Ada Types:
The Cornerstone of Simulation Models

By David C. Gross and Lynn D. Stuckey, Jr.

Boeing Defense and Space Group
Missiles and Space Division
Huntsville, Alabama

ABSTRACT

System simulation is the definition, control, and implementation of algorithmic models that replicate a system's real world behavior. Developing a useful simulation model requires a clear abstraction of the system. Software engineering supports abstraction by imposing a consistent structure on objects. One structural feature introduced by recent programming languages is strong [data] typing, aiming at two benefits: clarification of the design and enhancement of model verification. Strong typing clarifies the design by controlling the characteristics of an object, and enhances model verification by revealing errors early in the design cycle. Designers have traditionally viewed strong typing only as over-restricting the mixture of data units (e.g., meter versus degrees), an experience which has left a bad taste in many mouths. However, strong typing is a multifaceted tool which can apply to a broad range of software design problems. Simulation model designers can use Ada types to define, control, and implement models yielding:

(1) requirements consistency and traceability,
(2) interface definition/control,
(3) maintainability,
(4) reusability, and
(5) portability.

Because designers imagine and implement complex systems in parallel, projects can suffer from the fracturing effect of multiple visions of the final product. Strong typing can unify the system design, however, strong typing is only a tool -- the availability of which does not ensure its correct application. The challenge is to successfully implement it. This paper examines the successful use of Ada types for the design of simulation models, and points out the pitfalls of extreme approaches such as no-typing and over-typing. It presents Ada types as a scheme for enforcing a single system structure and as a foundation for generic simulation models. Finally, the paper discusses how types impact the software's lifecycle.

BACKGROUND

It comes as no surprise that designers implement the majority of a training device's *simulation* (as opposed to emulation) in software. Software is flexible, adaptable, and can simulate systems which have never existed. On the other hand, software has traditionally been a source of introduced errors and frustration for the system designers. The increased use of software for training simulation has pushed the state of the art in systems simulation.

System simulation is the discipline which attempts to construct useful models of real-world systems. The term "model" evokes images of stick and clay figures, or perhaps the plastic airplane models that children glue together; and this is precisely what we mean. A model is a representation of the system of interest, instantiated in a different medium and with insignificant differences from the "real" system. We construct the model to study some relevant aspect of the system such as its appearance, size, speed, range, mass, etc. Obviously, the model must be cheaper and easier to build than the real system, or we would use the real thing. The model should give us the ability to try experiments we cannot try with the real system, limiting our risk and expense.

It is a large intuitive leap from plastic toys to a multi-million line (and no doubt, multi-million dollar) software model of a weapon system. Nevertheless, the principles involved are the same. The primary difference is that the "map" from the model to the real system is substantially more esoteric than the map from a toy car to the family car. Building a software model creates a tension between the real system's physics and the simulation software's syntax; *both* of which exist only as images in the developer's mind. Modeling is a very creative act. The process by which a simulation maps an aircraft in flight to a software model, and then to ones and zeroes, has traditionally been the domain of arcane specialists. Particularly intimidating to the eventual user, this process generally requires three groups of "experts" (e.g., software, aerodynamic, and systems engineers) who do not even understand each other! This is not the kind of well-defined, disciplined process that inspires confidence in the resulting product.

The clarity of the model's map to the real system is the basic factor for determining the software's understandability, and in effect our confidence in the model. This "map" is formally called the simulation's system abstraction. Our confidence in the simulation is driven by how well we grasp and comprehend this abstraction. Therefore, the critical issues in system simulation development involve developing abstractions. How do we create an abstraction? How can we control the abstraction while the code is evolving? How can we build an abstraction that supports shifting requirements? How can we build an abstraction that communicates well with other parts of our team? Successful answers to these questions will result in production of high quality simulation models.

Fortuitously, the software engineering discipline, aimed at defining and refining the software process, has risen to prominence as a system simulation tool. In fact, one of the primary concerns of software engineering is applying abstraction to software projects. One important development of the software engineering effort is the Ada programming language. The Department of Defense sponsored the development of Ada specifically to address the goals of software engineering (i.e., modifiability, efficiency, understandability, and reliability). They intended for Ada to reach these goals by employing software engineering techniques such as abstraction, information hiding, modularity, uniformity, completeness, and confirmability. A brief investigation of the language will reveal that the rich "typing" feature is a primary means by which the language designers intended to include these qualities. Of course, it is by no means clear that they succeeded, or that the Ada types by themselves are sufficient for high quality system simulation. The following sections present a detailed discussion of modeling techniques as implemented in Ada via types. We begin with a brief introduction to Ada types. We follow with a discussion of software modeling. Finally, we present a look at implementing modeling via Ada types.

