TEAM SIMULATOR DEVELOPMENT: REUSEABLE ADA SIMULATION INTERFACES

by
Gary M. Kamsickas
Boeing Military Airplanes
Simulation and Training Systems
Huntsville, Alabama

ABSTRACT

The team development of a flight simulation device requires interfaces which contain enough detailed information and stability to allow each team member to independently develop and test their portion of the simulation device. The interfaces must be easy to understand and interpret in order to provide a high level of system maintainability and minimize interaction among team members. This paper discusses a methodology, developed as part of the Modular Simulator Design Program, a Tri-Service research and development project administered by the United States Air Force, which illustrates the steps involved in creating generic, reusable interfaces that can be used in a team development effort. This methdology uses features of the Ada programming language, such as strong typing and specific data structures, to solve problems encountered in reusable interface development. By using such a methodology in the development of standardized, reusable interfaces, a more cost-effective system can be provided for the user.

INTRODUCTION

In the team development of a flight simulator the interface between the activity of each development team is the most important aspect in building the simulation device. Unfourtunately, at the start of a program the interfaces are usually the most abstract and least known quantity of the project. This is because developing good simulation interfaces is a time-consuming, labor-intensive endeavor that requires a thorough understanding of the device being developed. If sufficient time and effort is not allocated to the design and development of the interfaces, the result can be interfaces which represent a source of confusion for team members instead of a source of enlightenment. If this occurs, the program could be hurt from the start of design until the completion of system integration.

A frequently used method for developing simulation interfaces, is to use previosly defined interfaces as a baseline. The basic desire for reusability is thereby established. This approach creates a problem in that these interfaces are not generic in design. If an interface is to be used from application to application, that interface must be designed to be as generic and reusable as possible. At the very least a reusable methodology for defining the interface must be established. Just as important, the interface must be easy to understand and interpret. It must provide a high level of system maintainability and a minimal amount of interaction among team members.

This paper discusses a methodology which was developed as part of the effort on the Modular Simulator Design Program (MSDP), a Tri-Service research and development project administered by the United States Air Force under contract number F33657-86-C-0149. This methodology utilizes features of the Ada programming language helpful in designing a generic, reusable interface. Important steps involved in creating an interface that contains enough detail and stability to allow for efficient team development of a simulation device are discussed, along with problems encountered in the development of reusable simulation interfaces. Although this methodology was developed for the MSDP, the majority of the concepts can be applied to the development of any simulation device interface.

WHAT IS THE INTERFACE ?

The interface is the highest level of communication among simulation math model software programs developed by team members. It defines the information requirements needed by the team members' software to perform its required functions. In the case of the MSDP, the team members developed loosely coupled, independent "Modules" which represent the functionality of a generic Weapons System Trainer. The hardware and software components of each of these modules could be dissimilar to other modules in the system. The modules communicate via a global bus configuration as shown in Figure 1. These modules are independently developed and represent, as a whole, the

functionality of the simulation. In order for the modules to interact and perform their respective functions in such a distributed system environment, the interfaces must be completely defined and strictly enforced.

In a distributed environment which uses a global bus architecture, every aspect of the interfaces must be defined. These aspects include data types, engineering units, data transmission rates, source and destination, and the grouping of data for transmission. The majority of these aspects also apply to a non-distributed environment, therefore making the interfaces reusable for a variety of applications.

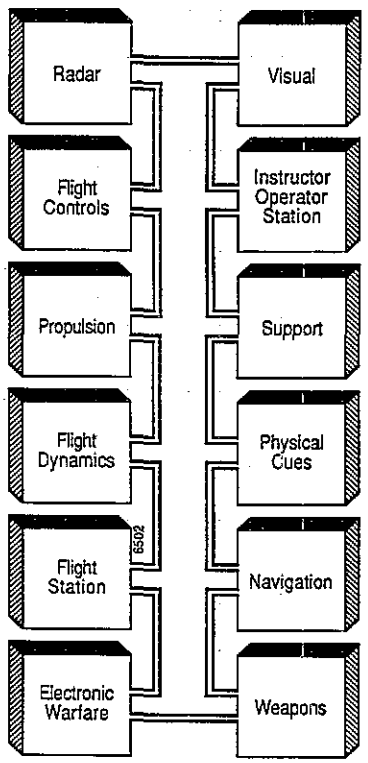

Figure 1
Modular Simulator Design Program
Module Allocation

WHY IS A REUSABLE INTERFACE IMPORTANT ?

