On Designing Parametrized Systems Using Ada

Michael Stark

Goddard Space Flight Center

1. Introduction

A parametrized system is a software system that can be configured by selecting generalized models and providing specific parameter values to fit those models into a standardized design. This is in contrast to the top-down development approach where a system is designed first, and software is reused only when it fits into the design. The term reconfigurable is used interchangeably with parametrized throughout the paper. This concept is particularly useful in a development environment such as the Goddard Space Flight Center (GSFC) Flight Dynamics Division (FDD), where successive systems have similar characteristics.

The FDD's Software Engineering Laboratory (SEL) has been examining reuse issues associated with Ada from the beginning of its Ada research in 1985. The lessons learned have been applied to operational Ada systems, leading to an immediate trend towards greater reuse than is typical for FORTRAN systems [McGarry 1989]. In addition, the Generic Simulator prototyping project (GENSIM) was a first effort at designing a parametrized simulator system. The lessons learned through the use of Ada and the GENSIM prototype are being applied to the Combined Operational Mission Planning and Attitude Support System (COMPASS), which is to be a reconfigurable system for a much larger portion of the flight dynamics domain. This paper will discuss the lessons learned from the GENSIM project, some of the reconfiguration concepts planned for COMPASS, and will define a model for the development of reconfigurable systems. This model provides techniques for realizing the potential for "Domain-Directed Reuse", as defined by Braun and Prieto-Diaz [Braun 1989].

The major motive for reconfigurable systems in the FDD is cost reduction. Having a well-tested set of reusable components may also increase reliability and shorten development schedules, but cost is the primary factor in this environment. Research done by the SEL indicates that verbatim software reuse (reuse without modification) can produce major cost savings. The cost of integrating a component that is reused verbatim is approximately 10 per cent of the cost of developing a new component from scratch [Solomon 1987]. Analysis done for GENSIM indicated that approximately 70 to 80 per cent of the code could be reused verbatim, and that this should cut simulator development costs in half [Markley 1987].

2. Reconfigurable Systems

This section focuses on the approaches taken and lessons learned from the GENSIM and COMPASS projects. These lessons influenced the reuse concepts and techniques defined in the subsequent sections of the paper.

2.1 GENSIM Overview

The GENSIM project was started in late 1986, and divided into two major phases. The first phase lasted until mid-1988, with the major products being the cost analysis cited above, mathematical specifications, and the high level system design. From mid-1988 to mid-1989 a small development team started implementing prototype software. The project was terminated before the prototype system was completed and evaluated, as COMPASS incorporates simulation requirements into its broader domain. Nonetheless, enough development work was done to learn some useful lessons.

The generic simulator design consists of a set of "modules" that plug into a standardized simulator architecture. Each of these modules was expected to have a corresponding mathematical specification, design data (object diagrams and Ada package specifications), and source code. The use of standardized specifications was intended to prevent the slight differences in specifications that often impede verbatim reuse. In addition, the GENSIM project intended to maintain test plans, data, and software for each module, so that changes in standard modules could be tested rapidly.

The simulator architecture is based on the designs of the first two Ada simulators developed in the FDD. The enhancements