ADA TYPING

Although this is not a tutorial on syntax or semantics of Ada types, a brief digression into the nature of typing in the Ada language is in order. Almost every language (including assembly languages) provides for some kind of data typing. Typing provides the programmer with the capability of implicitly classifying data. All data is represented in the computer by an underlying set of bits, but the type of the data provides an implicit definition for interpreting the information. For example, the bit pattern "01000001" could be interpreted as the integer 65 or the ASCII character "A". FORTRAN 77 supports a fairly typical set of types including integer, real, logical, complex, and character, along with arrays of any of these types. The programmer "declares" that he desires a variable of some type, and generally can specify the amount of memory allocated to that variable (bit, byte, word, etc).

Ada provides for the fundamental types mentioned above, as well as extending the concept. Although other languages introduced many of these concepts, Ada collected all of them and expanded their power in a language intended for common use. Most of the additional type features provide the capability to extend the inherent types available in the language. The exception is that Ada provides two real types, fixed point and floating point. Figure 1 illustrates the structure of Ada types.

The extension to typing with Ada provides powerful and expressive capabilities to the modeler. Ada controls access to data, even while it is being used, through of the "private" (and limited private) type extension. The private type permits procedures to perform limited operations on a data structure without gaining read/write access to the data. Ada also extends the concept of typing to intertask communication. The task type permits a Ada task to create and communicate with executing entities in the computer sys-

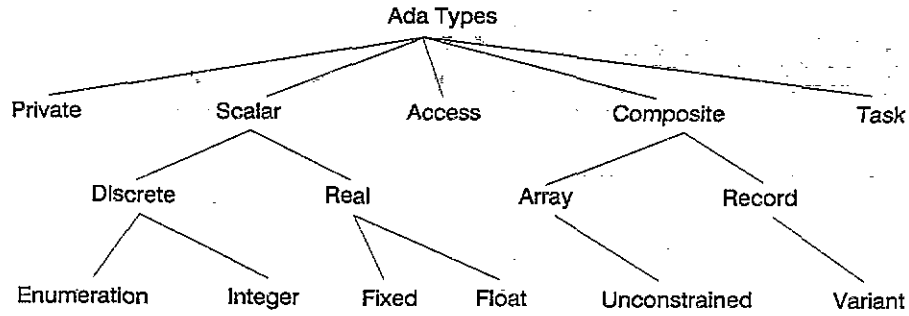


Figure 1

tem. Ada includes an access type which gives the programmer direct access to memory locations. Ada provides a subtle extension to the array type providing for unconstrained arrays, permitting the programmer to design an operation on an array of unknown length. The extension for record types permits the free association of data from mixed sources and types into a single "envelope" for manipulation. Variant records provide the capability to build a single record structure which can be tailored at declaration to support different, though similar entities (e.g., airplanes and trucks). The most pervasive extension beyond the fundamental types is the "enumeration" type. In an enumeration type, the programmer can specify the exact values permitted a variable of the new type. Figure 2 illustrates a new enumeration type declaration and declaration of two variables of that type.

```
type Colors is (Red, Yellow, Blue, Black, Grey);

Sky_Tone : Colors;
Sun_Tone : Colors range Red..Yellow;
```

Figure 2

Ada provides a set of operations for types that improve the programmer's ability to manipulate the data. Ada provides derived types to distinguish between data with similar appearance which should not mix (i.e., distance and temperature). Ada provides subtypes to further constrain an existing type (i.e., summer months from months in the year) while permitting ready conversion. The most powerful typing feature Ada provides is the rich set of type attributes. The language defines attributes over every discrete type such as 'first, 'last, 'succ, and 'pred which provide visibility into the type without coding the explicit value. Further, the special 'image and 'value attributes automatically translate between discrete type values and text representations. Ada provides every type with attributes such as 'address, 'size, 'constrained, and so forth. One might worry about converting between all these types, but Ada provides for casting between scalar types for conversion as well as unchecked conversion for any type. Finally, the programmer can specify the bit pattern for any type's representation, which provides access to the raw machine.

MODELING

A simulation model is a system, that is, a set of interrelated entities working together to achieve a common purpose. Obviously, the purpose of a simulation model is to mimic the behavior of the real system within a set of characteristics. These significant characteristics are the "state" variables of the simulation, because their value at any given point describes the state or condition of the system -- in so far as the simulation user cares.

The decomposition methodology employed for the model defines the set of model entities. In a classic functional decomposition, these are the functions of the real system: flight dynamics, weapons, electronic warfare, etc. In an object-oriented approach, the entities are the major object classes of the real system: terrain, culture, platforms, projectiles, etc. The capabilities of the user must drive the selection of methodologies. In our experience, we have found that most users relate well to an entity breakdown by objects, because they understand the notion of "laying my hand on it". Notice that this is really the issue of "how clear is your map" revisited.

