STRUCTURAL MODEL: Architecture for Software Designers

Robert G. Crispen and Lynn D. Stuckey, Jr.
Boeing Defense & Space Group
P.O. Box 240002 M/S JM-70
Huntsville, Alabama 35824

ABSTRACT

A structural model is the architectural map for a large software system or family of systems (domain). The structural model used in a domain represents the point of convergence for trade-offs between maintainability and performance, quality and efficiency. As such, different domains will likely have different structural models. The idea of a structural model evolved out of the Ada Simulator Validation Program (ASVP), which established the efficacy of Ada for real-time training simulation. In the years since this program a great deal of work has been done toward defining architectures and structural models in the air vehicle training system (flight simulation) domain. This is evidenced by recent initiatives such as the SEI's Structural Modeling, Mod Sim, STARS, DIS, ARPA's DSSA, etc. This paper discusses the concept of structural modeling within a software design methodology, the generic description of a structural model, and the structural model developed by Boeing Defense & Space Group, the Domain Architecture for Reuse in Training Systems (DARTS).

INTRODUCTION

The introduction and use of Ada has had a great impact on software engineering. The most recent influence has been in the area of software development methodologies. Historic methodologies consisted of several techniques or methods promised as silver bullets for software. Namely, the thought of software standards as an integral part of a methodology was foreign to most engineers. In practice the only meaningful standard was a coding standard. The reason for this lack of standards is directly attributable to the software languages being employed. They lacked the structure and consistency to support a standard. As Ada has become more and more the language of choice, the inclusion of software standards within a software methodology has grown.

The most important of these standards is the idea of a Structural Model standard. A structural model is new to the software process and falls directly out of a systems engineering process as it is applied to Ada software development.

The structural model is the framework through which components, attributes, and inter-relationships within the system are expressed. The structural model enforces a consistency in the software structure, thus aiding understanding. The structural model does not need to be confused with structural analysis. Structural analysis is a process of software component definition and refinement. In contrast, the structural model becomes an important part of the architecture in which the results of the structural analysis are implemented.

An *architecture*, as we intend to use the term, consists of (a) a partitioning strategy and (b) a coordination strategy[1]. The partitioning strategy leads to dividing the entire system into discrete, non-overlapping parts or components. The coordination strategy leads to explicitly defined interfaces between those parts. These two strategies provide an engineering approach to bridging the gap between the system as a whole (as represented by its specification) and the design (the plan to build the product from primitive parts, such as computer instructions, metal struts, and switches). The reach of an architecture can extend from a single system (an architecture that solves a unique problem) to an entire family or product line of systems. In the latter case, once the partitioning methods and coordination rules are determined, multiple products can be generated using the same methods and rules.

SOFTWARE DEVELOPMENT METHODOLOGY

Architecture development and its component, structural modeling, are members of a family of methodologies used in a defined, repeatable, improvable software development process. A methodology should spell out general steps to follow. It should be specific enough to give guidance but be general enough to apply to most software situations. It should not be

1. This definition is similar to the SEI's definition of structural model [SEI93]. However, as we hope to make clear, we consider the requirements for an architecture to include the content of the partitioning, not just a strategy for partitioning.

taken as a step-by-step way to develop entire systems; these *recipes* for how to get the work done simply do not exist. Management needs to realize that a directive to "use Schlaer & Mellor" or "use a structural model" has about the same content as a directive to "use an oscilloscope."

The systems approach to software development concentrates on the total system over its whole lifecycle. It addresses quality characteristics, methods, and standards, and provides a roadmap that integrates them into the whole. Figure 1 illustrates this concept. These components address all the considerations needed for software development. The lifecycle describes the phases in which software development takes place. Quality characteristics define the attributes that the software must exhibit in order to reach the software goals. The methods are the procedures employed in software development. The standards are used to guide and evaluate the software development process [STUCKEY92].

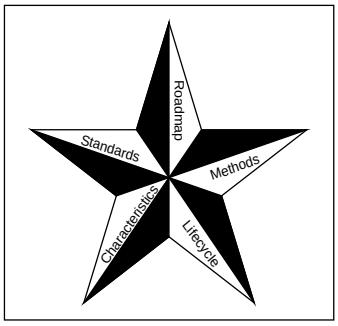

Figure 1. Components of a Software Development Methodology.