Simulator Architecture

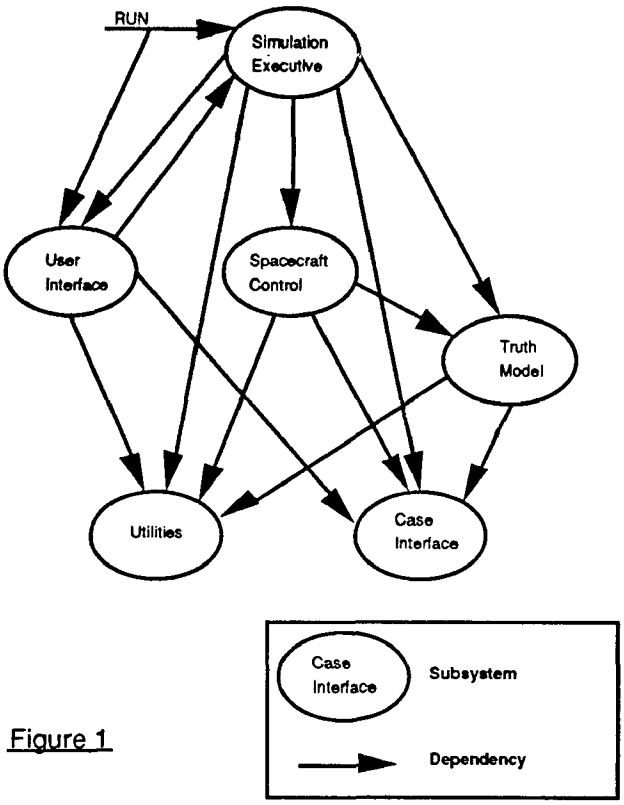


Figure 1

and changes to this architecture were intended to allow different sets of modules to be configured into a system, depending on the simulation requirements for a given satellite. It was possible to generalize the early designs, but because these were early designs, GENSIM incorporated some design flaws, even as others were removed. The major results of GENSIM were

1) The concept of reusing products from all life cycle phases presented no problems, and provided the anticipated benefit of standardizing mathematical specifications. The GENSIM team thoroughly specifies the individual simulator modules. However, the connections between modules were made at design time, despite the fact that they represented dependencies inherent in the problem. Note capturing these dependencies in the specification was not a problem, since the GENSIM team happened to be knowledgable enough to assure that a function needed by one module was provided by another. Nonetheless, problem domain dependencies should preferably be captured in the specifications, so that developers with less domain expertise will have the information they need. The COMPASS team is representing problem domain dependencies in their standardized specifications.

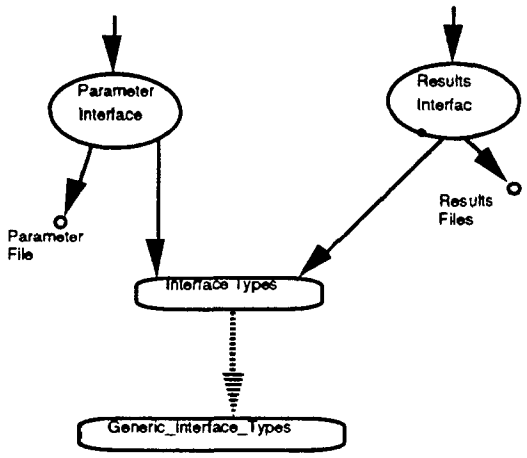
2) The configuration of a system is done by instantiating all the necessary generic Ada packages in the correct order. The GENSIM team instantiated each package as a library unit. In cases where the same set of packages are used in each system, generics can be combined so that a subsystem can be "instantiated" through the instantiation of a single generic package.

3) The legacy of the previous simulator architectures made the implementation of standardized components more difficult. In particular, the storage of inputs and results for a given simulation scenario could not be adequately generalized. This lesson is discussed in more detail in the next section.

2.2 GENSIM as a Standardized Architecture

The purpose of the flight dynamics simulators generalized by GENSIM is to test the flight dynamics control algorithms for a satellite before it is launched. Figure 1 shows the architecture for a spacecraft simulator built from GENSIM modules. This diagram shows the dependencies between major simulator subsystems. The Truth Model represents the "true" response of a spacecraft to its control system, and is configured using the components needed for a specific satellite. The Spacecraft Control subsystem contains new code that implements a particular satellite's control laws. The remaining subsystems are built to support these two subsystems, and must also be configurable to support varying sets of modules. This reconfigurability became especially cumbersome for the Case Interface, which is the subsystem that manages input data and results for simulation scenarios (cases). Figure 2 shows the two major parts of Case Interface. All simulation inputs are managed by Parameter Interface, and all results are managed by Results Interface. These two subsystems are accessed by both the user and the two simulation subsystems.