Simulation software defines the interrelationships between entities by the data structures used to communicate between them. There are three general approaches to this problem: (1) common global data, (2) parameter lists, and (3) message passing. To build well-accepted simulations, we

must levy constraints on these data structures. There are three desirable constraints for the data structures which define the interrelation between entities: (1) access, (2) quantitative, and (3) qualitative. The model must prevent access to data by entities which should not have it (e.g. threats should not know the "real" position of the target). The model must control the allowable ranges for data values (e.g., prevent the switch from exceeding the number of positions). Finally, the model must restrict time-oriented changes in the data (e.g., platforms should progress smoothly through space).

Software System Engineering

If the simulation model is a system, then it has a definite life cycle, with full development, production, and operation phases. Figure 3 illustrates the lifecycle for any system. The central motivation for adopting a systems engineering approach to design is that almost all designs overemphasize the development phase at the expense of production and operation. This is just as true of software designs. Model designs must support trainer integration (production). We need to develop models that support changes concurrently with the real system (operation).

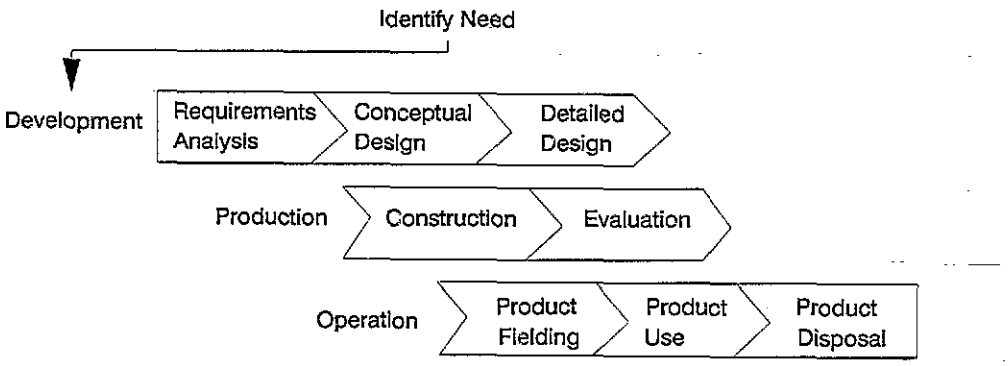


Figure 3

How do Ada types support model development? The answer is that they shift the defensive programming burden from the error-prone programmer to the compiler. The compiler has the advantage of repeatability -- it yields the same answer every time. Smart programmers code to protect their interests; checking for list overflows, illegal values, nonsense equations. But in Ada, this *explicit* defensive programming is shifted from the back of the programmer to *implicit* defensive programming in types enforced by the compiler. Real-time applications gain an additional benefit. Explicit defensive practices consume real-time computational resources; but implicit defensive practices are largely a function of compile time checks. We can go even further by removing the implicit run-time checks after the code is throughly tested, by setting a single compiler option.

Ada types provide support for specific modeling problems. In most languages, we model discrete states by assigning integers for each state; 1 for orbit, 2 for flyout, 3 for rendezvous, 4 for refueling, etc. The corresponding Ada type would be "type Tanker_State is (Orbit, Flyout, Rendezvous, Refueling);". Ada allows us to make this map explicit and to restrict the variables that contain the state to the appropriate values. Furthermore, Ada provides a way (via type modifiers) to explicitly state the range, interval size, and digits of precision for types, and variables within a type. This has the potential to eliminate, with a single declaration, thousands of "IF" statements scattered throughout the simulation which intend to protect data, each coded slightly differently. Through subtypes and derived types, Ada allows the programmer to directly model subtle differences between data. Via unconstrained types Ada allows the programmer to delay the decision of the final data size until it is available -- if it ever is. Again, this eliminates the need for explicit defensive programming in thousands of dispersed locations, each coded ever so slightly different. Finally, access types provide for data structures which dynamically grow and shrink according to need during the simulation. This stops the practice of always claiming the maximum memory and maintaining a pointer to the "last" used spot.

How do Ada types support model production? Software production is mostly an integration effort of code developed by widely dispersed

groups. Ada supports integration *in development* through types. Correctly designed Ada projects use a tightly coupled and controlled set of types packages. These "project type" packages define the initial baseline for the software. Every compilation must use the baseline set of types. Every time the developer recompiles his code, he must resolve his changes against the baseline. Essentially, these packages provide a virtual representation for the rest of the software.

