TESTING A TECHNOLOGY FOR REUSE

Brett W. Freemon and Robert G. Crispen
Boeing Defense and Space Group
Huntsville, Alabama

ABSTRACT

Organizations acquiring new systems often pay for the construction of the same components (design, documents, and code) over and over again. The historical premise of reusing existing components to save money, reduce risks, and improve schedule performance has so far proven more myth than reality. The technology customarily used to support reusability is a library of reusable components. Results from this kind of library have been disappointing in the real-time simulation realm. This library usually has no mechanism for showing interrelationships or structure of components. And as the complexity of components increases beyond control-law fragments, the components become less and less reusable and the size and contents of the library grow out of control, until a potential reuser finds that using the library is more work than rewriting a component from scratch.

The DARPA Software Technology for Adaptable, Reliable Systems (STARS) program is addressing these and other problems of reuse, and promoting technologies to achieve a vision of process-driven, domain-specific, reuse-based, technology-supported system development.

The Software Productivity Consortium (SPC), of which Boeing is a member, is pursuing a vision of transforming the production of software into an engineering discipline. One result has been the development by the Consortium of the Synthesis methodology, an innovative reuse-oriented approach to software development. Synthesis defines how, guided by the commonalities and variabilities of a family of systems, to create adaptable components and a process for choosing among the variabilities to create systems. Because this methodology exemplifies the major tenets of the STARS reuse vision, Synthesis is under investigation by STARS and Boeing.

STARS has funded a pilot effort by Boeing Simulation and Training Systems, the Center of Excellence for simulation and training for the Boeing Company, to use and evaluate the SPC Synthesis process. This project will determine variability information and elaborate a decision process for one domain of a generic flight simulator and test the decision process on an existing F-16 R&D simulator. This paper reports on the research performed and the preliminary results obtained.

ABOUT THE AUTHORS

Brett W. Freemon is a Senior Software Engineer with the Missiles and Space Division of the Boeing Defense and Space Group. He has worked on all areas of the life cycle of simulation and training systems from proposal through delivery and installation. He is currently working on a research and development program centering on software reuse technology. Mr. Freemon holds a Bachelor of Science degree from the Georgia Institute of Technology in Applied Physics.

Robert (Bob) G. Crispen is a Systems Analyst with the Missiles and Space Division of the Boeing Defense and Space Group. He worked on the Modular Simulator Program, the Ada Simulation Validation 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 in Liberal Arts/Psychology.

© 1992 The Boeing Company. All rights reserved.

TESTING A TECHNOLOGY FOR REUSE

Brett W. Freemon and Robert G. Crispen
Boeing Defense and Space Group
Huntsville, Alabama

INTRODUCTION

"The current Grail of software engineering is reuse." [Griswold91] Government and industry agree that reusable software parts will lower costs and risks and speed the delivery of products to the customer. Yet reusability has proved so far to be easier to specify than to implement.

Reuse cannot be an afterthought: "most people will agree with the idea that reusable software is hard to make. Some would even say that USABLE software is hard to make." [Sink91]. There is no question that reuse has to be planned for. "Reusability is first and foremost a design issue. If a system is not designed with reusability in mind, component interrelationships will be such that reusability cannot be attained no matter how rigorously coding or documentation rules are followed." [Ausnit85]

Yet, reuse has failed. When organizations have built libraries of reusable parts, the libraries either contain tiny control-law-style code fragments that are easier to redevelop from scratch than to find in the library, or they contain large code modules that have to be heavily modified to be reused. As the number of components increase each component is less reusable and the library becomes a data sink.

There are cultural problems as well: it is more fun to develop code than to find it; engineers are suspicious of code they did not personally write; engineers have training in how to code, but not in how to reuse [Davis92]. When managers grade employees on productivity, reuse is often neglected. One set of published criteria says, "The code that actually is copied from a source statement library is reused code and should be counted as such. If you are interested in development productivity it can be ignored." [Jones91]. One critic even noted that "in practice, effective reuse is more an achievement of good development environments than a strategy for software development."[Pintado90].

STARS DIRECTIONS TOWARD REUSE

The DARPA Software Technology for Adaptable, Reliable Systems (STARS) program is promoting reuse as a shift in perception by the software development community. A culture change is required from the technical and management perspectives in order to reap the productivity and quality benefits of reuse.

