AN ADA™ COMPATIBLE SPECIFICATION LANGUAGE

N.C.L. Beale - S.L. Peyton Jones

Beale Electronic Systems Ltd.,
Wraysbury, Berks, England.

ABSTRACT

This paper describes a notation for the formal specification of software packages. The main influences are the guarded commands of Dijkstra and the Algebraic Semantics of Guttag. However, a novel operator denoted by % is introduced, which allows algorithms to be abstracted in a specification, thereby creating a true specification language rather than another higher level language. The notation, called ADL/1, is designed to be used in conjunction with ADA™ but is equally suitable for other languages, and has been used for real time software written in Assembler and in a PASCAL-like language.

1. INTRODUCTION

The purpose of this paper is to describe work that we have been doing at Beale Electronic Systems Ltd. on a practical notation for specifying software packages. This work represents a logical extension of ADA's approach to some of the problems of programming in the large. In ADA a package is thought of as being a package specification and a package body. We are describing a notation for writing a semantic specification which describes the intended semantics without describing the implementation. We call our notation ADL/1.

Using this, we have built some real software such as a real time distributed multi-tasking operating system (written in Z80 Assembly language) and two programs written in a PASCAL-like language: an industrial data handling system and a plant data acquisition and control package. Thus we are describing a notation which has been used on medium sized practical problems. We have also found ADL/1 a useful tool for specifying fragments of program.

In this paper we begin by describing how to specify functions and program fragments. We move on to specifying packages and finally indicate how to specify tasks. We end with some conclusions. The paper assumes a basic familiarity with the ADA report.

2. SPECIFYING FRAGMENTS AND FUNCTIONS

2.1 Guarded Commands

The basic construct of ADL/1 is that of the guarded command, [Dijkstra 1976]. If P1..Pn are predicates and CL1..CLn are lists of commands then

```
if P1 -> CL1 ¦ P2 -> CL2 ¦ ... end if
```

or equivalently

```
( P1 -> CL1 ¦ P2 -> CL2 ¦ ... )
```

means:

if all the Pi are false then abort but if there exists an i such that Pi is true then execute CLi.

and

```
do P1 -> CL1 ¦ P2 -> CL2 ¦ ... end do
```

means:

if all the Pi are false then do nothing (null) but if there exists an i such that Pi is true then execute CLi and repeat.

Note that if more than one Pi is true the choice is arbitrary. The predicates P1..Pn are called guards, and the set

```
P1 -> CL1 ¦ P2 -> CL2 ¦... ¦Pn -> CLn
```

is called a guarded command set. We use else as a pronoun to stand for the negation of all the previous guards in a guarded command set.

™ADA and Ada are trademarks of the U.S. Department of Defense.

2.2 Predicates and the % operation.

Predicates in ADL/1 must not have side effects but are generally like ADA boolean expressions. However the power of predicates is greatly extended by the distinctive new operation of ADL/1, the percent operation. ADL/1 allows new free variables to be introduced in a predicate by prefixing them with a % sign.

The construct:

```
P(%x) -> command_list
```

means:

> If x can be found such that P(x) is satisfied, then execute the command_list. The scope of the newly bound variable x is the rest of the guard, together with the command list which it guards.

The % symbol is used on the first occurrence of the new variable only. It can be qualified with min or max, and the variable can be declared to be of a given type in the usual way. The syntax is:

```
⌊min ¦ max⌋%Variable_Name⌊:Type_Declaration⌋
```

For example:

```
P( max%x ) -> command_list
```

chooses a maximum x which satisfies P(x).

If there is more than one min or max in a predicate then values for all the variables are chosen simultaneously to satisfy the predicate (if possible), the leftmost min or max taking priority over the second and so on.

It should be noted that this percent operation is possible only in specification languages. Consider, for example, the following specification of a square root program, which carefully gives no clues as to implementation:

```
(ABS(%x ** 2 - n) <= EPSILON * x ->
   return x
¦ else -> raise NUMERIC_ERROR )
```

2.3 Choice Constructs.

Percent_Variables can appear in a guard in any place that a variable could appear, but they can also appear in a Choice_Construct of the form:

```
(⌊Choice_Qualifier⌋ Choice {,Choice}::Predicate )
```

where

```
Choice_Qualifier ::= all ¦ is ¦ set
Choice::= Percent_Variable ⌊IN Set_Expression⌋
```

examples are:

```
(%x::P(x))
(min%f1,%y in SY ::
  (max%f2,%x IN SX :: f2=F(x,y)) AND f1=f2)
(all%x::P(x)=>Q(x))
(is%y::R(y))
```

The first example chooses an x to satisfy P(x), and so is equivalent to P(%x). However the Choice_Construct makes explicit the range of possibilities over which the choice is to be made, and exactly what predicate is to be chosen.

The second example is the standard MiniMax problem, where the inner maximisation is free to choose f2 and x for a given value of y.

The third and fourth examples are the standard universal and existential quantifiers, and mean:

For all x such that P(x) we have Q(x)

and

There is a y such that R(y)

respectively.