Of course, the specific modeling benefits derived in model development spill over to help the rest of the software life cycle. The major production advantage is that the types have become an extension to Ada, precisely tailored to support the current project. Much of the exhaustive hand checking of data structures is replaced by completing a successful compilation. Our experience is that integration times for Ada systems have been sliced by an order of magnitude over comparable FORTRAN projects.

How do Ada types support model operation? Operation in the context of software systems involve the discovery of delivered bugs and capability upgrades. As mentioned, the user can turn type checking on or off to support problem investigation and solution testing. Our analysis has shown that more than 80% of Ada types in actual use are not fundamental types (real, integer, etc.). In fact most models depend heavily on the use of enumeration types. Since the algorithms for manipulation of these data structures need not depend on explicit knowledge of the underlying types, many capability changes simply involve changing the options in an enumeration list, and recompiling.

Design Clarity

The most difficult contrast to understand between hardware and software engineering is that for software, the design "is" the product. The implication of this is that the design's clarity is much more a driver of the product's quality than is the case for hardware. Software engineers sometimes state this idea as "the software is read many more times than it is written".

Given this, we can begin to see how important strong typing can be to improving programmer productivity. Which design is clearer, a set of variables all of type "REAL", or the same set with types "Temperature_In_Centigrade", "Airspeed_In_Knots", and "Power_In_Watts"? The multiple type set is clearer because it presents a higher fidelity map from the real system to the simulation; it reduces the mental burden to understand the design. Obviously, it is easier for a new programmer to understand the high fidelity map, and thereby make real contributions to the effort. Likewise, the high fidelity map is easier for the user to understand, increasing the chance that he will believe in the product.

Model Verification

Model verification is concerned with the effort to establish that the software is operational. The most simplistic verification is compiling the code. Clearly, the effectiveness of this verification is determined by the throughness of the compiler. The typical compilation only investigates the code for syntax errors. On the other hand, types empower the Ada compiler to investigate the semantics of the code. The Ada compiler will inspect every procedure call and every equation for types correlation. Perhaps the best part is that the programmer determines the degree of detail in this investigation by his choice of types (i.e., derived versus subtypes).

A number of more involved techniques are available for verification: modular design, peer review, traces, sample runs, animation, and data analysis. Private and generic types support a modular design by giving the code only the access needed to do the job. Ada types support peer reviews because the code is more expressive of the model -- the names and options for types generally are defined by the model's requirements document. The remaining techniques are supported by the software development environment.

Model Validity

Model validation is concerned with the effort to establish that the model accurately represents the real system. The critical validation criteria is that decisions based on the simulation should be the same as decisions based on the real system (if available). There is no such thing as an absolutely valid model, therefore we can validate it only for a given set of conditions. For example, a cockpit procedures trainer is not appropriate for combat training.

Given the expressive power of types, we can develop a model with a high face validity. The names of types, the options within types, the data ranges, and so forth come verbatim out of the model requirements document. The strongest advantage is that the entire data requirement is captured in a single types declaration. The reviewer is not forced to track down every use of a piece of data to see that the local code that

enforces the constraints on the information. This has the helpful effect of reducing the model's complexity for the validation process.

Model Credibility

A credible model is one accepted and used by the customer. Credibility is often overlooked by the developer -- he simply assumes the user will love it! On the other hand, users will occasionally take a non-verified and non-validated model as credible. Developers (who plan to stay in business) must strive to make their models credible. Models which conform to expert opinions, observations, and existing theory about the system tend to be highly credible. However, no one is going to take your word for it that your model possesses these qualities. The user desires to see for himself, and see it in your code. Once again, the expressive power of Ada types directly impacts the ability of non-experts to accept the model, increasing the model's credibility. The very syntax of types contribute to the model's credibility because their language and phrasing can exactly match the real system's self-description. Finally, the model with Ada types gains credibility from the discovery of many errors at compilation, meaning that fewer errors survive to be seen by the user.

IMPLEMENTATION

Implementation is the translation of a model from concept to reality. Types are the primary vehicle for implementing the model in Ada. Careful use of types results in software with desirable qualities such as requirements traceability, interface control, maintainability, reusability, and portability. We know these are desirable qualities because they result from applying the goals and techniques of software engineering. Ada types can be the tools we use to achieve the goals of software engineering.

