JOURNEY TO BABEL

PROGRAMMING LANGUAGE ISSUES IN IMPLEMENTING THE DISTRIBUTED INTERACTIVE SIMULATION PROTOCOLS

Frank J. Zinghini Jr.
Applied Visions, Inc.
Centerport, New York

David Berkman
Grumman Corporation
Great River, New York

ABSTRACT

Distributed Interactive Simulation (DIS) is the standard network protocol that promises to provide a common ground for the interplay of a variety of simulators in a single, integrated exercise. Many new training systems procurements require DIS support and, of course, mandate that such support be written in Ada. This in itself can be daunting, because Ada's rigid structure does not easily accommodate the complexity and diversity of information that constitutes the protocol. However, if the concept is to succeed it must *also* take into account the vast array of simulators that already exist, which will require implementation of DIS in all of the programming languages that preceded Ada.

This paper will examine the DIS protocols from the perspective of software implementation, and will discuss the problems and pitfalls of supporting those protocols in the variety of languages used in today's simulators. The features and quirks of these languages -- strong typing, variable-length record support, dynamic allocation, and so on -- will be explored for their effect on implementing the protocols. Specifically, Grumman's experience with implementing DIS in Ada, C, Pascal, and FORTRAN will be discussed. Each language has its own peculiar way of defining and manipulating data structures, and each has its good and bad points when applied to DIS.

ABOUT THE AUTHORS

Frank Zinghini is the founder and president of Applied Visions Inc., a software development and consulting company specializing in simulation, networking, and graphical computing. Frank holds a Masters in Computer Science and Bachelor of Electrical Engineering from the State University of New York at Stony Brook.

David Berkman is the lead Software Engineer for Grumman's Research and Development projects related to Distributed Interactive Simulation. Prior to this David was a Senior Software Engineer assigned to Grumman's UH-1 Flight Simulator Ada Feasibility Program. David holds a Bachelor of Computer Science from the State University of New York at Oswego.

JOURNEY TO BABEL

PROGRAMMING LANGUAGE ISSUES IN IMPLEMENTING THE DISTRIBUTED INTERACTIVE SIMULATION PROTOCOLS

Frank J. Zinghini Jr.
Applied Visions, Inc.
Centerport, New York

David Berkman
Grumman Corporation
Great River, New York

INTRODUCTION

The training community has embraced the concept of distributed, networked simulation as the key to the next generation of training technology. Much has been said of the need to move away from the realm of isolated simulators, and into a world of advanced networks of diverse trainers working together to achieve true team training. The efforts of the *Distributed Interactive Simulation* (DIS) standardization committee have made major strides in this direction, as evidenced by an emerging standard protocol for the interaction of simulators in a dynamic environment. There is little doubt that the future will see the integration of a vast array of simulators in complex, real-world team exercises.

Getting there, of course, will be half the fun. The goals of DIS are ambitious, to say the least. The complexity and quantity of information that must be exchanged will push the current state of the networking art to its limits, and as a result much attention is being paid to the physical considerations of moving all that data. However, the software implications of the DIS protocol are equally important, and are complicated by the variety of environments within which they must be implemented.

Central to the goals of the DIS effort is the potential inclusion of *all* of the training devices in the field today: not just new ones built specifically for use with DIS, but older and less-sophisticated trainers as well. From the perspective of the users of these training systems, this is good news. However, the implementors of these systems -- and especially the maintainers of existing trainers -- are faced with the task of actually making this all work within the confines of the programming languages and tools available on those systems.

This paper focuses on the implementation of the DIS protocols in the languages most often used in today's simulators. The features of these languages are explored for their effect on implementing the protocols.

