist to specify new applications in domain-specific terms; this is a primary emphasis of our project. It also does not seem to provide for tailoring of component composition to suit the application being built; composing component A with component B into component C will always produce the same behavior for C. We want to be more flexible in our compositions and allow A and B to be composed into C in one situation and C' in a different situation, depending on how the application specialist specifies the composition.

A.4.2 Automatic Programming Technologies for Avionics Software. The Lockheed Software Technology Center has developed the Automatic Programming Technologies for Avionics Software (APTAS) system pictured in Figure A.5 (29:2). The APTAS system, built for the target tracking domain, "takes a tracking system specification input via user interface with dynamic forms and a graphical editor, and synthesizes an executable tracker design" (29:1). An application specialist defines a new tracking application by answering questions which appear in pop-up, menu-like forms. His answers determine which additional questions are to be asked as he is guided through specifying a new tracker. When all pertinent specifications have been entered (defaults exist for questions which are left unanswered), the application specialist generates a software architecture for the new tracker via the architecture generator. A graphical user interface provides a "picture" of the application architecture and allows the user to change it interactively. After the application specialist is satisfied with the architecture just created, he generates executable code to implement that architecture via the synthesis engine (29). He can also invoke a run-time display which facilitates testing and analyzing the tracker just created.

The Tracking Taxonomy and Coding Design Knowledge Base is at the center of the APTAS system. It contains the system's specification forms, the primitive modules from which new trackers are constructed, and the composition rules which establish how primitive modules are to be

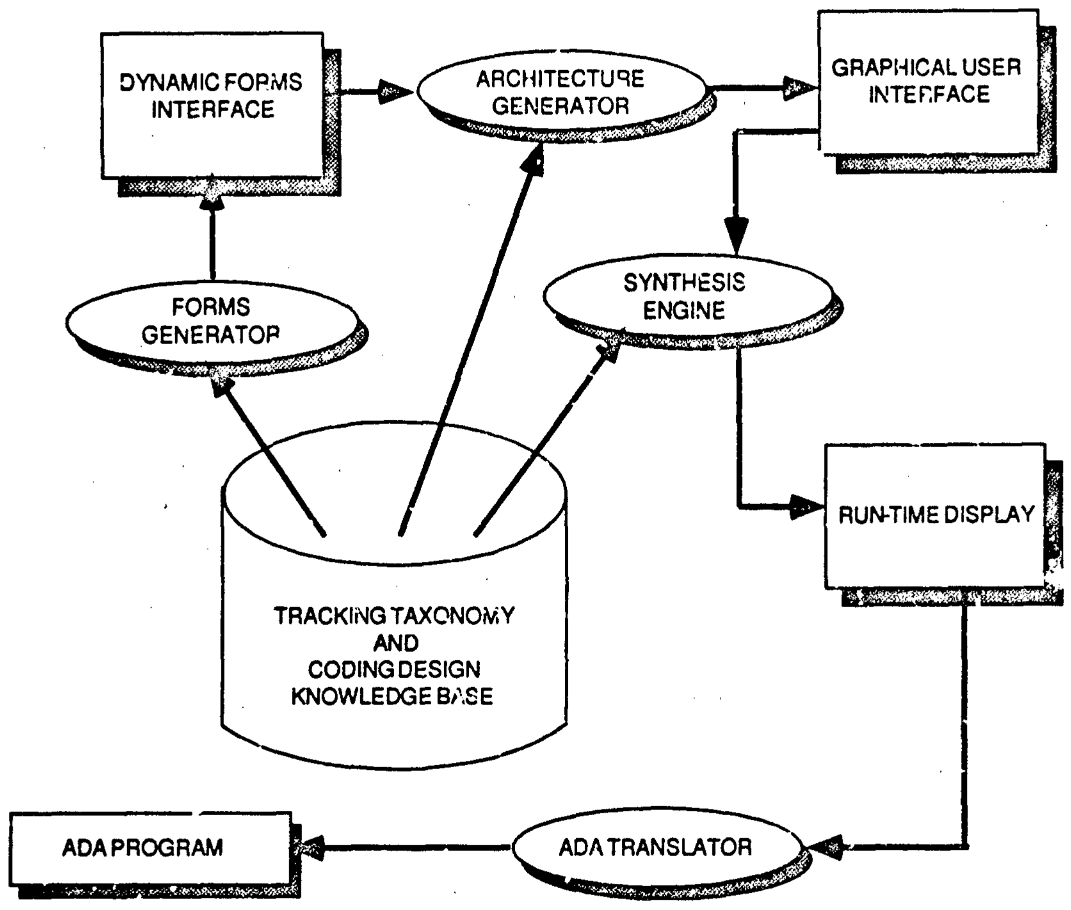


Figure A.5. APTAS

combined. The application specialist's answers to the questions on the specification forms progressively reduce the number of primitive modules which are candidates for incorporation into the new tracker. The architecture generated upon completion of the forms specification is synthesized into an executable intermediate language, Common Intermediate Design Language (CIDL). The CIDL code can be executed to demonstrate system behavior. If the system behaves as desired, the CIDL representation can then be transformed into Ada code. The use of an intermediate representation, such as CIDL, localizes the code translation function and enables languages other than Ada to be targeted more easily.

The APTAS primitive modules and their composition rules are also written in CIDL. Extending the system involves writing new primitive modules and incorporating references to these new modules into the appropriate composition rules and specification forms. This is generally considered to be a software engineer's task (rather than an application specialist's), as CIDL is a software specification language and few tools exist to simplify the process.

APTAS is strikingly similar to the system we envision. It clearly demonstrates that the concept of user-initiated composition and generation of domain-specific systems is feasible. It allows application specialists to specify new applications in domain-specific terms, by way of menu-like specification forms. It also provides a sophisticated graphical user interface which can be used to construct and/or edit the tracker system, as well as to view the structure of the architecture.

There are, however, some major differences between APTAS and the system we are developing. APTAS's use of a domain-specific language is implicit and embodied in its graphical user interface. Our domain-specific language, on the other hand, is explicit and its grammar is usable in both textual and graphical modes. We believe this provides advantages to both the software engineer and application specialist in terms of adaptability, flexibility, and ease of use. In addition, APTAS currently lacks a set of convenient tools to facilitate extending its knowledge base; such a toolset is an integral part of our system.

A.4.3 Model-Based Software Development. The Software Engineering Institute's (SEI) Software Architectures Engineering (SAE) Project has proposed a concept called Model-Based Software Development (MBSD) (28). Like Batory and O'Malley, MBSD strives to apply traditional engineering principles to software development by exploiting prior experience to solve similar problems. This prior experience is codified in models, "scalable units of reusable engineering experience" (28:11), which are stored in a technology base. In a mature engineering domain, the technology base will contain "all the components an engineer needs to predictably solve a class of problems, and the tools and methods needed to predictably fabricate a product from the components specified by the engineer" (28:4). Under MBSD, software development follows the engineering paradigm: reuse existing, mature models rather than starting from scratch for each new development. This involves much more than code reuse; the requirements analysis, design, and software architecture are reused each time the corresponding model is used.

MBSD uses a technology base, a repository of models and composition rules that share common engineering goals. Each model is mapped to a specification form and a software template for the target application language. The specification form is a text-based description which uniquely identifies a specific instance of a model. The software template is code containing place holders, which are replaced with information from the specification form (28:10).

As part of MBSD, the SEI uses the Object-Connection-Update (OCU) model as a consistent pattern of design, a software architecture. This model is especially suited to domains where the real world can be modeled as a collection of related systems and subsystems (28:17). Partitioning a system into subsystems provides different levels of abstraction, giving the flexibility to replace a subsystem with another that either provides a different function or has a different level of detail. In the OCU model, subsystems consist of a controller, a set of objects, an import area, and an export area as pictured in Figure A.6 (28:18).

1. Controller - Performs the mission of the subsystem by requesting operations from the objects it connects. A controller is passive, triggered by a call to perform its mission, and depends on the other subsystem components to accomplish that mission.
2. Objects - Model behavior of real-world entities and maintain individual state information. An object is passive, triggered by a call from the controller to which it is connected.
3. Import Area - Makes data external to the subsystem available to the controller and its objects.
4. Export Area - Makes data internal to the subsystem available to the other subsystems.

Both controllers and objects have standard procedural interfaces used by external controllers or application executives to invoke some action. Controllers have the following procedures (28:19):

1. Update - Updates the OCU network based on state data in the import area and furnishes new state data to the export area.
2. Stabilize - Puts the system in a state consistent with the current scenario.
3. Initialize - Loads the configuration, creates objects, and defines the OCU network.

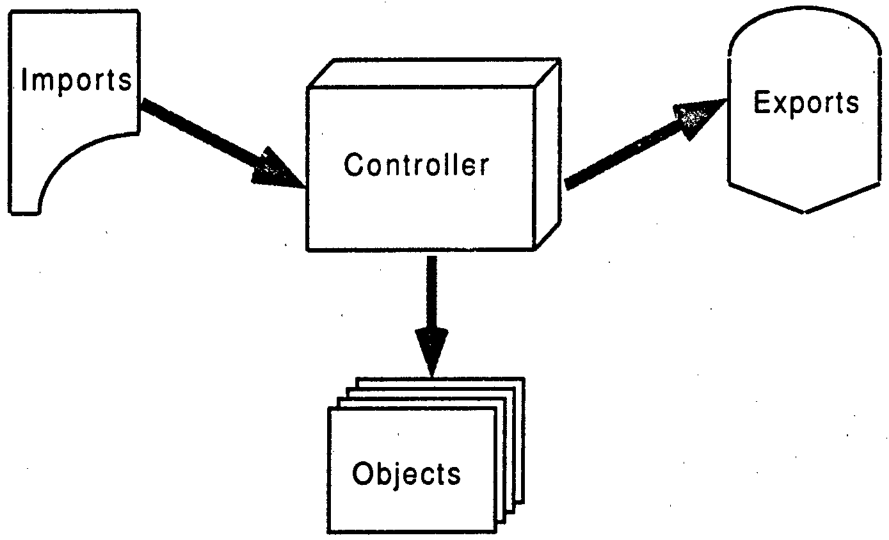


Figure A.6. OCU Subsystem Construction

4. Configure – Establishes the physical connection between import area and input data as well as export area and the output data.
5. Destroy – Deallocates the subsystem.