There are several reasons for developing a reusable interface but their common factor is cost. The purchaser of the simulation device, which in most cases is the user, wants to pay less. A reusable interface can reduce cost in two major areas: 1) the elimination of repeated development effort associated with a non-reusable interface and 2) the elimination of costs associated with the use of a poorly defined interface throughout the program. Of these, the second usually causes more damage to a project. A substantial amount of effort is wasted in communicating with team members to resolve confusion about the interfaces. An even greater cost is the expensive task of repairing interface misunderstandings during system integration and test. The cost associated with these areas can be significantly reduced by a clear and completly defined reusable interface. This will assist in meeting the users need for a lower cost simulation device.

HOW SHOULD THE INTERFACE BE DEFINED ?

Traditionally, simulation interfaces are simply described using the English language. This method can be ambiguous or difficult to interpret because English has an endless number of interpretations and rules. For example, the word `set' has over 100 different definitions. In an attempt to alleviate this problem, English interfaces have been supplemented with functional and physical diagrams or figures similar to computer language constructs. This can provide another source of error and confusion when a figure does not match its corresponding written definition. Even worse, there is no easy way to detect these errors or check the consistancy and quality of the interface.

Unlike the English language, computer languages operate on discrete and defineable rules of interpretation. These rules can also be interpreted and checked by a compiler. However, many computer languages are cryptic and difficult to understand, read, or use, which only adds to the problem. What is required is a computer language which has the basic universal readability of English but uses that English as a controlled and clearly defined symbology. The Ada language can support this requirement. In addition to providing a defined substitute for English, Ada is a controlled language. It is governed by a single specification (MIL-STD-1815) and therefore allows for a single interpretation of each interface's meaning. Ada compilers must be validated against this same specification, ensuring that each compiler interprets each statement as specified. The Ada compiler can then ensure that the interface is syntactically consistant, correct, and not ambiguous. Ada compilers can also provide a means of configuration control among team members. If the team members are required to compile against the established Ada interfaces, the compiler will enforce the interfaces. Even slight violations will be immediately apparent when the interface fails to compile into the system. The Ada compiler calls attention to the error enabling it to be corrected.

Using Ada to define the interface will

also allow the use of many unique data structures and design techniques, such as strong data typing, information hiding, and recompilation rules, that can promote a more generic, reusable interface. By using Ada, the interface will also be compatible with future simulator designs which are developed using the Ada language.

THE METHODOLOGY

The methodology used to develop a reuseable interface must encompass the concepts of and tradeoffs between maintainability, understandability and definition of the generic properties of the interface. One goal in establishing a reusable interface is to minimize changes. It needs to be recognized that even the best design will require some 'tuning', therefore maintainability must be a key consideration in the interface design. However, if the number of changes is minimized at the expense of change complexity, then maintainability is sacrificed and the interface is less likely to be reused or used at all on any project.

In order for an interface to be easily maintained it must be easy to understand. The methodology presented in this paper promotes understandability by using common data formats and structure throughout. This is then enhanced by using simple design concepts. If understanding the interface is accomplished, maintainability is increased significantly.

Again it should be noted that Ada promotes both understanding and maintainablity because it allows for very readable code and is controlled by a specification which defines its syntactical meaning.

The concept of promoting maintainability is also closely linked to the generic properties of the interface. A technique used to promote maintainability is to physically isolate those areas of the interface that have a high probability of change from application to application. These areas are the application specific portions of the interface. By identifying and separating the specific properties from the common properties of the interface, a generic interface is obtained. When this is accomplished the interface can then be reused on a specific application by simply modifiying the application specific portion of the interface.

The methodology discussed in the following paragraphs concentrates on developing an interface which is maintainable and reusable by making the interface easy to understand and by isolating the specific properties of the interface from the generic properties of the interface. The flow of the steps involved in defining a reusable interface is illustrated in Figure 2.