Case Interface Subsystem



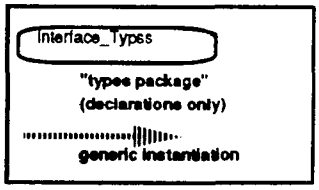


Figure 2

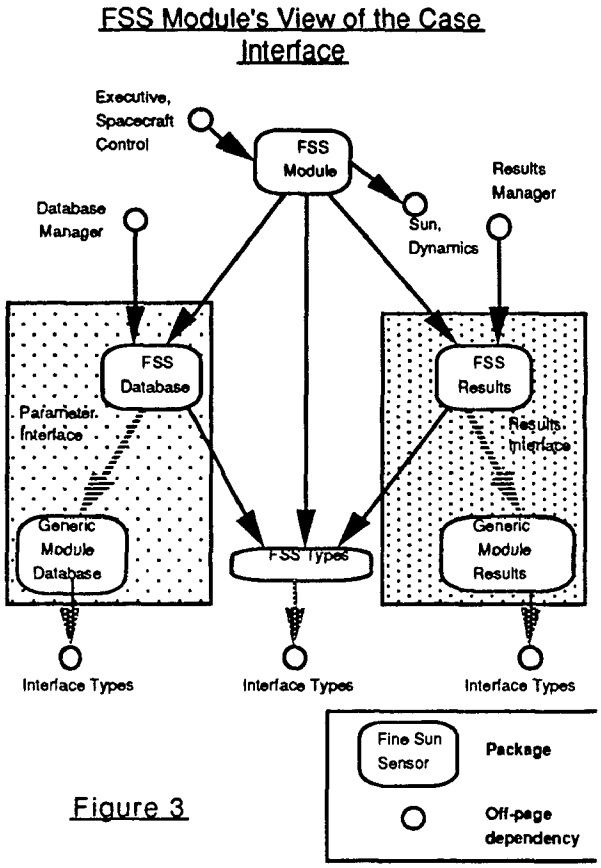


Figure 3

The GENSIM configuration concept called for the subsystems of the Case Interface to be built from components associated with each module. Figure 3 shows how a parameter and results database is created for a Fine Sun Sensor (FSS) module by instantiating standardized generics. The "FSS_Database" package is used by the module's initialization routine to get initial parameters, and the "FSS_Results" package is used by the module's computation routines to store simulated results. The shaded areas show that the individual components fit into the Case Interface packages. Figure 4 shows how several module databases fit into the Parameter Interface subsystem.

The advantage of this approach is that the packages Interface_Types and FSS_Types contain all the declarative information needed to include a module in a simulator configuration, and that standard types and protocols are used to achieve this. The configuration parameters include default values for module input parameters, flags indicating which parameters a user is allowed to change, and similar flags indicating what results a user may display during a simulation or print after a simulation. The disadvantages of this design approach are

1) the developer of a flight dynamics module has to be aware of all the complexities inherent in the simulator architecture, and all the dependencies shown in Figures 3 and 4, and

2) the parameters passed in and out of a package are limited to the data types defined by Interface Types. Module specific enumeration types (such as "type FSS_POWER is (OFF,ON)") cannot passed to the user except by using the 'POS attribute to convert to an integer which is then displayed.

Figure 5 shows an improvement to the architecture that addresses the first disadvantage. The package FSS_ADT exports an abstract data type (ADT) that implements all the modeling of the fine sun sensor. Now the state of the FSS module is based on this abstract data type, and the module's functionality is implemented by calling the operations on the type. This allows package FSS_ADT to be implemented by a developer who is aware of all the nuances of fine sun sensor modeling, and the FSS module can be implemented by a developer who is aware of all the nuances of the simulator architecture. In addition, FSS_ADT and all the other abstract data types defined for the flight dynamics simulation domain can be used to build a system with a completely different architecture, without changing a line of code in the packages that implement the modeling of the flight dynamics problem. An architecture that addresses the limitations imposed by Interface_Types can be built around such abstract data types, as is shown in section 4. The separation of problem domain and system architecture considerations is a key element of the reuse models described in section 3.