All objects have procedures analogous to those for controllers, but operating on a single object instance. Specifically, these procedures are (28:20):

1. Update – Calculates the new state based on input data and the current state.
2. Create – Creates a new instance of the object.
3. SetFunction – Changes or redefines the function used to calculate the state.
4. SetState – Directly changes the object's state.
5. Destroy – Deallocates the object.

These well-defined and consistent interfaces for controllers and objects facilitate and simplify the application composition process.

MBSD provides some significant insights upon which to base our research effort. Its focus on the reuse of validated, engineering experience is attractive and we have adopted the notion of storing such information in a technology base. The OCU model provides a realistic approach toward composing primitive objects into application-specific subsystems.

A.4.4 Extensible Domain Models[2]. The Kestrel Interactive Development System (KIDS) is a knowledge-based system that allows for the capture and development of domain knowledge (44). The representation of the domain knowledge constitutes a domain model, and these domain models are called domain theories. Essentially, the domain theory provides a formal language, natural to specialists in that domain, for specifying the problem they want to solve. The KIDS system provides support for constructing, extending, and composing domain theories, and over 90 theories have been built up in the system (44). Additionally, the set of domain theories developed during the domain modeling effort serves as the basis for software synthesis.

The foundations of the KIDS approach emerged from years of research into the specification and synthesis of programs (44). Concepts from algebra and mathematical logic are used to model application domains and synthesize verifiably correct software. Domain modeling entails the analysis of the domain into the basic types of objects, the operations on them, and their properties and relationships. The domain model is then expressed as a domain theory. Theories are useful for modeling application domains for the following reasons.

1. The basic concepts, objects, activities, properties, and relationships of the domain are captured by the types, operations, and axioms of a theory.
2. Any queries, responses, situation descriptions, hypothetical scenarios, etc. are expressed in the language defined by the domain theory.
3. The semantics of the application domain are captured by the axioms, inference rules, and specialized inference procedures associated with the domain theory.
4. Simulation, query answering, analysis, verification of properties, and synthesis of code are supported by inference within the domain theory.
5. Various operations on models such as abstraction, composition, and interconnection are supported by well-known theory operations of parameterization, importation, interpretation between theories, and others. Thus, a high degree of extensibility is obtained.

A.5 Specific System Concept

Several aspects of the system described in Section A.3 depend heavily on the choice of the models and tools used in the implementation. These selections may impact other parts of the

[2]This section was provided by Major Paul D. Bailor

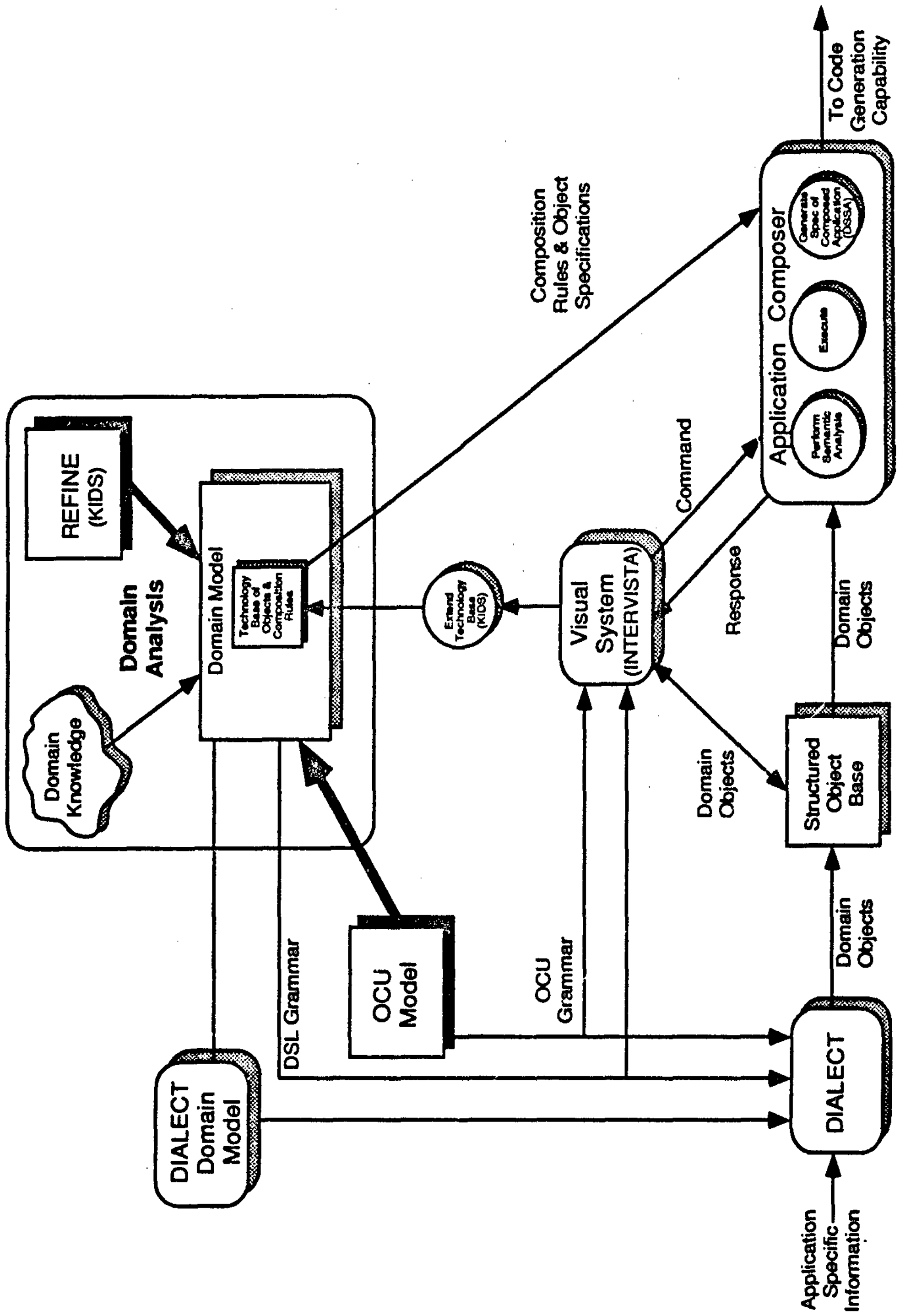
system. Figure A.7 is a modification of the system overview, incorporating the specific models and tools to be used. It represents Architect, the specific system which is to be implemented during this research effort.

A.5.1 System Overview. Figure A.7 illustrates how specific tools and models further define Architect. REFINE, as the domain modeling language, imposes its structure on the domain model (which will be represented in REFINE also). Input, written in the domain-specific and architecture grammars, is processed through a parser generated by DIALECT. DIALECT requires two inputs to generate a parser: a DIALECT domain model (a subset of the system domain model) and a grammar definition. The DIALECT parser creates abstract syntax trees in the structured object base. The visualizer will be implemented using INTERVISTA. The SEI's OCU model will serve as our software architecture model, providing a structure around which to generate our applications. KIDS will serve as a mechanism for realizing extensibility of the domain model and technology base.

A.5.2 Software Refinery. Software Refinery is a formal-based specification and programming environment developed by Kestrel Institute and available commercially from Reasoning Systems, Inc. We have selected this environment in which to implement Architect for several reasons, but the main factor in our decision is REFINE's powerful, integrated toolsets that allow rapid prototyping. This decision has many implications on how the system will operate, as we will show.

A.5.2.1 Capabilities. The REFINE environment consists of the following tools:

1. A programming language (REFINE) which includes set theory, logic, transformation rules, pattern matching, and procedures (41:1-2). The REFINE language provides a wide range of constructs from very high level to low level, making it suitable for various programming styles, including use as an executable specification language.
2. An object base which can be queried and modified through REFINE programs (41:1-2). "Object classes, types, functions and grammars are among the objects you can define and manipulate" (41:1-4) with several built-in and powerful object base manipulation tools.
3. A language definition facility (DIALECT) which allows design of languages using an extended Backus Naur Form notation. REFINE supplies a lexical analyzer, parser, pattern matcher, pattern constructor, and prettyprinter for the language (41:1-2).
4. A toolset (INTERVISTA) which is useful in creating a visual, window-based interactive user interface.



Domain Analysis
Domain Knowledge
REFINE (KIDS)
Domain Model
Technology Base of Objects & Composition Rules
Extend Technology Base (KIDS)
Visual System (INTERVISTA)
Composition Rules & Object Specifications
Command
Response
Application Composer
Perform Semantic Analysis
Execute
Generate Spec of Composed Application (DSSA)
To Code Generation Capability
Domain Objects
Structured Object Base
Domain Objects
Domain Objects
DSL Grammar
OCU Model
OCU Grammar
DIALECT Domain Model
DIALECT
Application Specific Information

Figure A.7. Overview of Specific System

A.5.2.2 Domain Modeling Language. Some domain modeling languages already exist for expressing domain knowledge within a formalized domain model; we considered two such languages: the Requirements Modeling Language (RML) and REFINE.

RML was designed as a research tool as part of the Taxis Project at the University of Toronto. It allows "direct and natural modeling of the world" (20:3) in an object-oriented manner which "captures and formalizes information that is left informal or not documented in current approaches" (20:1). RML can express "assertions (what should be true in the world), as well as entities (the 'things' in the world) and actions (happenings that cause change in the world)" (20:4). This is precisely the type of information we want to capture in our domain model.

