Selecting an Architecture for Use in Simulators

David C. Gross	Robert M. Seltzer	William V. Tucker
The Boeing Company	Naval Air Warfare Center	The Boeing Company
Modeling & Simulation	Training Systems Division	Modeling & Simulation
Huntsville, AL	Orlando, FL	Huntsville, AL

ABSTRACT

The introduction of advanced software engineering technologies has made architecture the central enabling concept for successful development of complex system software such as found in software intensive real-time simulators. Insightful selection and application of a software architecture seems to hold the key to the desirable trinity of software system development: better, cheaper, faster.

However, the promise offered by software system architecture raises the stakes associated with selecting one. Selecting the wrong architectural approach can jeopardize the entire project by increasing costs and delays. The barriers to selection are further raised when one considers the implications of reuse -- the architecture chosen is going to be with us for a long time, so it had better be good! Furthermore, collaborative development efforts involving multiple contractor teams and contractor/government teams raise barriers because the number stakeholders increase.

This paper explores the difficulties of selecting an appropriate architecture for use in real-time simulators, and proposes ways of overcoming them. We present a method for classifying a candidate architecture by partitioning representative architectures and noting examples of each class in fielded real-time simulators. We present an approach to overcoming the obstacles to selecting an architecture, with particular emphasis on communicating to stakeholders. The cornerstone of this approach is to gain agreement on the characteristics by which architectures should be judged and selected. Our observations are based on our collective experience on the Navy/STARS demonstration project, which involved joint government contractor teams. This project required the selection of an architecture intended for use in active system development for the next decade.

ABOUT THE AUTHORS

Mr. David C. Gross is a software systems engineer with Modeling and Simulation Technology within the Boeing Defense & Space Group. He has conducted applied research in the areas of software development process, software reuse/re-engineering, and software quality, as they apply to simulation. He is currently involved in applied research in advanced simulation technologies such as knowledge-based simulation, visualization, and graphical user interfaces. Mr. Gross holds a Bachelor of Science in Computer Science/Engineering from Auburn University and a Master of Operations Research at the University of Alabama at Huntsville. Mr. Gross is a doctoral student at the University of Central Florida. He can be reached at gross@mst.hv.boeing.com

Mr. Robert Seltzer obtained his B.S. in Aerospace Engineering from Polytechnic University in New York in 1983 and a Masters of Science Degree in Aeronautics and Astronautics from Purdue University in 1987. Mr. Seltzer spent eleven years as a Flying Qualities/Flight Dynamist Engineer at the Naval Air Warfare Center Aircraft Division, Warminster, Pennsylvania. His projects included real-time pilot in the loop and non-real-time aircraft research & design application simulation for air vehicles that included the A-6, F-14A, and X-31A. In 1994, he was assigned to the Naval Air Warfare Center Training Systems Division (NAWCTSD), Modeling and Simulation Integration Branch. At NAWCTSD, he has served as the flight dynamics domain analyst on the Navy STARS program, and worked to implement of software reuse within the Navy. Mr. Seltzer also holds a Master of Science degree in Engineering Management from National Technological University. He is a member of AIAA.

Mr. William V. Tucker is the Modeling & Simulation Technology Manager for Boeing Defense & Space Group, Concepts and Analyses organization. He has managed various trainer programs, including US, UK, and RSAF E-3, KC-135 and the Modular Simulator Design Program. He chairs the DIS exercise management and feedback subgroup. He holds a Bachelor of Science degree in Electrical Engineering from Wichita State University in Electrical Engineering. He can be reached at tucker@mst.hv.boeing.com

Selecting an Architecture for Use in Simulators

David C. Gross	Robert M. Seltzer	William V. Tucker
The Boeing Company	Naval Air Warfare Center	The Boeing Company
Modeling & Simulation	Training Systems Division	Modeling & Simulation
Huntsville, AL	Orlando, FL	Huntsville, AL

INTRODUCTION

The introduction of advanced software engineering technologies has made architecture the central enabling concept for successful development of complex system software such as found in software intensive real-time simulators. Insightful selection and application of a software architecture seem to hold the key to the desirable trinity of software system development: better, cheaper, faster. Architecture centric development efforts which are lead to better systems, because they promote requirements consistency and traceability. They lead to less costly systems because strong interface definition and control reduces the amount of code needed and communicates a clearer picture of the design space. Finally, they lead to faster system development because architecturally compliant standard components can be constructed and reused more easily. Indeed, one entire reuse scheme (product line development) depends on the assertion that an architecture spanning a product line may be defined that supports development of a variety of . systems as well as its continued maintenance, evolution, and future growth. The importance of architecture can be seen simply by considering major simulation initiatives such as High Level Architecture (HLA).