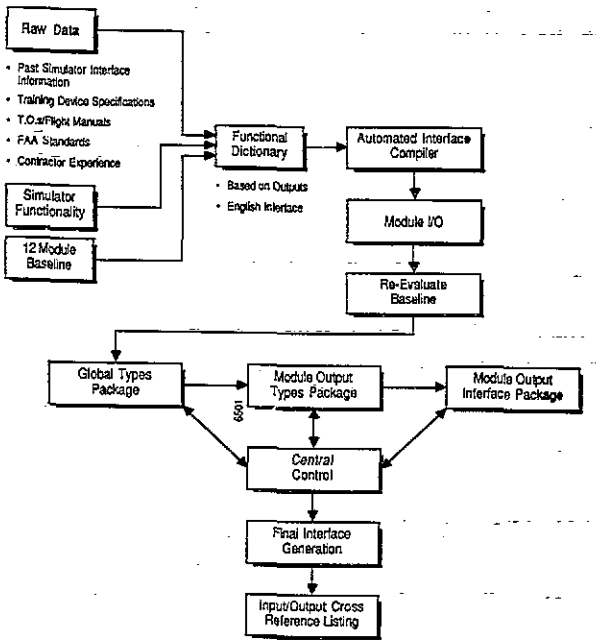

Figure 2
Reusable Interface Development Process

Establishing a Baseline

In order to establish generic and reusable interfaces, a set of common elements must be defined which represent the system as a whole and any system like it. For this discussion a Flight Simulator will serve as the system. The MSDP started with the twelve element system shown in Figure 1. In the MSDP, these elements are refered to as "Modules", they represent a high level of division or abstraction of the elements in a typical Weapon System Trainer. Optimally the number of elements in the system should be kept low and should represent some high level logical division of the system. The elements should also exhibit a minimal amount of connectivity.

It should be noted that there are many existing methods and theories, such as Object Oriented Design, Data Flow Analysis,and Functional Allocation, which can be used to partition a system. The MSDP used a combination of Data Flow Analysis and Functional Allocation that was based on the initial twelve module starting point. Regardless of the method used to partition the system, the initial partitioning of the elements may not be the best division for the system, but is required to establish a working baseline. After the functionality and structure of the system is throughly analyzed it may be apparent that the elements must be redefined to accommodate a better allocation.

Determining Functionality

In order to determine the best allocation of the elements in a generic system, several specific 'representative' systems must be analyzed. In the case of the MSDP, each system was divided into low-level discrete functions which represented and completly defined that system. The interfaces for each one of these low-level functions was also determined. Each function was then analyzed to detemine which element of the generic system would be responsible for that functionality. At first thought it would be considered prudent to select the element with the least number of interfaces thus minimizing communication between elements. However, other considerations should be made such as eliminating redundant interfaces, keeping common areas of expertise in one element, and locating databases. This may seem like a 'bottom-up' approach to the design but this method allowed for the specific properties of several model systems to be organized. If a top-down approach was taken in analyzing each existing representative system, it would have been difficult to divide the systems into the elements used for the baseline due to differences in existing design structure. It was much easier to define the existing systems in terms of their lowest functional entities. These entities could then be analyzed and distributed among the baseline elements of the generic system.

The analysis of the functionality data flow is a very slow and tedious process which requires vast simulation knowledge. The MSDP employed four subcontractors, who specialized in certain areas of simulation, and used inputs from simulation industry and government representatives to accomplish this task. It should be noted that a large portion of the effort expended on the MSDP entailed data flow analysis and functional allocation for the modules.

Reviewing the Baseline

After each element is established, the baseline must be re-evaluated. On the MSDP it was determined that the Support module did not exhibit enough unique functionality to remain a separate element and was combined with the IOS module.

When the baseline is finalized the external interfaces of each element must be determined and recorded. On MSDP the format in Figure 3 was used to record the interfaces and description of each low-level function. These functions and interfaces were then stored in a Functional Dictionary.