There are two important cultural changes on the management side. First, reuse must be a defined part of the way an organization does business. This must include the definition of processes for creating, managing and reusing software assets, and the incorporation of these processes into the organization's other defined processes. Second, an organization must lay out plans and goals and invest in the infrastructure that supplies both the expertise and the supporting technology in the areas relevant to its product lines.

There are two tenets to successful reuse on the technological side of STARS. First, the artifacts of the software development process (e.g. code, documentation, etc.) are more reusable if they are specific to one product line. Second, software engineering and reuse tools will be better utilized if they are selected and introduced to an organization along with the process for using them.

THE SPC SYNTHESIS PROCESS

The Software Productivity Consortium (SPC) is pursuing a similar vision: to transform the production of software into an engineering discipline. One of the tools they have developed to this end is the Synthesis process. The following is a summary of the Synthesis process [SPC92].

The Synthesis process stresses domain (i.e., product line) engineering. Synthesis is an approach for developing similar software applications based on systematic reuse of knowledge and products. Systematic reuse refers to not only reusable hardware, software and documentation, but a method or roadmap to follow in reusing results from previous successful programs.

Synthesis is an ordered approach to identifying the commonalities and variabilities among a set of similar systems and using them to formally guide reuse. Commonalities represent work that can be done once and then reused across all systems in the domain. Variabilities represent work which is not the same for all systems, but whose presence, absence, or adaptation is predictable throughout the domain. As the set of common and variable reusable components grows, a highly productive software development process can be defined. This process provides the necessary deliverables (e.g. requirements, design, code, and test procedures) for a given system by adapting and composing reusable components.

Synthesis is organized into two process that are closely related to each other: domain engineering and application engineering. See Figure 1. This structure separates the domain engineering issues of reusable component creation and process improvement from the application engineering issues of specifying and delivering products to customers.

Domain engineering can be broken down into two types of activities: domain analysis and domain implementation.

During domain analysis the domain is defined and formally described. The boundaries of the domain must be defined and a determination made as to the viability of the domain. The type of systems produced and the general characteristics of those systems are described in the Domain Definition. A justification analysis is conducted on the domain to determine whether additional investment is justified. The assumptions are built into a Decision Model that tells the engineer what the common and variable requirements are for the domain. This model shows how to specify systems by guiding the construction of deliverables for a domain specific architecture. This architecture is defined based on the domain definition and the components that make up the domain.

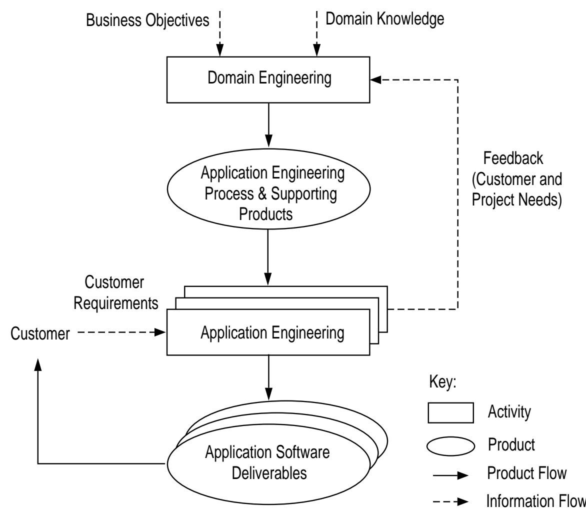

Figure 1 The Synthesis Process

© 1992 Software Productivity Consortium. All rights reserved.

During domain implementation the Decision Model is implemented in the components of the domain (code and documents). This is generally in the form of a macro language: if this characteristic of the domain is true, then the component will have this content; otherwise it will have that content. Standard policies and procedures are defined for Application Engineers to follow in their use of the Decision Model.

Application Engineers use the domain engineering products to specify a system and to produce deliverables. The process may be automated or manual; however the products are derived in a formal way from the specification. An Application Engineer never randomly searches through a reuse library. The reuse library structures searching to be directly correlated with the organization of the Decision Model. The development process indicates exactly which components are necessary and how they need to be adapted to satisfy the specification.