Choosing an appropriate software architecture requires that the technical team understand and address the following issues as part of a front-end architectural design analysis.

(1) The attributes of a software architecture.

(2) How these attributes are realized or not realized in the different classes of software architecture designs.

(3) How these architecture attributes impact the software system design.

(4) How to evaluate the candidate architectures relative to the design project.

(5) The project goals and requirements along with the hardware system requirements/ constraints.

Preeminence of Software in Simulation Systems.

Recent experience years has shown that the escalating cost of software development, procurement, and life cycle software maintenance has now reached the point where it far exceeds the cost of the hardware elements of a training system (see reference [8], pages 1-4). In the past, it was difficult to acquire computational hardware capable of supporting the needs of simulations. Graphical display systems were not capable of providing rapidly evolving graphics, processors were not capable of supporting real time context switches, interface devices capable of trapping inputs and modifying them in real-time were not readily available. Simulators used analog and eventually hybrid computers to meet these needs. This is no longer the case. Now, any number of manufacturers provide off the shelf components capable of meeting the demands of simulation systems. Software has become the critical component in a simulation system. This requires simulation designers to focus their efforts on adequately representing models and their interactions, rather than simply on getting a simulation to run. This trend accelerates with the introduction of truly open systems: products such as the open graphics library and languages such as Java.

Architecture as a Software Design Technique

Building architectures for software systems is a high level design process. Like many such activities, it is more art than science. At this point the software engineering community has not reached consensus as to what an architecture is, or what attributes make one better than another. For example, There is no common, universally accepted definition of architecture [7] . Never-the-less, architecture is at the center of many initiatives striving for ways of developing software intensive systems better, cheaper, and faster.

We adopt the definition of architecture as the simultaneous capture of two aspects of the system design: how the system is decomposed into components, and how those components interact. We view architecture as the *product* of the structural modeling *process*. The architecture specifies the following.

(1) The kinds of entities that will exist in the design

(2) How the real world is mapped into the software entities

(3) The communication between entities

If the architecture is implemented in the Ada software language, these issues might be re-phrased as follows.

(1) How do you package?

(2) What's in a package?

(3) How do packages communicate?

Certainly software architectures may be implemented in many different languages. Ideally, the needs of the problem will drive the choice of architecture, which in turn would influence the language choice. It is possible to answer these questions using all of the reasonable higher order languages (FORTRAN, Jovial, C, C++, and Ada) choices for real-time simulators. For example, in the C++ language, the foregoing questions would relate to the object class scheme. However, Ada offers a uniquely rich set of mechanisms for capturing architectures: types, functions, procedures, packages, tasks. The primary effect of Ada 95 is to further enrich the possibilities.

This definition of an architecture has been more formally stated. [9]

Software Architecture =

[Elements +

Forms +

Rationale/Constraints]

Importance of Architecture

Software architecture is important because it represents and defines the framework for the software code implementation. The architecture chosen will be the primary software design element impacting the maintainability, verifiability, expandability, flexibility, interoperability, and portability of the system software.

Over riding all of this is the fact that an architectural description makes a complex system intellectually tractable by characterizing it at a high level of abstraction. In particular, architectural design exposes top-level design decisions and lets the designer reason about how to satisfy system requirements as he assigns functionality to design elements. Additionally, architectural design lets designers exploit recurring organizational patterns. Such patterns, or architectural styles, ease the design process by providing standard solutions for certain classes of problems, by supporting the reuse of underlying implementations, and by permitting specialized analyses [6].

The complexity of software intensive systems has increased to the point where the software design problem goes beyond the algorithms and data structures of the computations [11]. Designing and specifying the overall system structure or architecture is now paramount. As defined in reference [11] some of the major architectural issues and properties describing an architecture include: gross organization and global control structure,; protocols for communication, synchronization, and data access; assignment of functionality to design elements; physical distribution; composition of design elements; scaling and performance.

As Mary Shaw explains [10], "the choice of architectural style and its associated notations can have far reaching consequences. The choice affects not only the system's description and its decomposition into components, but also its functionality and performance."

From our perspective, the architecture provides the framework for allocation of requirements, and thereby represents the end of the requirements analysis process -- at least for this iteration of the requirements. The architecture provides the context within which the design is implemented, and thereby represents the beginning of the design. Figure 1 illustrates this concept.