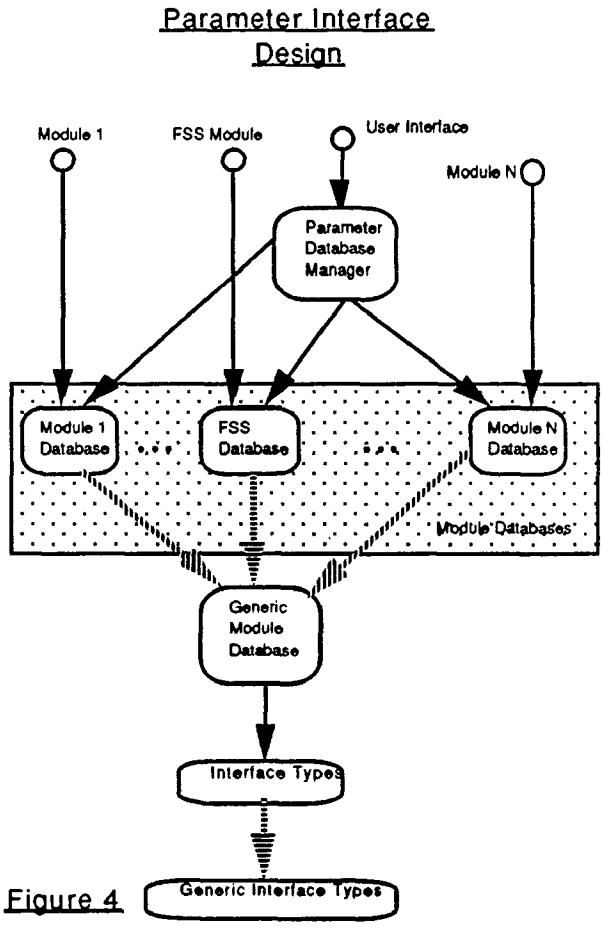


Figure 4

FSS Module's View of the Case Interface

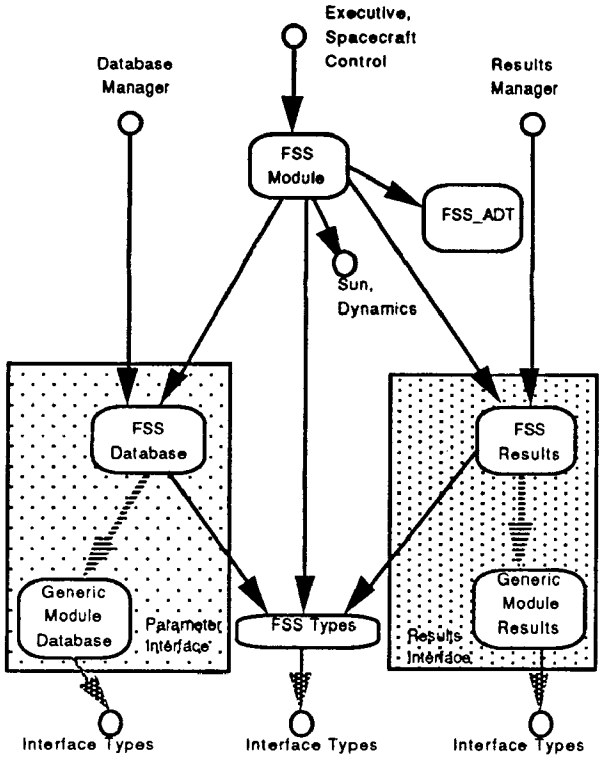


Figure 5

2.3 COMPASS

COMPASS is the second FDD project that is developing reconfigurable software. It has the same cost reduction goal as GENSIM, but covers a much larger problem domain.

COMPASS is intended to support the flight dynamics simulations area, mission planning and analysis both before and after launch, and spacecraft attitude support systems for mission operations. The estimated size of COMPASS is over a million lines (counting all carriage returns) of Ada source code, and is targeted to run on several different computers. This implies both being able to configure systems to run as distributed systems, and to be able to target the same functions to different platforms. These considerations have prompted refinements to the reuse model defined in [Booth 1989].

COMPASS has also involves defining standardized specifications to promote verbatim reuse. Unlike GENSIM, a standard specification methodology has been defined for COMPASS [Seidewitz 1989]. The COMPASS specification concepts are object-oriented, but contain restrictions tied to both reconfigurability and to project standards. For example, there is a restriction on the number of levels of superclasses and subclasses allowed in an inheritance hierarchy.