THE PILOT PROJECT

General Considerations

Since the SPC Synthesis process supports the STARS vision of reuse, STARS wanted to evaluate Synthesis with a pilot project. The Modular Simulator (MODSIM, or USAF designation HAVE MODULE) architecture [MSDP91a] was also of interest to STARS because the analysis which produced the MODSIM architecture was very similar to the Synthesis process of domain analysis. The MODSIM architecture closely resembles the Structural Model for Flight Simulators on the Software Engineering Institute (SEI) except that MODSIM focuses more on structuring communications between "segments". Messages pass between segments over a Virtual Network which may be implemented as a common data storage area, as it is in the Structural Model, or as an actual high-speed fiberoptic network as it is in the original MODSIM (see Figure 2). There is an on-going process at Boeing Simulators and Training Systems (S&TS) in Huntsville to convert from the original functional decomposition to an Object Focused Design in the MOD-

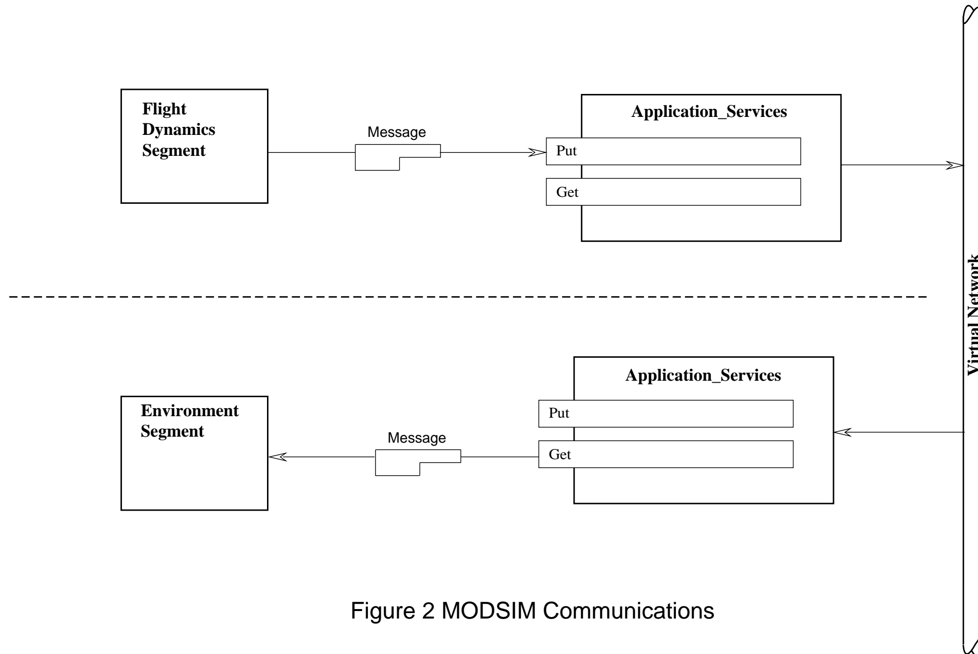

Figure 2 MODSIM Communications

SIM architecture.

On the basis that MODSIM is a sound reference product-line architecture and that the architecture is in the realm of flight simulation, STARS selected Boeing S&TS to evaluate the Synthesis process on a MODSIM testbed.

Approach

The authors in consultation with Rich McCabe and Jeff Facemire of the SPC and assisted by Bill Tucker and Gary Kamsickas of Boeing, chose a domain for the pilot program. The candidate domains were the twelve segments in the Modular Simulator architecture:

(1) Instructor Operator Station
(2) Flight Station
(3) Flight Controls
(4) Flight Dynamics
(5) Radar/Sensors
(6) Weapons
(7) Environment
(8) Electronic Warfare
(9) Navigation/Communication
(10) Propulsion
(11) Physical Cues
(12) Visual

Although the domain for the pilot study could have been as large as Boeing S&TS's entire product line, or as small as a single instrument on the control panel of a simulator, we determined that one segment from the list above was the proper size for our study in that it would be large enough to provide enough scope for significant testing of Synthesis, while being small enough to be doable in the time available. We used the following criteria to select the segment for our domain:

(1) Is the project doable as a 6 month effort?
(2) Can we demonstrate variability?
(3) Can the purchase of additional hardware be avoided?
(4) Is there a good mixture of variabilities and commonalities?
(5) Can the resulting component/process be reused by STARS?
(6) Will a demonstration of the process be impressive to industry?
(7) Is the domain of interest to the community?
(8) Can existing engineering be utilized?
(9) Will the resulting component be of future value?