The software standards component of the methodology provides for consistency in the software development process. Each of the standards addresses the style and layout of the code. These standards are used both for a guide as well as a review tool. The following elements expound the standards that a defined methodology might commonly include. They can be treated as a minimal requirements set for a software design methodology. The generalized elements include (a) Structural Model, (b) Data Structure Model, (c) Coding Standard, and (d) Verification Standard. It is imperative that each standard be followed exactly and enforced during each design review. Then at the end of the program these guides will be the most important documents in the maintenance of the software system.

DEFINITION OF A STRUCTURAL MODEL

A structural model indicates the form of the general solution of the system [SEI93]. It is a set of constraints on the solution space for the project (e.g., how data will be passed, how pieces of the system should be organized, and how these pieces relate to each other). The structural model specifies:

(1) The kinds of entities that will exist in the design (How do you package?).

(2) How the real world is mapped into the software entities (What's in a package?).

(3) The communication between entities (How do packages communicate?).

The question of how objects communicate is critical. In the real world, a system's interfaces are of paramount importance. How the subsystems plug, wire, bolt, or connect together is most of the design problem. With this in mind, it is astonishing that system software has done such a poor job of modeling interfaces. Part of the problem is that the software's decomposition has not represented the real world system. Another problem with interfacing has been the water bucket approach: throw all the interfaces into a bucket which anyone can access. This common global data approach has been a large source of error in most software systems that use it. For software to become functional without extensive wasted effort, the interfaces within the system must be both defined and controlled. The software itself must directly support true interface definition and control.

A "GOOD" ARCHITECTURE

What does a "good" system/software architecture look like? If architectures are different from one another, it ought to be possible to say that one architecture is better or worse than another in some meaningful sense. Nevertheless, we have seen very little in either the training systems literature or the software engineering literature on what makes one architecture better than another. There are assertions that one architecture or another is a "good thing," but there is little public scrutiny of criteria. We offer the following characteristics against which software architectures might be measured:

• A good architecture can be leveraged. It must show promise of lasting beyond present programs, rather than being a quick fix of specific current problems. It must be adaptable to easily fit many development methods. It must promote the highest levels of reuse maturity. It must hold up to changing requirements. And it must be scalable across a significant portion of the training systems domain.

• A good architecture is rational. It should promote and support a repeatable and improvable process for building out a specific member of the product family.

• A good architecture is affordable. It must be "efficient enough" in both time and memory. It must support large-scale cost and schedule improvements in both the short term and the long term. And it must have been defined, published, and demonstrated to work in order to reduce risk.

• A good architecture takes into account the complete domain of the problem. It must address visibility, interprocessor communications, time and memory requirements, frame balancing and processor balancing, and testing and debugging.

• A good architecture is consistent and enforces an interface contract between loosely coupled component models. It should permits subsystems to be developed independent of the source of the inputs and the destination of the outputs. It should allow new implementations of systems to be integrated into existing specifications.

• A good architecture encourages early development. In the case of data voids, systems for which data is missing can be stubbed, and the interface specification defines what must be known later about the system.

• A good architecture promotes system understanding. It must "look like" the problem space in some significant sense. It must be clear, and it must clearly meet both user and end customer requirements. Its quality and style should match what are considered sound systems and software engineering principles.

• A good architecture is a good citizen. It should not violate company or customer standards. It should be broadly accepted or acceptable in the community. It should be available in the public domain rather than being bound to a proprietary hardware or software system. And it must take advantage of military and international standards like the Ada programming language and ISO communications protocols.

ANCESTORS OF DARTS

The first use of the phrase "structural model" that we are aware of was at the Ada Simulation Validation Program (ASVP) workshop in Dayton, OH, at which Boeing was one of the two implementers [ASVP88]. Following that time, the Software Engineering Institute (SEI) developed their Air Vehicle Structural Model (AVSM) [SEI93]. Independently, Boeing was the prime contractor in a simulation industry-wide Modular Simulator program (Mod Sim) [MODSIM91]. We developed a Domain Architecture for Reuse in Training Systems (DARTS) as an internal research and development (IR&D) effort and are using it in a Department of Defense Advanced Research Projects Agency (ARPA) program called Software Technology for Adaptable, Reliable Systems (STARS) [CRISPEN92].

In the simplest terms, DARTS attempts to exploit both the reusable form achievable through structural modeling, and in particular the SEI's AVSM, along with the reusable content (e.g., systems engineering work such as system decomposition) of Mod Sim.

DARTS : AN EXAMPLE STRUCTURAL MODEL

DARTS is the software architecture/system architecture we have applied to the Air Vehicle Training Systems (AVTS) domain. 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**.** The devices within this domain are each made up of some subset of the domain segments or sub-domains.