On the MSDP an automated tool was used to aid in establishing the interface baseline. This tool was an Artificial Intelligence device which linked interfaces to functions and functions to modules. This tool was used to shuffle functionality among modules to optimize data flow and functional allocation while maintaining the consistancy of the interfaces.

```
Module: << insert module where function resides >>

Function: << insert name of function >>

Function Description: << insert description of function >>

Assumptions: <<insert data flow and allocation assumptions>>

Specific Mission Generation Requirements:
   << insert mission specific qualities of the function >>

Inputs: << list inputs required by the function >>

Outputs: << list outputs provided by the function >>


Output Definitions

Output Name:   << insert identifying name of output >>
Definition:    << insert definition of output >>
Destination:   << insert destinations for output >>
Data Size:     << insert size of output data >>
Update Rate:   << insert calculation frequency of output >>
Data Type:     << insert data type of output >>
Units:         << insert engineering units for output >>
```

Figure 3

Functional Dictionary Entry Format

Defining the Basics

In order to completely define an interface two basic properties of the data used in the interface must be specified. These properties are the basic machine characteristics of the data used among the elements and the engineering units which these data quantify.

All data used in a system, such as floating point numbers, exponents, bit orders, etc., have unique characteristics. These characteristics are usually not as important when operating in a single host computer environment since all software is using that machine's format. However, in a distributed environment which could contain computational systems which use dissimilar data formats, a standard set of data formats must be defined. By using the Ada representation specification feature, a set of base data types can be defined as shown in Figure 4. If all interfaces are defined using the base data types, the interface can quickly accommodate any changes in data format by simply modifying the base data types, thus making the interface much more reusable and maintainable. When using the distributed environment interface in a single host computer environment the base data types can be easily modified as shown in Figure 5. This accommodates a specific machines data format by removing (commenting out) the representation specifications and changing the Ada types' to subtypes' of the machine's predefined data formats.

```
Bit : constant := 1;

type Mod_Sim_IEEE_Float is digits 6;

type Mod_Sim_IEEE_Long_Float is digits 15;

type Mod_Sim_Integer is range (-2**15)..((2**15)-1);
     for Mod_Sim_Integer'Size use 16 * Bit;

type Mod_Sim_Short_Integer is range (-2**7)..((2**7)-1);
     for Mod_Sim_Short_Integer'Size use 8 * Bit;

type Mod_Sim_Boolean is ( True, False );
     for Mod_Sim_Boolean'Size use 8 * Bit;
     for Mod_Sim_Boolean use ( True  => 16#00#,
                               False => 16#FF# );
```

Figure 4

Typical Base Data Types for a Distributed Environment

```
-- Bit : constant := 1;

subtype Mod_Sim_IEEE_Float is Float;

subtype Mod_Sim_IEEE_Long_Float is Long_Float;

subtype Mod_Sim_Integer is Integer;

--    for Mod_Sim_Integer'Size use 16 * Bit;

subtype Mod_Sim_Short_Integer is Short_Integer;

--    for Mod_Sim_Short_Integer'Size use 8 * Bit;

subtype Mod_Sim_Boolean is Boolean;

--    for Mod_Sim_Boolean'Size use 8 * Bit;

--    for Mod_Sim_Boolean use ( True  => 16#00#,
--                              False => 16#FF# );
```

Figure 5

Modified Base Data Types for a Single Host Environment

In many flight simulator developments a common area of confusion lies in the definition of data units. In order to completely specify an interface, the engineering units of the data being passed among team member's elements must be specified. In most cases units are usually given with a comment in the interface. By making use of the Ada 'subtype' feature, a set of units can be defined that can be reused or added to as required, thereby enhancing maintainability and reducing confusion. As shown in Figure 6, 'base units' are determined, in this case Pressure, Temperature, Linear_Velocity and Length, and defined as a subtype of a previously defined base data type. Using the base unit, the actual units can be defined as a subtype of the base unit. The unit description can also be enhanced further using comments if desired. Some data elements cannot be expressed in this fashion; such as percents, ratios and normalized numbers, these can be expressed as shown in Figure 7. It is also common to have data elements which represent common groupings of data items these can be created as shown in Figure 8, they are also reusable. These units can then be used in defining the data contained in the interfaces.

```
subtype Pressure is Mod_Sim_IEEE_Float;
-- pressure units

subtype PSI is Pressure;
-- pounds per square inch
subtype Inches_Hg is Pressure;
-- inches of mercury (barometric pressure)


subtype Temperature is Mod_Sim_IEEE_Float;
-- temperature units

subtype Degrees_F is Temperature;
-- degrees Farenheight
subtype Degrees_C is Temperature;
-- degrees Centigrade


subtype Linear_Velocity is Mod_Sim_IEEE_Float;
-- linear unit/unit time

subtype Feet_per_Sec is Linear_Velocity;
-- feet/second
subtype Knots is Linear_Velocity;
-- nautical miles/hour


subtype Length is Mod_Sim_IEEE_Float;
-- linear units

subtype Feet is Length;
-- feet units
subtype Nautical_Miles is Length;
-- nautical mile units
```