Requirements Consistency and Traceability

One of the most crucial parts of model design is the ability to show consistent and traceable requirements. The only way to make the assessment of whether the simulation is a sufficient model is to be able to view and test the design requirements in the simulation itself. In the past, requirements were often overlooked or lost in the heat of the code and integration phases. Until the requirements become an integral part of the code, the implementation will diverge from the requirements. This divergence has historically been a problem because the resulting design fails to achieve its common goal: a system which fulfills the set of design requirements. Simulations that fulfill their entire design requirements are rare and simulations that have any direct traceability to these requirements in the code are rarer still. And lest we forget, traditional after-the-fact documentation is *not* requirements traceability. Traditional documentation is only dreams of what should have been in the code. True documentation is based exclusively on the code (not the comments). True documentation reveals the actual requirements implemented in the code.

Design requirements are a broad expression of what should and should not be done in a system. Types are the vehicle to express these requirements in code. The single greatest advantage of enforcing requirements through types is that we create compilable requirements. This gives us an *objective* test as to whether the design has implemented the requirements. This forces a system mindset from the beginning of the program. Compilable requirements require extended effort in analysis and design. But, they reduce problems and inefficiencies later in the program. By using types for requirements consistency and traceability, we are able to promote uniformity and confirmability in the simulation. Types can be a readable description of the system. Types can document the source of driving requirements. Types can also restrict object interaction in a way that is reflective of the real world. Types can police the simulation and training constraints. A types package can function as a central location for all system unique features.

Ada provides a number of typing features to encapsulate the requirements analysis in compilable code of which the most important is the enumeration type. Enumeration types provide the ability to really express requirements in code instead of hiding them with "magic numbers". Other types that are of use are (1) subtypes to restrict ranges without restricting interaction, (2) derived types to restrict interaction, (3) private types to restrict visibility, and (4) generic types to share algorithms among different instances of objects. The effect of all of this is to provide automatic universal data constraints, precisely as required, and with a large degree of visibility.

In general, requirements arise from three sources: intrinsic requirements of the implementation, design criteria, and system analysis. Requirements intrinsic to the implementation are captured by any language -- they are only required because

of it! The requirements traceability we have in mind addresses those arising from the design criteria and system analysis. The expressive power of Ada types provides a way to capture these requirements directly in the code in the natural language for the problem. A code example of the use of enumeration types to express a requirement rising from design criteria is found in Figure 4.

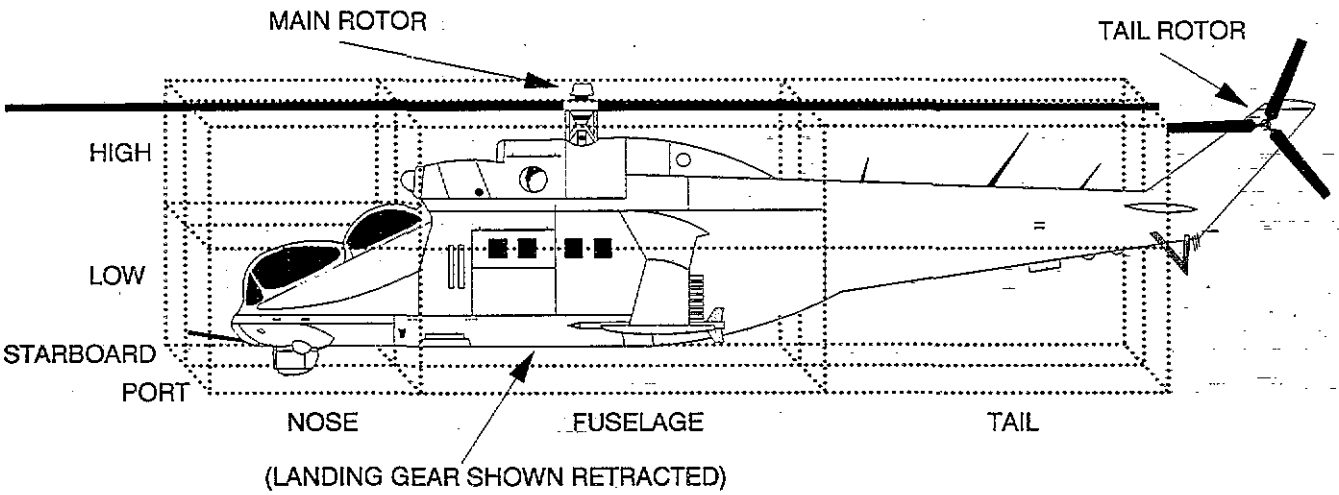
```
-- DOT Contract FA-75WA-3650
-- Programmable Test on Wind Shear

type FAA_Approved_Wind_Shear_Profiles is
  (Neutral_Logarithmic,  Frontal_2_Logan,
  Thunderstorm_2_Philadelphia, Thunderstrom_3,
  Thunderstorm_4,  Thunderstorm_5,
  Thunderstorm_6_JFK, Frontal_3,
  Thunderstorm_FAA_Mathematical);
```

Figure 4

What about a design requirement arising from system analysis? Figure 5 shows a drawing from a system specification and the resulting type declaration. The requirement states the need to determine the detonation of a weapon on the aircraft classified by one of fifteen zones on the body. This is to alert the affected systems of the need to consider the possibility of damage to their systems. The illustrated type implements this directly. A separate requirement to state the severity of the damage would be implemented by wrapping this type into a record with an additional field for severity.