The Environment domain was selected because it has the best fit with our criteria. Environment includes the tactical and natural environments external to the simulated vehicle. Also, the original MODSIM implementation did not include the Environment segment so that we were not working from a preconceived component. We first developed a definition of the domain:

"The Environment domain is a family of software segments that provides the simulation of the tactical and natural environments for a vehicle simulation which are external to the ownship. When the simulator is operating in a autonomous (stand-alone) mode, the entire external environment is simulated by this segment. When the simulator is operating in a Multiple Simulator Environment (MSE) mode, the segment shall serve as a network interface and import data about the external environment and export data about the ownship to other devices in accordance with established simulation standards (e.g., Distributed Interactive Simulation, SIMNET). The ENV segment provides MSE interaction, atmosphere modeling, external entity modeling, height above terrain calculations, threat weapon dynamics, damage assessment from ownship weapons, and database management. The ENV segment shall be capable of operating in a Modular Simulator System. The delineation of capabilities between the ENV segment and the rest of the vehicle simulation is made based on the criteria that capabilities which require no inherent data about the ownship will reside in the ENV segment. The ENV segment facilitates the introduction of the vehicle simulator into a MSE without changing the other segments in the vehicle simulation."

Following the Synthesis Guidebook [SPC91] we continued in the process by building and maintaining a glossary of terminology and identifying the assumptions that are appropriate to the Environment domain. The following are typical glossary entries:

Atmosphere - The gaseous medium surrounding the earth which defines the ambient air condition. This normally includes temperature, pressure, winds, icing, and dynamic pressure.

Autonomous - The simulation environment involving the stand-alone operation of a trainer in a training exercise.

Damage Assessment - The evaluation of injury to and the appropriate degradation of the system(s) affected.

The identification of assumptions is key to success in the Synthesis process. Assumptions are of two types: common characteristics that are the same for all products in the domain and variable characteristics that distinguish products in the domain. Table 1 is a sample of the assumptions for the Environment domain.

It was necessary to define the inputs and outputs (I/O) of the domain in order to complete our list of assumptions. We used the existing I/O interface definitions from the MODSIM specifications by allocating the appropriate interface objects to the Environment domain (e.g. threat weapon dynamics, weather conditions). We also had to determine the I/O of the individual objects in the Environment domain (e.g., the Atmosphere object requires the position and attitude data for the ownship and computes the atmospheric conditions data).

When we were satisfied with the domain definition and assumptions, we used them to produce a series of decisions to be made in a particular order. This set of decisions is called a decision model, and basically captures the parts of our assumptions that represent variabilities. For example, is this simulator required to compute moving model height above terrain? Will this simulator produce a constant atmosphere, will it produce a graduated standard day atmosphere, or will it have a fully-adjustable atmosphere model? Or, is the simulator always to be connected to a network, and not responsible for modeling the atmosphere.

Process Requirements were created to define the process that the Application Engineer (i.e. the person responsible for the design and development of a product) must follow in order to develop a product from the domain. In its developed form, the Process Requirements will be a series of menus and checklists incorporating all the variability questions in the Decision Model: Is there a requirement for an environment? If there is an environment, will the simulated vehicle be required to move in the environment? If movement is required, who can control the movement: (student, instructor, both)? Clearly there is a logical order here, as well as a set of dependencies: if there is no Environment segment, for example, none of the remaining questions need to be asked.

Now we have our definition of "decision model": decision model = commonalities (things that will be present in every possible product in the domain) + variabilities reduced to true/false or multiple choice questions + dependencies of questions on one another.

The decision model was first coded in Ada and compiled to give structure and prima facie demonstration of correctness. Later it was coded into a

TABLE 1

Assumptions that vary between components in the Environment Domain

The vehicle type being simulated:
- Fixed Wing Aircraft
- Rotary Wing Aircraft
- Ground Based (mobile)
- Ground Based (fixed)
- (etc.)...