DARTS includes both a body of systems engineering work (common content) and a structural model (a common form for components). We will not be discussing the former any further, except to note that a body of pre-existing systems engineering work has major possibilities for reuse which we are in the process of demonstrating under the STARS program.

The structural model for DARTS is shown in Figure 2. The DARTS structural model contains five major *structural elements*: the Virtual Network, the Module Executive(s) the Segment Executives, the Subsystem Controllers, and the Components.

Virtual Network

Definition The Virtual Network (VNET) is the means by which the other structural elements of the DARTS structural model communicate with one another.

Internal Design A set of *messages* is defined containing all data flow between each of the Segments (see below). Saying that the DARTS structural model is message-based has caused some concern among people who have suspicions about a pure Boochian design running in real time. We do not intend this meaning of "message" at all. Because the DARTS Structural Model has four levels of decomposition, and because the messages are used between one set of structural elements, data flow is both limited and quite orderly.

Each message is defined as an Ada type, with representation spec (since multiple computational systems and Ada compilers may be involved) and messages are grouped in package specs according to which Segment outputs that message. Additionally, DARTS defines the legal senders and receivers of a given message. For example, the Flight Station segment may send a Thrust Reverser Engaged message to the Propulsion segment. No other Segment may send this message, and no other Segment may attempt to receive this message.

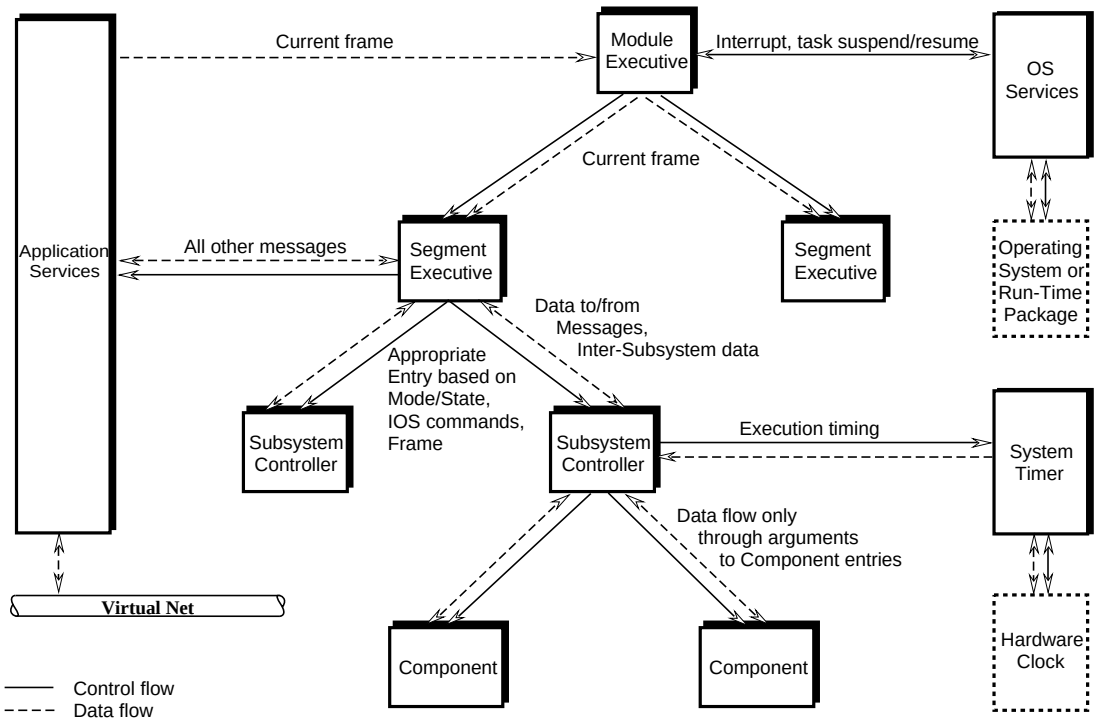

Figure 2. Domain Architecture for Reuse in Training Systems Overview

The VNET is accessed through a common set of Application Services. These services contain such obvious things as *Put*, *Get*, *I_Am*, and so on. The VNET has been deliberately designed with enough intelligence to perform some essential hiding of functionality. Thus, no Segment, Subsystem or Component need ever know what hardware it is running on.

The package body of Application Services takes care of implementation details. If two Segments are running in different Modules, they will need to communicate over some network (FDDI, Ethernet, and Reflective Memory have all been used). On the other hand, if the Segments reside in the same Module, it is almost certainly more efficient to leave the network out of things and communicate through a memory buffer.