2.4 Sets and strings.

ADL/1 makes extensive use of sets and (general) strings. Objects can be declared as sets or strings as follows:

```
SX,SY:set of INTEGER
Seq1,Seq2:string of REAL
```

and the operators '=', '&', '*' are extended to mean set equality, union and intersection.

Sets and strings can be created by Choice_Constructs of the form:

```
(set %X IN SX :: X < 99)
(string %X IN SX :: x < 99 )
```

These are read as "The set of X in SX such that X < 99" and "A string of X in SX such that X < 99" respectively.

The operator rem, standing for remove (overloading ADA's rem) works with sets and strings as follows:-

```
e:ELEMENT --some type
S:set of ELEMENT;
T:string of ELEMENT;
k:INTEGER;

S rem e means (set %y in S :: y /= e)
T rem e means (string %y in T :: y /= e)
k rem T means (string %y in T ::
                  T(%j) = y and j /= k )
```

In these, the scope of the new free variables is restricted to the Choice_Construct, whereas in an unqualified Choice_Construct the new variables are visible for the rest of the predicate and for the command list which it guards.

2.5 Examples

The following three examples illustrate some of the concepts used so far:-

```
procedure Schedule is
  if max % n :: Priority (%p) = n
         and Status(p)=Ready -> Run(p)
  ¦ else -> Run (Diagnostics)
  end if;
end Schedule;

function F_Inverse (x:outtype) return intype is
  ( F(%y)=x -> return y ¦ return null )
end F_Inverse;

type Queen is array (1..2) of 1..8;
predicate Threatens (q1,q2:Queen) is
  (is %i :: q1(i) = q2(i) )
      or ABS (q1(1)-q2(1)) = ABS (q1(2)-q2(2))  );
function F return  set of Queen is
  if (%A:set of Queen)'N = 8 and
   (all q1,q2 in A ::
          Threatens(q1,q2) => q1=q2 -> return A
  end if
end F;
```

3. SPECIFYING PACKAGES

A package is a group of related entities such as types, objects and subprograms , whose inner workings are concealed and protected from their users. The state of a package is the direct product of its visible state and its internal state, which is not accessible to the outside world. Our semantic specification must define the internal state and the intended semantics of any subprograms. We do this by defining an abstract package which has the same specification as the intended package body but whose internal data structures and algorithms are much simpler than the intended implementation. This abstract package has a set of possible behaviours, and a package body satisfies the specification if its behaviour set is a subset of the possible behaviours defined by the abstract package.

We call the procedures and functions which a package makes available to the outside world its operators. An operation is an operator together with its parameters.

3.1 The abstract state ($)

ADL/1 represents the internal state of a package by a string or set of operations, denoted by $.

The ADL/1 specification of a package describes:

a) What the possible internal states of the abstract package are

b) How the state of the abstract package changes as operations are applied to the abstract package.

c) How the out-parameters are related to the state and the in-parameters

3.2 Defining the internal state $

The internal state space of a package is defined as follows:

```
$ is [set]
  Operator_Declaration
  {;Operator_Declaration}
  end $
```

For example in:

```
$ is
  PUSH:INTEGER
  SIZE:RANGE 0..10000
end $;
```

Two possible values of $ are:

```
     PUSH(3),PUSH(4),SIZE(99)
and  PUSH(3),PUSH(4),PUSH(3),PUSH(20),SIZE(77)
```

$ is a string unless it has the qualifier set in which case it is a set.

3.3 Operations to change the current state

$ is a string or a set. It is therefore possible to use the operations defined in section 2.4 to modify $. We have found it convenient to introduce some notations to make these modifications easier to write, and these are described in this section. In the definitions that follow

Op stands for some operator
Opn stands for some operation
OpT stands for some operation_template

An operation_template being an operator some of whose parameters may be free variables (with % s) or replaced by ? . A ? is equivalent to %new_unique_name.

For example:

```
PUSH(%X), CREATE(?), PUSH(X), PUSH(4)
```

are operation_templates.

The basic way of obtaining information about the current state is through the '$>' operator. The predicate '$>OpT' (pronounced "$ contains OpT") is equivalent to:

```
(%IT :: $(IT) = OpT)
```

if $ is a string, and

```
(%IT in $ :: IT = OpT)
```

if $ is a set.

These are both true if and only if OpT in $ is true, but they introduce the pronoun IT which can be used later in the guard.

We can add an element to $ by writing

```
$ &= Opn      -- equivalent to $:=$ & Opn
```

We allow two further overloadings for rem :-

```
rem Opt    is equivalent to  $:= $ rem Opt
rem k      is equivalent to  $:= k rem $
```

and we let

```
set Op(x,y)      mean  rem Op(?,?); $&=Op(x,y)
set1 Op(x,y)     mean  rem Op(x,?); $&=Op(x,y)
set2 Op(x,y,z)   mean  rem Op(x,y,?);$&=Op(x,y,z)
```

3.4 Specification of Packages

We have described how the abstract state is defined and modified. By using the notations of section 2 we can define the intended effect of functions and procedures. Exceptions, in so far as they are visible to the user, can be raised explicitly. Thus we are able with these concepts to specify arbitrary Ada packages.

