Problem Encountered with the Variant Record Construct in Ada

STATUTORILY EXEMPT

This paper presents a problem encountered in applying the Ada variant record construct in the design of the data link layer protocol. It also discusses and demonstrates possible solutions to this problem.

INTRODUCTION

To reduce their design complexity, most networks are organized as a series of layers, with each layer built upon its predecessor. The number of layers, the name of each layer, and the function of each layer may differ from network to network. In all networks, however, the purpose of each layer is to offer certain services to the higher layers. The rules and conventions defining the services in a layer (n) are collectively known as the "layer n protocol."

The Open Systems Interconnection (OSI) reference model consists of a seven layer protocol architecture: application, presentation, session, transport, network, data link, and physical layers. The X.25 Defense Data Network (DDN) standard protocol [4], with the International Telephone and Telegraph Consultative Committee (CCITT) Recommendation X.25 [5], was chosen as the data link layer protocol to be used in a protocols project initiated in the Computer Network Security Division of the National Computer Security Center.

The Ada language was chosen for implementing the X.25 protocol for this project. While applying the Ada variant record construct in this design, an unexpected problem emerged. This paper describes the problem (the coresident homograph error), methods explored in search of a solution, and the method found to be the solution to the problem.

BACKGROUND

A record is simply a construct composed of several constructs. A variant record is a record which has more than one possible structure. Thus, a variant record is only a special type of record. In Ada, a record is a composite object; that is, a collection of possibly different types of components. For example:

```
type DATE_RECORD is
    record
        YEAR: INTEGER range 0..4000;
        MONTH: MONTH_NAME;
        DAY: INTEGER range 1..31;
    end record;
```

The record is called DATE_RECORD, and YEAR, MONTH, and DAY make up its component list. The ranges and MONTH_NAME make up the types of each part of the component list.

A variant record is a composite object consisting of a component list. Part of this component list remains constant while the rest may vary. The variation of the component list is based on a discriminant, a syntactically distinguished component of a record. For example:

```
type DATE_VARIANT_RECORD is (JULIAN, YEAR_MONTH_DAY);
type DATE_RECORD (TYPE_OF_DATE: DATE_VARIANT_RECORD) is
          record
                    YEAR: INTEGER range 0..4000;
                    MONTH: MONTH_NAME;
                    case TYPE_OF_DATE is
                      when YEAR_MONTH_DAY= >
                        DAY: INTEGER range 1..31;
                      when JULIAN = >
                        DAY: INTEGER range 1..366;
                    end case;
                    end record;
```

In this example, DATE_RECORD is again the name of the record. The component list consists of YEAR, MONTH, and TYPE_OF_DATE. The discriminant, TYPE_OF_DATE, may vary, and DAY type may then change. If TYPE_OF_DATE is to be YEAR_MONTH_DAY then type DAY will have an integer range of 1 to 31. If TYPE_OF_DATE is to be JULIAN, then type DAY will have an integer range of 1 to 366. The variant record clearly allows for more flexibility in describing ideas that appear structurally similar.

THE PROBLEM

The concept of variant records can easily be applied to packets defined in the CCITT Recommendation X.25. The X.25 protocol has several types of packets. These packets have some similarities and some differences. Using the variant record construct to describe packets that had fields with constant data types and fields with different data types was preferred. Thus, some parts of the component list remained constant, some parts changed, and some parts kept the same name but changed their types. When applying this to the previous DATA_VARIANT_RECORD example, MONTH and YEAR remain constant, and types for TYPE_OF_DATE changed, and some of the types for TYPE_OF_DATE (i.e., DAY := INTEGER range 1...31) change. Consider the X.25 packet structure in figure 1.

In figure 1, PACKET_TYPE_RECORD is a variant record with GENERAL_FORMAT_IDENTIFIER, LOGICAL_CHANNEL_GROUP_NUMBER, and LOGICAL_CHANNEL_NUMBER remaining constant. Here, KIND_OF_PACKET changes and the type for PACKET_TYPE_IDENTIFIER changes, varying from type PENTAD to type OCTAD. Clearly, these packets have different functions, yet the general structures of the packets remain basically identical, and some of the same operations are found on different types of packets. For this reason, a variant record construct best describes the abstraction of the packet. However, in applying this abstraction a problem was discovered with the way the author tried to use variant records.