Even though both RML and REFINE appear to be capable of expressing the kind of information we require in the domain model, we chose REFINE as our domain modeling language for the following reasons:

1. REFINE provides an integrated environment including programming constructs and powerful object base manipulation tools. Use of REFINE's existing tools eliminated the need to write our own, allowing more time to be spent on the research itself.
2. RML is not an executable language; no compilers currently exist. To use RML, we would be forced to develop a compiler, a considerable overhead to our project. As REFINE is also capable of expressing the information we require, it is unclear what added benefits RML could provide to justify this additional expense.
3. The REFINE environment includes compatible tools (DIALECT and INTERVISTA) useful in other portions of the system.
4. REFINE is a commercially available and supported product.
5. Members of the research team already possessed a working knowledge of REFINE.

A.5.2.3 Parser. "DIALECT is a tool for manipulating formal languages" (42:1-1). A part of the REFINE software development environment, DIALECT generates appropriate lexical analyzers, parsers and pretty-printers for user-specified, context-free grammars. Valid input is parsed and stored as abstract syntax trees in the REFINE object base, according to the structure established in the DIALECT domain model. The DIALECT domain model defines object classes, object attributes, and the structure of the instances in the object base. DIALECT also supports grammar inheritance, allowing for a base language with several variations or "dialects." In Architect, the architecture grammar acts as the common base, and the domain-specific grammar specifies a particular variation. DIALECT does impose restrictions on the grammars. Since DIALECT generates an LALR(1) parser, the grammar must be consistent with this type of parser. Also, the productions in the grammar must correspond to the structure defined in the domain model. Altering some productions may require updating the DIALECT domain model.

A.5.2.4 Structured Object Base. The structured object base was implemented using the REFINE object base. REFINE includes many tools which, when combined with REFINE code, provide all of the functions necessary to manipulate the structured object base. However, the object base must be accessed through the REFINE environment.

A.5.2.5 Technology Base. Models in the technology base were represented as REFINE code and stored in REFINE's object base. Although separate conceptually, the technology base and structured object base are not physically separate. Access is controlled by Architect to avoid any confusion.

A.5.2.6 Visual System. INTERVISTA provides a tool set with which to generate a window-based graphical user interface. It is compatible with the other REFINE tools; therefore, it is easily integrated. INTERVISTA can access the REFINE object base, so all its required data is readily available.

A.5.3 Object-Connection-Update Model. We have selected the Software Engineering Institute's Object-Connection-Update (OCU) model for our software architecture model. As such, it provides a framework for composing applications – a standardized pattern of design for all applications and their components. The OCU model's consistent interfaces enable all components to be accessed in the same manner and its intercomponent communication scheme ensures that each component can readily access the external data needed for its processing. Currently, our focus is on implementing the subsystem aspect of the OCU model: the hardware interface portion of the model will be addressed in follow-on research efforts.

The choice of the OCU model for our software architecture model had certain implications for Architect.

1. Terminology – In keeping with the OCU model, we will refer to domain primitive objects as "objects," compositions of objects as "subsystems," the locus of control of a subsystem as a "controller," and the overall application itself as an "executive" (see (2) for a more detailed discussion of the executive). External data needed by an object are "input-data," whereas data to be made externally available are "output-data." An "import area" serves as a focal point for all external data needed by the subsystem and an "export area" is the focal point for all internal data to be made available to other subsystems. The OCU model's names for the object and controller procedural interfaces have also been retained.
2. Use of a Technology Base – Although the concept of storing reusable domain knowledge or models in a technology base is not unique to the OCU model, it is a fundamental component of Model-Based Software Development of which the OCU model is a part.
3. Domain Analysis – The OCU model deals with objects and subsystems. This imposes a constraint on the domain engineer and will impact the manner in which domain analysis is conducted. Under the OCU model, the domain engineer must model the domain in terms of subsystems which can be composed from lower-level, more primitive objects. Many domains can be naturally modeled in such a way; with other domains, a new mindset may be needed to incorporate the subsystem/object requirements of the OCU model. Alternatively, an additional class of software architectures may need to be defined.
4. Definition of Domain Objects – The OCU model requires that all objects be defined in the same manner. Each object has state data, other descriptive information, input-data/output-data definitions, and the following procedural interfaces: Update, Create, SetFunction, SetState, and Destroy. These requirements dictate how the objects will be constructed, severely limiting implementation choices. However, it is this very limitation which provides the flexibility that allows the domain objects to be successfully composed to satisfy the application specialist's specification.
5. Definition of Architectural Fragments – The OCU model requires that all architectural fragments (subsystems) be described in the same way. All subsystems have an import area, export area, controll and objects. Each controller has the following procedural interfaces: Update, Stabilize, Initialize, Configure, and Destroy. As with the objects, this apparent limitation on implementation choices actually provides great flexibility in composing subsystems and combining them into a complete application.
6. Composition Rules – The standardized object/subsystem definitions and interfaces of the OCU model simplify application composition. There are no inherent restrictions preventing one component from being combined with another; all composition rules are domain-specific and do not derive from the software architecture.
7. Intercomponent Communication – The OCU model establishes and enforces a standard method for intercomponent communication. Communication external to the subsystem is localized in the import area which obtains the necessary input-data for all objects within the subsystem. This localization of communication concerns within the narrow guidelines imposed by this scheme simplifies intermodule communication: subsystems can readily obtain needed

external information in a consistent manner and changes in the low-level implementation of the communication process are hidden from the subsystems/objects.

8. Structure of the Resulting Application Specification – Obviously, the specification produced by the application composer is impacted by the choice of a software architecture model. The OCU model produces an application (an "executive") which is composed of subsystems. These subsystems can be decomposed into objects and lower-level subsystems, if appropriate. This hierarchical structure is preserved in the generated specification.

The OCU model is the result of years of research and experimentation by the SEI. It has been used successfully in the flight simulator, missile, and engineering simulator domains (13) and appears to provide a suitable structure for composing applications within our application composition system.

A.6 Conclusion

Software engineering may be on the brink of a new era, an era in which software engineers develop knowledge about generating software systems and application specialists actually create the software systems using familiar, domain-oriented terms. Our research, which builds on important work already accomplished by various researchers, is designed to demonstrate the feasibility of such an application composer.

Appendix B. Sample Session

This appendix contains a sample session in which we build an application in the digital circuits domain, a binary array multiplier. A schematic diagram for a binary array multiplier is given in figure B.1.

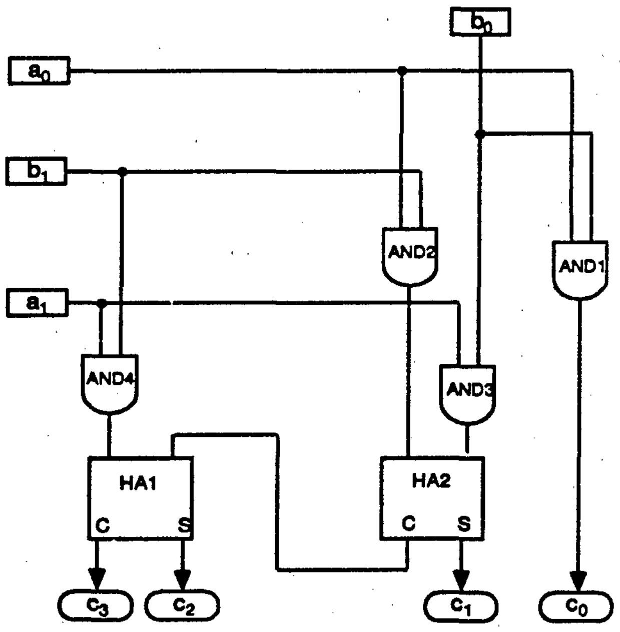


Figure B.1. Binary Array Multiplier Schematic Diagram

B.1 Starting AVSI

Assuming that the Software Refinery system has already been loaded, in the Emacs REFINE window, enter:

```
(load "1")
```

This file contains a LISP function that loads the DIALECT, INTERVISTA, Architect, and AVSI files. After this file has been loaded, enter:

```
(1)
```

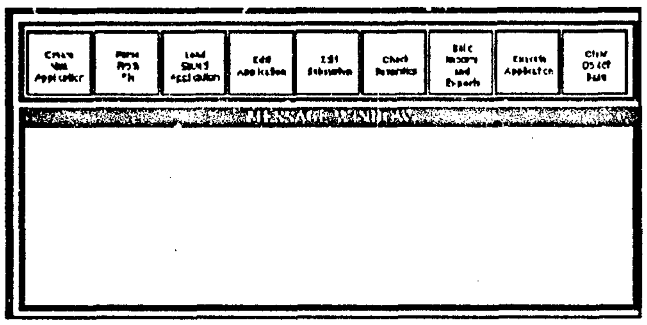


Figure B.2. AVSI Main Window

It will take a few minutes for the files to load. Upon completion, a prompt appears:

```
Load Complete
Type "(AVSI)" to start AVSI
```

Enter the command:

```
(AVSI)
```

The visual specification files are now loaded and the control window (refer to B.2) appears in the upper left-hand corner of the screen. The upper portion of this window contains "buttons" that represent the AVSI's main functions for composing an application. The lower portion contains a message area where various user messages are displayed during the application composition process.

B.2 Create New Application

To create a new application,

1. Click any mouse button on the button labeled "Create New Application."
2. A pop-up window appears and prompts, "Select Domain." Click on the menu item "CIRCUITS."
3. A pop-up window appears with the prompt, "Enter name of application." Type

   ```
   binary-array-mult
   ```

4. The name can be entered by hitting the "return" key or by clicking on "do it" at the bottom of the pop-up window.

B.3 Edit Application

Editing an application is comprised of two separate operations, editing an application's components, and editing an application's update algorithm.

