A Program Development System

Execution Supervisor

James E. Archer, Jr.
Andrew I. Shore

TR79-397

Department of Computer Science
Cornell University
Ithaca, New York 14853

This work was supported in part by a grant from the NSF MCS77-08198.

A Program Development System Execution Supervisor

James E. Archer, Jr.
Andrew I. Shore
Department of Computer Science
Cornell University

The Cornell Program Development System is an experimental vehicle to explore the applicability of highly cooperative tactics to a contemporary development environment. The CPDS provides significant facilities for modifying and immediately executing programs. The execution supervisor and the internal user program representation it uses to implement these facilities are described.

The Cornell Program Development System is an experimental vehicle to explore the applicability of highly cooperative tactics in a contemporary development environment. These tactics have been extensively explored in earlier Cornell compilers, but have not previously been exploited in a highly interactive environment with full-screen display capability. Both the user interface and the overall system architecture are described in other reports [5, 6]. The purpose of this report is to describe, in more detail, one of the more significant and innovative modules in the system.

The CPDS is faced with a demanding task. For each small unit of the program (essentially for each "statement"), the PDS must be prepared to:

a. display the unit, in source form, to the user
b. evaluate the syntactic correctness of the unit, and its compatibility with the context in which it appears
c. execute the unit, essentially immediately
d. have this unit replaced with another, and still be able to perform the forgoing functions
e. have other units be replaced that affect the interpretation of this unit, and still be able to perform the forgoing functions.

Much of the burden of this task falls on a module called the "execution supervisor", and on the internal representation of the user program that was designed to support this module. These are the subject of this report.

The CPDS has been designed to be essentially language-independent in its architecture, although it is highly language-cognizant in its behavior. That is, while its user interface appears to be completely integrated with a particular host language, in fact the system is readily adaptable to any modern block-structured language of modest size. PASCAL [8], or a highly disciplined subset of PL/I [7] are obvious candidates, but an instructional subset of Ada [3, 1] has been chosen for the prototype implementation, primarily to become familiar with this new language. The details and examples in this report are all based on this Ada/CS prototype of the CPDS.

Architecture

The structure of the CPDS system mirrors the user's view of the entire system. The fundamental entities in the system are simply the user terminal and files. A user session consists of a series of operations initiated from the terminal to change one or more of the files in the system memory. System memory appears to be homogeneous; no distinction is made between primary and secondary memory. Each file change is (optionally) displayed on the user terminal screen for review and possible modification. The description of the CPDS architecture [6] is necessary to fully understand the motivation of the execution supervisor implementation.

The execution supervisor is called to perform the execution of a procedure or section of commands. The primary data objects manipulated are an environment file, which includes a statement location descriptor, values of all variables and system control information, one or more program files, and any data files used, including terminal input and display. The execution supervisor is responsible for maintaining the environment file in such a manner to allow execution to be discontinued and restarted at any time. Based on user commands, check (variable modification history) and trace files are created which can be reflected on the user terminal to follow program execution. Trace output is in the form of a cursor following execution through a displayed copy of the program as it executes.

The basic flow of control is:

```
loop
    fetch next statement from procedure file;
    exit when interrupted or errors or file empty;
    execute instruction;
    modify CHECK and TRACE files (as required);
end loop
```

Six types of situations will terminate the execution interval:

1. Normal program termination.
2. The "pause" key was hit. This raises the interrupt flag which ES checks before beginning each new statement execution.
3. Limited length execution has run its course.
4. Unable to write to file because window is full; waiting for page turn.
5. File contained insufficient data; waiting for terminal input.
6. An error has been encountered.

Operation is synchronous in the sense that execution is never "surprised" in the middle of a statement, though error conditions may cause a statement to be suspended before it is completed. A statement whose execution is interrupted in the middle is restarted from the beginning. For input/output operations, execution is stopped and restarted at the beginning of transmission of each individual data item.

Program Representation

Programs are stored in a directly interpretable form specifically for the purpose of allowing free movement from program entry and modification to execution, and back. In this way, it is possible to create the appearance that the program text is being executed directly. The code structure is constrained to support the immediacy of interaction, retain the ability to be halted arbitrarily, and still sacrifice as little as possible in terms of execution speed.