Problems arise if the variant record is allowed to change some parts of the component list while keeping the names of the components within each part of the list the same. In Ada, this is called a coresident homograph error. This kind of error would result in the DATE_VARIANT_RECORD example because the component DAY was named DAY in each

```
type PACKET_TYPE_RECORD
    (KIND OF_PACKET:KINDS_OF_PACKET_TYPE) is
    record
        GENERAL_FORMAT_IDENTIFIER:                 TETRAD;
        LOGICAL_CHANNEL_GROUP_NUMBER:              TETRAD;
        LOGICAL_CHANNEL_NUMBER:                    OCTAD;
        case KIND_OF_PACKET is
            .
            .
            .
            when DTE_REJECT_PACKET =>
                PACKET_RECEIVE_SEQUENCE_NUMBER:              TETRAD;
                PACKET_TYPE_IDENTIFIER:                      PENTAD;
            when RESET_REQUEST_PACKET =>
                PACKET_TYPE_IDENTIFIER:            OCTAD;
                RESETTING_CODE:                    OCTAD;
                DIAGNOSTIC_CODE:                   OCTAD;
            .
            .
            .
        end case,
end record;
```

Fig. 1. Example of X.25 Packet Structure

part of the component list. In figure 1, the use of the name PACKET_TYPE_IDENTIFIER as a component of a DTE_REJECT_PACKET and as a component of a RESET_REQUEST_PACKET would again result in a coresident homograph error. The rest of this paper shall examine several different methods which attempt to use Ada variant records.

THE SOLUTION

The Ada Language Reference Manual (LRM) [3] states that "each of two declarations is said to be a homograph of the other if both declarations have the same identifier and overloading is allowed for at most one of the two." Further, according to the *Rationale for the Design of the Ada Programming Language* [6], "a record type with a variant part specifies several alternative variants of the type." Another name for this occurrence is a coresident homograph. Clearly, the set of possible record values is the union of the sets of values possible for the alternative variants. In other words, the components of the component list cannot be called the same name even if it seems logical to do so.

The following data structure is used to define the contents of the records that are used in demonstrating the problem and its solution.

Record A	Record B	Record C	Record D
first: integer second: float	first: integer third: float	fourth: integer	first: float third: float

Method 1

Method 1 shows a logical way to construct the variant record. This method supports the data abstraction desired in defining the records. It shows the case where there were four parts to the component list of the variant record and each part contains a component that is either an integer data type or a float data type. (See figure 2.)

```
procedure sample__1 is
      type Q is (A, B, C, D) ;
      type Rec (X : Q : = A) is
         record
             case X is
                   when A = >
                        First   :  Integer;
                        Second :  Float;
                   when B = >
                        First   :  Integer;
                        Third   :  Float;
                   when C = >
                        Fourth  : =   Integer;
                   when D = >
                        First   :  Float;
                        Third   :  Float;
             end case;
         end record;

Foo : Rec (B);
begin
      null;
end Sample__1;
```

**Referencing Mechanism: Foo.First (object name.name of the desired choice of the record).

Fig. 2. Method 1

This method does not work. When trying to verify the semantics of the code, errors specifying coresident homographs occur. Since the component "first" is referenced in both choice A and choice B, the component "first" in choice B is a coresident homograph of the component "first" in choice A. Further, the fact that component "first" was of type float in choice D and of type integer in choice A was irrelevant. It was the name "first" that gave rise to the error.

Method 2

Method 2 consists of regrouping the different components of the variant record into choices with common components extracted. (See figure 3.) Look at records A, B, and C where component "first" remained an integer data type. In this case, a further case structure was needed to separate "second" and "third" into the choice A and the choice B. The component "fourth" was a part of choice C because it was not common to record A or record B.

```
procedure Sample__2 is
      type Q is (A, B, C,) ;
      type Rec (X : Q := A) is
         record
            case X is
                  when A|B = >
                     First   :  Integer;
                     case X is
                      when A = >
                           Second  :   Float;
                      when B = >
                           Third   :=  Float;
                      when others = >
                           null;
                     end case;
                  when C = >
                     Fourth :  Integer;
            end case;
         end record;
Foo : Rec (B);
begin
      null:
end Sample__2;

**Referencing Mechanism: Foo.First
```

Fig. 3. Method 2

This variant record structure was syntactically and semantically correct as long as the data types for the A, B, and C choices remained unique to each of the choices. However, this solution did not consider the instances where the data types were not the same for more than one of the variant choices. For example, consider a case where "first" was a component of data type integer for choice A and "first" was also a component of data type float for choice D. This type of variant record structure could not be coded because the data type of the component "first" does not remain a constant type throughout the record.

Even if all the data types of the components remained constant, Method 2 still would not work. Consider a data structure consisting of record A with components "first" and "second," record B with components "second" and "third," and record C with components "first" and "third." The author regrouped the common components and was left with choices A and B grouped together because they had component "second" in common, but choice C also had component "first" in common with choice A. Thus, this code could not be completed because component "first" had already been referenced as a part of choice A|B and could not be referenced again.

Method 3

Method 3 explored the idea of creating just one record containing all of the variant components. This method would work but would not allow the logical abstraction of the record to be kept. Moreover, instances could not be coded where the data types of the components were not the same for more than one of the variant choices. (See figure 4.)

```
procedure Sample__3 is
      type Q is (A, B, C) ;
      type Rec (X : Q : = A) is
         record
                        First   :  Integer;
                        Second :  Float;
                        Third   :  Float;
                        Fourth :  Integer;
         end record;

Foo : Rec (3);
begin
      null;
end Sample__3;

**Referencing Mechanism:  Foo.First
```

Fig. 4. Method 3

Each component within the record could only be one data type and every record contained every component. Every time choice A, B, or C was called, the entire record was accessed whether or not it was needed. Further, record D could not be coded because component "first" had already been declared an integer data type and, therefore, could not be declared a float data type. This method did not entail the use of the variant record and necessitated a waste of memory space.