A simple example may illustrate this. Suppose that the Control Segment outputs a message for the Flight Dynamics Segment and the Physical Cues Segment to freeze the aircraft position. The Control Segment calls *Put* for that message, and at some later time the Flight Dynamics and Physical Cues segments call *Get* for that message. If Flight Dynamics is in the same Module as Control, but Physical Cues is in a different Module, they will obviously use different mechanisms to get that message (the former will get the message from the memory buffer, and the latter will get it from the network's buffer). Nonetheless, they use the same *Get* call to Application Services, and the details of the message passing are hidden from them.

That describes the *Get* side. What about the *Put* side? Application Services knows that one of the Segments receiving the message is in the same Module and one is in a different Module, so it invokes both the real network interface and the simulated (memory-buffer) interface. If all the Segments defined as receiving the message were in the same Module, Application Services would not have to talk to the real network interface for that message.

One additional word is required on the inner workings of *Get*. *Get* says to Application Services, "Do you have any new data for this message?" Suppose a Segment calls *Put* at what we call half-rate: only on odd-numbered iterations, or only on even-numbered iterations. Suppose another Segment calls *Get* at what we call max rate: every iteration. If there is no new data in the message, the receiving segment may choose to not even execute the Subsystems that deal with that data as input. Similarly, Segments in different Modules can go out of synchronization, or perhaps a Module shuts down or freezes. The VNET architecture copes with these events gracefully.

VNET is quite an elaborate mechanism for passing data between Segments, particularly when all the Segments reside in one Module. Other structural models, such as the SEI's Air Vehicle Structural Model use shared or reflective memory to implement a similar level of communication. Why have we chosen VNET, then?

First, we have measured the amount of time the VNET takes in various configurations. As we hoped, the overwhelming majority of time in our software was spent in processing the data, not in sending it over the VNET. We have developed a rule of

thumb that if a segment spends as *little* as ten times the execution time processing the data as it does getting the data from the VNET, the Segment probably doesn't need that data.

Second, as computational requirements change and it becomes necessary to add or subtract CPU power or to place different segments in Modules, it is worth noting that the change of a handful of lines in the Module Executives is the *only* change that is required to make this happen. We once spent two weeks doing this on a"simpler", shared-memory architecture, versus less than an hour on a DARTS architecture.

Third, a VNET permits combinations of computational systems which are impossible with shared memory structural models. Shared memory often ties the builder of a system to one or a handful of vendors of shared memory hardware.

Finally, MIPS are becoming as cheap as MBytes. We are no longer in the position of having to jettison good ideas in order to save instruction cycles, though it still makes sense to eliminate overt time wasters.

We believe the message-based communication mechanism of DARTS is a better solution. We also understand that achieving universal agreement on this point is unlikely.

Module Executive

Definition A Module is a computational system with associated resources such as storage devices, displays, network interfaces, and so on. However abstract we'd like our structural model to be, at the end of the day it has to run on a real computer. Multiple-CPU computers are *generally* modules, the rule being "do you have to communicate with the other CPU(s) as though they were in different boxes?" (e.g., over a network). The purpose of the Module Executive is to cause the lower-level design elements to execute.

Internal Design There is one Module Executive for every Module. All operating system and hardware dependent functions such as interrupt, task suspend and resume, and so on, are located here. A Module Executive running on Brand X computers may look quite different from a Module Executive running on Brand Y computers, though of course its functionality will be the same.

The Module Executive "causes" the Segment Executives to execute. This is kept deliberately ambiguous, since the right way of doing this on a given program may be to call the Segment Executives as subprograms, or it may be to schedule their execution as independent tasks. Because data flow between the Module Executive and Segment Executives is one-way and small (the clock tick message passes from the Module Executive to its Segment), it does not stand in the way of implementing the right choice for a program.

Segment Executives

Definition A Segment is a major grouping of functionality. Early in the Mod Sim program, it was determined that some functions and objects "go together" in the sense that (a) there are major data flows between them, and (b) there are order dependencies between them. Subsystems and Components which go together in this sense are grouped into a Segment. These functions and objects were gathered together into 12 Segments, and this represents some of the pre-done systems engineering work that makes DARTS a robust candidate for the development of reusable software.

Internal Design. The Segment Executives are responsible for all communications over the VNET (apart from the clock tick message). By isolating the VNET communications functions in the Segment Executives, the lower-level elements (Subsystem Controller and Component) may be reused from similar software for other architectures. For example, we are currently doing IR&D work to attach an Environment Segment to a simulator which doesn't use the DARTS architecture at all.