The Grumman DIS Interface Unit (DIU)

This paper is based on work performed at Grumman for the development of a prototype of a DIS-based simulation network (see *A Prototype of a Simulation Network Using the Distributed Interactive Simulation Network Standard* in these proceedings). That system proved to be an interesting microcosm of the types of situations developers will face in the future: The bulk of the DIS software for that system was developed in

Ada under Unix, while portions of the Unix-based software were developed in C due to restrictions on the use of Ada in the Unix used for that project. A third part of the system was developed for the PC, written in Pascal and running under Microsoft Windows. In this last case, Windows was chosen for its graphics capabilities, and for its similarity to the Motif interface used in the Unix system; Pascal was chosen as the development language for its similarity to Ada, given that there are no Ada compilers currently available for the Windows environment.

Thus, even within the relative sterility of a laboratory environment, we were faced with the need to develop DIS software in three programming languages, and under two different operating systems.

THE DISTRIBUTED INTERACTIVE SIMULATION (DIS) PROTOCOL

What makes the DIS standard particularly difficult to implement, in *any* programming language, is the vast scope of what the protocol tries to achieve. The quantity and variety of information that must be manipulated in order to successfully describe a dynamic and complex tactical exercise will stress the data management capabilities of any programming environment available today.

The DIS standard is built around the definition of a core set of messages, called Protocol Data Units, or PDUs. These PDUs attempt to accommodate all of the possible variations on the theme of weapons system simulation, and as such contain large and varied amounts of information.

Language-Independent Specification of the DIS PDUs

Because the DIS standards committee recognizes the need to implement the protocol in a variety of ways, they have developed a language-independent format for the definition of the PDUs. This format is something of a lowest common denominator, which defines the physical structure and content of a message, without reference to any particular programming language.

The DIS specification presents the PDUs as a structured table of word and bit fields. Fields which represent quantities (such as velocity, location, *etc.*) are defined in terms of their data representation, units, and range. Fields which identify something, such as a weapon type or a country ID, are defined as a set of integer values corresponding to all possible selections. It is the task of the software developer to translate these concrete definitions into a more abstract form suitable to the programming language at hand.

IMPLEMENTING DIS WITH A PROGRAMMING LANGUAGE

The challenge in translating the language-independent concrete definitions of the DIS PDUs into an actual programming language is in matching the data modeling capabilities of the language to the content and format requirements of the PDUs. This is not as straightforward as it may seem: one of the most important functions of a programming language is to provide for the *abstraction* of data, so that the specifics of its concrete implementation are hidden from the programmer. In order to adhere to the DIS standard, we must be able to understand that abstraction, and coerce it into providing the required physical implementation.

Knowing how to influence a compiler's implementation of data structures requires an understanding of the overall concepts behind those structures, and of the particular language's technique for implementing those concepts. Given this understanding, we can work backwards, defining our abstract data in such a way that the physical realization of the data will match the DIS requirement. You will probably notice that this process is not only dependent on the programming language itself, but is in fact sensitive to a particular implementation of that language.

Review of Data Structures

Before considering the characteristics of specific programming languages, it is useful to review the topic of data structures at the conceptual level. The fundamental concepts of structured data transcend any particular realization in a programming language, and viewing the DIS PDU structures in these high-level terms will make their realization in a language more apparent.

In the evolution of programming languages, the modeling of process was developed first, while data was considered as something that just went along with the processing. We eventually learned, through the development of structured programming, that all processing could be decomposed into a sequence of standard processing constructs: blocks, iterations (loops), conditionals (if-then-else and case statements), and so on. While much of this seems intuitively obvious today, it was a breakthrough in its day, leading the way from hopelessly complex "spaghetti" code (in the age of GOTOs), to the orderly expression of a process as a sequence of steps.

It was not long after this shift in the expression of process that we began to realize the importance of data, and in particular how it relates to processing. We quickly realized that the same concepts used to refine and formalize the process model could be applied to data modeling. In fact, if the two models matched, the resulting program would be simple and elegant. Thus, we now view the structure of data in the same terms as with processing.