3. Reuse Concepts

To be able to design reconfigurable systems, it is necessary to have some underlying principles that can be used as design guidelines. The major concept defined in this paper is a Layered Reuse Model that categorizes components by function and defines dependencies among these components. The initial model was developed as a result of the work done on GENSIM and on an operational system, the Upper Atmosphere Research Satellite (UARS) Telemetry Simulator (UARSTELS) [Booth 1989]. This model was primarily driven by the need to separate problem domain and system architecture considerations, as is discussed in section 2. This model does not address how to incorporate very general components that have potential use across several problem domains and/or architectures, nor does it address the separation of system dependent features from potentially portable code. The latter omission became obvious when a multiplatform system such as COMPASS was considered.

The Layered Reuse Model

Major Layers	Levels	Examples
Architecture Levels	System Architecture Templates	Case_Interface
	System Modules	FSS_Module
Problem Domain Levels	Domain Definition Classes	FSS_ADT (Fine sun sensor abstract data type)
	Domain Language Classes	Linear_Algebra
Service Levels	System Independent Services	Booch Components (TM)
	System Dependent Services	DEC math library package

Figure 6

To address the above issues, a "services" layer was added to the model. This services layer is split into a system dependent and a system independent layer. The updated reuse model is shown in Figure 6. A component in a given layer can only depend on components in layers below it, as is the case in any good layered model. The layers are defined as follows:

System Architecture Templates -- Components at this level provide a template into which modules fit. These can be reconfigurable subsystems such as the GENSIM Case Interface discussed above, or they can be standard components that do not depend on the particular configuration. In GENSIM the Display Interface and the Plot Interface were designed to be

standard software, with any needed configuration data being provided by input files, rather than generic instantiation.

System Modules -- This layer contains components that are designed to fit into a standard design. These modules are built from components at the problem domain and service levels.

Domain Definition Classes -- These components define classes in the problem domain that are identified through domain analysis. They are generally implemented as Ada packages exporting abstract data types, as is discussed above.

Domain Language Classes -- Components at this level capture the vocabulary of a particular domain, in other words, these classes capture the knowledge and language that domain experts use to express the specifications for domain definition classes. In the flight dynamics domain, such classes would include "vector", "matrix", "orbit", and "attitude". The domain analyst would use these simpler classes to define more complex classes such as "Fine Sun Sensor".

System Independent Services -- This layer contains components that can be used in implementing both the problem domain layer and architecture layer components. They are usually usable in more than one problem domain and/or more than one system architecture. Components at this level include the generic data structures and tools provided by the Booch Components (TM) [Booch 1987], as well as portable interfaces to general services such as DEC's screen management routines. These portable interfaces can be moved to different computers, and new code or a different commercial product can be used to implement the same functions. Thus one ends up with multiple non-portable implementations of a single abstraction. Calls to this package should act the same, even if they are implemented in a machine dependent manner.

System Dependent Services -- This layer contains all the components that are dependent on a particular computer or operating system. This generally includes all non-Ada code, as most other languages have different non-standard extensions on different machines. This also includes Ada code that incorporates system dependent features such as Direct_IO files created with a non-null FORM parameter. These system dependent features should have system independent interfaces at a higher level.

The improved model takes an object-oriented approach to specifying the problem domain. The domain definition classes and domain language classes form the two major groupings within the problem domain. Each of these two groups are also organized with the more domain specific classes depending on the more general classes. For example, the flight-dynamics classes "orbit" and "attitude" depend on the more general classes "vector" and "matrix".

The layered reuse model does not depend on Ada, but the Ada language contains features that support this model well. The use of generic packages allows each of the problem domain classes to be implemented as a generic unit that is completely decoupled from all other classes. In addition, the generic formal definitions associated with a package capture all the information about dependencies in a single location, as well as distributing external references throughout the code. Another useful feature is the separation of package specifications (and subprogram and task specifications as well) from their implementations. This is useful in hiding system dependent services, which can then have the system independent part defined at the appropriate layer. For example, the interface to a system dependent math library would be classified within the problem domain, and the interface to system-dependent screen management routines could be system independent services. The 5 top levels in this model would then contain system independent Ada code, which would be expected to be completely portable. This is not a consequence of attempting to make the highest layers portable, but rather is a benefit of isolating the known system dependencies, and using a standardized programming language. Using Ada leads naturally to having most reusable components also be portable. Similar portability may be attainable using C. It is almost certainly not attainable with FORTRAN, as the dialects vary too greatly between machines.