The Segment Executives are also responsible for mode and state control logic (total freeze, reposition, run mode, and so on).

The Segment Executives schedule the execution of their Subsystem Controllers by using a scheduling table mechanism. Functions such as malfunction insertion and mode/state change are handled in the main execution thread in DARTS: a mode or state change message in DARTS is a message like any other, though it is processed by the Segment Executives.

The Segment Executives in DARTS call upon the appropriate aperiodic entries in each of the Subsystem Controllers, based on the receipt of the appropriate control messages through the VNET.

Segment Executives execute a large amount of highly predictable code. If a Segment is defined as the sender of a message in our interface specs, then it will identify itself to the VNET as a sender of that message, send that message, and so on. Additionally, mode and state control logic is almost certain to be the same for every Segment. We have developed utility programs which generate a "basically working" Segment Executive for every Segment which may then be adapted to take order dependencies and different iteration rates for Subsystems into account.

Subsystem Controllers

Definition Subsystems originally corresponded to the functions allocated to Segments in the Mod Sim architecture, which at the request of the customer was a traditional functional decomposition. Since that time, we have done additional work on abstracting the objects in the system in a more object-focused or object-abstracted methodology.

We have had surprisingly little difficulty in doing this, which demonstrates the truism that in the real world functions aren't performed by spirits or demons; they're performed by objects. This also shows that the original functional analysis was done fairly skillfully[2]. Often the methodological difficulties with functional decomposition are a straw man: the issue is not between functional decomposition and object-abstracted decomposition, but between bad decomposition and good decomposition.

Good decomposition in the AVTS domain, whatever the methodology is called, will recognize the twin truths that (a) the system has functional requirements (if you would like us to develop a system that has no functional requirements, we would be *very* interested in taking your money), and (b) the next levels of decomposition will require the simulation of real-world objects (we will cheerfully take your money to build a simulator that doesn't have to simulate anything). Seriously, though, an architecture that illuminates the functional requirements and the simulated objects is likelier to drive a successful program than an architecture that obscures them.

Subsystem Controllers are implemented as in the AVSM. In the AVSM data flows out of Subsystems through a shared memory based Export Area, while in DARTS (largely because of the correspondence between Subsystems and interface messages) the Segment Executive provides the Subsystem Controller with data from messages and builds messages to send to the VNET.

Internal Design The subsystem controller is at the middle of the DARTS hierarchy. Its entry points are standardized. Every subsystem controller has the same entry points. This does not mean that all entry points will be used in every subsystem controller. It may be, for example, that there is not a malfunction in a given subsystem, in which case the *Process_Malfunction* entry won't do anything, and will never be called.

The complexity of the controller is largely based on the segment. Segments with subsystems that are physically related behave relatively sophisticated subsystem controllers. In contrast, segments with more logically related subsystems can lead to rather simple subsystem controllers. One example that comes to mind is a recent IR&D simulator which fired missiles. The flight of each missile was the province of the lower-level component. So the subsystem controller was responsible for determining how many missiles were flying and for calling the lower-level components the required number of times with the appropriate missile state data.

Initialize, *Reconfigure*, *Set_Parameter*, *Hold_Parameter* and *Process_Malfunction* are aperiodic entries, called by the segment exec, generally based on system state transitions triggered by the receipt of messages from the Control segment. *Import*, *Update*, and *Stabilize* are called by the segment executive periodically, and could be called every frame, every other frame, every fourth frame, or whatever the requirements of the program demand.

The *Update* entry point performs required processing, including activation of a subset of constituent components, as necessary, to determine subsystem state and export the resultant state data. *Update* is commonly called iteratively, at a rate between 20 and 120 Hz. The *Import* entry point collects time-critical data required for *Update* or *Stabilize* from export mechanism. The *Stabilize* entry point performs required processing to implement state convergence of the subsystem and report subsystem stable or not stable in an OUT parameter.

The *Initialize* entry point loads a set of state data into controller and/or components to bring the subsystem to a known state (either initial or final state, when hardware must shut down gracefully). The *Reconfigure* entry point retrieves and loads state data, loads databases, builds tables, etc., which bring the subsystem to a required mission state. The *Set_Parameter* entry point loads provided subsystem parameter values. It calls on the appropriate components' *Set_Parameter* entries to set the parameter in question. The *Hold_Parameter* entry point freezes a selected output parameter at its current value and prevent further update of that parameter until released. The *Process_Malfunction* entry point sets appropriate state data in the subsystem controller or invoke applicable components to activate or deactivate malfunction.