Each user interaction dealing with the procedure is related to a record or set of records in the procedure. The appearance of the procedure on the screen is controlled by a decoder which reads the procedure records and materializes them into human-readable text. In order to facilitate this display, each statement in the procedure is allocated as its own record. The statements are organized into a doubly-linked, logically sequential file with additional pointers to represent the alternate execution paths of control structures. A more tree-like structure, based on the parse tree for the procedure, might be more easily built and maintained, but the access for window display would be greatly complicated.

Statement Records consist of a fixed length part (the stub), followed by a variable length part holding expressions or "phrases". The statement stub is used to maintain the structure of the program, link the statements together, and provide information necessary for the successful materialization of the statement for user inspection. Attached to each stub are a series of syllables which represent the execution of the statement. The syllables are code for a low-level stack-oriented virtual machine sufficient to implement PL/CS, Ada/CS, or PASCAL.

The following diagram depicts the salient portions of the statement structure. All pointers are file system addresses, not memory pointers.

```
                           <lexical back pointer>
                                     ^
                                     |
+------------------+--------------+-|-+---+-----------------+----...----+
| <statement type> | <flag bits>  | X | _ | <phrase pntrs>  | <phrases> |
+------------------+--------------+-|-+-|-+-----------------+----...----+
                                     |   |
                                     |   v
                                     |  <exec pointer>
                                     v
                           <lexical next pointer>
```

<statement type> identifies the statement to the decoder. All statements are not necessarily represented on the screen (e.g. "endthen", "endif", "endwhen").

<flag bits> are used to represent all sorts of information, including display characteristics, known errors in the statement, and other indicators of use to the various system functions.

<lexical back pointer> points to the lexically previous statement.

<lexical next pointer> points to the next lexical statement. Represents one of the possible execution flows out of the statement.

<exec pointer> (optional) represents a second execution flow out of the statement which is not present in single-exit statements such as assignment. For example, in an **if** statement, <exec> points to the first statement in the **else**-clause if it is present, or to the **end if** if there is no **else**-clause. In a repetitive **loop**, <exec> points to the matching **end loop**.

<phrase pointers> are offsets into the variable-length part of the statement -- the "phrases". The number of such pointers varies from one statement type to another, but is always constant for a given statement type. Phrase pointers are actually kept as local goto opcodes to make them most closely similar to phrase constructs themselves.

<phrases> make up the truly variable-length part of the statement. They represent the actual postfix code that will be interpreted by the stack-oriented expression interpreter.

During the interpretation of a procedure, execution proceeds by fetching each opcode (and any implied operands) from the current phrase position. Execution will pass to the next statement when an opcode is encountered that indicates that a new statement is to be fetched. All new statement fetches are performed on the basis of the <next> or <exec> pointers. All interpretation is done on the basis of the opcodes present in the phrases. At no time does the execution supervisor take the type of the statement being processed into account.

The normal execution sequence for most statements is to proceed to the next statement and begin executing at the beginning. The ES equivalent is to follow the <next> pointer and begin execution with the first phrase. The <next> pointer always points to the next statement to appear in the procedure, regardless of the statement type. For statements with only a single phrase, the first phrase position will begin directly with the opcodes necessary to perform its functions; for a more complicated statement (for instance, **while...loop**, the first phrase position will consist of a local goto to a later position within this statement. This goto mechanism is employed to make it possible for the phrase positions to be fixed, while still making it possible to proceed with interpretation without discriminating on the basis of the type of the underlying statement.

For an **if** statement, execution proceeds either along the <next> pointer (the condition is true) or along the <exec> pointer, executing the first phrase of the target statement in either case. The statements in the **then** (**else**) portion of the **if** are followed by an "endthen" ("endif") statement to simplify flow of control and statement insertion. The <exec> pointer of the "endthen" statement points to the corresponding "endif". The code for the "endthen" is the single opcode indicating that execution should proceed through the <next> pointer of the statement reached by following its <exec> pointer. By this device, the execution sequence for simple statements is always the same, regardless of the context into which they are placed.

Similar mechanisms are required for the repetitive constructs, of which the most complicated is **for ... loop**.