Interface Definition/Control

In the real world, a system's interfaces are of paramount importance. How the subsystems plug, wire, bolt, or connect together is most of the design problem. With this in mind, it is astonishing that system simulation has done such a poor job of modeling interfacing. Part of the problem is that the simulation's decomposition has not represented the real world system. Designs based on functional decomposition do not lend themselves to comparison with the real world interfaces. The increasing prominence of an object-orientation is bringing interface modeling into focus. Another problem with interfacing has been the water bucket approach; throw all the interfaces into a bucket which anyone can access. This common global data approach has been a large source of error in most simulations that use them. For a simulation to become functional without extensive wasted effort, the interface within the simulation must be both defined and controlled. The simulation software itself must directly support true interface definition/control.




```
type Damage_Location is (Main_Rotor,                Tail_Rotor,          Landing_Gear,
  Low_Nose_Port,      Low_Nose_Starboard,     High_Nose_Port,      High_Nose_Starboard,
  Low_Fuselage_Port,  Low_Fuselage_Starboard, High_Fuselage_Port,  High_Fuselage_Starboard,
  Low_Tail_Port,      Low_Tail_Starboard,     High_Tail_Port,      High_Tail_Starboard);
```

Figure 5

Ada types provide for interface definition and control via compilable interfaces. This directly parallels the real world practice of interface definition. Any type can be an interface, but records are especially useful. Record types are a collection of basic types, directly modeling the interrelationship of diverse data concerned with a common subject. This collection captures in code the interface between system components. These interface types form the parameter lists of subprogram specifications. When compiled, these specifications form an interface *contract* between the subprogram and its user. The interface types result from a dataflow analysis and become the compilable program design language (PDL). Ada types provide direct support for designing the interface model.

Another concept in interface definition/control is the understandability of the interface. When we look at an electrical plug, we understand the types of its interfaces, i.e., neutral, ground, and hot. We also understand to what it would interface. The same must be true of simulation software. Via Ada types, we can describe the interface, its units, its purpose, or even its origin. Notice that use of interface types in software supports the direct production of the interface definition document (IDD). The IDD derives from the code -- not some extraneous piece of documentation. An example of a compilable interface is shown in Figure 6.

```
type Moving_Model_State is record
  Position    : Gaming_Area_Position_Components;
  Orientation : Angular_Position_Components;
  Velocity    : Gaming_Area_Velocity_Components;
  Rotation    : Angular_Velocity_COmponents;
end record;
.
.
.
procedure Update_Position (
  Model : in out Moving_Model_State);
```

Figure 6

Maintainability

Maintainability is the degree of difficulty in continuing to use the software in the face of changing equipment, requirements, and personnel over the life of the project. We desire maintainability because the operational cost of software can be significantly greater than the development cost. If a simulation is not maintainable, the cost of changes will be excessive throughout its life. Notice that typically just twenty percent of the software's lifecycle cost is expended in the development phase. Further, we desire maintainability because the design engineers change over the lifecycle of a simulation. New design engineers require maintainable code to be productive. Unmaintainable software introduces a substantial learning curve resulting from the design decisions and requirements lost at the time of departure of the original designer. Finally, we desire maintainable code to improve the software documentation, which determines how well our design is understood. We can not depend on traditional documentation; in most cases it does not reflect the actual code or decisions behind the code. Maintainable code is a step toward self-documenting code.

How can we build maintainability into our design? There are four steps to implementing a design change:

1. understand all explicit and implicit design decisions,
2. design around present structure,
3. prove no error propagation, and
4. validate new feature.

Ada types directly support each step. We can make it easy to visualize a current design by expressing the design through Ada's rich variety of types. Automatic exception checking can protect the original design from introduced errors. And, types obviously provide the same validation support to new design that they did to the original design. A good example is the enumeration type discussed earlier. If we add a new color, its location and its effect on the rest of the system is evident. The visibility into the implementation provided by Ada types is the basis for maintainability. A second powerful application of types for maintainability is variant records. With variant records, we can define a new entity in the simulation as similar to, or an extension of an existing entity. Obviously, the portions that the new and existing entities share have already been validated, which reduces the workload for the change.