To add a subsystem-object to the application,

1. Click any mouse button on the button labeled "Edit Application".
2. A pop-up menu appears with the prompt "Choose Application." Click on the menu item "BINARY-ARRAY-MULT."
3. A pop-up menu appears with the prompt "Choose:" Click on the menu item "Edit Application Components." A window appears with a single icon labeled, "APPLICATION-OBJ BINARY-ARRAY-MULT" (refer to Figure B.3).
4. Click on the diagram surface (anywhere within the window except within any icon's boundaries) of the new window. A pop-up menu will appear.
5. Select "Create New Subsystem."
6. A pop-up window appears, with the prompt "Enter a name:" Enter

   ```
   driver
   ```
7. A new icon appears and is attached to the mouse cursor. Place the icon below the application-object icon by moving the cursor to the desired location and clicking.
8. Click any mouse button on the newly created subsystem icon and select the menu option "Link to Source."
9. The mouse cursor changes from an arrow to an oval with a dot in it. This signifies that an object needs to be selected. Place the mouse cursor on the application-object's icon and click

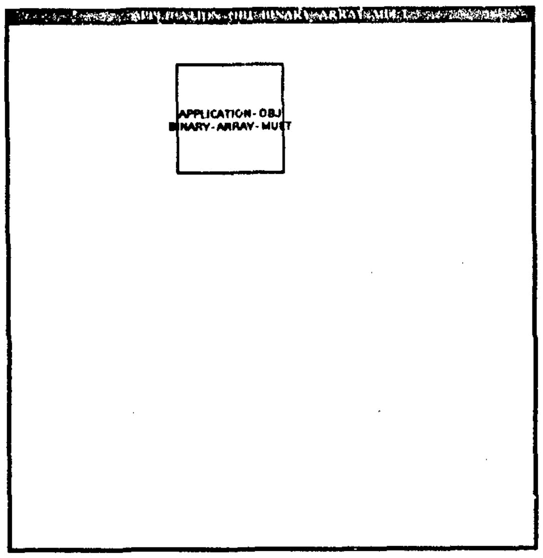


Figure B.3. Edit-application-objects window

any mouse button. A link appears between the application-object's icon and the subsystem-object's icon, as in figure B.4

10. Close the edit-application-objects window by clicking on the diagram surface and selecting "Deactivate"

To edit the application-object's update-algorithm.

1. Click any mouse button on the button labeled "Edit Application".
2. A pop-up menu appears with the prompt "Choose Application." Click on the menu item "BINARY-ARRAY-MULT."
3. A pop-up menu appears with the prompt "Choose:" Click on the menu item "Edit Application Update." Two windows appear, one contains the update algorithm's diagrammatic view, and the other contains its textual view. The diagram window contains two text icons, labeled "Start" and "End," and there is a dotted arrow pointing from the start-icon to the end-icon (refer to figure B.5).
4. Click on the "nub" which is placed on the dotted arrow midway between the start and end icons.

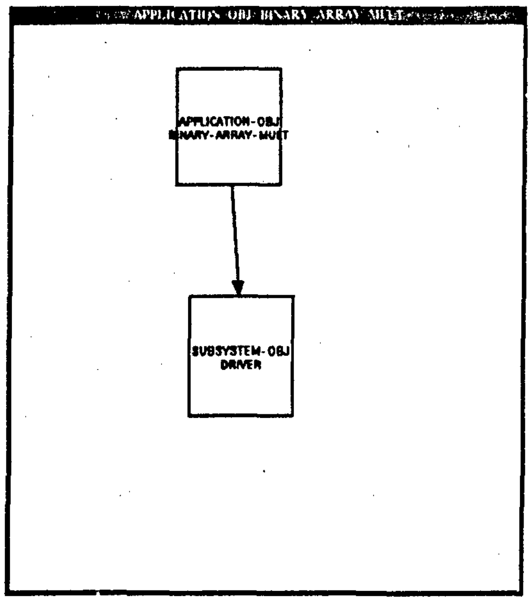


Figure B.4. Edit-application-objects window

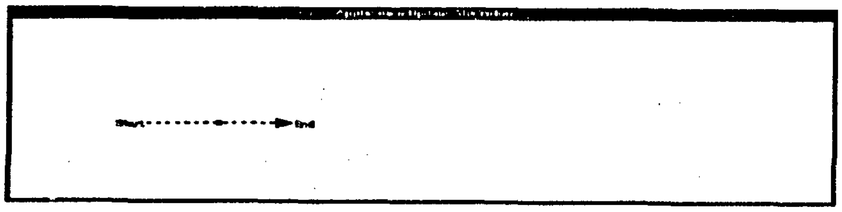


Figure B.5. Edit-Update-Algorithm Window with Null statement sequence

5. Select "Insert Statement" from the pop-up menu.

6. Select statement update-call statement by clicking on "Update-call-obj" from the pop-up menu. An icon is automatically placed midway between the start and end icons.

7. Click on the new update icon and select "Edit."

8. A pop-up window appears with the prompt, "Enter Operand Name," and lists the current operand name, which is "*UNDEFINED*." Replace the current operand name with:

   Driver

   The application update algorithm is shown in figure B.6. Note the textual representation is

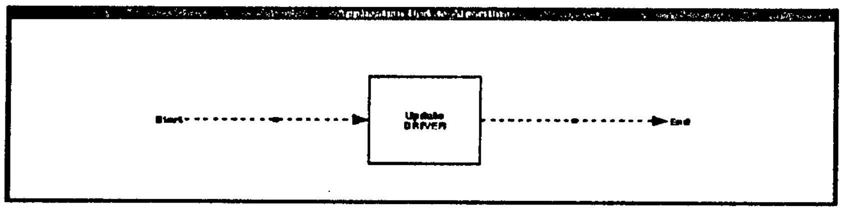


Figure B.6. Edit-Update-Algorithm Window with update statement

   automatically updated to reflect each change in the diagram window.

9. Close the edit-update-algorithm windows by clicking on the black title bar at the top of the windows and selecting "deactivate."

B.4 Editing a Subsystem

With minor differences, editing a subsystem is very similar to editing an application. In this section, we will add create primitive objects and a nested subsystem for the subsystem created in the previous section.

The subsystem, "driver" controls four switches, four LEDS, and the binary array multiplier, a separate subsystem. To add these objects, perform the following steps:

1. Click on the "Edit Subsystem" button in the control panel window.
2. Click on the menu item "DRIVER." A window now appears, the edit-subsystem window, which contains a representation of a subsystem similar to the standard OCU picture of a subsystem (see figure B.7).

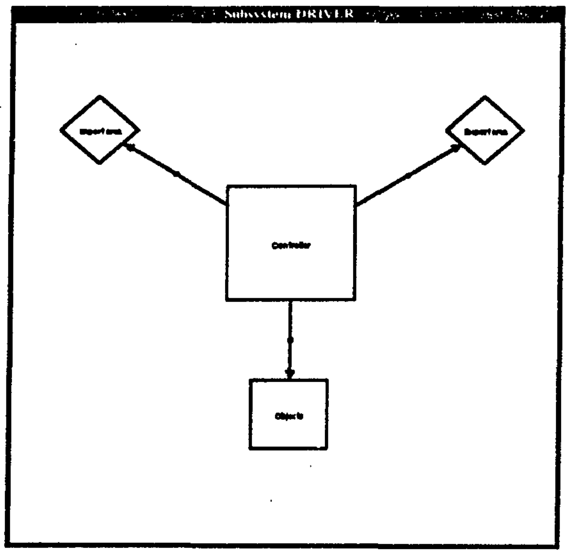


Figure B.7. Edit-subsystem window

3. Click on the "objects" icon in the edit-subsystem window. The edit-subsystem-objects window for this subsystem appears, and contains a single icon, labeled "SUBSYSTEM-OBJ DRIVER." A second window, the technology-base window, appears and contains an icon for each primitive-object in the current domain (see figure B.8).

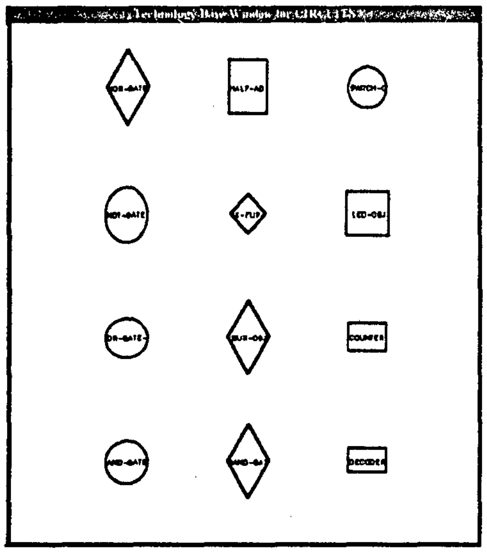


Figure B.8. Technology Base Window

4. Click on the diagram surface of the edit-subsystem-objects window and select the "create new subsystem" menu item.

5. Name the new subsystem

    BAM

6. Place the subsystem-icon somewhere on the diagram window.

7. Click on the new subsystem-icon (BAM), and select the menu item "link to source."

8. Link the new subsystem-icon (BAM) to its controlling subsystem (DRIVER).

Add the primitive objects:

1. Click on the icon in the technology-base window labeled "Switch-Obj."
2. A Switch-icon is created and attached to the mouse cursor. Place this icon on in the edit-subsystem-objects window near the subsystem-icon labeled "DRIVER."
3. Name the switch by typing in the pop-up window,

   A0

4. Click on the Switch-icon and select the menu item, "Link to source."
5. Connect the Switch-icon to its controlling subsystem's icon (DRIVER)
6. Follow the above steps to create, place, and link to the DRIVER subsystem, three more switch objects, named A1, B0, and B1.
7. Create four LE.) objects, named C0, C1, C2, and C3, and link them to the DRIVER subsystem, as in Figure B.9
8. Force a screen redraw by clicking on the edit-subsystem-object's diagram surface and selecting the menu item "Redraw." The subsystem is scaled and redrawn as in figure B.10.
9. You may, at any time, pretty-print an object by clicking on its icon and choosing the menu selection "pretty-print object." Figure B.11 shows a pretty-print of DRIVER.
10. Kill edit-subsystem-objects window by clicking its black title bar and selecting "deactivate."

Create an update algorithm by performing the following steps:

1. Click on the "Controller" icon in the edit-subsystem window.
2. Click on the "nub" which is placed on the dotted arrow midway between the start and end icons.
3. Select "Insert Statement" from the pop-up menu.