3.5 Example.

We are now ready to consider an example of an ADL/1 package specification. The example is taken from the Ada manual [Ada 1980], page 7.8.

```
-- We begin with the Ada package specification

package Table_Manager is
type Item is
record
  Order_Num : INTEGER;
  Item_Code : INTEGER;
  Item_Type : CHARACTER;
  Quantity  : INTEGER;
end record;
Null_Item : constant Item :=
  (Order_Num | Item_Code | Quantity => 0,
           Item_Type => " ");
procedure Insert( New_Item:in Item );
procedure Retrieve( First_Item:out Item );
Table_Full : exception;  -- May be raised by Insert
end;

-- Now the ADL/1 specification

semantics Table_Manager is
  Size: constant := 2000;
 $ is set
  i:Item
  end $;
  procedure Insert(n:Item) is
    ($'N < Size-> $&= I(n)
    |else->raise Table_Full     )
  end Insert;
  procedure Retrieve(out First:Item) is
    ($>I(min%o,%c,%t,%q)->rem IT;
               First:= (o,c,t,q)
    |else->First:= Null_Item      )
  end Retrieve
 end Table_Manager;
```

It is instructive to compare this 14 line specification with the 24 line partial package body given in the Ada report.

4. FURTHER REMARKS

In this section we briefly discuss some further implications of the work we are describing, which cannot be developed fully in this paper due to lack of space.

4.1 Specifying Tasks

Using the Ada entry handling constructs we can specify Ada tasks in ADL/1 using only the notations described above. However ADL/1 can also be used for specifying task semantics (not necessarily of Ada tasks) in a more direct way.

To do this we extend the notion of Operation to include a priority level.

```
n * Opn
```

denotes an operation which is Opn at priority level n.

We can now consider a task as performing a sequence of atomic (that is, indivisible) operations, which are executed by the following procedure (called an axiom grinder).

```
do $> max%n * %Opn ->
       rem IT; Perform (Opn)
else -> null      --infinite loop
end do
```

By including commands of the form

```
$&= 3*Opn
```

in the command list for performing the operations, it is possible to specify chains of operations without overspecifying them.

4.2 Mathematical Foundation

The theoretical foundations of ADL/1 are the notion of a string, assignment, the guarded command and the % operator. The semantics of assignment and the guarded command are given in [Dijkstra 1976] in terms of weakest preconditions.

If C is a command and P a predicate then

```
wp (C,P)
```

is the weakest predicate such that if wp(C,P) is true and C is executed then C will terminate and P will be true.

In this notation, the fundamental semantic rule for % is:

```
wp( G(%x) -> C, P) =
   (is %y :: G(y)) and G(x) => wp(C,P)
```

This is not circular because the %y is only our notation for the standard existential quantifier.

The % operator is in fact closely related to the Tau operator of [Bourbaki 1968], which is used to develop the whole foundations of mathematics.

4.3 Ada Commpatibility

ADL/1 is designed to be compatible with Ada, and thus to provide a unified notation for writing specifications, programs and predicates.

We have added $ and % to the basic characters and

-> , &= and ::

to the compound symbols.

The ADL/1 words

max , min , set , semantics and string

are all used in contexts where identifiers cannot appear, and thus do not have to be new reserved words.

5. CONCLUSION

We have described the basic syntax and semantics of ADL/1, and provided an example of its application.

The use of the % operator permits algorithm abstraction analagous to the data abstraction facilities in many modern languages.

ADL/1 has sufficient facilities for non-determinacy to specify software without over-specifying it.

One problem with most specification languages is that they become unwieldy for large programs, a property they share with programming languages themselves. ADL/1 suffers much less from this than most languages. For instance, a three page ADL/1 specification specifies about 2000 lines of a PASCAL-like language. ADL/1 is significantly more compact that OBJ [Gougen 1979], which is one of the best published specification languages we have seen.

By providing an integrated Ada compatible notation for specifying and reasoning about programs, it has proved to be helpful to us in medium scale industrial control software. We believe it is a useful tool for any software within the Ada application domain.

ACKNOWLEGEMENTS

This work has benefited particularly from conversations with Martin Beale, Veronique Donzeau-Gouge, Manny Lehmann and Maurice Sloman. The comments of the referees have also been most helpful.

REFERENCES

[ADA 1980] Reference Manual for the Ada Programming Language. US DoD July 1980

[Bourbaki 1968] Bourbaki N Theory of sets Addison Wesley 1968

[Dijkstra 1976] Dijkstra E. W. A Discipline of Programming Prentice Hall 1976

[Gougen 1979] Gougen J.A. An Introduction to OBJ Proc. IEEE/ACM Symposium Specification of Reliable Software 1979 pp 170-189.

[Guttag 1978] Guttag J.V. et al Abstract Data Types and Software Validation. Comm ACM Vol 21 No 12 [Dec. 1978] pp 1048-1064

The programs written with the aid of ADL/1 are described in the following:

S.L. Peyton Jones and N.C.L. Beale An Advanced Concurrent Processing Operating System for Multiple Microcomputers. Unpublished paper.

BPMS Batch Production Monitoring System Manual

PVM Process Variable Manager Data Sheet

Available from the authors.