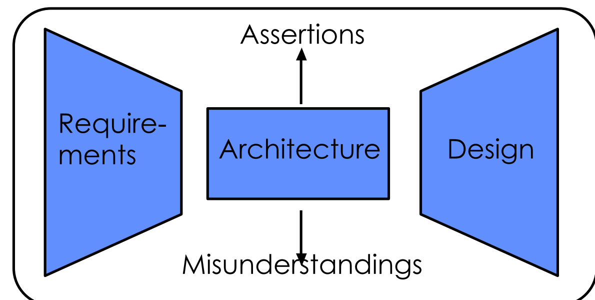

Figure 1: Architecture in Design Process

Good architectures serve as a way to communicate the design, to make assertions about the system. These assertions form a common frame of reference for everyone working on the project. Of course, poorly conceived architectures tend to create misunderstandings and confusion within the development team. If the architecture succeeds, then the team can look forward to:

(a) Doing less work

(b) Building better products

(c) Finishing products quicker

(d) Creating entry gates for this business

Other research [3] has clearly shown that architecture, as a central point in the design, has the ability to directly impact the "ilities" sought after by the software design: efficiency, reusability, maintainability, verifiability, expandability, flexibility, interoperability, and portability.

ADOPTION CHALLENGES

Given the relatively widespread acceptance of architecture as a software design technique, why do so few projects have one and why is it difficult for organizations to adopt one? In our experience, particularly on the Navy/STARS Demonstration Project, we have discovered a number of reasons for this difficulty.

Ignorance of Fundamentals.

The Capability Maturity Model [3] developed by the Software Engineering Institute is having great influence, among other reasons, because it recognizes that software development organizations *mature*. There is no such thing as instant maturity. New organizations have to overcome their institutional ignorance; they have to learn how to work together, to use in place processes, to define new processes, to understand what the customer needs and so forth. The organization may need to learn and adopt different techniques and innovations. For example, we have seen projects with the majority of the software development team having little or no experience with Ada. Since the project required Ada, they had to climb this learning curve.

No Experience with Architecture.

Inexperienced software engineering teams that have not been exposed to the various styles and variants of potential candidate architectures will stick with familiar approaches -- we typically use what we know. This is not necessarily bad since the adoption of an unfamiliar architecture carries with it the burden of a learning curve, a burden a project may not be able to afford in up-front schedule time. However, selecting a "comfortable" architecture can be a serious detriment if the selection is performed without objectively evaluating the potential merits of other candidate architectural solutions. On many simulator projects, most of the software team will be experienced only with executive/subprogram/ global data architectures. Even worse from a project perspective is *denying* the architecture decision. Some software designers do not recognize architecture selection as a critical decision in the development process. On the contrary, it is a decision of prime importance, and failing to make explicitly make the choice is to decide by default.

Not-Invented-Here.

Every new technology must overcome the hurdle of people's natural resistance to change. This is particularly painful when a new organization is assembled from the pieces of several different organizations. This will increasingly be the case in today's environment of highly integrated contractor/government organizations. We have experienced resistance to adopting an architecture from some team members because it originated with a particular contributing organizations to a project.

Inability to Commit.

Because the selection of the architecture in many ways represents the essence of the software code to be developed, it is very important for the *entire* team to "buy in" to the architecture selection. It is frequently difficult to get the team lined up behind an approach. One surprising cause of this we have seen is the presence of many bright and creative people, who always want to find just a little better solution. We have also seen this problem on projects with a large number of contributing organizations. This results in weaker management structure, and a loss of a sense of "who's in charge". Closely related is the problem of mis-timing the decision to commit. Whereas an architecture permeates through out the decision process, and the inclination is to select and adopt one as soon as possible, one must resist until the project has matured to the point where it is able to meaningful use the decision. This is particularly important for new software development organizations. If trade studies affecting architecture selection were conducted so early that the results were frequently not used or really understood.

Lack of Institutional Memory

Sometimes the inability of an organization to select an architecture arises from a lack of institutional memory. The most frequent cause of this problem is high turnover rates. If the people who made the decision are gone, then there is little corporate memory of the decision. A second cause is frequent changes in the organization's charter. An organization continually shifting directions is tempted to revisit all previous decisions. We have experienced both of these problems on various projects.

Immaturity of Architecture as a Design Technique

Architectural design as a science is still maturing and the "hard" engineering data needed to correlate

architectural properties with application specific software performance is still being sought. Architectural properties of interest include global control structures, message passing, synchronization and data access, assignment of functionality to design elements, and so forth. Application software performance issues include memory, time efficiency, fault-tolerance, and reusability. Despite this lack of maturity, architecture decisions are generally made early in the lifecycle of a software system, and as such become very hard to change. All of this makes the selection difficult.

Immaturity of Architecture as a Product

Adopting an architecture is not like adopting a language. There are no well established standards for what constitutes a well-defined and documented architecture. Architectures are simply not matured to the point where they are off-the-shelf products. Generally, there are no manuals, books, training courses, or legacy code applying the architecture. The demonstration project was fortunate to have legacy implementations of its adopted architecture from which to leverage.

