1 Introduction

The Modula-2 programming language is useful in teaching computer science concepts beyond the introductory level because it is strongly typed and supports such constructs as external modules, separate compilation, low level interfaces, and co-routines. However, understanding the more complex concepts such as information hiding, scope, and quasi concurrent processes requires information on how the software actually executes. This paper presents a model which can be used to look at program execution in terms of the source code and thus avoids the complication of dealing with the complexities at the assembler/compiler level. It is an extension of work done with other languages (Birtwistle, et.al, 1981, 1986). The model illustrates what actually occurs and introduces concepts of "block instances", static and dynamic links and the binding rule as building blocks. The combination of these can be used to produce a visual model of the execution.

2 Basic Components

2.1 Block Instances

A block of code (code between a *begin* and *end* or a procedure) can be visualized as a **block instance**. A block instance has three components — a heading, declarations, and executable code or body. The heading contains a unique identifier, a static link and a dynamic link. The declaration portion contains information on types, variables and procedures which are to be used within the block. It may also contain the addresses indicating where the value is stored. The body contains the executable code. Since the execution of the program moves sequentially through the code, a pointer (called *sequence control* or *SC*) indicates the currently executing statement. Block instances do not usually exist until *SC* moves to the beginning of a block code or to a procedure call.

In visualizing Modula-2 execution, each module is visualized as having its own stack which contains the active block instances for that module, rather than coping with one large stack. The order of stack initialization is such that modules which are referenced by other modules are initialized first. Implementation modules require a partner definition module. For our purposes, these are incorporated into the declaration portion of the block instance for the equivalent implementation module at run time. (During compile it is only necessary to know what the values and their structures are but it is not necessary to create the entity or assign storage at compile time). These features and the additional components are best discussed using examples.

Suppose that we have a simple program which assigns a value to an integer variable and writes this value to the screen as shown below. Figure 1 provides a visual representation of the block instances at the time that execution has called *WriteInt* in the program module.

```
MODULE ex1;

FROM InOut IMPORT WriteInt;

VAR   x : INTEGER;

BEGIN

   x := 3;

   WriteInt(x,5);

END ex1.
```

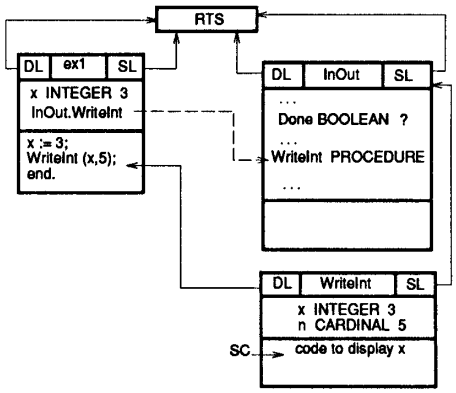

Figure 1

When the program is executed, the run-time system (RTS) is called and some book-keeping is done and then the execution begins. We first create a stack for the module *InOut* since it is used by *ex1* and must be created first. Initialization produces a block instance for the module. The dynamic link, **DL**, provides the position to which execution will return when all requirements for the block instance are complete and it therefore points back to the RTS. The static link, **SL**, points back to the block where the declarations exist which are required to produce the block new instance. Since the RTS can also be considered as a block instance, SL points back to the RTS. The declarations for *InOut* contain the variables, constants, and procedures for the block instance. Note that it is

only here that the address of any variable which is declared exists. For purposes of the model, we show the value of the variable in this section of the block instance. Any initialization which might be required in the body of the implementation module is also accomplished at the time that the block instance is created. In this case, this has been left blank since any of these operations are transparent to the user for a library module.

Similarly, a block instance is created for *ex1* with DL and SL pointing back to the RTS. The declarations include an integer x and a procedure imported from InOut called WriteInt. The declarations do not include the code for the procedure but only a link to the procedure declaration in InOut (shown with a broken line). The value of x is not yet determined. When all of the declarations are complete sequence control, *SC*, is positioned at the first instruction which sets the value of x to *3*.