Figure 6

Engineering Data Unit Definitions

```
subtype Ratio is Mod_Sim_IEEE_Float;

subtype Percent is Mod_Sim_IEEE_Float range 0.0..100.0;

subtype Normalized_Number is Mod_Sim_IEEE_Float
                             range 0.0..1.0;

subtype Signed_Normalized_Number is Mod_Sim_IEEE_Float
                                    range -1.0..1.0;
```

Figure 7

Unitless Engineering Data Definitions

```
type Linear_Position_Components is
 record
   Longitudinal_Position : Feet;
   Lateral_Position : Feet;
   Vertical_Position: Feet;
end record;

type Linear_Velocity_Vector is
 record
   Longitudinal_Velocity : Feet_per_Sec;
   Lateral_Velocity : Feet_per_Sec;
   Vertical_Velocity : Feet_per_Sec;
end record;
```

Figure 8

Common Data Grouping Definitions

Developing the Reuseable Interface

Generic, Reusable Data Structures

Using the data units, subtypes and types, each team member must determine how to describe their element's interfaces in a generic manner. In order to accomplish this, the specific attributes of the interface must be separated from the generic attributes of the interface. This is accomplished by using enumeration types, a feature of the Ada language, to form a list. The enumeration types are used to contain and isolate the specific attributes of the interface. The enumeration types are then referenced to add the specific quality to the generic interface structures. This is best described using the simple example shown in Figure 9.

```
type Aircraft_Fuel_Tank is                    -- Specific Part
     ( Left_Inboard,
       Center,
       Right_Inboard,
       Reserve_1,
       Reserve_2 );


type Fuel_Tank_Data is                        -- Reusable Part
 record
   Fuel_Tank_Temperature : Degrees_F;
   Fuel_Tank_Quantity : Gallons;
   Fuel_Tank_Flow_Rate : Lbs_Per_Hr;
   Fuel_Tank_Centroid : Linear_Position_Components;
end record;

type Fuel_Tank_Data_Array is array ( Aircraft_Fuel_Tank )
                          of Fuel_Tank_Data;
```

Figure 9

Simple Reusable Data Structure

An element common to all aircraft is the fuel tank. However, each type of aircraft has a different number of fuel tanks and each fuel tank has a specific name. But there is usually common data about a fuel tank such as fuel temperature, fuel tank quantity, consumption rates, and the centroid of the fuel in the tank which are required in any application. An enumeration type is created which contains the specific tanks for the aircraft, these are the specific attributes. The generic data structures are then defined based on the enumeration type. By using this method effectively, an entire set of interfaces can be designed. When an interface for a new application is required the enumeration types are changed to reflect the new configuration. It should be noted that this is a very simplified example and much thought is required to design an entire set of interfaces. The designer must consider which interface aspects are truely generic and which are truely specific. Consideration must also be given to how the interface will be used and its requirements.

Data Transmission

In a distributed environment with a global bus architecture such as the MSDP, data must be transmitted or sent between the elements of the system. In single host systems the data can be transmitted by other methods such as shared or reflective memory or simply parameter passing. In the simulation devices in use today, data for the simulation models is updated at a specific rate which is dependent upon the required fidelity of the model. In building a generic/reuseable interface it is important to define the rate at which data will be updated between development teams to avoid data transmission problems. This will alliviate the problems associated with data being updated and passed but not used, gaps appearing in data transmission, and data not being updated often enough, causing glitches in the simulation.

While reviewing the interfaces for the simulation device functionality it was discovered that the maximum update rate for data in many simulation devices was 60-64 Hz. Higher data rates were sometimes found when special rates were required for driving specialized hardware items. In order to accomodate the majority of simulation devices a maximum data rate of 64 Hz was established with lower rates of 32, 16, 8, and 4 Hz. However, it is still allowable to have specialized rates for a specific application if required.