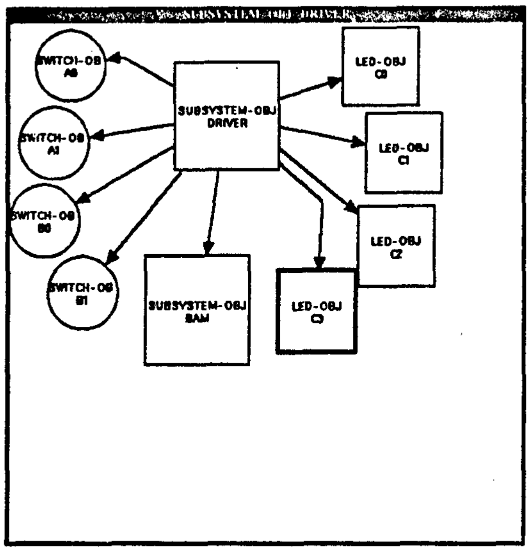


Figure B.9. DRIVER Subsystem, with primitive objects and nested subsystem

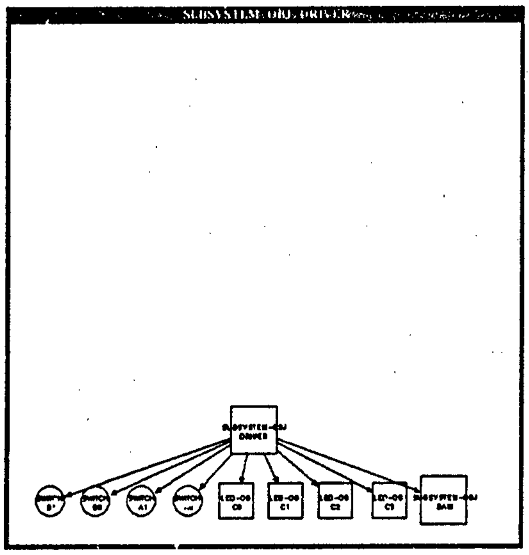
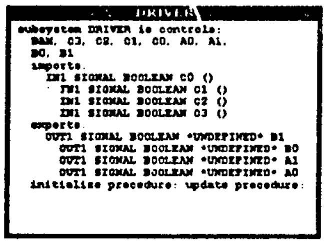


Figure B.10. Redrawn, scaled DRIVER Subsystem



```
subsystem DRIVER is controls:
  BAM, C3, C2, C1, C0, A0, A1,
  B0, B1
  imports.
    IN1 SIGNAL BOOLEAN C0 ()
      IN1 SIGNAL BOOLEAN C1 ()
      IN1 SIGNAL BOOLEAN C2 ()
      IN1 SIGNAL BOOLEAN C3 ()
  exports.
    OUT1 SIGNAL BOOLEAN *UNDEFINED* B1
      OUT1 SIGNAL BOOLEAN *UNDEFINED* B0
      OUT1 SIGNAL BOOLEAN *UNDEFINED* A1
      OUT1 SIGNAL BOOLEAN *UNDEFINED* A0
  initialize procedure: update procedure:
```

Figure B.11. Pretty-print of subsystem DRIVER

4. Select statement update-call statement by clicking on "Update-call-obj" from the pop-up menu. An icon is automatically placed midway between the s. .t and end icons.

5. Click on the new update icon and select "Edit."

6. A pop-up window appears with the prompt, "Enter Operand Name." Type:

```
A0
```

7. New statements may be inserted between two icons by clicking on the "nubs" positioned on the link that is between the icons. Insert several more update-call statements by following the previous steps, creating the following sequence (note that the order of the statements *is* important):

```
update A0
update A1
update B0
update B1
update BAM
update C0
update C1
update C2
update C3
```

8. Deactivate the update algorithm windows.

The subsystem, "BAM" controls four AND-gates and two half-adders. To add these objects, perform the following steps:

1. Click on the "Edit Subsystem" button in the control panel window.

2. Click on the menu item "BAM."

3. Click on the "objects" icon in the edit-subsystem window. The edit-subsystem-objects window for this subsystem appears, and contains a single icon, labeled "SUBSYSTEM-OBJ BAM."

4. Click on the icon in the technology-base window labeled "AND-gate-obj."

5. An AND-gate-icon is created and attached to the mouse cursor. Place this icon on in the edit-subsystem-objects window near the subsystem-icon labeled "BAM."

6. Name the switch by typing in the pop-up window,

```
and-1
```

7. Click on the And-gate icon and select the menu item, "Link to source."

8. Connect the And-gate icon to its controlling subsystem's icon (BAM)

9. Follow the above steps to create, place, and link to the BAM subsystem, three more AND-gate objects, named and-2, and-3, and-4.

10. Create two Half-adder objects, named ha-1 and ha-2, and link them to the BAM subsystem.

11. Force a screen redraw by clicking on the edit-subsystem-object's diagram surface and selecting the menu item "Redraw."

12. Kill edit-subsystem-objects window

Create an update algorithm by performing the following steps:

1. Click on the "Controller" icon in the edit-subsystem window.

2. Click on the "nub" which is placed on the dotted arrow midway between the start and end icons.

3. Select "Insert Statement" from the pop-up menu.

4. Select statement update-call statement by clicking on "Update-call-obj" from the pop-up menu. An icon is automatically placed midway between the start and end icons.

5. Click on the new update icon and select "Edit."

6. A pop-up window appears with the prompt, "Enter Operand Name." Type:

   ```
   and-1
   ```

7. Insert several more update-call statements by following the previous steps, creating the following sequence:

   ```
   update and-1
   update and-2
   update and-3
   update and-4
   update ha-2
   update ha-1
   ```

8. Close all edit-update windows

B.5 Connecting Subsystems' Imports and Exports

To connect subsystems' import and export areas perform the following steps:

1. Click on the control panel button labeled "Build Imports and Exports." The import/export window appears, and contains an icon group for each subsystem in the application (refer to figure B.12). Each icon group contains three icons: a rectangle and two circles. The

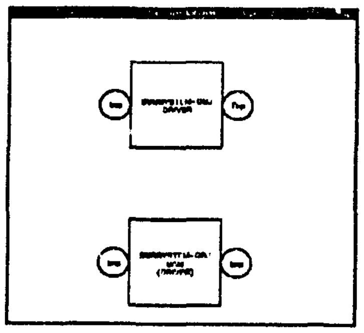


Figure B.12. Subsystem Icon Groups

rectangle is the subsystem icon, and is labeled with the subsystem name. If a subsystem is a nested subsystem, its controlling subsystem appears in parentheses. The two circles are the import-area (labeled "Imp.") and export-area (labeled "Exp.") icons.

2. Click on the subsystem icon (the rectangular icon) labeled BAM. Choose "Make internal connections" from the pop-up menu. The import-export window now contains an icon group for each primitive object in the subsystem, BAM (refer to figure B.13).

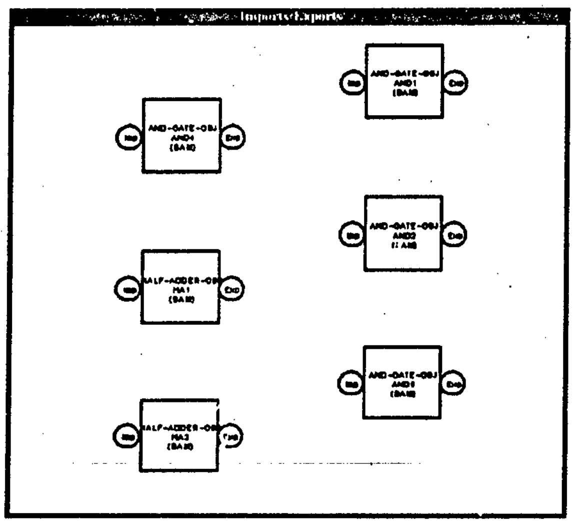


Figure B.13. Icon Groups for Primitives

3. Click on HA1's "Imp" icon. A window opens that lists the import objects in the BAM subsystem that are associated with HA1.

4. Click on the item named "IN2." Observe in the diagram window that dashed arrows appear, showing all the potential sources for this import item.

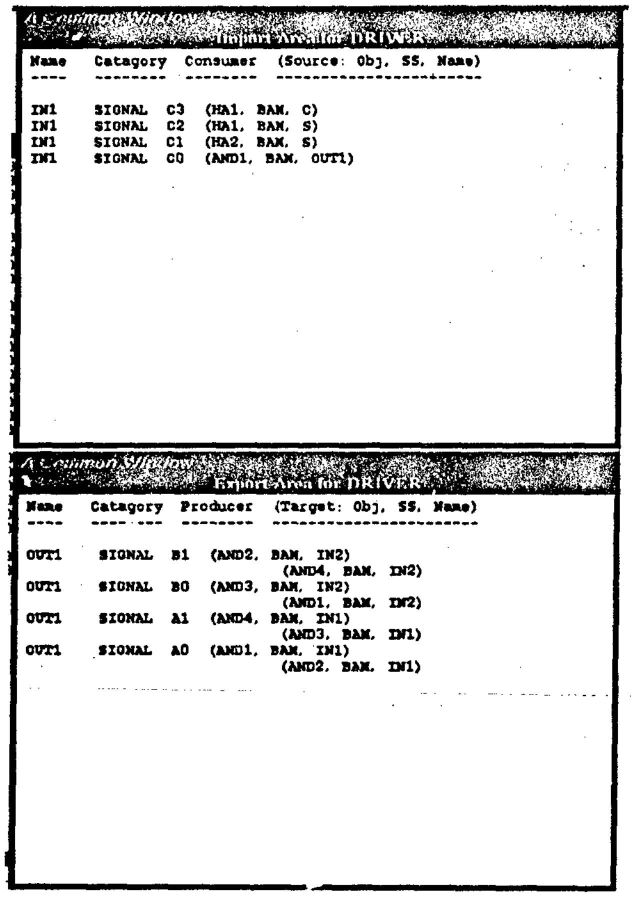
5. Click on HA2's "Exp" icon. A window opens that lists the export objects in the BAM subsystem that are associated with HA2.