Portability

Portability is the ability to transport software between computers, people, projects, and companies. Portable code is a goal of simulation that is often only considered late in the lifecycle. But early consideration of portability has a number of advantages. Portable code enables convenient platform changes. Portable code increases pro-

grammer productivity because the effort is concentrated on modeling the system instead of clever coding for the machine. Portable code reduces life cycle cost. Software costs are typically significantly higher after a simulation is fielded because of design changes (see maintainability) and equipment changes. Designs that depend on the nuances of particular machines or compilers or support tools do not hold up well over their lifespan. Also, non-portable code reduces the competitive position of an organization, which is in the position of continually redeveloping the wheel.

Ada significantly supports portability simply through its charter. Ada is a controlled, standardized language. This one fact has done a lot for portable software. Ada provides two specific type features for portability; user defined types and the extensive use of self-derived types. Through user defined types, Ada allows a programmer to define his own types' basis. This allows an engineer to remove his dependence on compiler implementation. The second characteristic of Ada dealing with portability is the huge set of new types we can introduce into the language. Enumerations, records, tasks, etc., are just a few of the examples of different types that we can use to model the simulation. With this proliferation of types, a simulation is not tied to a few machine dependent types to express its model. This volume of possible types is an advantage when producing portable code. The capability to derive and define our own set of types thus limits our risk exposure to the machine.

However, Ada has not removed all dangers. Different compiler vendors are allowed to implement fundamental subtypes under different names and sizes. Figure 7 shows an example of the names two different compiler vendors used for the various integer subtypes available. Clearly, code depending on these types would have its meaning changed as it moved between systems

Integer Width	Predefined Type Name	
	Compiler A	Compiler B
32 Bit	Long_Integer	Integer
16 Bit	Integer	Short_Integer
8 Bit	Short_Integer	Tiny_Integer

Figure 7

and might not even compile. We can avoid this loss of portability by defining our own fundamental types for integer and real numbers, and using extensive subtyping.

Reusability

Reusability is defined simply as the ability to use software again in new applications. Reusability requires portability. The potential for savings and increased profit from reusable code has provoked many studies. The benefits of reusability are not in question nor do they warrant listing. But, reusability is not as simple as it sounds. For example, restricting code to a single function does not always result in reusable code -- nor is just being generic enough. Reusable code possesses certain essential attributes such as definition of both purpose and interface. It must not be based on magic numbers. A reusable design must separate the control and the physics of a problem.

One of the most obvious ways in which Ada supports reusable code in through generics. Generics (based on types) provide the capability to develop a single algorithm for use in a wide variety of situations. One example is a user menu, which builds a menu, prompts the user, and guarantees a valid response -- one procedure for any enumeration type. Enumerations by themselves increase programmer productivity because they reduce the understandability load (trying to remember what the value 1 means here -- is it color or switch position?).

The use of attributes is perhaps the best way Ada types address reusability. When a simulation is written based on the attributes of a type, it is driven by requirements, not by parameters. The implementation algorithms can work based on the attributes of types instead of an explicit value. An example of this is aerodynamic lift parameters. Given an enumeration type defining the lift surfaces on an aircraft, then the "Compute_Lift_Characteristics" algorithm can compute lift for each surface in the type, rather than a local value. If the code is to be reused, the only change required is a modification to the list of surfaces in the type (and the database prescribing the characteristics of the surface). Any part of the implementation that considers the control surfaces is unchanged. See Figure 8 for a code example.

```
type Control_Surfaces is
  (Left_Flaperon,
  Left_Horizontal_Stablizer,
  Left_Leading_Edge_Flap,
  Left_Speedbrake,
  Right_Flaperon,
  Right_Leading_Edge_Flap,
  Right_Speedbrake,
  Rudder,
  Nosegear,
  Main_Gear);
```

Figure 8

We can further support reusability by controlling the *strength* of our types. Ada supports strong (very restrictive) and weak (non-restrictive) typing. Strong typing can insulate the data structure completely, but the misuse of strong typing can cause far more problems than it solves. We can use subtyping to express the requirement without unnecessarily interfering with data transfer in equations. Consider an equation for converting indicated airspeed into true airspeed. The equation will probably involve constants, a pressure measure, and a temperature measure. If the pressure or temperature has been created as "new" types (versus "sub"-types), Ada will not permit direct conversion. This kind of overly strong typing can cause the designer to commit all manner of bad design to work around his mistake. Properly understood, types packages are tailored extensions to the language. Anyone needing the type should have it, and the controls should be at the level of additions or revisions to the types. The rule of thumb is "all the visibility needed and no more".

Pitfalls

Ada types are not a panacea for software design. A type based implementation still requires careful design. There are a number of pitfalls that can occur with mindless typing.

1. Optimizing individual parts of a system will not result in an optimized system as a whole. We must keep the big picture in mind.