A portion of data used in a simulation device changes very infrequently. Flight deck crew inputs or inputs from the instructor fall into this category. In order to reduce unnecessary data transmission, this data requires transmission only when it changes state. This data is given a special rate called send-on-change'.

Data Grouping and Format

Grouping data for transmission to other elements in the system is important for

several reasons. The first is that it significantly reduces the complexity of the interface by reducing the number of individual interfaces to be tracked and processed throughout the system. It is obviously much harder to keep track of 500 interfaces than 50 packets of data. This type of grouping adds a layer of abstraction to the interface which makes it simpler to understand. Another reason to group data is for run-time efficiency. In many cases it is much more expediant to pass a group of data as a single parameter than to pass several single data elements. The best reason for grouping is that if it is done effectively it can make the interfaces much easier to maintain and understand.

Each system has certain design requirements or limitations to meet and therefore different criteria for grouping. The grouping method used for the MSDP will be discussed along with how it might apply to any team development environment.

MSDP data elements were grouped into packets for transmission in order to more efficiently make use of the available bus bandwidth. Grouping data elements reduces the amount of bus bandwidth that is used for data transmission overhead.

In order for grouping to promote reusability, it must follow a set of logical rules. Three principle areas of grouping should be considered: 1) by function, 2) by data update rate and 3) by data destination.

Each element in the system supports a defined set of functions logically grouped to form that element. The functions require input and output from other functions both internal and external to the element. If a certain function was not required for some application, it is advantageous to know its interfaces and group them so that they may be removed easily along with the function. Therefore, for the sake of maintaining the reusable interface, the primary grouping of data should be by function.

Since each function is capable of producing or updating data at various rates, it is recommended that the functional groupings be subgrouped by data update rate. If data are not grouped by update rate then all data for that function would be transmitted by the highest update rate for that function. Slower data would then be calculated and transmitted more often than required. In the case of the MSDP this would also waste a large amount of bus bandwidth. In any system this wastes computational time. It is not efficient to supply data to an element that will not use it when it is supplied. By allowing this to occur, a great deal of execution time would be spent transmitting useless data around the system.

The third grouping, by destination, was ruled out for the MSDP. The argument for further grouping by destination was driven by the concepts of data hiding and data visibility. These concepts contend that data should only be sent or visible to those modules or elements that require it. The major drawback to this extra grouping for MSDP is that data would be repeated within various output packages because several modules could require the same single piece of data. This would cause the bus to be over utilized by passing redundant data. In addition, when using a global bus architecture as in the MSDP, data access is controlled by making controlled data transmissions among the modules. In this environment modules only receive data that is passed directly to them and cannot gain access to other data.

In a single host environment, where data hiding is essential for control of the system, it is highly recommended that data be grouped by destination. The price paid in additional execution time is small compared to the cost of having uncontrolled data visibility in the system.

An exception to grouping is send-on-change data. This data, as its name implies, is only transmitted when it changes. Thus significantly reducing bus traffic in the case of the MSDP by not transmitting redundant data.

An Ada record structure template implementing the data grouping rules can be developed for all team members to use in defining their interfaces. The template increases maintainabliity by again creating commonality in the interface. The template used for the MSDP is shown in Figure 10.

It should be noted that it is only necessary to define the input side or the output side of the interface. Defining the interface by the inputs creates a fundamental problem. If there is more than one element requiring the input, the team members responsible for defining that input could define different input formats, thus confusing the team member responsible for providing the output. Therefore, the interfaces should be defined by output to avoid confusion and reduce redundant interface definitions.

```
type <<insert function name>>_<<insert data update rate>> is
 record
   << insert record field for data element >>;
   -- << insert Destination Codes >>

    .
    .
    .

end record;

--Destination:<<insert Destination Codes for entire record>>

Destination Codes:   PRO = Propulsion
                     IOS = Instructor/Operator Station
                     PHC = Physical Cues
                     WPN = Weapons
                     EW  = Electronic Warfare
                     FS  = Flight Station
                     FD  = Flight Dynamics
                     FC  = Flight Controls
                     RDR = Radar
                     VIS = Visual
                     NAV = Navigation/Communication
```