The segments that are to be included:
- Flight Station
- Flight Controls
- Flight Dynamics
- (etc.)...

The Environment Domain objects:
- (1) Environment Segment Support
- (2) Multiple Simulator Environment Interaction
- (3) Atmosphere
- (4) Height Above Terrain
- (5) Occulting
- (6) Ownship Weapons' Damage Assessment
- (7) Threat Weapon Dynamics
- (8) External Entity
- (9) External Entity's Chaff and Flairs
- (10) Database Management
- (11) Threat Environment Database
- (12) Navigation Environment
- (13) Terrain Database (Visual and Radar)
- (14) Collision Detection

The type of atmospheric data:
- Constant
- Predetermined Scale
- Adjustable

Assumptions that are common to all components in the Environment Domain

There is a subset of Environment Domain objects required for any environment segment:
- Support
- Atmosphere
- Navigation Environment

The Atmosphere Object will provide pressure, temperature, wind, and precipitation data.

less formal macro language.

When the Application Engineer goes through the process defined in the Process Requirements (which for the pilot program we implemented in paper and pencil form) in order to design a particular product, the output of that process will be decision variables such as:

An_Environment_Segment_Is_Required (True, False)
Vehicle_Movement_Is_Required (True, False)
Movement_Is_Controlled_By (Student, Instructor, Both)

which have some value assigned to them (An_Environment_Segment_Is_Required would have the value "True"). In cases where the adaptation is not routine or where the component has not been designed yet, the decision process will output instructions to the Application Engineer on what new work needs to be done to complete the adaptation. For example, the decision model may have a decision variable whose value could be "Other" (this is especially appropriate for the Weapons domain), and the results of which would be an instruction to the Application Engineer to generate the code and documents for this Other weapon.

Constructing the Environment Segment

The first application of the decision model was in the area of software requirements. We generated software requirements for all the known members of the Environment domain, starting with the generic Environment specification produced in the MODSIM follow-on program [MSDP91b]. Decision rules like those mentioned above were applied to this requirements specification (called the Product Requirements) to tailor this document for a given product. In the pilot program this was done very informally, and more emphasis was placed on adapting code components. As we shall show, the process is identical for adaptable code and adaptable documents.

The authors applied the Synthesis process to create a new Environment segment in an R&D F-16 flight simulator. The lab simulator uses the MODSIM architecture with the virtual network implemented as an FDDI network connecting the following segments.

(1) Flight Controls
(2) Flight Dynamics
(3) Flight Station
(4) Instructor Operator Station
(5) Navigation and Communication
(6) Propulsion
(7) Visual

The pilot project replaced the Propulsion segment with a new Environment segment (moving basic engine throttle response to Flight Dynamics) in order to make use of existing hardware.

As a result of applying the decision rules of the Synthesis process, the following objects apply to the Environment component for the lab simulator. The sub-objects which were determined from the decision rules to be relevant to the lab simulator appear in parentheses beside the objects. Each sub-object corresponds to a message sent by Environment over the Virtual Network.

(1) Environment Segment Support (Mode and state control, clock tick, on-line diagnostics)
(2) Atmosphere (atmosphere and weather)
(3) Height Above Terrain (ownship)
(4) Navigation Environment (navaids and airports)
(5) Collision Detection

The next step in the development of the Environment segment was to create a Segment Executive which would serve as the top-level driver for the Environment code. The MODSIM executive was already designed with some elementary reusability in mind. The functions that are common to all segments, such as connecting to the clock, suspending, resuming, and overrun detection were contained in the top-level executive which was used by every segment. All that was required on the executive level was to supply the body of the Scheduler package, plus the code called by the Scheduler (i.e., the code that performed the function of each of the objects allocated to Environment).

Because Environment was a newly-created segment, some of the code for its objects was contained in code belonging to other segments. It was necessary to move this code to the Environment segment and encapsulate it. For example, Flight Dynamics had originally computed Atmosphere. It was necessary to move most of this code to Environment, then replace the call to Atmosphere in Flight Dynamics with code that read the Atmosphere data coming from Environment. As we will discuss below, it was also necessary to perform some additional computations in this Flight Dynamics Atmosphere stub in order to compute some variables that were not properly part of Atmosphere (e.g., Mach number) but which had been computed in the Atmosphere code.