Structured Data: Records. We are accustomed to viewing software as a hierarchy of nested functions, with each successive step down the hierarchy more specific and detailed that the one above. This concept is modeled in data by the Record structure (known simply as a Structure in some languages). A Record provides for encapsulation and abstraction of data. For example, given a record containing the location of an aircraft as a Latitude, Longitude and Altitude, that data can either be treated individually, or as a higher-level abstraction; for example, two locations can be compared without the need to know how they are expressed internally.

Iterated Data: Arrays. A common construct in programming languages is that of iteration: the so-called "for loop", wherein the same set of instructions are executed a finite number of times. The data structure which corresponds to this construct this is the array, which is a sequential collection of homogenous elements.

Selective Data: Enumerations. Functions in a program are usually selectively executed, based on various conditions and choices. Very often these choices are based on simple conditions, such as what type of object is represented by some data (for example, is it an aircraft or a tank). This type of processing usually leads to sequences of if-then tests or, more frequently, a selective execution statement known as a *case* (or *switch*) statement. Selective data identifies the alternatives in situations like this, and was traditionally implemented as simple numeric codes (1 means aircraft, 2 means tank, and so on).

While numeric codes are serviceable for selective data, they are not sufficient: an important benefit of data abstraction is that it allows compilers to weed out mistakes by binding the definition of the data to its purpose. In the above example, what would happen if we used a code of 3? There's no way for the compiler to know that it's wrong. What we would rather do is define a "Vehicle" data type that has the values Aircraft and Tank, and only use variables of that type to represent that selection. These values would, of course, be represented internally as numbers, but that is not visible to the programmer. Instead, these are treated as a new data type, and variables declared of this type can only take on values that are a member of that type. This concept is supported by all modern languages, and is known as an *enumeration* type.

As we discussed earlier, the DIS PDUs are defined using numeric codes for their selec-

tion values (a necessity for language independence). In order to continue to benefit from the improved type checking associated with enumeration types, control over the internal assignment of numeric values to the members of an enumeration is a key concern in implementing DIS in a given language.

Conditional Data Structures: Variant Records. Just as a functional problem can be expressed as a sequence of decisions and alternatives, so too can the data associated with such a problem. The concept of a *variant record* recognizes the fact that the structure of data is not always clear-cut: very often the meaning of one piece of a record is dependent on some other piece.

Variant records generally consist of three parts: the invariant part, which is the same for all instances of the record; the discriminant, which is used to indicate which variant applies; and the variant part, which is the part of the structure that can take on different meanings based on the discriminant.

For example, while both an aircraft and a tank have a position and a velocity, pitch/roll/yaw may only be of concern for the aircraft, while turret position applies only to the tank. We could generalize a vehicle state as a variant record containing location and velocity as invariants, with Vehicle type as a discriminant, and the remaining values as variant parts based on that discriminant.

The size of a variant data structure is always the greatest of all possible variations (it can also be viewed as the union of all of the variants, plus the invariant part; hence the use of the term *union* to denote a variant record in C). The specification and implementation of variant records differs widely among the languages considered here.

Review of Programming Languages

This paper addresses the four most common programming languages in use today: Ada, C, Pascal, and FORTRAN. Each presents a slightly different level of sophistication in data modeling, from limited support of standard FORTRAN, to the rigid and formal features of Ada.

In this section, we offer an overview of the capabilities of the four languages, and how they will be applied to the implementation of DIS. In the next section, we will actually apply these languages to a sample DIS PDU, to see the effects of these features.

Ada. While many view Ada as the most restrictive of languages, it turns out to be one of the most flexible when it comes to controlling the creation of data structures. This stems largely from Ada's focus as a language for embedded systems development: the language includes features to control the realization of data structures. This control is embodied in the Ada "for/use" construct, with which the programmer is given the ability to control such things as:

- The addresses of variables
- The assignment of bits within a record to fields of that record
- The assignment of numeric values to the members of an enumeration type

These features make Ada particularly well suited to the implementation of the DIS protocols, because of the need to match specific bit-field layouts of a PDU, and the usefulness of developing enumeration types to match the numeric values defined in the DIS specification.