Components

Definition As in the AVSM, the lowest level element identified in the architecture is called the Component. Each of the Components corresponds to an Object in the OOA sense. In DARTS, as in the AVSM, all knowledge about the operation and state of Components is contained within the Components. And no knowledge about the external environment (simulation control commands, presence or absence of other Components, computational environment) is contained within the Components. Components compute the state of the objects they simulate in a purely abstract, and therefore reusable, manner. These rules, which are among the most attractive features of the AVSM and DARTS structural models, comprise "knowledge firewalls".

Internal Design Just as in the AVSM, all data flow between Components takes place through the subprogram calls for each of the entries in the Components. As the SEI points out, this set of entries is both necessary and sufficient to permit the knowledge firewalls described above to operate.

It is the responsibility of the individual Components to provide "safe" input values for themselves. Thus, the *Initialize* entry for each Component provides both input and output data. Once the

2. Virtually every company in the simulation and training systems industry participated in the analysis. Boeing performed considerable data flow simulations and all participants presented their findings to an Interface Standards Working Group.

Component has executed its *Initialize* function, it may execute without error even though no input data has yet arrived over the VNET. This eliminates all worries about which Components or Subsystems need to execute before others in order to avoid erroneous data being processed.

The *Update* entry point performs required processing to determine component state and export the resultant state data. The *Set_Parameter* entry point loads provided component parameter value. The *Process_Malfunction* entry point sets appropriate state data in component to activate or deactivate malfunction. The *Initialize* entry point loads a set of state data into component to bring component to a known state. A component which is initializing is responsible not only for setting its internal state and an initial set of outputs, but is also responsible for setting input parameters for itself. If the component is not supplied with any new data from other components, it guarantees that those inputs will not cause it to crash or to do anything strange.

DARTS Impact on Design

Templates One way that DARTS differs from other architectures is that every subsystem controller is identical to every other subsystem controller. They have the same periodic and aperiodic entry points. Likewise every component has the same entry points. Every subsystem looks like every other subsystem from the outside, and every component looks like every other component from the outside. What entries are needed to do all the work of the object? The answer is here, no more and no less.

Because every one of the subsystem controllers is like every other, and every one of the components is like every other, these elements can be thought of as reusable templates. It's important to realize that if a brand-new component or subsystem is to be created, especially as a domain engineer, you have a head start already. The form is defined and an outline of the function it has to perform is known. So the effort is very focused.

Segment Interfacing The segment exec performs all the communications between segments, with the single exception of the clock tick message which is supplied by the module executive so that all the segments in the module are executing the same frame at the same time. That means that subsystems and components don't have to worry about communicating with other segments. In fact, subsystems and components aren't allowed to communicate outside the segment. Segment executives also perform all the mode and state logic, and the frame logic. So subsystems and components don't have to worry about that either.

Adapting the Architecture Making modifications within DARTS is a simple matter promoted by its design. The simplest example is at the component level. A component is pure code which passes all its data in and out through it subprogram entry parameters. The component keeps all knowledge about the item it's simulating, and is thus an object in the pure object-oriented sense. This allows a plug and play approach to this level of the training system.

Another example is found in the definition of the system decomposition. Decomposition of the training system is defined and implemented. It is not an amorphous blob of objects that are placed willy nilly. The segments and subsystems become more of a holding tank for components. As such they are, once defined, static throughout the domain. For instance, a Navigation segment will have a Radio Aids subsystem that contains components such as TACANs or VORs. This follows a standard object view of the world. If a TACAN model is not required then it is removed, but the Segment and Subsystem remain static.

The idea of service functions is the final example of adapting the architecture. In training systems there can arise requirements to provide a standard service function in part of one segment. The segment identified can change. As such the architecture provides a place holder for the function and identifies its placement as a domain variability. This type of flexibility makes DARTS more responsive to the training system customer.

Advantages of DARTS

The performance of DARTS, according to our current understanding, against the criteria we discussed at the start of this paper is summarized in Figure 3. Some advantages of DARTS which were captured from its progenitors (AVSM and Mod Sim) deserve special mention.

• The Subsystem Controllers and Components are based on reusable templates. Every Subsystem looks like every other Subsystem and every Component looks like every other Component, in that they have the same subprogram entries and the same package structure.

• Components are structured to be widely reusable. Since Components have no knowledge of their environments, they should be reusable in the widest possible context.

• Delivery is more predictable, since the components are all nameable and locatable very early in the program. Each of the Subsystems and Components can be tracked from a very early date in the program, so that reaction to delays and data voids have lower impact.