The encapsulation strategy we used was to make each object an Ada package, and to create a subprogram call for each sub-object. For example, we created a package Height_Above_Terrain and included subprograms Compute_Ownship_Height_Above_Terrain and Compute_Moving_Model_Height_Above_Terrain. Each of these subprograms is called from the Scheduler. An alternative which is equally valid is to have only one subprogram, Compute_Height_Above_Terrain, which is callable from the scheduler. That subprogram would call the subprograms for ownship and moving models. The approach we chose enabled us to allocate one sub-object for each message to be sent by the segment, so that we could call the subprograms in a sensible and consistent order:

(a) Copy in the data needed by the sub-object from the Virtual Network or from other local sources.
(b) Compute the sub-object's outputs
(c) Send the output message to the Virtual Network

Code similar to the example above was generated for each sub-object. The approach we chose was also a trifle faster, since it avoided one level of subprogram call. With respect to adaptability, we saw no difference between the approaches, and if the alternative were chosen, the impact on the adaptable design would be very small.

The second design concept followed directly from the design rules of MODSIM: data objects are stored only in the scheduler body, and no object communicates with another object except through its subprogram parameters. This is another case where MODSIM is identical to the Structural Model. Figure 3 illustrates this design.

The next step in moving code into the Environment segment was to determine the inputs and

Ada Code:

```
From_Net.Copy_In_Atmosphere(Atmosphere_Ins);
Atmosphere.Compute_Atmosphere(
    Atmosphere_Ins, Atmosphere_Outs);
To_Net.Send_Atmosphere(Atmosphere_Outs);
```

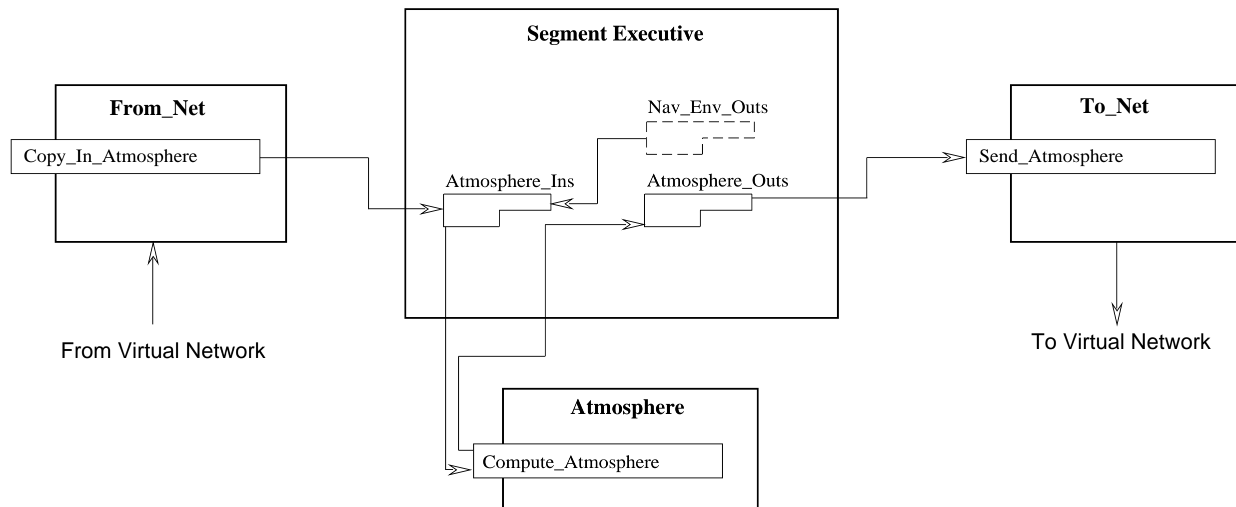

Figure 3 MODSIM Dataflow Design

outputs for each sub-object. Where the code had already followed the rules above (no inputs or outputs except through parameters), the work was already done. The Atmosphere software from Flight Dynamics, however, produced a special challenge. This code was taken from an elderly F-16 simulator in Fortran which used common blocks for data transfer. It was necessary to go into the code, line by line, and determine what the Atmosphere sub-object's inputs and outputs were. Thus, we learned afresh what we had already known: code that does not follow the Structural Model/MODSIM rules is not reusable in any strong sense.