Method 4

Method 4 shows the only solution to this naming or coresident homograph problem. (See figure 5.) With this method, separate records were created for each component grouping desired, and the data types needed for each component were declared within each record. Then a variant record was created with choices A, B, C, and D. The data types of these choices were the component groupings of the previous records.

For example, RECORD_TYPE_A was a record type with components "first" and "second." Similarly, RECORD_TYPE_B, RECORD_TYPE_C, and RECORD_TYPE_D were each separate records. A variant record was created with components A, B, C, and D. The data types of these components were RECORD_TYPE_A, RECORD_TYPE_B,

```
procedure Sample__4 is
      type Q is (A, B, C, D) ;

      type Record__Type__A is
         record
                        First   :  Integer;
                        Second  :  Float;
         end record;

      type Record__Type__B is
         record
                        First   :  Integer;
                        Third   :  Float;
         end record;

      type Record__Type__C is
         record
                        Fourth  :  Integer;
         end record;

      type Record__Type__D is
         record
                        First   :  Float;
                        Third   :  Float;
         end record;

type Rec (X : Q : = A)  is
         record
            case X is
                 when A = >
                      A : Record__Type__A;
                 when B = >
                      B : Record__Type__B;
                 when C = >
                      C : Record__Type__C;
                 when D = >
                      D : Record__Type__D;
            end case;
         end record;

Foo : Rec (B);
begin
      null;
end Sample__4;
```

**Referencing Mechanism: Foo.B.First

Fig. 5. Method 4

RECORD_TYPE_C, and RECORD_TYPE_D, which were the names of the previously created records. To access component "first" with data type integer and component "second" with data type float, choice A first had to be selected from the variant record. Choice A, in turn, had component A with data type RECORD_TYPE_A. When data type RECORD_TYPE_A was referenced, a record RECORD_TYPE_A was found with components "first" and "second."

Ideally, the name of a component reflects the abstract definition of its usage. The "first" components could have differing data types yet still represent the same logical thing, but each variant version of the record is defined by two record types. Changing a data type for any choice required referencing the underlying record structure. Thus, another level of indirection has been added, and all of the referencing mechanisms have to reflect the two record types.

CONCLUSION

The naming or coresident homograph problem has presented some problems in implementing the Ada variant record. The wording of the variant record description in the Ada Language Reference Manual is ambiguous. The actual implementation of the variant record is counter-intuitive. When the author began to code the design, she was not even aware of the possibility of a problem. This unpredictable error can only be found when the variant record is actually implemented.

Fortunately, the compiler detected the error very early in the development process. The Ada Compiler Validation Capability test suite [2] does contain a test (#B82C01A) which insures that every validated Ada compiler will deal with the variant record construct error in exactly the same way.

The idea of a variant record is to be able to access one component or another regardless of which choice is referenced. Method 1 could actually work if Ada did not disallow it. The compiler knows which variant record component is being accessed without the extra record construct that is necessary in Ada (see Appendix for proof). As described in this paper, Method 4 can be used for variant records and allows the record structures to be easily understood by reading the source code. The only undesired trait is the extra level of indirection. The Ada Language Reference Manual should unambiguously highlight the way that the variant record construct works. The manual's current definition of a variant record is not sufficient to entail its proper usage.

STATUTORILY EXEMPT

BIBLIOGRAPHY

[1] Booch, G. *Software Engineering with Ada.* Department of Computer Science, USAF Academy, Colorado, 1983.

[2] Data & Analysis Center for Software. *Revisions to the Compiler Implementer's Guide.* Griffiss AFB, New York, July 1984.

[3] Department of Defense. Ada Joint Program Office. *Reference Manual for the Ada Programming Language.* ANSI/MIL-STD-1815A-1983. Washington, D.C., October 1983, Sec. 8.3.

[4] Defense Communications Agency. Defense Data Network Program Management Office. *Defense Data Network X.25 Host Interface Specification.* Washington, D.C., December 1983.

[5] International Telephone and Telegraph Consultative Committee (CCITT). "Interface between Data Terminal Equipment (DTE) and Data Circuit-terminating Equipment (DCE) for Terminals Operating in the Packet Mode on Public Data Networks," *Recommendation X.25,* 1980.

[6] *Rationale for the Design of the Ada Programming Language*. SIGPLAN Notices, Vol. 14, No. 6, New York: Association for Computing Machinery, Inc., June 1979.

[7] Stallings, W. *Data and Computer Communications*. New York: Macmillan Publishing Co., 1985.

[8] Tanenbaum, A. *Computer Networks*. Englewood Cliffs, New Jersey: Prentice-Hall, 1981.