1. The entry phrase. Perform initialization. Go to phrase 3.

2. The increment phrase. Increment the index variable. Go to phrase 3.

3. The test phrase. Test for termination. If loop continues, follow <next> pointer, otherwise uninitialize index variable and follow the <next> pointer of the "endloop" statement pointed to by <exec>.

The code for the "endloop" statement causes a goto to its <exec> pointer, which points back to the "forloop" statement, starting at phrase 2. An **exit** from a loop follows the <next> pointer of the "endloop" statement in the same manner the test phrase does. The coding for iterative loops in Ada/CS, PL/CS, and PASCAL differs on the handling of the loop index at loop exit.

The following is an example of an **for ... loop** with an imbedded **if:**

```
            |
            |
+---------+-|-+
| S1      | X |             +---------------+
+---------+-|-+             |               |
            |               V               |
+---------+-|-+---+------+------+------+    |
| forloop | X | _ | init | incr | test |    |
+---------+-|-+-|-+------+------+------+    |
            |   '--------------+            |
+---------+-|-+---+------+     |    +-------+
| if      | X | _ | test |     |    |
+---------+-|-+-|-+------+     |    |
            |   '----+         |    |
+---------+-|-+      |         |    |
| S2      | X |      |         |    |
+---------+-|-+      |         |    |
            |        |         |    |
+---------+-|-+---+  |         |    |
| endthen | X | _ |  |         |    |
+---------+-|-+-|-+  |         |    |
            |   '----|--+      |    |
+---------+-|-+      |  |      |    |
| S3      | X |<-----+  |      |    |
+---------+-|-+         |      |    |
            |           |      |    |
+---------+-|-+---+     |      |    |
| exit    | X | _ |     |      |    |
+---------+-|-+-|-+     |      |    |
            |   '-------|------+    |
+---------+-|-+         |      |    |
| endif   | X |<--------+      V    |
+---------+-|-+                |    |
            |                  |    |
+---------+-|-+                |    |
| S4      | X |                |    |
+---------+-|-+                |    |
            |                  |    |
+---------+-|-+---+            |    |
| END     | X | _ |<-----------+    |
+---------+-|-+-|-+                 |
            |   '-------------------+
+---------+-|-+
| S5      | X |
+---------+-|-+
            |
            V
```

```
S1;
for I in 1..10 loop
   if (C2) then
      S2;
   else
      S3;
      exit;
   end if;
   S4;
end loop;
S5;
```

The choice of a program representation similar to the underlying machine operations makes execution efficiency more easily practicable. Since the system retains no free procedure text to remind it of the input form of the procedure, this information must be derivable from the executable form. Many of the display characteristics of the program are purposely removed from the user's concerns. For PL/CS, the system types directly reflect the underlying machine types and within the system's formatting rules the display form for the program is unique. Ada, and to a lesser extent PASCAL, provide facilities for attaching user-specific names to the underlying concepts, specifically naming types, specifying enumerations, etc. Most of this additional naming is handled directly by the display routines, but some additional cases require the addition of display-specific information to the executable portion of the statements. The frequency of these additions is not such that the user will notice their presence in execution speed.

Addressing Structure

In order to allow a session to be terminated at any time, CPDS maintains all system components as files. As a result, all variable references are to relative file addresses, rather than to actual memory locations. Since the same internal form must serve both for execution and display, all relevant information concerning the variable must be accessible from the same representation:

1. The current value of the variable.

2. Whether the variable is to be CHECKed or not.

3. The name of the variable.

These requirements are presented in order of decreasing frequency and ease of access. Variables are accessed by fixed offset from the current environment, much as is usually done in a compiled environment. Dynamically allocated variables (arrays, strings) and non-local (variables in an Ada **package** or PL/CS EXTERNAL storage) are referenced through indirect addresses stored at the corresponding offset. All variable addressing is on the basis of non-negative offsets from the base of the current environment.

Each location in the current environment has a corresponding bit in the CHECK area of the environment which serves to indicate whether changes in its value should be reflected on the CHECK file (and the user terminal). CHECK output results only when the variable is marked [illegible] the system is in the appropriate display mode. Both types of CHECK status information are controlled by executable commands and can change during the execution of the procedure. CHECK output control is a characteristic of the procedure instantiation. Since the CHECK status of a variable may differ in stacked copies of the same procedure, this information cannot be stored statically.