2.2 Binding Rule

Locating an identifier (variable, constant, procedure) requires the application of the *binding rule*. This rule can be stated as:

> Look for the quantity in the declarations of the current block instance. If it is found then the search ends. If it is not found, then follow the SL to the block instance and look in the declarations here. This process is continued until the search is successful.

Thus, x is found in the declarations of the current block and the value is placed here. When the call is made to *WriteInt*, the current block instance is searched for the procedure. Because the import statement has indicated that the procedure exists in another module and has set a link to that module, execution then moves to *InOut* and a block instance for the procedure created in the stack for *InOut*. Since the values in the paramemter list of the procedure are passed by value, the

declarations for *WriteInt* have assigned variable names corresponding to the formal parameter list and the values passed to the procedure are placed in the appropriate positions in *WriteInt*. Note that passing by value does create another set of values in the newly created block instance and the original values are unchanged in the calling block instance.

When the procedure has written the appropriate value to the screen, the code reaches an *end* statement. Whenever an *end* statement which terminates the block is encountered, the block instance is removed and execution continues at the position indicated by the DL. In this case, execution returns to the calling position, and since this statement is complete, moves to the end of the program. Since this *end* statement terminates the block instance for *ex1*, the instance is removed and control passes to the RTS as indicated by DL. When the RTS regains control after the removal of the program module, all other stacks (modules which were initialized) are also removed and a final clean-up done.

3 Reference and Value Parameters

When a procedure is called, this results in the creation of a block instance with the DL showing the return position when the procedure is complete and the SL indicating the block instance which contains the declaration of the procedure which can be used to create the block instance. If the procedure has no parameters, then the binding rule uses SL to find the position (block instance) where the variables in the procedure are stored. In creating the block instance for procedures with parameters, the declaration of the procedure contains the formal parameters and their types. However, reference and value parameters are treated slightly differently. When a formal parameter is called by value, the formal parameter name is given to the variable within the declaration of the

block instance and the value which this parameter has on the call is inserted into the declaration. If, however, the parameter is a reference parameter, the address of the storage location is placed into the formal parameter name position rather than the value itself. This is shown with an @ and then a broken line is used to indicate the position where the actual values are stored.

Consider the following simple program.

```
MODULE norm;
FROM RealIO IMPORT ReadReal, WriteReal;
TYPE vec = ARRAY [1..5] OF REAL;
VAR v : vec;
    success : BOOLEAN;
PROCEDURE readvec (VAR x : vec; n : INTEGER);
  VAR k : INTEGER;
  BEGIN
     FOR k := 1 TO n DO
        ReadReal(x[k]);
     END;
END readvec;
PROCEDURE writevec ( x : vec; n : INTEGER);
  VAR k : INTEGER;
  BEGIN
     FOR k := 1 TO n DO
        WriteReal(x[k], 10, 2);
     END;
```

```
END writevec;

BEGIN

   readvec (v, 5);

   writevec (v, 5;

END norm.
```