Complications Raised by Reuse

As mentioned, one of the greatest motivators for committing to an architecture, is the potential benefits of: leveraging architectural components for reuse. However we do not in fact see much progress toward building systems from reusable parts. Why? One answer is that the assumptions about their intended environment are implicit and either don't match the actual environment or conflict with other parts [5]. Stated another way, not just any architecture lends itself to reuse.

If software is to be reusable, then it must anticipate the needs of systems which have not even been envisioned. Van Der Linden [11] identifies three requirements that an architecture must met to be "future proof" (i.e., reusable) as follows.

(1) Cover a family of products.

(2) Accommodate integration of new services,

(3) Allow exchange of hardware and controlling software.

These requirements have recently led to development and adoption of Domain Specific Software Architectures (DSSA). DSSA's provide an organizational structure specifically tailored to a family of applications such as avionics. command and control, or air vehicle training systems. DSSA's strengths lie in providing a context within which significant reuse of software assets can occur, including source code, test cases, support software, documentation, and so forth. This significant reuse potential becomes realizable using DSSA's by specializing the architecture to the domain, making it possible to increase the power of structures [6].

All of the foregoing problems create challenges to adopting architectures, challenges which are not unknown to the software research community. The introduction to *IEEE Software's* special issue [6] on architecture last year points out the need for a "sound mathematical basis to explore the potential benefits and limitations of proposed architectures and to ultimately prove a system's properties on the basis of the properties of the individual components." . While we wait for the development of such methods, design teams have to make architectural designs today. The objective of this paper is to give them a process that gets them some of the help they need to make the right decision.

CLASSIFICATION OF ARCHITECTURES

It will be useful to provide a classification of architectural styles. Given our adopted definition, the classification of architecture is delineated by how items, or "chunks" of software, are decomposed and how they interact. It follows that three general classes of architecture design styles falls out. They are: (a) Flat, (b) Rigorously Stratified, and (c) Cooperative Objects. Of course, most real systems use some hybrid of these styles. Each approach has its advocates and each is represented by successfully fielded devices.

Flat

Flat architectures, illustrated in Figure 2, typically partition the system into two levels: an executive and its servants. The executive handles responsibility for interfacing to the operating system and (generally) hardware, and invokes the servant subprogram according to a schedule table. It is generally the most natural model supported by languages with limited abstraction such as FORTRAN. Generally, such architectures are implemented with globally available data structures (e.g., DATAPOOL). Flat architectures have a number of advantages including (a) high efficiency, (b) debugging ease, and (c) ease of load balancing. The chief disadvantage is their lack of information hiding, which leads to various problems associated with responsibility for data.

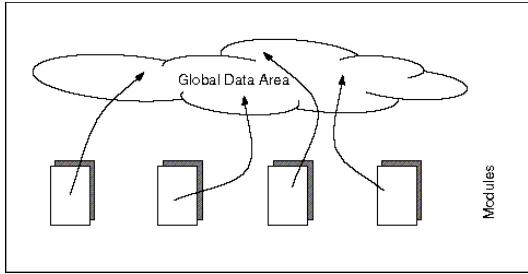

Figure 2: Class 1 Structural Hierarchy

Rigorously Stratified

Rigorously stratified architectures, illustrated in Figure 3, partition the system into an arbitrary number of layers. Each layer has a specific responsibility. Communication via entry point invocation and data exchange is typically tightly constrained. The Air Force's Structural Model falls into this class [1]. This is a natural architecture for implementing systems in highly abstracted languages such as Ada 83. Such architectures have the advantages of strong information hiding, uniformity, and localization. These advantages lead to a preference for this architecture for distributed development teams. The prime disadvantage of this approach is reduced computational efficiency, although measurements suggest that this penalty is actually very small. Poorly though out implementations can create difficulties in system verification and validation.