6. Click on the item named "C."
7. Observe that the "source" and "target" entries are filled in in the text windows indicating the connection just made. Also observe that the dashed arrows disappear in the diagram window.
8. Follow the above steps to establish the rest of the "local" connections:

Source	Target
——	——
HA1.IN1	AND4.OUT1
HA2,IN1	AND2.OUT1
HA2,IN2	AND3.OUT1

9. Close the Import/Export Diagram window.
10. Click on the "Build Imports/Exports" button on the control panel. This brings up the original display. Now we will make the subsystem-to-subsystem connections.
11. Click on Subsystem DRIVER's "Imp" icon. This brings up a text window that lists all of DRIVER's import objects. Note that all of these import objects have the same name and may be distinguished by the "consumer" field.
12. Click on the item named "IN1" for consumer C3.
13. Click on subsystem BAM's "Exp" icon to bring up its export area test window.
14. Click on the item named "C" for Producer HA1 and observe that source and target fields are filled in, indicating the connection.
15. Repeat the above steps the establish the following connections shown if figures B.14 and B.15.

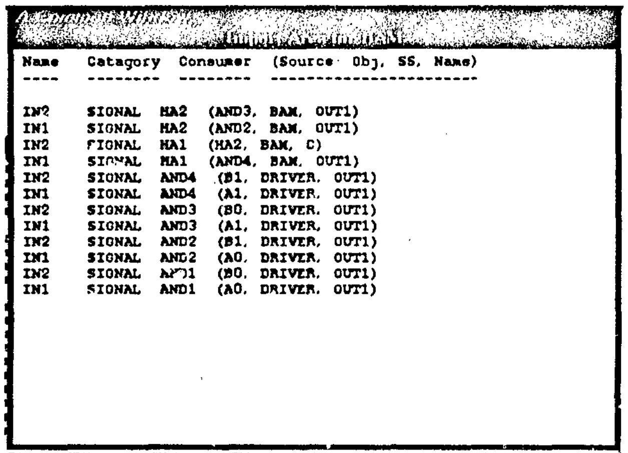
16. Close all windows except for the control-panel window.



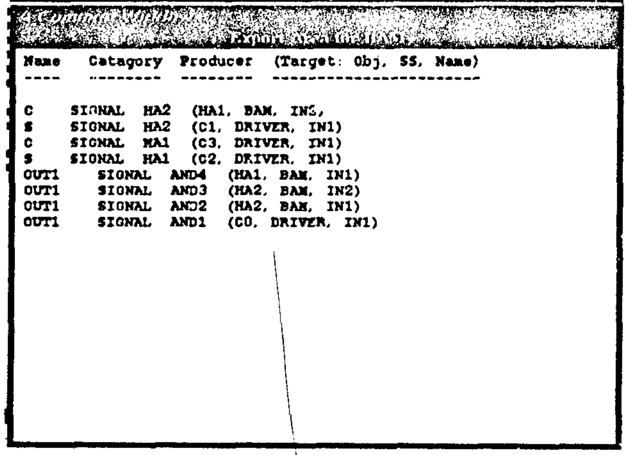
```
Name    Catagory  Consumer  (Source: Obj, SS, Name)
----    --------  --------  -----------------------

IN1     SIGNAL  C3  (HA1, BAM, C)
IN1     SIGNAL  C2  (HA1, BAM, S)
IN1     SIGNAL  C1  (HA2, BAM, S)
IN1     SIGNAL  C0  (AND1, BAM, OUT1)
```

```
Name    Catagory  Producer  (Target: Obj, SS, Name)
----    --------  --------  -----------------------

OUT1     SIGNAL  B1  (AND2, BAM, IN2)
                          (AND4, BAM, IN2)
OUT1     SIGNAL  B0  (AND3, BAM, IN2)
                          (AND1, BAM, IN2)
OUT1     SIGNAL  A1  (AND4, BAM, IN1)
                          (AND3, BAM, IN1)
OUT1     SIGNAL  A0  (AND1, BAM, IN1)
                          (AND2, BAM, IN1)
```

Figure B.14. Import/Export Connections for Driver



```
Name   Catagory  Consumer  (Source  Obj, SS, Name)
----   --------  --------  ----------------------

IN2    SIGNAL  HA2   (AND3, BAM, OUT1)
IN1    SIGNAL  HA2   (AND2, BAM, OUT1)
IN2    SIGNAL  HA1   (HA2, BAM, C)
IN1    SIGNAL  HA1   (AND4, BAM, OUT1)
IN2    SIGNAL  AND4   (B1, DRIVER, OUT1)
IN1    SIGNAL  AND4   (A1, DRIVER, OUT1)
IN2    SIGNAL  AND3   (B0, DRIVER, OUT1)
IN1    SIGNAL  AND3   (A1, DRIVER, OUT1)
IN2    SIGNAL  AND2   (B1, DRIVER, OUT1)
IN1    SIGNAL  AND2   (A0, DRIVER, OUT1)
IN2    SIGNAL  AND1   (B0, DRIVER, OUT1)
IN1    SIGNAL  AND1   (A0, DRIVER, OUT1)
```



```
Name   Catagory  Producer  (Target: Obj, SS, Name)
----   --------  --------  ----------------------

C     SIGNAL  HA2  (HA1, BAM, IN2,
S     SIGNAL  HA2  (C1, DRIVER, IN1)
C     SIGNAL  HA1  (C3, DRIVER, IN1)
S     SIGNAL  HA1  (C2, DRIVER, IN1)
OUT1     SIGNAL  AND4  (HA1, BAM, IN1)
OUT1     SIGNAL  AND3  (HA2, BAM, IN2)
OUT1     SIGNAL  AND2  (HA2, BAM, IN1)
OUT1     SIGNAL  AND1  (C0, DRIVER, IN1)
```

Figure B.15. Import/Export Connections for BAM

B.6 Semantic Checks

Whenever the import and export areas are connected, as in the previous section, semantic checks are automatically run. However, semantic checks may be run at any time by clicking on the control panel button labeled "Check Semantics." The results of the semantic checks may be viewed in the EMACS window.

B.7 Execute Application

Click on the control panel button labeled "Execute Application." The results are display in the EMACS window.

Change switch settings with the object attribute editor by performing the following steps:

1. Click on the control panel's "Edit Subsystem" button.
2. Choose "DRIVER" from the pop-up window.
3. Click on the "Objects" icon in the edit-subsystem window.
4. Click on the switch icon labeled "A0."
5. Choose "View/Edit attributes" from the pop-up window. A window appears, listing A0's attributes.
6. Click on the attribute, "Position." A pop-up window appears, listing the current value of the switch.
7. Enter a new value for the switch position by typing

   ```
   ru::off
   ```

   (The "ru::" prefix is the package name and is required for symbols. It is not required for other data types such as numbers and strings)
8. Change the values for any other switches in the same manner as above.

9. Click the "Execute" button again and observe the results in the EMACS window.

Appendix C. System Files

This appendix contains a listing of the Lisp file used to load Architect and AVS. The order in which the files are loaded in this file also indicates the required compilation order.

```
(defun l()

% Load system files for Dialect and Intervista

  (load-system "dialect" "1-0")
  (load-system "intervista" "1-0")

% Load Architect files

  (load "./DSL/lisp-utilities.lisp")
  (load "./OCU-dm/dm-ocu")
  (load "./OCU-dm/gram-ocu")
  (load "./DSL/globals")
  (load "./DSL/obj-utilities")
  (load "./DSL/read-utilities")
  (load "./DSL/erase")
  (load "./DSL/menu")
  (load "./DSL/display-files")
  (load "./DSL/modify-obj")
  (load "./DSL/save")
  (load "./DSL/generic")
  (load "./DSL/build-generic")
  (load "./DSL/complete")
  (load "./OCU/set-debug")
  (load "./OCU/imports-exports")
  (load "./OCU/eval-expr")
  (load "./OCU/execute")
  (load "./OCU/semantic-checks")
  (load "./load-logic-domain")
  (load "./load-gizmo-domain")

% Load Logic Domain

  (load "./CIRCUITS-TECH-BASE/and-gate")
  (load "./CIRCUITS-TECH-BASE/or-gate")
  (load "./CIRCUITS-TECH-BASE/nand-gate")
  (load "./CIRCUITS-TECH-BASE/nor-gate")
  (load "./CIRCUITS-TECH-BASE/not-gate")
  (load "./CIRCUITS-TECH-BASE/switch")
  (load "./CIRCUITS-TECH-BASE/jk-flip-flop")
  (load "./CIRCUITS-TECH-BASE/led")
  (load "./CIRCUITS-TECH-BASE/counter")
  (load "./CIRCUITS-TECH-BASE/decoder")
  (load "./CIRCUITS-TECH-BASE/half-adder")
  (load "./CIRCUITS-TECH-BASE/mux")
  (load "./GIZMOS-TECH-BASE/gizmo-gram-dsl")

% Load Gizmos Domain

  (load "./GIZMOS-TECH-BASE/contraption")
  (load "./GIZMOS-TECH-BASE/gadget")
  (load "./GIZMOS-TECH-BASE/glibsnitz")
  (load "./GIZMOS-TECH-BASE/thing")
  (load "./GIZMOS-TECH-BASE/widget")
  (load "./GIZMOS-TECH-BASE/gizmo-gram-dsl")

% Load AVS files (these files are in the
%     REFINE-USER-REFINE-INTERFACE package

  (load "ri-user-pkg")
  (in-package 'ru-ri)

  (load "lisp-file-utils")
  (load "tech-base")
  (load "vsl-dm")
  (load "vsl-gr")
  (load "vsl-utils")
  (load "edit-expression")
  (load "viz-utils")
  (load "edit-update")
  (load "edit-attr")
  (load "create-obj")
  (load "edit-ss")
  (load "edit-applic")
  (load "viz")
  (load "imp-exp")


% Parse VSL description files

  (in-grammar 'ru::viz)
  (pvf "vsl-circuits.re")
  (pvf "vsl-gizmos.re")

 )
```

Appendix D. VSL Specification of Digital Circuits Domain

This appendix Contains the domain model and grammar for the Visual Specification Language, and contains the VSL descrintion for the CIRCUITS domain:

D.1 VSL Domain Model

```
!! in-package("RU")
!! in-grammar('user)


var Viz-Spec-Obj             : object-class subtype-of user-object
var    Class-spec-Obj        : object-class subtype-of Viz-Spec-Obj
var       Icon-Attr-Obj      : object-class subtype-of Class-spec-Obj
var       Edit-Attr-Obj      : object-class subtype-of Class-spec-Obj

var Class-Specs     : map(Viz-Spec-Obj, seq(Class-Spec-Obj)) = {||}

var Class-Name      : map(Class-Spec-Obj, symbol) = {||}
var Icon-Attributes : map(Class-Spec-Obj, seq(Icon-Attr-Obj)) = {||}
var Edit-Attributes : map(Class-Spec-Obj, seq(Edit-Attr-Obj))  = {||}


var Edit-attr-name  : map(Edit-Attr-Obj, symbol) = {||}
var Edit-attr-type  : map(Edit-Attr-Obj, symbol) = {||}

var Icon-attr-name  : map(Icon-Attr-Obj, symbol) = {||}
var Icon-attr-val   : map(Icon-Attr-Obj, any-type) = {||}



%% Icon attribute definitions

%var Icon-Active?              : map(Viz-Spec-Obj, symbol) = {||}
%var Icon-Shape                : map(Viz-Spec-Obj, symbol)  = {||}
%var Icon-Size-Factor          : map(Viz-Spec-Obj, real)    = {||}
%var Icon-Height-Width-Ratio   : map(Viz-Spec-Obj, real)    = {||}
%var Icon-Label-Function       : map(Viz-Spec-Obj, symbol)  = {||}
%var Icon-Clip-Icon-Label?     : map(Viz-Spec-Obj, symbol) = {||}
%var Icon-Mouse-Sensitive?     : map(Viz-Spec-Obj, symbol) = {||}
```

D.2 VSL Grammar

```
!! in-package("RU")
!! in-grammar('syntax)

Grammar VIZ

   no-patterns

   start-classes Viz-Spec-Obj
   file-classes  Viz-Spec-Obj


   Productions

   Viz-Spec-Obj ::=
      ["visual" "specs" "for" name "are" [Class-Specs * ";"]]
   builds Viz-Spec-Obj,

   Class-Spec-Obj   ::=
      ["attributes" "for" Class-name "are"
            "icon" ":" [Icon-Attributes * ";"]
            "edit" ":" [Edit-Attributes * ";"]
         "end"
      ]
   builds Class-Spec-Obj,


   Icon-Attr-Obj ::=
      [icon-attr-name "=" icon-attr-val]
   builds Icon-Attr-Obj,


   Edit-Attr-Obj ::=
     [edit-attr-name ":" edit-attr-type]
   builds Edit-Attr-Obj

   symbol-start-chars
      "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ:"

   symbol-continue-chars
      "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ1234567890-?:"

end
```

D.3 VSL Description for the CIRCUITS Domain

```
Visual Specs for Circuits are

   attributes for And-Gate-Obj are
      Icon :
         icon-type = ellipse;
         active?   = true;
         size-factor = 1.1;
         height-width-ratio = 1.0;
         label = class-and-name;
         clip-icon-label? = false;
         mouse-sensitive? = true
      Edit :
         name : symbol;
         delay : integer;
         manufacturer : string;
         mil-spec? : boolean;
         power-level : real
   end;

   attributes for Or-Gate-Obj are
      Icon :
         icon-type = ellipse;
         active?   = true;
         size-factor = 1.1;
         height-width-ratio = 0.95;
         label = class-and-name;
         clip-icon-label? = false;
         mouse-sensitive? = true
      Edit :
         name : symbol;
         delay : integer;
         manufacturer : string;
         mil-spec? : boolean;
         power-level : real
   end;

   attributes for Not-Gate-Obj are
      Icon :
         icon-type = ellipse;
         active?   = true;
         size-factor = 1.1;
         height-width-ratio = 1.3;
         label = class-and-name;
         clip-icon-label? = false;
         mouse-sensitive? = true
      Edit :
         name : symbol;
         delay : integer;
         manufacturer : string;
         mil-spec? : boolean;
         power-level : real
   end;

   attributes for Nor-Gate-Obj are
   Icon :
      icon-type = diamond;
      active?   = true;
      size-factor = 1.1;
      height-width-ratio = 1.7;
      label = class-and-name;
      clip-icon-label? = false;
      mouse-sensitive? = true
   Edit :
      name : symbol;
      delay : integer;
      manufacturer : string;
      mil-spec? : boolean;
      power-level : real
end;

attributes for Nand-Gate-Obj are
   Icon :
      icon-type = diamond;
      active?   = true;
      size-factor = 1.1;
      height-width-ratio = 1.7;
      label = class-and-name;
      clip-icon-label? = false;
      mouse-sensitive? = true
   Edit :
      name : symbol;
      delay : integer;
      manufacturer : string;
      mil-spec? : boolean;
      power-level : real
end;

attributes for Mux-Obj are
   Icon :
      icon-type = diamond;
      active?   = true;
      size-factor = 1.1;
      height-width-ratio = 1.7;
      label = class-and-name;
      clip-icon-label? = false;
      mouse-sensitive? = true
   Edit :
      name : symbol;
      delay : integer;
      manufacturer : string;
      mil-spec? : boolean;
      power-level : real
end;

attributes for JK-Flip-Flop-Obj are
   Icon :
      icon-type = diamond;
      active?   = true;
      size-factor = 0.9;
      height-width-ratio = 1.1;
```

```
        label = class-and-name;
        clip-icon-label? = false;
        mouse-sensitive? = true
     Edit :
        name : symbol;
        delay : integer;
        set-up-delay : integer;
        hold-delay : integer;
        manufacturer : string;
        mil-spec? : boolean;
        power-level : real
end;


attributes for Half-Adder-Obj are
     Icon :
        icon-type = box;
        active?   = true;
        size-factor = 1.0;
        height-width-ratio = 1.3;
        label = class-and-name;
        clip-icon-label? = false;
        mouse-sensitive? = true
     Edit :
        name : symbol;
        delay : integer;
        manufacturer : string;
        mil-spec? : boolean;
        power-level : real
end;

attributes for Decoder-Obj are
     Icon :
        icon-type = box;
        active?   = true;
        size-factor = 1.0;
        height-width-ratio = 0.6;
        label = class-and-name;
        clip-icon-label? = false;
        mouse-sensitive? = true
     Edit :
        name : symbol;
        delay : integer;
        manufacturer : string;
        mil-spec? : boolean;
        power-level : real
  end;

attributes for Counter-Obj are
     Icon :
        icon-type = box;
        active?   = true;
        size-factor = 1.0;
        height-width-ratio = 0.7;
        label = class-and-name;
        clip-icon-label? = false;
```

```
      mouse-sensitive? = true
   Edit :
     name : symbol;
     count : integer;
     delay : integer;
     manufacturer : string;
     mil-spec? : boolean;
     power-level : real
end;

attributes for Led-Obj are
   Icon :
      icon-type = box;
      active?   = true;
      size-factor = 1.1;
      height-width-ratio = 1.0;
      label = class-and-name;
      clip-icon-label? = false;
      mouse-sensitive? = true
   Edit :
     name : symbol;
     manufacturer : string;
     color : symbol
end;

attributes for Switch-Obj are
   Icon :
      icon-type = ellipse;
      active?   = true;
      size-factor = 1.0;
      height-width-ratio = 1.0;
      label = class-and-name;
      clip-icon-label? = false;
      mouse-sensitive? = true
   Edit :
      name : symbol;
      debounced : boolean;
      manufacturer : string;
      delay : integer;
      position : symbol

end
```

Appendix E. REFINE *Source Code for AVSI*

The REFINE source code for AVSI may be obtained, upon request, from:

Maj Paul Bailor
AFIT/ENG
2950 P Street
Wright-Patterson AFB, OH 45433-7765

(513)255-9263
DSN 785-9263
email: pbailor@afit.af.mil

Bibliography

1. Ambler, Allen L. and Margaret M. Burnett. "Influence of Visual Technology on the Evolution of Language Environments," *IEEE Computer*, 9–22 (October 1989).

2. Anderson, Captain Cynthia G. *Creating and Manipulating Formalized Software Architectures to Support a Domain-Oriented Application Composition System.* MS thesis, AFIT/GCS/ENG/92D-01, School of Engineering, Air Force Institute of Technology(AU), Wright-Patterson AFB, OH, December 1992.

3. Arefi, Farahangiz and others. "Automatically Generating Visual Syntax-Directed Editors," *Communications of the ACM*, 349–360 (March 1990).

4. ASD/RWWW. *Joint Modeling and Simulation System (J-MASS): System Concept Document.* Technical Report, CROSSBOW-S Architecture Technical Working Group, November 1991.

5. Bailor, Paul D. and others. "Formalization and Visualization of Domain-Specific Software Architectures," *AAAI-92 Workshop an Automated Software Design, AAAI Conference*. 6 – 11. 1992.

6. Barstow, David R. "Domain-Specific Automatic Programming," *IEEE Transactions on Software Engineering*, *11*:1321– 1326 (November 1985).

7. Batory, Don and Sean O'Malley. *The Design and Implementation of Hierarchical Software Systems with Reusable Components.* Technical Report TR-91-22, Austin, Texas: University of Texas, January 1992.

8. Booch, Grady. *Object-Oriented Design: With Applications.* Redwood City, California: Benjamin/Cummings, 1991.

9. Boom, Mary and Brad Mallare. *Formalization and Transformation of Informal Analysis Models Into Executable* REFINE *Specifications.* MS thesis, AFIT/GCS/ENG/92D, School of Engineering, Air Force Institute of Technology(AU), Wright-Patterson AFB, OH, December 1992.

10. Brown, Gretchen P. and others. "Program Visualization: Graphical Support for Software Development," *IEEE Computer*, 27–35 (August 1985).

11. Chang, Shi-Kuo. "Principles of Visual Languages." *Visual Programming Systems* edited by Shi-Kuo Chang, 1–59, Prentice Hall, 1990.