Ada does, of course, have its drawbacks as well. Perhaps one of the more difficult to deal with is the fact that it requires all variant record types to have an actual discriminant field. This is as opposed to other languages which either do not require a discriminant at all (as is the case with C), or allow the possibility of a discriminant *type* without actually defining a *field* of that type (such as Pascal). This feature is useful in situations where it is necessary to view the same exact location in memory in a number of ways, without the overhead of a dis-

criminant field. A simple example of such a case is the common need to view a single location in memory as, say, either a floating-point number or a straight binary value.

C. The C programming language has become a dominant force in the commercial sector and, in the early days of Ada waivers, found widespread use in trainers as well. Until recently, its object-oriented offspring, C++, was seriously considered as a challenger to Ada's dominance in defense systems (an Air Force study proved Ada's cost effectiveness, and the challenge has faded).

The original C language was designed as a loosely-typed systems implementation language, giving programmers the ability to manipulate the environment fairly freely. For example, the standard mechanism for defining enumeration types can include (as an option) numeric values for the members of the type.

Since the original C is loosely typed, it does not afford any of Ada's type-checking benefits. The currently proposed ANSI standard version of C (upon which C++ is based) does include strong typing with extensive checking, but that version of the language is not yet in wide use in simulators.

The one weakness in C, from the perspective of DIS, is the lack of control over the storage size allocated to data. Where Ada gives us the ability to specify the number of bits to be allocated to, say, a field of a record, in C we must rely on knowledge of the size of various data types *for a given implementation of the language*. This is a great source of difficulty in working with C: for example, the size of the integer data type *int* is dependent on the target system: 16 bits for 16-bit processors, and 32 bits for 32-bit processors. Fortunately, the modifiers *short* and *long*, when applied to *int*, have achieved a *de facto* standardization of 16 and 32 bits respectively. ANSI standardization efforts for the C language promise to minimize these problems in the future.

Pascal. Pascal was the philosophical forerunner of Ada, and as such shares many of the same data modeling concepts. However, it does not have the embedded-systems focus of Ada, and therefore lacks the low-level controls present in Ada. As a result, it is often difficult, if not impossible, to accurately control the size and orientation of fields within a record. Similarly, the only way to control the assignment of numerical values to enumerations is by "padding" the enumeration type with extra values to force a particular ordering.

FORTRAN. FORTRAN is probably the most common language among existing simulators. Unfortunately, it is also the least hospitable to the implementation of the DIS protocol. FORTRAN's data modeling capabilities are extremely limited: it supports scalar and array types.

Most FORTRANs, including the current ANSI standard definition of the language, do not support structured data, while those that do don't support variant records. Programmers must still resort to the EQUIVALENCE statement to achieve any sort of structure or variance in the data.

Finally, the closest FORTRAN comes to *enumeration types is the* PARAMETER statement, which defines nothing more than simple named constants. While this certainly makes it easy to control the correspondence between names and numeric values, it does not offer any of the benefits of the stronger typing in the other languages.

Do-It-Yourself Data Structures. Just because a particular language does not directly support all of the basic data structures does not mean that they cannot be implemented in that language. All languages have in them the building blocks for creating the basic data structures defined above.

The most common example of this homebrew approach to data structures is the use of the FORTRAN EQUIVALENCE to implement variant record types. Although many programmers may not have realized it when

they were doing it, equivalencing several variables to the same location in memory is nothing more than a variant data structure. The difficulty in FORTRAN is that there is no control in the language over access to such variant data, and there is no direct language support for things such as sizing.

Specific Language Implementations

This section will explore the application of each of the languages to the DIS protocol. The best way to illustrate this is with an example; unfortunately, the complexity and depth of the DIS PDUs makes using an actual PDU impossible for a paper of this scope. Instead, we will define a sample PDU that is representative of the types of problems we encounter in DIS. We will present the definition of this PDU in the same language-independent format as in the DIS specification, and then we present the implementation of the corresponding data structures in the four languages we have been considering.