Sample Template Populated

```
type Atmosphere_16Hz is
 record
   Air_Density_Ratio:              Ratio;        --NAV,PRO
   Ambient_Air_Pressure:           PSI;          --NAV,FS,PRO
   Ambient_Air_Temperature:        Degrees_C;    --NAV,FS,PRO
   Dynamic_Pressure:               PSI;          --NAV,FC
   Impact_Pressure:                PSI;          --NAV
   Pressure_Altitude:              Feet;         --NAV,PRO,FC
   Sea_Level_Barometric_Pressure:  Inches_Hg;    --NAV
   Static_Pressure_Ratio:          Ratio;        --NAV
   Total_Air_Pressure:             PSI;          --NAV
end record;

--Destination: NAV, FS, PRO, FC
```

Figure 10

Data Grouping Template Example

Developing a Package Structure

An important feature of the Ada language is the package structure. The package allows for a convienient method of grouping information. In order to provide commonality among team members' interfaces, a global types package needs to be established. The global types package format is shown in Figure 11. It contains the information that all team members need to access. This includes base data formats, data units, common groupings of data types which describe the configuration of the specific aircraft, and interfaces which are used on a system level.

In order to increase maintainability and reusability by isolating the areas of change, the global types package is divided into two major sections, Aircraft/Simulator Specific Types and Aircraft/Simulator Reusable Types. The Aircraft/Simulator Specific Types section contains the data representations which will change from application to application. The Aircraft/Simulator Reusable Types section contains data types that will not change for a specific application.

After the global types package is completed, each team member develops an output types package as shown in Figure 12. This package also employs the same two major sections as the global types package to enhance maintainability. It contains the type definitions for the element's output records in the format shown in Figure 10. In order to keep the package more readable and organized, the output records should be listed by function.

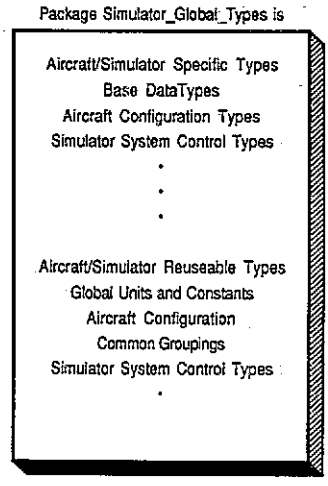

Figure 11
Global Types Package Format

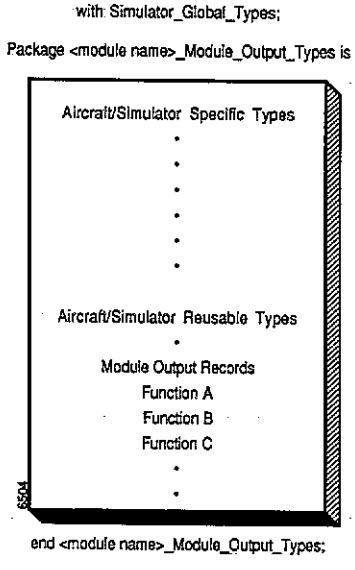

Figure 12
Output Types Package Format

Upon completion of data type definitions and output record structuring, a final package should be created which contains the actual objects or variables of those data types. The format of this package is shown in Figure 13. This package is again organized by function to create commonality and support maintainability. For each function, objects are created for the output record types used by that function. In addition, output objects are also created for send-on-change data provided by the function.

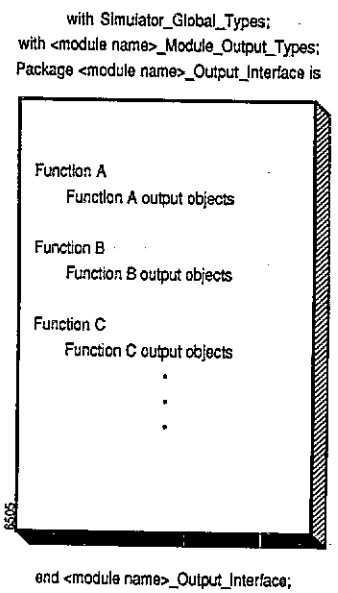

Figure 13
Output Interface Package Format

When all data types packages and objects are defined, the Ada compiler can then be used to perform a final check for any syntactical errors in the interface.