Send-On-Change messages are a characteristic of the MODSIM design for discrete events. For example, a Collision message is never sent unless a change in collision status occurs (the vehicle has either collided with something, or the vehicle is no longer colliding with something). The Collision object software determines whether or not a new message needs to be sent, and sets the flag accordingly in its output interface. The pseudocode above becomes:

(a) Copy in the data needed by the sub-object from the Virtual Network or from other local sources.

(b) Compute the sub-object's outputs

(c) If the sub-object says that a new message should be sent, send the output message to the Virtual Network

Note that Send-On-Change is a property of the message itself, and not of the implementation. Therefore, in every conceivable MODSIM simulator we will use the pseudocode above for send-on-change messages and we will use the pseudocode described previously for data which is sent regularly (iterative messages).

Another data item that is sometimes required in an output interface is an inter-object output. These are not common. It turns out that if an object has data to communicate with another object in the same segment, it usually needs to communicate this data to other segments as well. However, this is not always the case, and when objects require exclusive communication with other objects within the same component, this data becomes part of the output interface for the sub-object that computes the data.

In Figure 3 we show a hypothetical example where the Navigation Environment sub-object has computed a result needed by the Height Above Terrain sub-object. In this case, data is simply transferred from the outputs of Navigation Environment to the inputs of Height Above Terrain.

Applying the Decision Model to Components

The goal of the pilot program was to validate that the Synthesis process could be applied to the simulation arena. This done by using the sub-objects and design rules we have described as building blocks and by constructing a good decision model and a thorough set of process requirements. Thus, the proper objects and interfacing will be built into the required component. The Application Engineer simply applies the specific product requirements to determine what work products are produced.

In order to make code adaptable, some sort of macro language needs to be embedded in the Ada source code. This macro language reflects the decision model derived for the domain. In the following pseudocode example, the pseudo macro language is preceded by "ADAPT":

```
ADAPT if THE_ENVIRONMENT_SEGMENT_COMPUTES_ATMOSPHERE
From_Net.Copy_In_Atmosphere(Atmosphere_Ins);
call Atmosphere(Atmosphere_Ins, Atmosphere_Outs);
To_Net.Send(Atmosphere_Outs);
ADAPT endif
```

The effect of this macro language is to generate the source code calling the Atmosphere routine and sending the message. The Application Engineer determines what type of Atmosphere (i.e. constant, standard day, or adjustable) that is required for the Environment segment.

Thus, we can see very broad outlines of one approach to automating code adaptation now: the Application Engineer uses some mechanism (perhaps an X-Windows-based script) to generate decision variables, and when the decision variables chosen are applied to the adaptable source code files, the macro language processor generates the appropriate adapted source code for our Environment segment.

We chose the awk [Aho88] macro language to implement our decision rules in the Environment code. We chose awk because it is free on Unix systems, it is non-proprietary, it supports iteration, and it is at least a de-facto open-systems standard.

Notice also that the very same mechanism applies to the documents required for the program. A very simple-minded example for a System Requirements Specification might be like this:

The Environment segment will contain the following objects:

(a) Environment support

```
ADAPT if THE_ENVIRONMENT_SEGMENT_COMP-
UTES_ATMOSPHERE
```

(b) Atmosphere

```
ADAPT endif
```

The point is that both code and document adaptation can certainly be automated. All that is required, with a domain specific decision model, is that the decision model be created, and that the Application Engineer be given some means of making his or her choices in the decision model for the particular simulator.

CONCLUSIONS

Our main conclusion is that the Synthesis process is effective when applied to real-time vehicle simulators. This process supports the main tenets of the STARS reuse philosophy.

The pilot program has demonstrated that for code to be strongly reusable, it must follow the design rules typified in the MODSIM and Structural Model architectures. A considerable amount of work must be performed on the old style of common-block FORTRAN code to turn it into reusable components. However, the investment in reuse will pay off very quickly when the customer changes his mind about some part of a component's functionality and you, in a matter of minutes or hours, generate a new component and all the associated documentation.