Sample PDU Specification. Our sample PDU will be based on the "vehicle state" example we used earlier. This PDU will illustrate a number of common problems with DIS, including nested data structure definitions, a variant record type, and non-contiguous values for an enumeration. Note that this sample PDU is not representative of the actual DIS Entity State PDU (which is *much* more complex, and does not make use of variant records).

The sample Vehicle State PDU is defined as follows:

FIELD SIZE (Bits)	VEHICLE STATE PDU	
96	Location	X 32-bit Floating Pt Y 32-bit Floating Pt Z 32-bit Floating Pt
32	Velocity	FPS 32-bit Floating Pt
8	Kind	8-bit Enumeration
48 (Union of all)		
Kind = 1 ->	Roll Pitch Yaw	Deg 16-bit Int (0-360) Deg 16-bit Int (0-360) Deg 16-bit Int (0-360)
Kind = 2 ->	Turret Pos	Deg 16-bit Int (0-360)
Kind = 3 ->	Category	8-bit Enumeration

Where:
Kind = 1 Aircraft, 2 Tank, 4 Ship
Category = 1 Surface Ship, 2 Submarine

Ada. Implementing our sample PDU in Ada illustrates most of the data modeling features of the language discussed earlier. Note the application of the for...use construct to control the definition of the enumeration types, and to ensure that the sizes of the fields of the records match the requirement. Among the four languages, Ada allows the greatest level of control in this manner. It is also, unfortunately, the most verbose as well:

```
type VEHICLE_TYPE is (Aircraft,Tank,Ship);
for VEHICLE_TYPE use (1, 2, 4);
for SHIP_TYPE'SIZE use 8;

type SHIP_TYPE is (Surface, SubSurface);
for SHIP_TYPE use (0, 1);
for SHIP_TYPE'SIZE use 8;

type DEGREES is new INTEGER 0..360;
for DEGREES'SIZE use 32;

type FLOAT_32 is new FLOAT;
for FLOAT_32'SIZE use 32;

subtype FEET_PER_SECOND is FLOAT_32;

type LOCATION_TYPE is
  record
    X: FLOAT_32;
    Y: FLOAT_32;
    Z: FLOAT_32;
  end record;
```

```
type ORIENTATION_TYPE is
  record
    ROLL : DEGREES;
    PITCH: DEGREES;
    YAW  : DEGREES;
  end record;

type VEHICLE_STATE_PDU_TYPE (KIND: VEHI-
CLE_TYPE) is
  record
    LOCATION: LOCATION_TYPE;
    VELOCITY: FEET_PER_SECOND;
    case KIND is
      when Aircraft =>
        ORIENTATION : ORIENTATION_TYPE;
      when Tank =>
        TURRET_POS: DEGREES;
      when Ship =>
        CATEGORY : SHIP_TYPE;
    end case;
  end record;
```

Pascal. To achieve the same definitions in Pascal, we need to know more about how the compiler does things, because we have less direct control over how it creates its data. In particular, notice how we must "pad" the definition of the Vehicle type enumeration to force the required numbering scheme; this is based on the knowledge that standard Pascal numbers enumeration values consecutively starting from zero. Also, you will notice that we cannot control the size of scalar types such as real or integer: in this example, we simply assume that the target device is a 32-bit processor. We also rely on Pascal's standard behavior of packing subranges and enumerations into their smallest possible storage element, so that the enumerations will occupy a single byte, and TDegrees will take two:

```
type
  TVehicle  = (V_Pad_0, Aircraft, Tank,
               V_Pad_3, Ship);
  TShip     = (Surface, SubSurface);
  TDegrees  = 0..360;
  TFloat_32 = real;
  TFeetPerSecond = TFloat_32;

  TLocation = record
    X: TFloat_32;
    Y: TFloat_32;
    Z: TFloat_32;
  end;

  TOrientation = record
    Roll : TDegrees;
    Pitch: TDegrees;
    Yaw  : TDegrees;
  end;

  TVehicleStatePDU = record
    Location: TLocation;
    Velocity: TFeetPerSecond;
    case (Kind: TVehicle) of
      Aircraft:(Orientation: TOrientation);
      Tank    :(TurretPos  : TDegrees);
      Ship    :(Category   : TShip)
    end;
```