• DARTS is designed to permit Segments to be easily subcontracted. Companies with expertise in visual systems, electronic warfare, or weapons but which have little or no training system experience can compete to build an individual or set of appropriate Segments. The ability of Segments to be tested as stand-alone entities lowers both prime and subcontractor risk at acceptance.

Leverage
- Major elements proven on multiple programs (F-16 Mod Sim, USAF Structural Model)
- Not tied to any CASE tool or computer vendor
- Subsystem specs and bodies and Component specs may be automatically generated
- Reuse of Components across multiple architectures
- Scalability by plug-replacement of Components, Subsystems, Segments
- Segments may be eliminated or combined for product-line variations
- Based on industry-wide Domain Engineering effort

Simplifies System Understanding
- Small number of well-defined elements (12 Segments, Subsystems, Components)
- Structure maps to requirements analysis (early management visibility into software)
- Software engineering principles from SEI, SPC and STARS

Rational
- Subsystems and Components are a toolset, not a straitjacket
- Results of early systems engineering activities flow into design
- Interface specifications clarify requirements, guide design

Affordable
- Much systems engineering work is already done, simply by selecting the architecture
- Parallel development, testing improve schedule performance
- Lower integration time proven in F-16 simulator program
- Architecture proven in F-16 program to be fast, cheap enough for 50 Hz WST
- Exact specification of computer power, best computer architecture for segment

Good Citizen
- Based on standards in public domain (FDDI, XTP, Ada)
- Mod Sim and Structural Model government-sponsored for simulation industry
- ISWG got wide consensus from government, simulation industry
- Contact with ARPA DSSA program through STARS

Figure 3. Summary of DARTS Advantages

• As requirements change, within a range of simulators, or as follow-ons require more or less computational power, DARTS permits near-zero-effort addition or deletion of Segments and of computational power allocated to a Segment. Segments may be moved from one Module to another, again with near-zero effort. When this change is anticipated, a hardware architecture can be chosen for the affected Segments that permits the simple plug-replacement of CPUs with less powerful or more powerful CPUs.

• Interfaces between Segments are strictly specified in compilable Ada. Adaptation of these reusable interfaces to the requirements of a specific program is accomplished by the decision model for the domain, and we have demonstrated that it is easy to automate this process.

Disadvantages of DARTS

• DARTS, like the AVSM, absolutely requires data flow control, which takes engineering hours. Our slogan has been, "If you want to control data flow, you've got to control data flow." The generic, adaptable interface specification provides a great deal of help in this control process, and utilities associated with DARTS automate much of the tedious coding work like message setup and connection.

• DARTS allocates systems functional requirements to subsystems and permits object or functional decomposition of components. O-O purists may see this as a red flag.

• DARTS is domain-specific. As such, organizations working in different domains may not realize the full benefits of DARTS.

EXPRESSING A STRUCTURAL MODEL IN Ada

Previous software languages have not been sufficient for the expression of a structural model. They did not provide the reliable structure and standardization that must be expressed in a defined software architecture. A structural model expressed in C or FORTRAN for example is a dog with no bite! Ada, on the other hand, provides the base of language features, standardization, and a systems engineering approach that are required to fully express an architecture as part of an organizations software development methodology.

Standardization

The standardization of Ada contributes to the expression of a structural model in a number of ways. Portability is the most recognizable. For a structural model to be useful it must be portable in two ways, (a) between environments, and (b) within a domain of products. Portability between environments addresses the classic need to be able to transport code from computer to computer without subjecting the project to reliability issues. Ada is the only language that *can* provide such assurances. The issue of domain portability addresses the need for an architecture to apply to a family of products. Ada allows for the adaptation of the structural model to facilitate this.

Language Features

There are many language features of Ada that contribute to the implementation and enforcement of a structural model. Of prime importance are packages, separates, types, and the ability to restrict certain context clauses.

Packaging is really the cornerstone of structural modeling. Without Ada packages, a structural model is nothing more than cheap philosophy. The architecture illustrates a packaging scheme for the domain. Each level of architecture (segment, subsystem, and component) is implemented by Ada packages.

SEPARATEs allow an expressed architecture to be physically capable of coherent and logical implementation. We would be overwhelmed by the enormity and complexity of the code if we were required to lump all the code at each level of the architecture into one subprogram. Based on the complexity of an object, it is conceivable for the various entry points to be implemented via separates. In some instances each lower level of the architecture could be implemented as separate packages, components separate from subsystems separate from segments. The ability to use separates allows for true achievement of information hiding and separation of concerns.