4. Example

This section presents an improved GENSIM design as an example of how to use the layered model. This new design is presented at the same level of detail as the original GENSIM design presented in section 2. Figure 7 shows the improved simulator design.

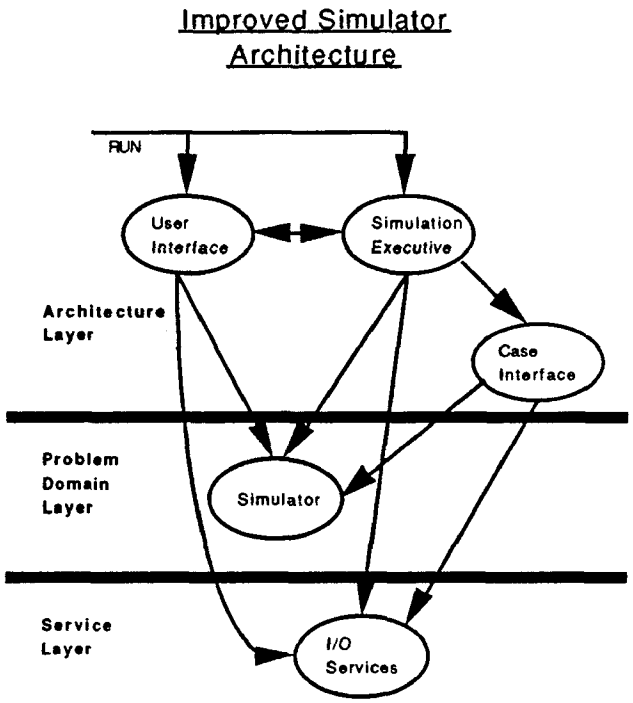


Figure 7

The key differences in this design are the location of the Case Interface subsystem and the new I/O services subsystem. In additon, the Spacecraft Control, Truth Model, and Utilities subsystem are combined into the Simulator subsystem. Figure 8 shows that the dependencies between these three subsystems are the same as in the original architecture (Figure 1), but that now none of these subsystems depends on Case Interface.

This extra design level is not carried through to implementation. Subsystems may be implemented as a single package which provides an interface to all the subsystem's components, but in this case the Simulator subsystem is merely

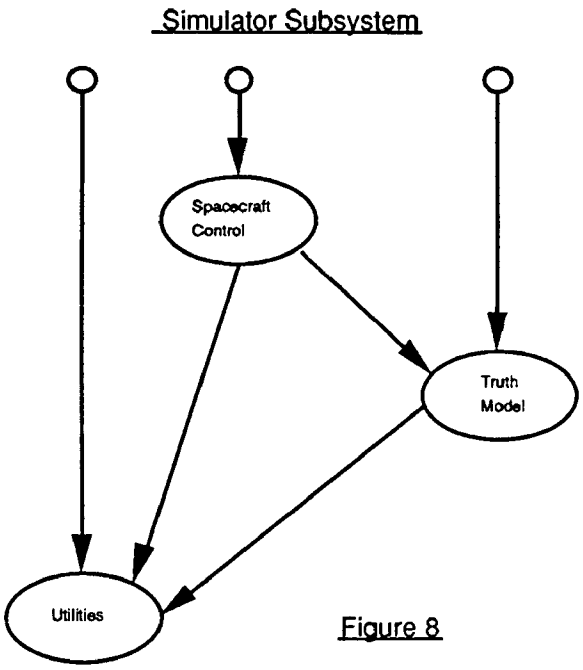


Figure 8

a logical grouping intended to reduce the design complexity.