2. There is a tendency to misuse features to define complex or unusual data structures merely to facilitate a "clever coding" technique. We must avoid coding artistry but apply sound software engineering.

3. A haphazard use of types has a direct affect on the maintainability of software. We must avoid duplicating names or applications.

4. The abuse of strong typing can cause inefficient, unreadable, and unmaintainable software. Strong typing is strong medicine; we must need its power before employing it.

5. Reliance on system fundamental types is unportable. Predefined types change in both name and representation between compiler manufacturers.

6. Many times, extremely similar types will creep into the design as the software develops. This will reduce the design clarity.

7. Common global data and message passing represent extremes approaches to interface control. We should seek the balance and clarity of design which parameter lists yield.

Developers can misuse types. But we already know that we must *design* software to achieve our goals. Too often in the past, engineers have given lip service to design and have then proceeded to hack out a simulation. Design is not doing things the way they have always been done. It is not making the same decisions over and over again because it worked years ago. It is employing a systems viewpoint and transforming requirements into a verified, valid, and credible model. Clearly, Ada types can play an important role in this process.

A Typing Scheme

Given the richness of the Ada typing features, we desire a consistent and unified approached to implementing types in a simulator.

1. Define your own fundamental types from the intrinsic values --
   "type Integer_32_Bit is new Integer
   range (-2**31)..((2**31)-1);".

2. Derive all of the subtypes from your new fundamental type --
   "subtype Integer_16_Bit is
   Integer_32_Bit range -32_768..32_767;".

3. Subtype wherever possible to avoid overly strong typing --
"subtype Moving_Model_Number is Integer_8_Bit;"

4. Define the interface between every object as a single type containing all of the needed information.

5. You need more enumerations than you think you do.

6. All two alternative events are not boolean (True, False) --
use enumerates as appropriate (On, Off).

7. Type names should be complete and expressive of the information --
"type Landing_Gear_State is
(Locked_Up, Up, Retracting,
Extending, Down, Locked_Down);".

8. Use a single package for the global simulator types at the top of the design.

9. Package the types defining the interfaces between components at a given tier in a single package one tier above the components.

10. The purpose of types is to map the design to the real world.

11. The types should express their driving requirements (as applicable), design criteria, program specification, etc.

12. Begin a project's code by prototyping the types. On the other hand, expect the types to evolve with the program.

13. There must be an owner of each types package to police additions and revisions.

14. Write code which depends on attributes instead of explicit values. Such code supports design changes simply through changing the types rather than requiring changes throughout the code.

15. Types packages, unlike other packages in the system, should be "withed" and "used" for direct access. The types are not data but a tailored extension of the language, a fundamental resource for the design.

The underlying responsibility of the types' creator is to express the design and map the simulation to the real world. Types provide a powerful simulation modeling tool, but carry a responsibility. Whereas under-typing cannot hope to fulfill the design goals, over-typing can obscure the design. A well-controlled approach to typing can produce the desired balance. However, this control depends on goodwill and agreement between team members about the typing scheme.

CONCLUSION

In the course of our experience with Ada, we have seen that designs, utilizing Ada types, correctly implemented, exhibit certain desirable qualities. Ada types can provide the glue which holds the model together. But such results are by no means a forgone conclusion. The availability of Ada types does not relieve the software engineer of the responsibility to design the product. Quality software is not the result of happenchance or black magic; it is the result of an applied software engineering approach. The developer must create a model for a set of defined goals, and then implement it with a well-defined process in order to achieve these goals. The types selected for the implementation can encapsulate the system requirements and design. Application of Ada types can be a long step toward achieving the stated goals for a simulation model. Strong inherent language features like these are a requirement for the large complex training systems being constructed today.

ABOUT THE AUTHORS

David C. Gross is a software systems engineer with the Boeing Defense and Space group in the Missiles and Space Division. He has worked in all phases of simulation and training systems from proposal through delivery. He is currently involved in basic research for software engineering and applied research for system simulation. Mr. Gross has a Bachelor of Science in Computer Science/Engineering from Auburn University and is working toward a Master of Operations Research at the University of Alabama at Huntsville.

Lynn D. Stuckey, Jr. is a software systems engineer with Boeing Defense and Space Group in the Missiles and Space Division. He has been responsible for software design, code, test, and integration on several Boeing simulation projects including the Ada Simulator Validation Program and the Modular Simulator System. He is currently involved in research and development activities dealing with software development for weapon and threat simulators. Mr. Stuckey holds a Bachelor of Science degree in Electrical Engineering from the University of Alabama, in Huntsville and is working toward a Master of Systems Engineering at the University of Alabama, in Huntsville.