CONTROLLING INTERFACE DEVELOPMENT EFFORT

In a team interface development it is extremley difficult to coordinate team members in a common task. This is due to varying skill levels, misinterpretations, lack of examples, etc. It is important for the sake of maintainability and understandability that each team member strictly adhere to all interface format requirements and concepts. Therefore, a central source of control must be established to coordinate the interface effort. This central source should consist of one person, if possible, that fully understands the interface methodology and development rules. This person is responsible for answering questions from the other team members and coordinating the interface development effort.

LESSONS LEARNED

When using this methodology, and the concepts discussed in this paper on the MSDP, several important lessons were learned.

Specifing the interfaces in Ada proved very beneficial. Initially the interfaces were written in the format shown in Figure 3. It was determined that these interfaces were not as generic as required by the program. Defining the interfaces in Ada involved design work and forced the designer to be concise in the definition. Each team member was forced to think about the interfaces and how they would be used in different applications. By using Ada the interface was designed and refined to meet its requirements and not just thrown together.

The use of a central collection point for the interface development was key to the success of the effort. In addition to serving as a coordinator, this person had access to each team member's interface design. The central coordinator could then pick and choose the best methods for defining data and discuss these methods with all team members to arrive at the best interface possible.

When the interfaces were completed and compiled they consisted of about 100 pages of Ada code. Although the Ada code is easy to read, it is sometimes difficult for a person unfamiliar with the design to locate a single interface. To eliminate this problem a derivitive of the tool used to establish the baseline was built. This tool created an input/output cross reference map for all the interfaces. An example is shown in Figure 14. This reduced the 100 pages of Ada code to about 10 pages of cross reference map which was much easier to use.

Upon review of this methodology, it should be understood that it was developed for the MSDP. If this methodology is used in a non-distributed environment more emphasis should be placed on information hiding. As discussed, the global bus serves as a data control device for the MSDP. In a single host environment this would not exist. Therefore, serious consideration must be exercised in the areas of data grouping and package structure. As an initial suggestion, data should be grouped by destination as well as functionality and data update rate. In addition, the output interface package could also be divided into several smaller packages based on data destination.

	EW	FC	FD	FS	IOS	NAV	PHC	PRO	RDR	VIS	WPN
Record: Atmosphere_16Hz		I	O	I		I		I			
Variable: Air_Density_Ratio			O			I		I			
Variable: Ambient_Air_Pressure			O	I		I		I			
Variable: Ambient_Air_Temperature			O	I		I		I			
Variable: Dynamic_Pressure		I	O			I					
Variable: Impact_Pressure			O			I					
Variable: Pressure_Altitude		I	O			I		I			
Variable: Sea_Level_Barometric_Pressure			O			I					
Variable: Static_Pressure_Ratio			O			I					
Variable: Total_Air_Pressure			O			I					
	EW	**FC**	**FD**	**FS**	**IOS**	**NAV**	**PHC**	**PRO**	**RDR**	**VIS**	**WPN**

Figure 14

I/O Cross Reference Map Example

CONCLUSION

The methodology described in this paper is not a revolutionary concept or collection of new ideas. It simply takes into account the realities of the interface. The most important reality is the need to accept change. Once this fact is accepted the goal is to minimize the impact of those changes and account for their effects.

The development of an interface which is as generic and reusable as possible is essential to the team development of simulation devices. The final interface product of this methodology incorporates the ability to adapt to most applications. In future applications, which are not now apparent, the format of the interface allows for changes to be easily made to accommodate the required data. Through development of a standardized interface as discussed in this paper, the user's needs can be met for more cost-effective flight simulation systems.

REFERENCES

Boeing document number D495-10402-1 Interim Report on Research and Development for Modular Simulator System Phase III Program, Part 1

Boeing specification number S495-10400 System Specification for the Generic Modular Simulator System Volume I, Systems Integration

ABOUT THE AUTHOR

Mr. Gary M. Kamsickas is a Software Systems Engineer with the Simulation and Training Systems organization of Boeing Military Airplanes in Huntsville, Alabama. He has been responsible for software design, code, test and integration on several Boeing simulator projects, including the Ada Simulator Validation Program and the Modular Simulator Design Program (MSDP). He is currently the Chief Engineer for the MSDP and involved in the design validation of the modular simulator concept. Mr. Kamsickas holds a Bachelor of Science degree in Electrical Engineering from Michigan Technological University, Houghton, Michigan.