The value of a variable is accessible as quickly as possible and the CHECK information almost as easily. The least accessible information is that required to actually create the CHECK output. The name to be attached to the variable is accessible through a symbol table procedure call with the offset as its argument. If the variable modification causes CHECK output, the time required to process the output will dwarf the time required to find the name to print. When no output is required, the name is unnecessary and no time is lost.

The runtime environment is a distinct file which consists of the runtime stack, a static area, and the control information for the current execution. The stack is used (consistent with normal language processor usage) for automatic variables, parameters, call history, etc. The static area is used to store both internal and external static variables; a portion of the static area is set aside for each of the procedures. Because the runtime environment is a distinct file, and because procedure files are dynamically loaded, "static" variables must actually be allocated as new procedures are called. The control information includes the current procedure set, global file information, and other long-term interpreter structures.

<u>Runtime Environment</u>

The runtime environment is made up of a linked list of storage units. Each procedure instantiation is represented by either one or two such units.

```
high          *=====================================*
addresses     /   dynamically allocated area        /
           *-->*=====================================*
           |
           |   *=====================================*
           |   /   parameters, locals, temporaries   /
           |   *-------------------------------------*<-----*
           |   /   check flags (bit map of locals)   /
           |   /-------------------------------------/
           |   /   accessible environment display    /
           |   /-------------------------------------/<-----*
           |   /   operand stack  (grows down ↓)     /
           |   |-------------------------------------|
           *---+-* dynamic area pointer              |
               |-------------------------------------|
base pointer   | environment maintenance information |
-----------&gt;   |=====================================|
```

The environment frame for a procedure consists of four logical areas:

1. Environment maintenance information. This is a set of descriptors for the location of the previous environment element, the next environment element, the procedure identifier, the return location, and the display of lexically containing environments. The static area pointer is the only "display" pointer for the PL/CS language version since there are no internal procedures. Ada/CS includes a pointer in the display for variables declared in **packages**.

2. Local variables, temporaries, and parameters. Contains the actual variable locations for local and **in** parameter scalars, and file addresses for **in out** and **out** parameters and local and **in** parameter arrays and strings.

3. Operand stack. Area used for stack expression evaluation.

4. Dynamic variable area. Actual storage for the bodies of arrays and strings. String temporaries are also stored here.

At the start of each execution interval, a real-memory pointers are set to the commonly referenced base locations so that fixed offsets from these pointers can be used for all actual references. Although no real-memory addresses can be included in saved structures, these temporary values materially speed execution.

The configuration of the environment element is essentially fixed at procedure entry. Since procedure modification (insertion or deletion of variable declarations) can materially change the configuration of the environment element, there must be provision for such modifications. Most of these changes are performed by the editor when the changes are made. It is possible that a changed procedure will be restarted from the point where execution last stopped. In this case, the storage map for the procedure in the environment must match that provided for by actual variable references. This adjustment can always be made by reformatting each of the stacked environments as the changes are made. Since environment activations follow a strict stack discipline, it is possible to only reset the environment configuration when the procedure is reactivated, either after editting or upon procedure return. Readjustment of the environment may be complicated, but the check for its necessity is very simple and contributes negligible overhead. A more complete discussion of this and related issues is given in [4].

Static Area

A single static area is allocated for the environment. In this area go all of the variables that are declared static in PL/CS and all variables contained in **package**. This area is potentially extended at each new procedure fetch as new static variables are declared. The only distinction between external and internal static variables (PL/CS) is in the scope of the name. The discussion of procedure fetching provides more detail on the process of static storage allocation and the differences between PL/CS and Ada in this regard.

Variable Access

At execution, a real memory pointer is maintained to the current stack element and to the display area. All addressing is relative to the current environment pointer (local) and the file addresses stored in the display (lexical). Addressing can be direct or indirect. Indirect addressing is used where the addressed memory location contains the file address of the object to actually be accessed. Reference parameters (Ada **in out** or **out**, PASCAL **var**) are accessed indirectly through the corresponding local area variable. Static variables are accessed indirectly to the static area. Parameters and static variables in containing procedures are accessed indirectly through their lexically accessed variable locations. Since procedure storage is logically distinct, a separate instruction set is provided for copying constants onto the stack.