C. In the C implementation of the sample PDU, we once again have direct control over the definition of the enumeration types, however we still do not have direct control over the size of scalar types. Again, we assume a 32-bit processor, and use the "short" modifier to force a 16-bit size where needed. Note also how C's implementation of variant records differs from Pascal and Ada: here, we must define the variant part as a nested record, so that referencing those fields requires and additional qualifier ("extra_data"):

```
enum VEHICLE= (Aircraft=1, Tank=2, Ship=4);
enum SHIP   = (Surface=0, SubSurface=1);

typedef short int DEGREES;
typedef float     FLOAT_32;
typedef FLOAT_32  FEET_PER_SECOND;

typedef struct {
    FLOAT_32 X;
    FLOAT_32 Y;
    FLOAT_32 Z;
} LOCATION;

typedef struct
    DEGREES roll;
    DEGREES pitch;
    DEGREES yaw;
} ORIENTATION;

typedef struct
    LOCATION        location;
    FEET_PER_SECOND velocity;
    VEHICLE         kind;
    union {
      ORIENTATION orientation
      DEGREES     turret_pos;
      SHIP        ship;
    } extra_data;
} VEHICLE_STATE_PDU;
```

FORTRAN. Implementing the sample PDU illustrates the mapping of the multi-level structures onto the flat data models of FORTRAN. Here, we use the EQUIVALENCE statement to define a collection of discrete variables, and then "place" them over the appropriate positions of a PDU message buffer:

```
 PARAMETER (Aircraft = 1),(Tank = 2),
+          (Ship = 4),
+          (Surface = 0),(SubSurface = 1)

 INTEGER*1   VS_PDU(23)

 REAL*4      LOCATION(3)
 REAL*4      X, Y, Z
 EQUIVALENCE (X, LOCATION(1)),
+            (Y, LOCATION(2)),
+            (Z, LOCATION(3))
 EQUIVALENCE (LOCATION(1), VS_PDU(1))
```

```
      REAL*4      VELOCITY
      EQUIVALENCE (VELOCITY, VEH_STATE_PDU(13))
      INTEGER*1   KIND
      EQUIVALENCE (KIND, VS_PDU(17))

C Variant parts all occupy the same space

      INTEGER*2   ORIENTATION(3)
      INTEGER*2   ROLL, PITCH, YAW
      EQUIVALENCE (ROLL , ORIENTATION(1)),
     +            (PITCH, ORIENTATION(2)),
     +            (YAW,   ORIENTATION(3))
      EQUIVALENCE (ORIENTATION(1), VS_PDU(18))

      INTEGER*2   TURRET_POS
      EQUIVALENCE (TURRET_POS, VS_PDU(18))

      INTEGER*1   SHIP
      EQUIVALENCE (TURRET_POS, VS_PDU(18))
```

As you can see, the use of EQUIVALENCE to create multi-level data structures leads to *complex declarations that are tightly bound* to the structure of the PDU, and therefore are difficult to maintain.

CONCLUSION

There is no denying the importance of the emergence of DIS, nor is there any likelihood of slowing its advance. Implementing DIS on new trainers in Ada does not pose a problem: it will prove to be difficult but educational and rewarding, and the standardization of Ada will quickly lead to a *large volume of reusable DIS Ada code in* the industry. It is the implementation of DIS in *existing* trainers that will prove troublesome: the preponderance of FORTRAN-based trainers in the field (and assembly-language systems as well) will make the task of bringing old trainers into the DIS universe difficult and risk-prone. We must continue to focus our efforts on isolating these older systems from the details and complexity of DIS, as Grumman is doing with the DIU.