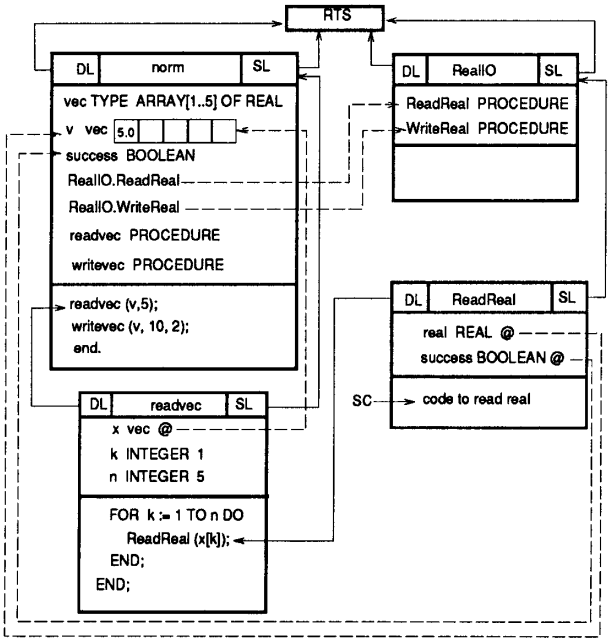

Figure 2

Figure 2 indicates the block instances at the time that *readvec* is called and the first read has just been completed. As shown previously, two stacks are created. The stack for *RealIO* is created first, followed by the stack for *norm*. The block instance for procedure *readvec* is created from the definitions in *norm*. Note that the formal parameter for the array (called x) is called by reference so the formal value indiates that the position where the values are stored is in the block instance for *norm*. The values for k and n in the formal parameter list are given the values which existed at the proce-

dure call, and the loop begins executing. The call to *ReadReal* results in a search for the identifier in the current block and since it is not found here the search continues in the block indicated by SL (i.e. *norm*). Because this is an IMPORTED procedure, the linker has already determined that it exists in the stack defined by the block instance *RealIO*. Thus a block instance for *ReadReal* is produced in this stack using the information from the declarations of *RealIO*. Since both of the parameters for the procedure are called by reference, the block instance which is to contain the values is indicated. In this case, it is a block instance in another stack. When *ReadReal* has obtained the first value, this is stored in the block *norm* as shown in the diagram. Note that the DL indicates that SC moves back to the *readvec* block instance. At that time, the block instance for *ReadReal* will disappear, the value of k incremented, and another call made to *ReadReal* which repeats the process of creating *ReadReal*. When *readvec* is complete, the block instance is deleted, along with the values for k and n. The formal value for the array (x) also disappears but since it indicated that the values read were to be stored in *norm*, the array v still contains the values. SC returns to the statement which called *readvec* and since this is now complete, it moves to the next statement to write the values.

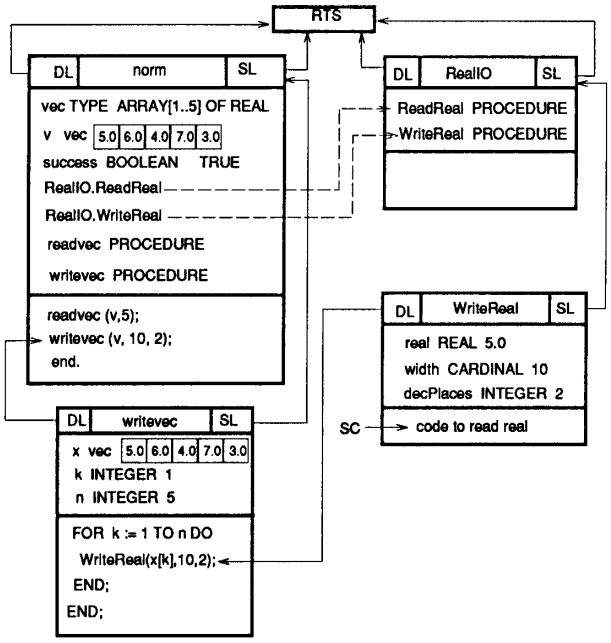

Figure 3

Figure 3 shows the block instances when *writevec* has been called and the first value is to be written to the screen. Note that the values for the array are now stored in the block instance for *writevec* as a local copy in array x since these are called by value. When *WriteReal* is called, all are value parameters and thus local copies are made for *real, width,* and *decPlaces.* The value for *real* is found using the binding rule which locates the first value of x in the block instance *writevec.*