The actual values used for access are offsets (in terms of minimum-size items) from the beginning of the local area or the contents of one of the display pointers. The initial implementation is limited to offsets 0 through 255. Extension to higher maximum offsets would not be difficult, but seems highly unlikely for reasonable programs. The implementation has carefully been planned not to depend on a particular choice of sizes for the basic numeric types.

Array descriptors are stored to facilitate addressing and bounds checking. The descriptor includes the number of subscripts, a direct pointer to the body of the array (which is stored in "normal" rectangular array form), the high and low bounds, and the pre-multiplied width between successive subscripts. The array bounds are stored backwards because this makes them accessible in the same (stacked) order as the subscript values.

Interpreted Machine

The basis of the CPDS execution supervisor is a stack-oriented, statically-typed postfix expression interpreter. Programs consist of statements whose execution is described by postfix expressions. This form is particularly easy to interpret efficiently, and is adequate (with the addition of very few non-functional, display-only operations) for re-creating a canonical source-image of the program. Statement stubs contain additional non-executable information necessary for effective source-image reconstruction. No attempt will be made to detail the entire operation set, which is mostly "standard".

CPDS departs from the "standard" form of assignment statement evaluation which can be oversimplified as: push left-hand-side address, push right-hand-side value, assign. Calculating the (logical) address of the left-hand-side is relatively expensive, and the process must be reversed to create a real-memory address at assigment. Further, the location being assigned to has become anonymous. Its address is known, but not its name. For diagnostic purposes, the name is important for CHECK output and other diagnostic facilities. The new order of evaluation for an array reference is: push values of subscript expressions, push right-hand-side value, store result performing subscript calculations. This method has a potential drawback in that subscript formation is not performed until assignment, where tradition would have noted any subscript errors before evaluating the right-hand-side. If user experience indicates that this is a problem, subscript checking can be added as a separate "operation" to be performed prior to right-hand-side evaluation. If right-hand-side evaluation could change the values of the subscript expressions, left-hand-side address formation could not be delayed.

Both Ada and PL/CS support string operations (Ada does not limit these to vectors of characers) that do not readily fit into the stack-oriented structure adopted. Although the implementations of the two sets of operations are different in many details, the primary difficulties are similar. Two types of complications arise from string operations: overlapping of source and destination strings, and the creation of string temporaries. These are often, but not always related.

The problem of assigning temporaries is handled for arithmetic by the stack mechanism. To see why this solution is not attractive, consider the Ada assignment:

```
A := B & C;
```

In a strict stack-oriented method, the concatenation operator would convert the stacked values of B and C into their concatenation. Assume B and C are each four characters long and that the length of the string is stacked along with the value, a snapshot of the expression stack before and after would look as follows:

```
        | B[3]   B[4]  |              | C[3]   C[4]  |
        | B[1]   B[2]  |              | C[1]   C[2]  |
        |      4       |              | B[3]   B[4]  |
        | C[3]   C[4]  |              | B[1]   B[2]  |
        | C[1]   C[2]  |      TOS ==> |      8       |
TOS ==> |      4       |              |              |
```

This operation requires a complete reshuffling of the stack in a manner unparalleled in arithmetic operations. Note that "growing" the stack in the opposite direction makes handling variable-length items almost impossible (without negative offset addressing) since the stack increment (decrement) depends on the size of the previous top of stack instead of the current operation.

To solve this problem, Teitelbaum [11] stores strings in reverse order, so that the above would look like:

```
        | B[2]   B[1]  |              | B[2]   B[1]  |
        | B[3]   B[4]  |              | B[3]   B[4]  |
        |      4       |              | C[2]   C[1]  |
        | C[2]   C[1]  |              | C[3]   C[4]  |
        | C[3]   C[4]  |     TOS ==>  |      8       |
TOS ==> |      4       |              |              |
```

This solution reduces the shuffling required considerably, but not entirely and requires that strings be stored differently in the stack than in variables.