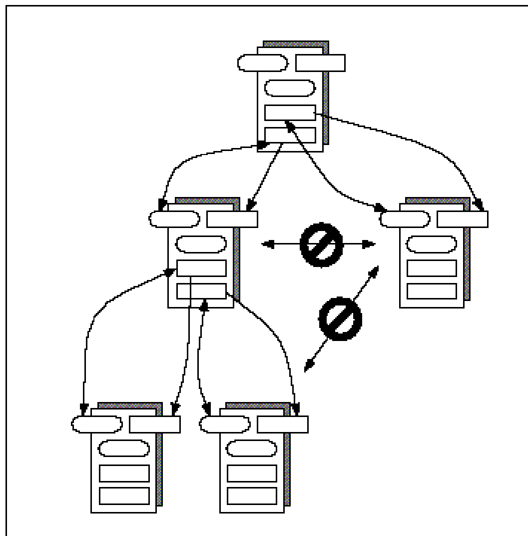

Figure 3: Class 2 Structural Hierarchy

Cooperating Objects

Cooperating objects, illustrated in Figure 4, are the preferred approach of object -oriented "purists". In this view, every software item is an independent actor or entity Cooperation between objects occurs through message passing -- there is no executive to dictate behavior. This architectural class has proved particularly well suited for very high level of abstraction. For example, Distributed Interactive Simulation and HLA fall into this class. This is a natural architecture for implementing in highly object-oriented languages such as SmallTalk, and can be implemented in languages such as C++ and Ada 95. The primary advantage of this approach lays in the unique feature of inheritance. In our figure, inheritance might be best represented as a third dimension, reaching into the page. Inheritance provides for the rapid creation of new objects from the parts of existing ones thus providing an indirect approach for reuse. The disadvantage of the approach lay with its relatively large computational inefficiencies.

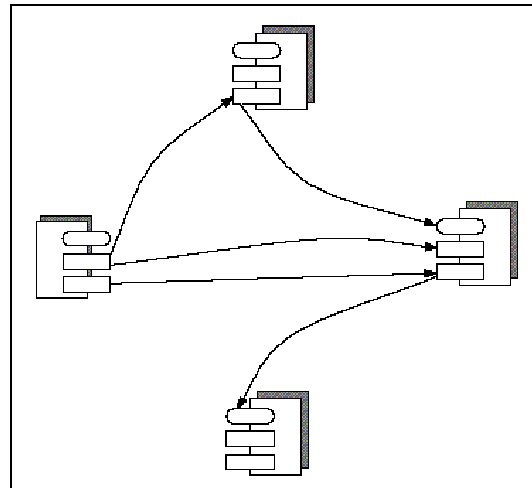

Figure 4: Class 3 Structural Hierarchy

SELECTING ARCHITECTURES

Selecting an architecture is really an exercise in multi-attribute decision making. The best way to make such decisions is by exercising a well defined process. A process is a n-tuple of people, tools , standards, and procedures. The most important part of any process are the people -- obviously the software development team, but also other less obvious stakeholders in this decision. The failure to include all of the stakeholders in the process is the quickest way to fail.

The following is our suggested approach to making the architecture decision. First, we present an example architecture to facilitate the discussion.

The Navy/STARS Architecture

The Navy/STARS team selected and applied an architecture in developing the Air Vehicle Training Systems (AVTS) domain called, "Domain Architecture for Reuse in Training Systems" or DARTS . The objective of the Navy/STARS Demonstration Project was

to construct a domain of reusable assets that support a product line, or family of related devices. The AVTS domain is a family of air vehicle training devices that provides the simulation, stimulation, and/or emulation of all the components and systems for a real-time air vehicle training systems. This domain encompasses the systems necessary to provide training devices that a trainee uses to become familiar with the operator station configuration and/or flight characteristics of the application air vehicle, gain proficiency in executing normal procedures, recognizing malfunctions/ abnormal indications and executing the corresponding standard/emergency procedures, and executing mission procedures. Since the architecture was intended for use in a complex domain with a project lifecycle in decades, the architectural decision was particularly important.

Interface Scheme. The interface scheme for DARTS is shown in Figure 5. DARTS contains five major structural elements: the Virtual Network (VNET), the Module Executive(s) the Segment Executives, the Subsystem Controllers, and the Components. The VNET provides for the only communication path between segments. The Module Executive provide for the housekeeping of executing a task on a specific processor. The Segment Executives provide for the execution of assigned subsystems and message traffic to and from the VNET. The Subsystem Executives provide for executing their assigned Components and managing data states. The Component simulate discrete entities within the simulation.

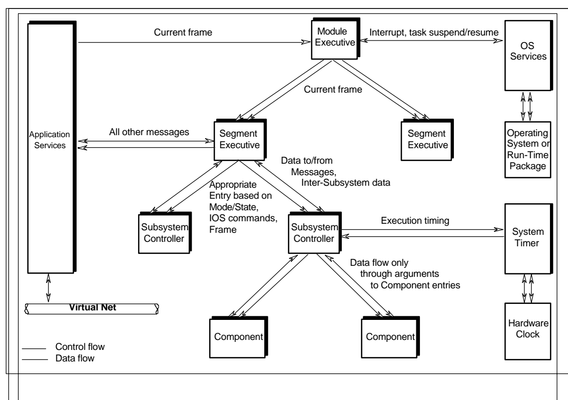

Figure 5: The DARTS Structural Hierarchy

Partitioning Scheme. The AVTS domain divides into thirteen subdomains. Twelve of these provide general simulation functionality: environment, radar, flight station, flight controls, flight dynamics, propulsion, weapons, instructor/operator station, navigation/ communication, electronic warfare, visual, physical cues. The thirteenth subdomain provides various executive services and utilizes for the others. A subdomain purpose is to provide the required segment, subsystems, and components for a particular simulator's needs. The organization of a subdomain into subsystems and components was the result of engineering analysis within the context of the subdomain: some functional analysis, some object-oriented analysis, and mostly hybrid.