4 Name Parameters

The structure of Modula-2 makes it possible to group procedures which have the same formal parameter list structure into a single type. Thus each procedure must have the same size parameter list with the same types for the parameters. This type can then be used as a parameter in other

procedures, with the appropriate name of the desired procedure inserted at call time. Consider the example below.

```
MODULE tab;
FROM InOut IMPORT WriteCard, WriteLn;
TYPE
    CardFunc = PROCEDURE(CARDINAL) : CARDINAL;
PROCEDURE Tabulate( F : CardFunc; Limit : CARDINAL);
VAR
    N : CARDINAL;
BEGIN
    FOR N := 0 TO Limit DO
      WriteCard(N,10);
      WriteCard(F(F(N)),10);
      WriteLn;
    END;
END Tabulate;
PROCEDURE Sum (N : CARDINAL) : CARDINAL;
BEGIN
    RETURN (N * (N+1) DIV 2);
END Sum;
PROCEDURE SumSq(N :CARDINAL) : CARDINAL;
BEGIN
    RETURN (N * (N+1) * (2 * N + 1) DIV 6);
```

```
END SumSq;
BEGIN
    Tabulate(Sum,5);
    Tabulate(SumSq,5);
END tab.
```

Procedures *Sum* and *SumSq* are both of type *CardFunc* since they both have one cardinal value parameter and return a cardinal value. Thus, procedure *Tabulate* which has a parameter *F* of type *CardFunc* is called, either *Sum* or *SumSq* can be used in the evaluation to produce the number which is printed to the screen. The procedure is called by name. Figure 4 shows the block instances as they exist at the time that *WriteCard(F(F(N)),10)* is called.

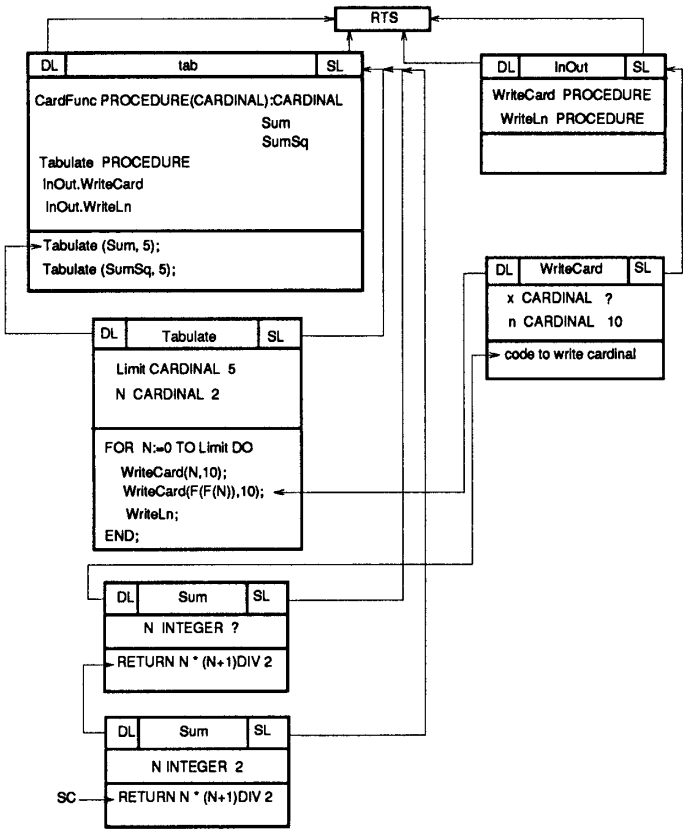

Figure 4

Note that in the declarations of *tab*, the procedures *Sum* and *SumSq* have been placed in the type *CardFunc* to identify them not only as procedures but also as procedures of a specific type which can be called by name as a parameter in other procedures. The call to *Tabulate* results in a block instance for this procedure. At this point, N has been incremented twice and the value of N has been written to the screen. We are now executing the second call to *WriteCard*. This has resulted in the block instance for *WriteCard* in the *InOut* stack. Since the parameters passed are by value, a local copy is made for the two values. The value of n has been stored but the value of x is not yet determined. This requires a call to the procedure *Sum* which results in a new block