The solution adopted is to defer actual concatenation until the operand value is required. The string equivalent of a push value creates a string descriptor (address and length) corresponding to the string (or slice) and places it on the top of the stack. The concatenation operator combines the two descriptors on the top of stack into a new one which represents the concatenated string. The assignment and comparison operations are more complicated than would otherwise be the case since they must deal with descriptors rather than actual strings. Temporaries are still required to handle function values and nested string expressions (extremely rare). It is possible to prevent overlapping source and target strings in assigment without allocating temporaries in the environment file because the operation cannot be interrupted. In most cases it is syntactically determinable that there is no overlap, and no check for overlap need be made. Single-operand right-hand-sides can be copied without the use of any sort of temporary by correctly choosing the direction in which the copy operation is to take place. A similar process can be employed to avoid temporaries for more complicated cases, but the complexity of the operation far outways any benefits for the current purpose.

Procedure Invocation

Ada and Pascal language definitions, and batch PL/CS implementation restrictions are sufficient to guarantee that all type-checking can be done at compile time. While this is a useful restriction to place on the language and its expected usage, it presupposes that the program is complete at the time of execution. Since CPDS is at great pains to allow the execution of partially complete programs, it must be prepared to deal with missing procedures, unspecified parameters, etc. gracefully. The current section deals with the simpler part of the problem, making the invocations work. The next section deals with the complications introduced by allowing incomplete programs to be executed.

Two distinct type-related problems are raised by separately maintained procedures: parameter typing and function value typing. The number and variety of possible accessing mechanisms for parameter matching makes it more economical to perform all parameter checking at runtime. This is possible because both the call-side and the entry-side must declare the parameter and these declarations can be easily checked. (Additional, static, checks can be made on entry to warn the user of possible internal usage incompatibilities, but these do not reduce the need for runtime checks.)

The value returned by a function is more difficult to check, especially where explicit function declaration is not required (PL/CS). It is possible to predict the type of value the function will return if it has been declared before the invocation is entered, but subsequent modification of the function cannot feasibly (in an interactive system) notify all referencing procedures of the change. As a result, any type prediction must be verified at runtime. Similarly, parameter count and parameter types must be checked at actual invocation. Since the underlying expression interpreter is statically typed, the expression code for the statement includes the predicted type of the function. When the function is actually invoked, a type-mismatch causes an error-return from the execution supervisor. For this particular error, the repair is often possible without nofication to the user (the problem is viewed as an incorrect prediction rather than as an error). This strategy is easily implemented for PL/CS since the limited set of types, all language-defined, are freely intercovertible. Ada and PASCAL allow user-defined types, but do not allow free interconversion for all operations, making the chances of success somewhat lower.

The actual procedure invocation is performed by a series of expression syllables each of which perform a specific part of the required sequence. The "call-side" operations are:

1. Push (argument, type) pairs onto the top of the stack in lexical order.

2. Push the argument count and the type of the procedure.

3. Save the file address of the current procedure and invocation.

4. Fetch the invoked procedure as the current procedure.

The following "entry-side" operations are interpreted in the new procedure.

5. Allocate a new environment frame suitable for this procedure.

6. Copy actual parameter values/references while popping them from the call-side operand stack.

7. Copy the display and reconcile any "static area" references.

8. Allocate local array and string variables.

After the body of the entry-side procedure has executed:

9. Entry-side returns a value by copying it back onto the call-side top of stack and releases its own stack frame.

Execution can now proceed in the call-side statement.

Actual parameter formation requires that the address of any referenceable parameters be pushed onto the top of the stack. Expression evaluations, including function calls can proceed normally through the calling sequence using the local operand stack. The type that accompanies the pushed actual includes a value/reference indicator to facilitate parameter copying. For arrays and strings, the value passed in the actual is always a descriptor. The copy operation on the entry side is responsible for performing parameter type-checking.

All parameters are required to type-match exactly. For READONLY parameters, PL/CS extends this to mean that implicit conversions may be performed. For reference parameters (Ada **in out** and **out**, PASCAL **var**), types must match and the actual parameter must be legal for use on the left-hand-side of a local (call-side) assignment. The PL/CS version does not create temporary variables to accomodate unassignable reference parameters. To do so exposes the novice programmer to a parameter assignment anomaly in which a change to the value of the formal is not reflected in the actual. To ease this restriction for parameters which are not modified, the PL/CS version of the system assumes READONLY for all parameters for which usage will allow the restriction.