A complete discussion of DARTS, its advantages, and disadvantages can be found in reference [5].

Selection Process

Software development is a complex process which involves many decisions like architecture selection whose outcome is not self evident. Initial contemplation of such decisions typically suggests several alternative solutions, each of which appear to satisfy some set of minimal requirements, but exactly which solution provides the maximum benefit leverage is not clear. Therefore, the choice between them depends on quantified estimates of their benefit to the domain. In these cases, our preferred practice is to conduct a trade study process. The following discussion outlines our suggested process and means of documenting the results. Figure 6 illustrates the Architecture Selection Process.

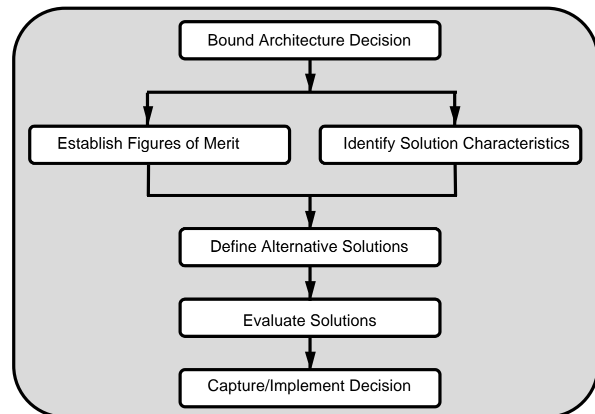

Figure 6: An Architecture Selection Process

Solution Characteristics

Solution Characteristics should capture the attributes that any viable solution for this decision must address, therefore they define what really constitutes an architecture. If there are minimal requirements for the solution, they should be expressed in the solution characteristics. Each solution characteristic should be defined and placed in context.

The simplest answer, according to our definition of an architecture, is that the proposed solution must provide a partitioning scheme and an interfacing scheme. These are frequently provided in the form of some kind of data/control flow diagram (recall Figure 5), and a set of assertions about functional or object oriented analysis. Does this really constitute a sufficiently defined software architecture?

Most would say “no”. Certainly an architecture must provide an idea of the control and data flow requirements within a system but these are not the only uses of the architecture. In fact, the architecture provides onewith many “views” of the software system. Kruchten [9] suggests that an architecture provides “4+1” views of the system as follows.

(1) The logical view, describing the system’s object model or entity-relationship.

(2) The process view, describing the system’s concurrent and synchronization.

(3) The physical view, describing the system’s mapping onto hardware.

(4) The development view., describing the static organization.

The “+1” view is captured in a set of scenarios. [9]

The advantage of the architecture as multiple views approach is that it directly expressed the idea that the role of architecture is to communicate about the system. All of the views are really bundles or classes of perspectives. We assert that any architecture than can provide these views of the system is an adequate architecture. We illustration these views in Figures 7, 8, 9, and 10 of the DARTS architecture as a demonstration of their use.

Figures of Merit

Figures of Merit should define and constrain the basis for deciding between alternative solutions. Figures of merit must be clear and complete so as to avoid confusion over their meaning or a sense that they do not fully express the problem of selecting a solution. The purpose in utilizing a variety of figures of merit is to balance the competing demands on the solution. The figures of merit are purposely scored and weighted to avoid the problem of selecting a solution that solves one aspects of the problem, without addressing (or perhaps damaging) other interests. As such, the trade study is a systems-oriented approach to making domain decisions.