The MODSIM architecture is almost ideally suited for adaptability and reusability and when used in conjunction with the Synthesis process most of the work in defining the domain is already done. Having the interfaces between segments defined greatly reduces the time required to assemble the assumptions and therefore is the key to a successful domain definition, the decision model. This process does require the additional definition of object and sub-object interfaces in order to complete the decision model However, the majority of these interfaces are already defined for each MODSIM segment and all that is required is to establish what objects or sub-objects generate and use the data.

The "naive" kind of reusability is still worth pursuing. For example, the MODSIM executive was structured so that segment executives for the various segments would have all but one package body in common. The non-naive adaptability model of reuse was harder to implement. The Scheduler and the object packages were redesigned several times in the pilot program. For example, it

is certainly possible to migrate the decision model down into the lowest level of the code in each object. We chose instead to keep as many decisions at the Scheduler level as we could. Our principal criterion was to minimize the amount of adaptability macro code. The design that resulted from this choice was cleaner and probably clearer than previous segment designs. Thus, a kind of secondary reusability fell out of our work with Synthesis.

Domain Engineering is much like expert system development in that there is no substitute for domain knowledge. In the flight simulation domain, it helps tremendously if you have worked on a lot of simulator programs. We were constantly seeking help from people knowledgeable about the domain and recognized that future iterations of the Synthesis process would be aided by more domain expertise.

The construction of the decision model and the process requirements is easier when you keep in mind who is to be making the decisions. Good systems engineering practices also help the success of the decision model in representing the possible family of components within a domain.

The success of the pilot project in producing a roadmap with which to build families of segments, the generation of a working Environment component, and this paper give firm evidence to the STARS philosophy that geographically distributed people can work well together as a team.

In order to produce consistent components across multiple domains (e.g. an Environment segment, an Instructor/Operator Station, etc.) a macro language must be chosen and adhered to. If you use the same tools and style when defining each domain and component, it will be much easier to address changes to the domain in the future. Also, the system will be more maintainable if you chose non-proprietary tools and macro language.

REFERENCES

[Griswold91] Griswold, William G., "Will an Interpretive Language Speed Up CAD Application Development?", Posting to Usenet, 7 December 1991.

[Sink91] Sink, Eric Wayne, "Reserve Demobilization System Built Around Reuse", Posting to Usenet, 18 June 1991.

[Ausnit85] Ausnit, Christine, Christine Braun, Sterling Eanes, John Goodenough, Richard Simpson, Ada Reusability Guidelines, April, 1985, SoftTech, Inc., Waltham, MA.

[Davis92] Davis, Margaret J., Boeing/STARS Program Reuse Technical Lead, personal communication (email) May 12, 1992.

[Jones91] Jones, Capers, Applied Software Measurement, McGraw-Hill, 1991.

[Pintado90] Pintado, A. Xavier, "Selection and Exploration in an Object-Oriented Environment" in Tsichritzis, D. C., Object Management Geneva: Centre Universitaire d'Informatique, July 1990.

[MSDP91a] Modular Simulator Design Program, Phase III, Part 2 Final Report, August 91, The Boeing Company, D495-10437-1.

[MSDP91b] System/Segment Specification for the Generic Modular Simulator System, Volumes I-XIV, June 1991, The Boeing Company, S495-10400.

[Aho88] Aho, A. V., Kernighan, B. W. & Weinberger, P. J., The Awk Programming Language, Addison-Wesley, 1988.

[SPC92] Introducing Systematic Reuse To The Command And Control Division Of Rockwell International, May 91, Software Productivity Consortium, SPC-9202-N.

[SPC91] Synthesis Guidebook Volume I Methodology Definition, December 91, Software Productivity Consortium, SPC-91122-MC.

[TRF291] TRF2 Metaprogramming Tool User Guide, August 91, Software Productivity Consortium, SPC-91132-MC.

ACKNOWLEDGMENTS

We are grateful to the following engineers at Boeing S&TS who shared their expertise with us, greatly improving the quality of our decision model: Alan John Hicks, Gary Kamsickas, William Tucker, C. E. Peterson. Thanks are also due to Nich Ruprecht of the Institut fuer Informatik der Universitaet Freiburg who, in response to a request for help on Usenet, provided the awk program that we used for adaptability. We would also like to thank K.C. King, Margaret Davis, Jeff Facemire, and Rich McCabe for their insightful comments and suggestions.