Figure 7 also shows the three major layers of the reuse model. In this design, the I/O services consist of standard Ada packages such as Text_IO or Direct_IO, and an interface to DEC's Screen Management Guidelines (SMG) routines. Figure 9 shows the

interrelationship between the FSS module and the simulator architecture. Here the abstract data type for a sensor is created by instantiating a generic package. The generic ADT is designed so that all external dependencies are captured in the generic formal part. These dependencies include types provided by the simulator's Math_Types package, and functions to select information from the Sun and Dynamics modules. The FSS_Objects package uses the ADT (private type) exported by the FSS_ADT package to define its package state, and the FSS_Parameters_Display package uses visible types exported by FSS_ADT to define parameter screens. The FSS_Parameter_Displays package also instantiates Enumeration_IO using "type FSS_POWER is (OFF,ON)" as the actual parameter. This removes the reliance on using the 'POS attribute of enumerated types that has been a feature of all FDD simulators up until now.

Figure 10 shows how the FSS_Parameter_Display package fits into the design of the Case Editor subsystem. The Case Editor subsystem is the part of the User_Interface that allows a user to change any of the initial parameters for a simulation. The Parameter_Editor package tracks which displays the user has selected and calls the appropriate parameter display package. The difference is that now the User_Interface controls the initialization of simulation parameters, rather than the simulator components requesting intial values from a database contained within the Case Interface.

In this example, the use of the layered model removes the Truth Model's complex dependencies on the Case Interface packages shown in Figure 3. This enables the Simulator subsystem components to be usable within more than one architecture. The placing of the system architecture subsystems above the Simulator subsystem also allows general purpose service layer components to be enhanced as needed to integrate a given module into the system architecture. The FSS_Parameter_Display demonstrates this concept by using Enumeration_IO to add to the general IO services.

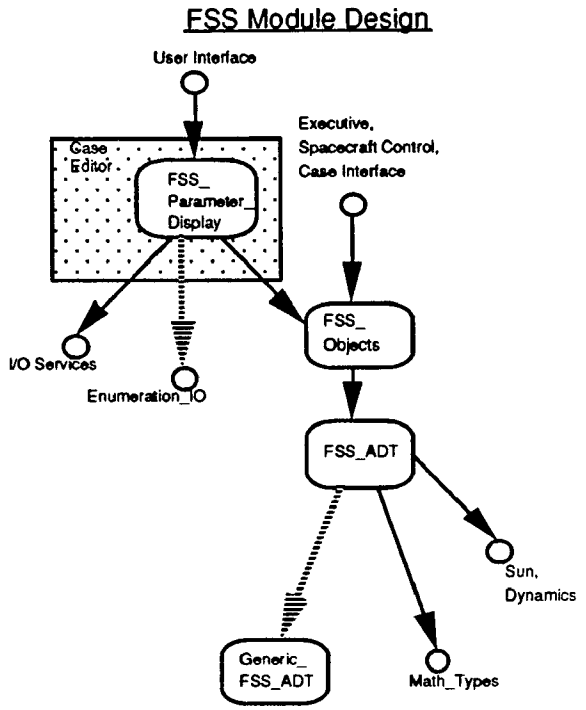


Figure 9

5. Future Directions

This paper describes a general reuse model for designing reconfigurable systems. The next step is to map the layered reuse model to Ada design and implementation concepts. The high-level designs presented in this paper use generic packages to help parametrize systems. There are many possible ways to incorporate generic packages into a larger design. These "reuse in the small" techniques include nesting generic instantiations, nesting generic definitions, and creating dependencies between library instantiations [Booth 1989]. This paper has used the last technique so that while generic instantiations are coupled, each of the generic units is completely decoupled from the others.