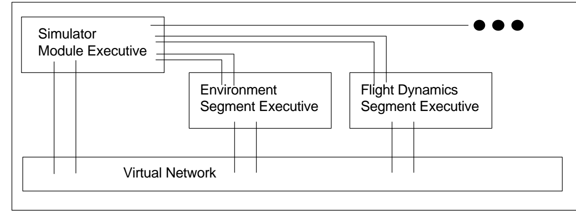

Figure 7: An Example Logical View of AVTS

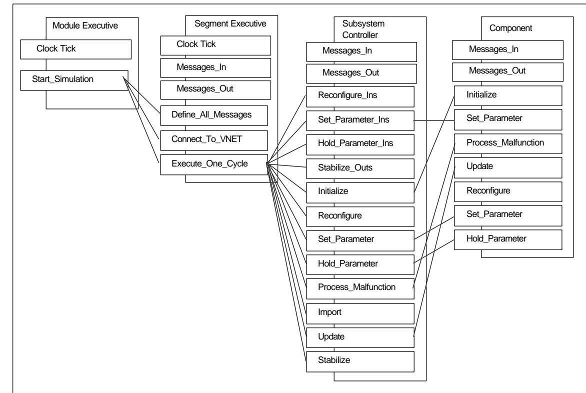

Figure 8: An Example Process View of AVTS

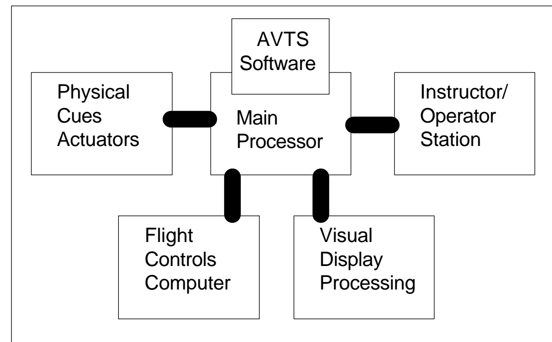

Figure 9: An Example Physical View of AVTS

Flight Dynamics Segment
fd_segment_executive_ads.a
fd_segment_executive_adb.a
fd_execute_one_cycle_ada.a
Equations of Motion Subsystem
fd_eom_types_constants_ads.a
fd_equations_of_motion_ads.a
fd_equations_of_motion_adb.a
Body Dynamics Component
fd_body_dynamics_ads.a
fd_body_dynamics_adb.a
fd_translation_dynamics_ada.a and so forth.

Figure 10: An Example Development View of AVTS

We must specify a scoring mechanism for each figure of merit. The scoring provides (a) a means for ranking alternative solutions, and (b) a means for weighting each figure of merit against all other figures of merit. Therefore, the figures of merit should specify the share of total score that each category of figures of merit represents. The purpose of this is to show that, despite the varying numbers of specific figure of merit in any given category, the categories have a weight that expresses their relative importance.

Obvious candidates for sources of figures of merit include the following.

(1) Operational Requirements.

(2) Contractual Requirements.

(3) Best Accepted Commercial Practices.

(4) Cost of Implementation.

(5) Producibility.

(6) Supportability.

(7) Lifecycle Cost.

Many of these can be conveniently scored as discrete, objective criteria. For example, one contractual requirement for the demonstration project was the ability to incorporate legacy FORTRAN subroutines. The ability of DARTS to meet this requirement was completely objective. On the other hand many figures of merit are a lot softer. Figure 11 illustrates three sets of hard to measure requirements for software, and some of the relationships between them.

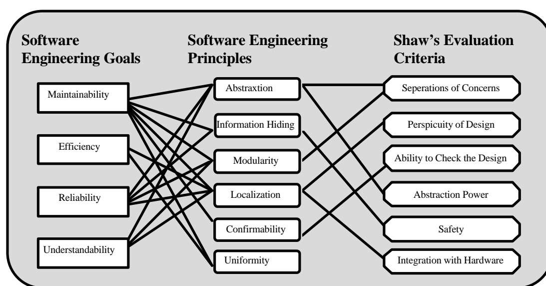

Figure 11: Hard to Measure Figures of Merit

While the scoring, even for these hard to measure things must be meaningful, the most important thing is that we get buy in from the entire team as to how the figures of merit will be scored.

We recommend that the figures of merit be based on the principles of software engineering [2]. These principles are qualitative characteristics within the software that help measure progress toward quality. While the principles apply in every phase of the lifecycle, they seem particularly appropriate for architectural design. The following discussion reviews each principle, and scores DARTS using a simple high | medium | low scheme.

Abstraction. This principle deals with how one views the system. The essence of abstraction is to extract essential properties while omitting nonessential details. DARTS demonstrates strong abstraction in its breakdown of subdomains which indicates the essential properties of the AVTS domain. Score: *medium*.

Information Hiding. This principle is that details in certain parts of the system should not affect other parts of the system. Information hiding therefore conceals how an object or operation is implemented. DARTS practices information hiding by restricting control and data flows to, up, and down the segment/ subsystem/component hierarchy. Score: *high*.

Modularity. This principle is the purposeful structuring of the physical architecture of the software system. Modularity deals with how the structure of an object can make the attainment of some purpose easier. DARTS enforces modularity through the requirement for segments, subsystems, and components. Score: *high*.

Localization. This principle helps to create modules that are loosely coupled to the outside world and cohesively strong internally. It is concerned with physical proximity of software components. The subdomain partitioning of DARTS is a functional grouping. Score: *medium*.