instance under *Tabulate*. The SL for *Sum* is used to find the definition for the creation of the block instance. The other quantity *N*, exists in the definition for *Tabulate*. Since the value needed is not yet determined, another block instance must be created for *Sum*, but this time it has been passed a value of *2*. The statement is then executed and a value returned to the point indicated by DL and the block instance deleted. Since this statement has not yet been completed, the value of *3* is the inserted so that this can be completed, the value obtained (6), and then passed to the point indicated by the DL before deleting the block instance for *Sum*. *WriteCard* can then complete its execution and return to the point indicated in *Tabulate* and the block instance for *WriteCard* deleted.

5 Co-routines

In imperative languages, code is normally executed sequentially and it is the aim of good languages to contain only controlled goto type of statements. Thus when the statement contains a call to another procedure, a new block instance is created and statements are executed seequentially in this block instance as well. In addition, the DL is automatically set to the statement which resulted in the creation of the new instance so that execution will resume here. Then if the execution of that statement is complete we move on, otherwise, as is the case for functions, execution of the statement can continue to completion. Modula-2 also allows for co-routines. In this case, it is necessary to find a means of retaining a block instance when execution resumes at some other position in the program. It is also necessary to find a way to cope with DL since we may want a choice of where to go next rather than to go to the statement which called the block instance. This is accomplished through a module called *SYSTEM* which interfaces with the RTS and has access

to information and processes which are normally controlled here.

Consider the following code.

```
MODULE CoRoutine;
FROM SYSTEM IMPORT ADDRESS, WORD, TRANSFER, NEWPROCESS,ADR;
FROM InOut IMPORT WriteCard;
VAR  num : CARDINAL;
     p1, p2, Exit : ADDRESS; (* for coroutines*)
     WorkOne, WorkThree : ARRAY [1 .. 1000] OF WORD;(*workspace*)
PROCEDURE IncOne;
BEGIN
 LOOP
   INC (num);
   WriteCard(num,5);
   TRANSFER (p1,p2);
 END;
END IncOne;
PROCEDURE IncThree;
BEGIN
 LOOP
   INC(num,3);
   WriteCard(num,5);
   IF num < 15 THEN
      TRANSFER (p2,p1);
```

```
    ELSE
       TRANSFER (p2,Exit);
    END;
  END;
END IncThree;
BEGIN
    num := 0;
    NEWPROCESS (IncOne, ADR(WorkOne), SIZE(WorkOne), p1);
    NEWPROCESS (IncThree, ADR(WorkThree), SIZE(WorkThree), p2);
    TRANSFER (Exit, p1);
END CoRoutine.
```

Figure 5 shows the execution at the point where the call to *TRANSFER(p2,p1)* in *IncThree* is just being completed. At this point, *num* has a value of 4 and the control of execution is returning to *IncOne*.