Good Ada typing is the basis for understandable, maintainable, and reusable code. This is especially true of the code that expresses the software architecture. System understanding is an evaluation criteria for an architecture and it is largely achieved through types. A major advantage in the architecture arrived at through types is a form of data inheritance. A segment message can be built up as a series of records of records. This decomposition of types can mirror the physical decomposition of the system. This allows for ease in interface control that is consistent with the overall architecture.

Finally, the ability to restrict scope and visibility is required in an architecture and provided by Ada through the use of the *with* context clause and the restriction of the *use* context clause. This allows for better clarity of design and facilitates both system integration and maintenance.

Systems Engineering Mindset

The main systems engineering benefit of Ada is very basic. Systems engineering asserts that a system that is made up of optimized components will itself not be optimized. In other words, the system has to be optimized as a whole -- not as separate individual pieces. Ada was developed with this principle in mind. As such, it does not over optimize the pursuit of one software engineering goal versus another. This is very important when striving to produce an architecture that facilitates all the software engineering goals of modifiability, efficiency, reliability, and understandability.

EXPERIENCE WITH DARTS

Boeing has used the predecessors of DARTS on ASVP and Mod Sim. Under our IR&D program, we have used DARTS for a simple F-15 simulator, for an Avenger fire unit, and for the networking portion of a missile simulation program.

We have implemented DARTS on 10 VMEbus modules running VxWorks, connected by the FDDI fiber optic network. We have also implemented it on a Sun running SunOS, various Silicon Graphics machines running Irix, and a VAXstation running VMS.

While there are special joys involved in implementing DARTS on various operating systems and hardware architectures, we came out of the experience without ever saying that the architecture was broken or in any way needed to be bypassed to meet a particular requirement. We note that the missile and fire unit simulation domain is different enough from the AVTS domain to benefit from the kind of analysis that produced the DARTS architecture. Still, and much to our surprise, the DARTS structural model seemed to be nicely adaptable to the missile simulation domain, indicating that there may be some features of the DARTS structural model which are more universal than we are currently claiming them to be.

CONCLUSION

Our general experience is that (1) other domains can experience the same benefits as AVTS -- with a structural model suited to their specific domain, (2) Ada really is a necessary and sufficient language for expressing a structural model, and (3) Structural Modeling provides an essential part of an overall system architecture for creating high quality large scale software systems.

An architectural map for a software program or domain is essential to the production of quality software. But the existence of such an architecture is not enough. It must be an integral part of the software development methodology. It must be a standard with which to measure software quality as well as a guideline for software development.

REFERENCES

[ASVP88] The Boeing Company. "Ada Simulator Validation Program Final Report", D495-49506-1, Sept. 1988.

[CRISPEN92] Crispen, Robert G., et. al. "DARTS: A Domain Architecture for Reuse in Training Systems", 15th I/ITSEC Proceedings, Nov. 1992.

[MODSIM91] The Boeing Company. "System Segment Specification for the Generic Modular Simulator System", Volumes I-XIV. S495-10400, June 1991.

[SEI93] Software Engineering Institute. "Structural Modeling Guidebook (Draft)", January 1993.

[STUCKEY92] Stuckey, Jr., Lynn D. "A Systems Engineering Approach To Software Development", A Masters Thesis, The University of Alabama in Huntsville. May 1992.

ABOUT THE AUTHORS

Robert G. Crispen is a Systems Analyst with the Missiles & Space Division of the Boeing Defense & Space Group. He worked on the Modular Simulator Program, the Ada Simulation Validation Program, the STARS program, and is currently a researcher in an IR&D program at Boeing. Before joining Boeing, he was a Senior Systems Design Engineer on commercial flight simulators at GMI in Tulsa, OK. He holds a Bachelor of Arts degree from the Johns Hopkins University. Email: crispen@foxy.hv.boeing.com

Lynn D. Stuckey, Jr. is a software systems engineer with the Missiles & Space Division of the Boeing Defense & Space Group. He has been responsible for software design, code, test, and integration on several Boeing simulation projects. He is currently involved in research and development activities dealing with software reuse in the domain of air vehicle training systems. Mr. Stuckey holds a Bachelor of Science degree in Electrical Engineering from the University of Alabama, in Huntsville and holds a Master of Systems Engineering from the University of Alabama, in Huntsville. His thesis presents a systems engineering approach to software development. Mr. Stuckey is a doctoral student at the University of Central Florida. Email: stuckey@plato.ds.boeing.com