Uniformity. This principle simply means that the software utilizes a consistent notation and is free of any unnecessary differences. DARTS requires that each software element match a predefined invocation specification, manipulate messages in exactly the same way, and that components do not retain state data. Score: *high*.

Confirmability. This principle implies that the system is designed so that it can be readily tested. It increases the credibility of the design by building validation into the code. It is possible to build and test DARTS components without the rest of the system being present. Score: *low*.

Alternative Solutions Alternative Solutions bound the set of various alternative, acceptable solutions for the decision. Any patterns in the way solutions were generated (for example, major and minor variations on themes) should be discussed. A explanation of how each alternative solution meets each solution characteristics

must be given. Every alternative that fits the solution characteristics should be considered.

Each alternative is scored against the figures of merit. There are any number of techniques for combining these scores, however, we particularly found the Kiviat diagram (i.e., a multi-dimensional graph) useful. Figure 12 illustrates such a diagram, in which we have arbitrarily assigned numerical values to the previous scores. Obviously, this diagram means little without comparing it to alternatives. These diagrams conveniently indicate the overall best solution (i.e., the chart with the largest diagram) and where each solution is strong and weak.

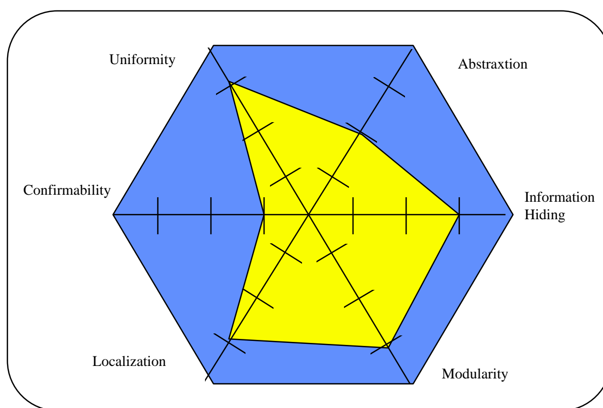

Figure 12: Example Kiviat Diagram

Implementation

Once the selection has been made, the engineering staff must be willing to enforce the decision. The best way to do this is by adopting a structural model standard. The AVTS Structural Model standard was one of the four primary software development standards developed and applied in the demonstration project. Each piece of software developed was verified against the standard before acceptance. Furthermore, we examined the calling structure and data flows by analyzing the design with computer aided software engineering tools. We highly recommend this kind of automated checking of the design against the standard.

CONCLUSION

Architecture is a strong design technique, which offers a great deal of potential for improving our ability to deliver better, cheaper, faster software. However, these benefits come at a cost. The software organization must be willing and able to consider alternative architecture, the merits of these alternatives, and choose between them. Once the selection has been made, the organization must be sufficiently mature to keep its commitment. The key to implementing our selection approach is linking the architectural figures of merit to system software performance, in terms of memory, time-efficiency, fault-tolerance, reusability, and development cost requirements. Developing these links is the key to making the selection of architectures *"more science than art. Unfortunately this research area still warrants significant more research before this can be achieved.*

REFERENCES

[1] Abowd, Gregory D. et al. *Structural Modeling: An Application Framework and Development Process for Flight Simulators*, ESC-TR-93-192, August 1993.

[2] AVTS Software Design Methodology, January 1994.

[3] Baumert, John H. and Mark McWhinney. Software Measures and the Capability Maturity Model, ESC-TR-92-025, 1992.

[4] Boasson, Maarten. *The Artistry of Software Architecture," IEEE Software, November 1995, pp 13-16.*

[5] Crispen, Robert G. and Lynn D. Stuckey Jr. "Structural Model" Architecture for Software Designers", *Proceedings of Tri-Ada 1994*.

[6] Garlan, David et al. "Architectural Mismatch: Why Reuse Is So Hard", *IEEE Software*, November 1995, pp 17-26.

[7] Garlan, D., and M. Shaw, "An Introduction to Software Architecture," *Advances in Software Engineering and Knowledge Engineering*, Volume 2, World Scientific Publishing Company, 1993.

[8] Guidelines for Successful Acquisition and Management of Software Intensive Ssytems, Volume1, Department of the Air Force, Software Technology Support center, February 1995.

[9] Kruchten, Philippe B. "The 4+1 View of Architecture", *IEEE Software*, November 1995, pp 42-50.

[10] Shaw, Mary. "Comparing Architectural Design Styles", *IEEE Software*, November 1995, pp 27-41.

[11] Van Der Linden, Frank et al. "Creating Architectures with Building Blocks", *IEEE Software*, November 1995, pp 51-60.