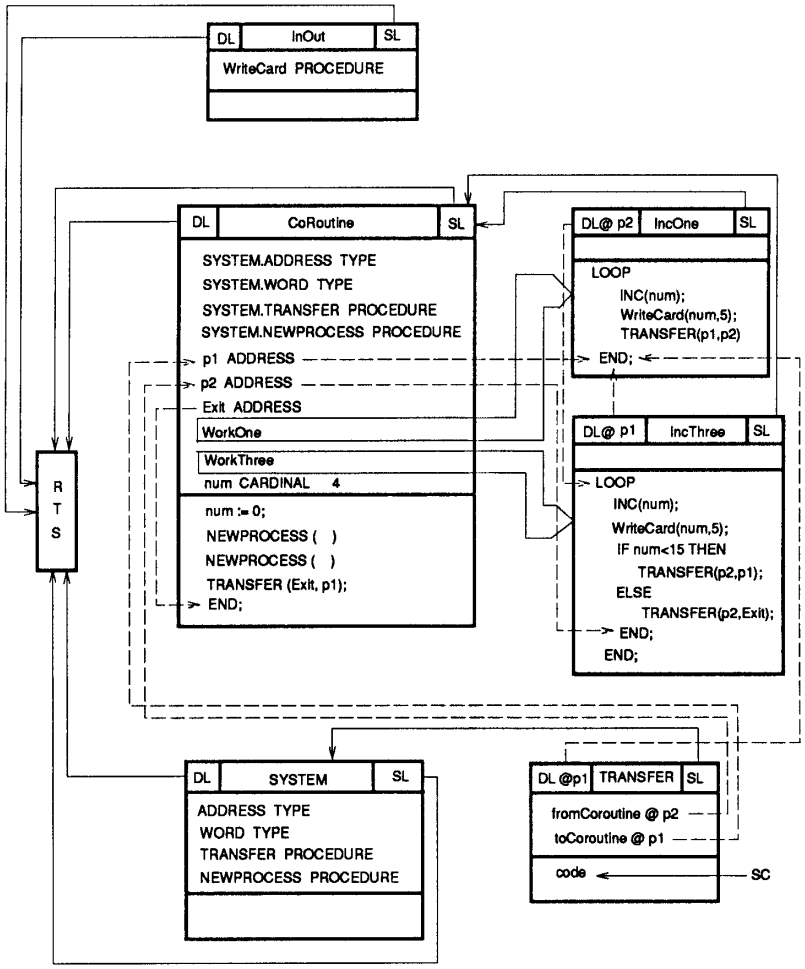

Figure 5

When the program began execution, *InOut* and *SYSTEM* were initialized, followed by *CoRoutine*. *p1*, *p2*, and *Exit* are all indicators which can be used to position SC. These variables are set at the time a NEWPROCESS procedure is executed, or at the time that a TRANSFER procedure is executed. When execution begins, *num* is given its initial value and then a new process (*IncOne*) is created. This results in a block instance for this procedure, but there is an important difference.

This block instance is created within the declarations of *CoRoutine* in the workspace array called *WorkOne*. At the same time, *p1* also is modified to show that SC would move to the first statement in the procedure at the time that the procedure was the one which was active. Similarly, this is accomplished for the procedure *IncThree*. At this point the DL is not set. SL points back to the *CoRoutine* declarations so that any variables not newly defined can be accessed within the main progam. When the TRANSFER statement is executed, *Exit* will be modified to contain the information to move SC to the statement after the TRANSFER call in the main program and SC moves to the position indicated by *p1* which is the beginning of the code in *IncOne*. When the code indicates that execution control should be passed to *IncThree*, DL is modified by *TRANSFER* to indicate that it should move to the position indicated by *p2*. Then execution moves through the statements until the TRANSFER call is made.

Figure 5 shows the situation at this point. Note that transfer modifies the values of *p1, p2* and the DL for *IncThree* and *TRANSFER* so that following DL will get to the appropriate point in *IncOne*.

Because the block instances exist in *CoRoutine* and *TRANSFER* modifies the DL for the appropriate block instances, the block instances do not disappear since they never reach the END statement for the block instance. When SC finally is transferred back to the main program, the end will be reached and the RTS will delete all block instances.

6 Conclusions

A simpler version of this model has been used with Simula and Pascal to help understand what actually happens when program execute. It is particularly useful to explain parameter passing

and recursion in these languages. The extensions included here in a brief form have been used successfully as a way of helping students understand scope, separate modules, information hiding and parameter passing in Modula-2. The model can be extended to show other low level interfaces and has been used successfully in giving a good overview of what might be some critical issues in compiler construction.

7 References

Birtwistle, G., Dahl, O-J., Myrhang, B., and Nygaard, K. (1982) *Simula Begin.* Sweden : Student Literatur.