The layered reuse model provides a sound basis for project management. By strictly separating the problem domain issues from the system architecture issues, a manager can assign the appropriate experts to implement packages within each layer of the model. Improving the allocation of personnel to tasks should

improve both productivity and software quality. As this model is used, an understanding of what proportion of a system falls into

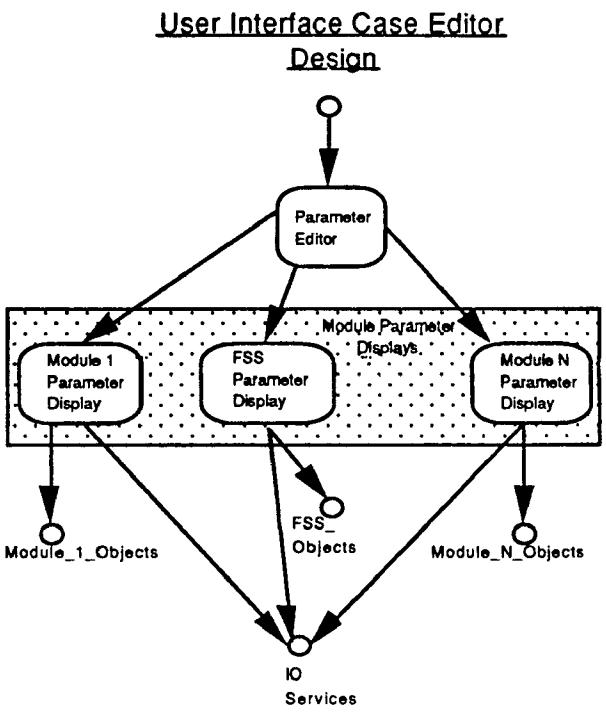


Figure 10

which layer will evolve.

The layered reuse model also can be used to understand which software is most critical. Layered models have seen the most use in operating system design. The kernel of an operating system typically requires the most attention, despite the fact it is a relatively small proportion of the code. This is because all other layers depend on its correctness and efficiency. The analogous layers in the reuse model are the service layers and the domain language layer. Additional evidence for the assertion is that the FDD has observed performance degradation in its Ada simulators due to the inefficient implementation of mathematical utilities packages.

In addition to the performance problems observed above, there is a concern that layered implementation models may be inherently slow due to the addition of extra levels of procedure calls to accomplish the same work. The FDD encountered this problem with a commercially provided graphics interface that provides the same FORTRAN interface routines on a VAX or an IBM mainframe. Whether this is due to extra procedure calls or generally inefficient implementation is unclear. Ada addresses the former problem by providing pragma Inline. The latter problem must be addressed by improving the software. If the software design and implementation is done properly, the layered reuse model should not degrade performance.

6. Conclusion

In "Domain-Directed Reuse", Braun and Prieto-Diaz extract properties that are common to applications (such as compiler design) where a high degree of reuse is already being obtained [Braun 1989]. These properties are a focus on a particular application domain, assumptions about system architecture constraints, and a set of generalized and well defined interfaces. The layered reuse model provides design concepts for examining applications domains and defining standardized architectures. These techniques will help realize the potential inherent in the concept of domain directed reuse.

References

[Booch 1987] Booch, G. Software Components With Ada, Menlo Park, Calif., Benjamin/Commings, 1987.

[Braun 1989] Braun, C. and R. Prieto-Diaz, "Domian-Directed Reuse," Proceedings of the Fourteenth Annual Software Engineering Workshop, November, 1989.

[Booth 1989] Booth, E. and M. Stark, "Using Ada Generics to Maximize Verbatim Software Reuse," Proceedings of TRI-Ada '89, October 1989.

[Markley 1988] Markley, F. L. , C. Mendelsohn, M. Stark and M. Woodard, "Impact Study of Generic Simulator Software (GENSIM) on Attitude Dynamics Simulator Development Within The Systems Development Branch," Unpublished FDD Study, 1988.

[McGarry 1989] McGarry, F., S. Waligora, and T. McDermott, "Experiences in the Software Engineering Laboratory (SEL) Applying Software Measurement," Proceedings of the Fourteenth Annual Software Engineering Workshop, November, 1989.

[Seidewitz 1989] Seidewitz, E. Combined Operational Mission Planning and Attitude Support System (COMPASS) Specification Concepts, Goddard Space Flight Center Flight Dynamics Division, COMPASS-102, 1989.

[Solomon 1987] Solomon, D. and W. Agresti, "Profile of Software Reuse in the Flight Dynamics Environment," Computer Sciences Corporation, CSC/TM-87/6062, November 1987.