Dynamic Loading

When a procedure is fetched for the first time, it is necessary to make sure that its external characteristics are consistent with those of the other active procedures and establish uniform access to shared data objects. For PASCAL procedures are the only name-bound objects in the system. Ada/CS and PL/CS both allow the declaration of shared memory, though the different mechanisms require different solutions. Both languages are designed to allow compile-time checking of variable correpondances; the complication of introducing procedures into an active environment is entirely due to the decision to delay procedure references as long as possible.

A separately stored procedure (or outer level procedure of an Ada package) constitutes its own logical address space. Its access to the remainder to the environment is through parameters, other procedures and shared data (PL/CS EXTERNAL variables, Ada packages). (Program references to data files are not based on variable names and are unaffected by procedure linkage[2].)

A new PL/CS procedure may reference known EXTERNAL variables, in which case the declared types must match. The INITIAL attribute of the declaration must be compatible with those given in previously encountered declarations. It is required that all specified initial values match (any or all of the declarations can omit the INITIAL attribute). This requires checking against a saved initial value, since the current value may have been assigned (or otherwise modified). It is sufficient to save the designation of the initializing procedure, somewhat complicating the check, but saving large amounts of space for array initializations. New EXTERNAL variables must be added to the static area. Indirect linkages must be formed for all of the EXTERNAL variables and space allocated for any internal STATIC variables.

Ada linkage is somewhat simpler for data values, largely because there is only one possible source of variable initialization. On the other hand, it is significantly more difficult to defer the loading of packages than procedures. Since the package body contains the information necessary to allocate its variables, the package must be fetched prior actual to variable usage. Further, in order to allocate variable storage for the procedure, it may be necessary to have access to the types defined in the package. As a result of these consideratins, CPDS defers fetching a particular package until the first procedure requiring it is fetched. If only the procedures defined in the package are referenced, the package is not fetched until the procedure is called (or some other fetch operation requires its presence).

Summary

The execution supervisor for CPDS makes it possible to provide a particularly flexible environment for program development and debugging. A balance is found between efficient runtime representation and flexible program representation. The nature of the development system, particularly the highly interactive execution and program entry process and the range of restart situations, impose a number of additional requirements on the structure of the system. The maintenance of the environment as a permanent file structure not only facilitates restart, but increases the feasibility of symbolic debugging operations involving the environment. The implementation of dynamic loading makes it possible to have the benefits of type-checking while still allowing considerable freedom with regard to the order of program development.

References

1. "Preliminary Ada Reference Manual", SIGPLAN Notices 14, 6, June 1979.

2. Archer, J., "A File System Extension to Micro-PL/CS", Technical Report 79-366, Department of Computer Science, Cornell University.

3. Archer, J., "An Instructional Subset of the Programming Language Ada", Technical Report 79-395, Department of Computer Science, Cornell University.

4. Archer, J., "Interleaved Execution and Modification", Technical Report 79-398, Department of Computer Science, Cornell University.

5. Archer, J., and R. Conway, "A Program Development System for a Subset of Ada", in preparation.

6. Archer, J., R. Conway, A. Shore and L. Silver, "The System Architecture for the Ada/CS Program Development System", Technical Report 79-399, Department of Computer Science, Cornell University.

7. Constable, R. L. and R. W. Conway, "PL/CS -- A Disciplined Subset of PL/I", Technical Report 76-293, Department of Computer Science, Cornell University.

8. Jensen, K. and N. Wirth, PASCAL User Manual and Report, 2nd ed. Springer-Verlag, 1975.

9. Klint, P., "Interpretation Techniques", IW 110/79, Afdeling Informatica, Mathematisch Centrum, Amsterdam.

10. Tanenbaum, A. S., "Implications of Structured Programming for Machine Architecture", CACM 21, 3, 1979.

11. Teitlebaum, R. T., "The Cornell Program Synthesizer, a Micro-processor Implementation of PL/CS", Technical Report 79-370, Department of Computer Science, Cornell University.