12. Cypher, Allen and Marilyn Stelzner. "Graphical Knowledge-Based Model Editors." *Intelligent User Interfaces* edited by Joseph W. Sullivan and Sherman W. Tyler, 403–420, ACM Press, 1991.

13. D'Ippolito, Richard and Kenneth Lee. "Modeling Software Systems by Domains." *Tenth Automating Software Design Workshop.* American Association for Artificial Intelligence, April 1992.

14. D'Ippolito, Richard S. "Using Models in Software Engineering." *Proceedings: TRI-Ada '89.* 256–265. 1989.

15. D'Ippolito, Richard S. and Charles P. Plinta. "Software Development Using Models," *ACM Sigsoft Software Engineering Notes* (October 1989).

16. Eades, Peter and Lin Xuemin. "How to Draw a Directed Graph." *IEEE Workshop on Visual Languages*. 13–17. 1989.

17. Eisenstadt, Marc and others. "Visual Knowledge Engineering," *IEEE Transactions on Software Engineering*, *16*:1164–1177 (October 1990).

18. El-Kassas, S. "Visual Languages: Their Definition and Applications in System Development," *Microprocessing and Microprogramming*, *32*:383–391 (1991).

19. Fischer, Charles N. and Richard J. LeBlanc, Jr. *Crafting a Compiler with C.* Redwood City, CA: Benjamin/Cummings Publishing Company, Inc, 1991.

20. Greenspan, Sol J. *Requirements Modeling: A Knowledge Representation Approach to Software Requirements Definition.* PhD dissertation, University of Toronto, Toronto, Ontario, Canada, 1984.

21. Huang, Kuan-Tsae. "Visual Interface Design Systems." *Visual Programming Systems* edited by Shi-Kuo Chang, 60–143. Prentice Hall, 1990.

22. Ichikawa, Tadao and Masakito Hirakawa. "Iconic Programming: Where to Go?," *IEEE Software*, 63–68 (November 1990).

23. Iscoe, Neil. "Domain Modeling – Evolving Research." *Proceedings of the Sixth Annual Knowldege-Based Software Engineering Conference.* 300 – 304. 1991.

24. Iscoe, Neil Allen. *Domain-Specific Programming: An Object-Oriented and Knowledge-based Approach to Specification and Generation.* PhD dissertation, The University of Texas at Austin, Austin Texas, 1990.

25. Kang, Kyo C. and others. *Feature-Oriented Domain Analysis (FODA) Feasibility Study.* Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, November 1990 (AD-A235 785).

26. Korth, Henry F. and Abraham Silberschatz. *Database System Concepts, 2nd edition.* New York, NY: McGraw-Hill, Inc., 1991.

27. Langloss, Randall K. *Graph-Based Visualization of Formal Specification and Domain Specific Languages.* MS thesis, AFIT/GCS/ENG/91D, School of Engineering, Air Force Institute of Technology(AU), Wright-Patterson AFB, OH, December 1991.

28. Lee, Kenneth J. and others. *Model-Based Software Development (Draft).* Technical Report CMU/SEI-92-SR-00, Software Engineering Institute, December 1991.

29. Lockheed Software Technology Center. *Software User's Manual for the Automatic Programming Technologies For Avionics Software (APTAS) System.* Technical Report, Palo Alto, CA: Lockheed Software Technology Center, June 1991.

30. Lowry, Michael R. "Software Engineering in the Twenty-first Century." *Automating Software Design*, edited by Michael R. Lowry and Robert D. McCartney. 627–654. Menlo Park, CA: AAAI Press, 1991.

31. Miller, James R. and others. "Introduction." *Intelligent User Interfaces* edited by Joseph W. Sullivan and Sherman W. Tyler, 1–10, ACM Press, 1991.

32. Moriconi, Mark and Dwight F. Hare. "Visualizing Program Designs Through Pegasys," *IEEE Computer*, 72–85 (August 1985).

33. Neal, Jeanette G. and Stuart C. Shapiro. "Intelligent Multi-Media Interface Technology." *Intelligent User Interfaces* edited by Joseph W. Sullivan and Sherman W. Tyler, 11–43, ACM Press, 1991.

34. Neighbors, James M. "The Draco Approach to Constructing Software from Reusable Components," *IEEE Transactions on Software Engineering*, *10*:564–574 (September 1984).

35. Peterson, A. Spencer. "Coming to Terms with Software Reuse: A Model-based Approach," *ACM SIGSOFT Software Engineering Notes*, *16*:45–51 (April 1991).

36. Prieto-Díaz, Rubén. "Domain Analysis: An Introduction," *ACM SIGSOFT Software Engineering Notes*, *15*:47–54 (April 1990).

37. Prieto-Díaz, Rubén. "Domain Analysis for Reusability." *Proceedings of the 11th Annual International Computer Software and Application Conference.* 23–29. IEEE Computer Society Press, 1990.

38. Protsko, L. Beth and others. "Towards the Automatic Generation of Software Diagrams," *IEEE Transactions on Software Engineering*, *17*:10–21 (January 1991).

39. Randour, Capt Mary Anne. *Creating and Manipulating a Domain-Specific Formal Object Base to Support a Domain-Oriented Application Composition System.* MS thesis, AFIT/GCS/ENG/92D, School of Engineering, Air Force Institute of Technology(AU), Wright-Patterson AFB, OH, December 1992.

40. Reasoning Systems Inc., Palo Alto, CA. *INTERVISTA^{TM} User's Guide.* For INTERVISTA^{TM} Version 1.0.

41. Reasoning Systems Inc., Palo Alto, CA. *REFINE^{TM} User's Guide.* For REFINE^{TM} Version 3.0.

42. Reasoning Systems, Inc. *DIALECT User's Guide.* Palo Alto, CA, July 1990.

43. Shu, Nan. *Visual Programming.* New York: Van Nostrand Reinhold Company, 1988.

44. Smith, Douglas R. "KIDS - A Knowledge-Based Software Development System." *Automating Software Design*, edited by Michael R. Lowry and Robert D. McCartney. Chapter 19. Menlo Park, CA: AAAI Press/MIT Press, 1991.

45. Sommerville, Ian. *Software Engineering.* New York: Addison-Wesley, 1989.

46. Spicer, Kelly L. *Mapping an Object-Oriented Requirements Analysis to a Design Architecture that Supports Reuse.* MS thesis, AFIT/GCS/ENG/90D, School of Engineering, Air Force Institute of Technology(AU), Wright-Patterson AFB, OH, December 1990.

47. Tamassia, Roberto and others. "Automatic Graph Drawing and Readability of Diagrams," *IEEE Transactions on Systems, Man, and Cybernetics*, *18*:61–79 (January/February 1988).

48. Teague, Alan H. and Henson Graves. *The Graphical System Description Language and Development Environment Version 2.0 (Draft).* Technical Report O/96-10 B/254E, Software Technology Center Lockheed Palo Alto Research Labs, April 1992.

Vita

Second Lieutenant Timothy L. Weide was born November 21, 1959 in Houston, Texas and graduated from Trevor G. Browne High School in Phoenix, Arizona in 1977. He enlisted in the United States Air Force in August 1983 and completed technical training for computer maintenance at Keesler AFB, Mississippi in May, 1984. He spent the years from 1984 through 1988 as a computer maintenance technician for the Joint Surveillance System, Northwest Air Defense Sector, McChord AFB, Washington. In January, 1989 he entered the Airmen Education and Commissioning Program to pursue a Bachelor of Science degree in Computer Science. He graduated, Summa Cum Laude, from Arizona State University in May, 1991. In June, 1991, attended Officer Training School at Lackland AFB, Texas and upon receiving his commission in September, 1991 he entered the Air Force Institute of Technology at Wright-Patterson AFB, Ohio to pursue a Master of Science degree in Computer Engineering.

Permanent address: 255 Morning Sun Dr.
Woodland Park, CO 80863

REPORT DOCUMENTATION PAGE	*Form Approved* *OMB No. 0704-0188*

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE March 1993	3. REPORT TYPE AND DATES COVERED Master's Thesis

4. TITLE AND SUBTITLE DEVELOPMENT OF A VISUAL SYSTEM INTERFACE TO SUPPORT A DOMAIN-ORIENTED APPLICATION COMPOSITION SYSTEM	5. FUNDING NUMBERS
6. AUTHOR(S) Timothy L. Weide, Second Lieutenant, USAF	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Air Force Institute of Technology, WPAFB OH 45433-6583	8. PERFORMING ORGANIZATION REPORT NUMBER AFIT/GCS/ENG/93M-05
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) ASC/RWWW Wright-Patterson AFB, OH 45433-6583	10. SPONSORING/MONITORING AGENCY REPORT NUMBER
11. SUPPLEMENTARY NOTES	
12a. DISTRIBUTION/AVAILABILITY STATEMENT Approved for public release; distribution unlimited	12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

This research designed and prototyped a visual system interface to generate, display, and modify domain-oriented application specifications. A visual system interface, called the Architect Visual System Interface (AVSI), supplements a text-based environment, called Architect, previously developed by two other students. Using canonical formal specifications of domain objects, Architect rapidly composes these specifications into a software application and executes a prototype of that application as a means to demonstrate its correctness before any programming language specific code is generated. This thesis investigates visual techniques for populating, manipulating, viewing, and composing these software application specifications within the formal object base scheme required by Architect. A Visual Specification Language (VSL) was developed to define the visual display characteristics of domain objects. AVSI provides automatic diagram layout, and also produces a textual display in a domain-specific language. The Software Refinery environment, including its graphical interface tool INTERVISTA, was used to develop techniques for visualizing application data and for manipulating the formal object base. AVSI was validated with a well-understood domain, digital logic, and was found to significantly enhance Architect's application composition process.

14. SUBJECT TERMS computers, computer programs, software engineering, visual languages, visual programming systems, specifications, domain-specific languages, domain modeling, application composition systems, software architecture models	15. NUMBER OF PAGES 151
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

NSN 7540-01-280-5500

Standard Form 298 (Rev 2-89)
Prescribed by ANSI Std. Z39-18
298-102