

Language Properties To Improve Productivity And Speed

by: H.F. Ledgard¹ & W.C. Cave²

As of August 27, 2011

INTRODUCTION

We estimate that, in a competitive product organization, 65% to 85% of software development effort is spent working with existing software modules. Therefore, the ease with which one can understand and modify an existing module directly impacts software productivity. This impact is amplified when changes are made by someone other than the original author. In many applications, one must also be concerned about run-time speed. This paper analyzes language properties that affect both productivity and run-time speed.

Prior papers, e.g., [2], [3], and [4], have addressed the issues of software productivity and speed using a CAD software development environment. The focus of those papers was on visualization of architecture, where separation of data from instructions was shown to provide for precise delineation of modules and their independence properties. This paper follows the separation theme, but focuses on the language issues.

Figure 1 depicts the interplay of external documentation, architecture, and language for the CAD software environment described in the above mentioned papers. Three separate languages are shown at the bottom of the figure: one for declaring data, one for stating instructions, and one for run-time control. These languages were developed specifically to maximize both productivity and run-time speed. The principles and concepts behind these languages provide the basis for this analysis.

Software requires external documentation to understand complex algorithms. In most programming environments, just gauging the amount of external documentation required is difficult. Enforcing the production of external documentation is harder. Maintaining correlation between external documentation and code is most difficult. Understandable code reduces the need for external documentation as well as the time spent trying to understand complex algorithms written by another author.

We focus here on the properties of programming languages that affect the speed and understandability of production code, and the approach to language design that ensures these properties.

SOME BASIC OBSERVATIONS

Language And Information Theory

“... English is arguably the largest and most complex language in the world (measured in number of words and idioms), but also one of the most successful,” [9]. It dominates the world of free trade. Considering its tiny island origin, its survival is attributed to the ability of its users to communicate reliably.

1 The University of Toledo

2 Visual Software International

The two major issues in communications are reliability and speed. Fast and reliable transfer of information is the goal of information theory, as evolved by Shannon, [8] and others. Reliability of information transfers is increased by adding redundancy or additional information. This may be as simple as sending the same message twice, or using additional words, such as articles, adjectives, or adverbs. In wireless communications, one may double the size of the original data stream to ensure reliable transfers. English is considered to have a high degree of redundancy compared to most other languages, implying it is more likely that information is transferred reliably.

The Problem With Terse Programming Languages

Studies comparing interactive languages have shown that errors increase as statements move from good English to a more terse form, [5]. If we compare COBOL, FORTRAN and C-based languages (C, C++, C#, Java, etc.), we derive the following conclusion: COBOL is verbose, FORTRAN is OK, C-based languages are terse. A misperception is that verbose correlates to slower run-times. In fact, these properties are totally unrelated. Since source language is translated to machine language, the burden is on the translator. Reliability and speed can be improved simultaneously. If we are after reliability, verbose is best. If we are after speed, tests have shown the superiority of COBOL for transferring information. This is due to the ease of creating and referencing complex hierarchical data structures as described below. Tests have also shown FORTRAN to be faster than C for inverting large matrices.

Some languages encourage minimization of keystrokes with their spartan syntax. This is tantamount to minimizing internal documentation and therefore understandability of the code. Such languages may reduce the time to type a line of code, a small part of the time spent on a module. Anyone who has worked with spartan syntax should confirm that the time lost trying to understand, test and debug complex algorithms will likely far exceed that spent saving keystrokes. Our experience working with complex systems is that *terse* is the wrong direction.

One can argue the alternative of producing documentation inside code with large blocks of comments. This is an indicator of a poor language - one that has to be *heavily annotated* to be understood by the reader. Good languages minimize the need for comments while providing a high degree of understandability directly from the code.

Visualizing Scope

The architectural approach suggested by Figure 1 and described in [2], [3], and [4] provides a visualization of how data is shared by instructions. All data must appear in elements called "*resources*". *Dedicated* resources are used only by a single "process" (group of instructions), and have only a single connect line to that process. *Shared* resources are connected to those processes which share that data.

Forcing every data element into this structure is a great simplification. It ensures immediate visibility of what instructions have access to what data. It forces designers to put more thought into architectures that afford easy restructuring. More importantly, one can determine from a quick visual inspection of an engineering drawing who shares a resource, and therefore any subset of a data structure. An architect can minimize the connectivity between

processes at the drawing level. Since it is the connectivity that determines independence, this serves to maximize independence of modules, supporting parallel processor designs.

Many organizations impose coding restrictions above and beyond those in a given language to attempt to achieve some level of independence. However, without preprocessors, and with access to individual variables as in C-based languages, one must track down all the functions that contain data definitions of each individual variable used by a function, to enforce a standard, or to locate a problem in that function. The potential for very high connectivity makes it difficult to ensure independence of software modules.

Separation of data from instructions produces an entirely new view of scope, one that is specified precisely in the architectural drawing. Since there is a direct mapping from architecture to code, the language need not be concerned with scope. More importantly, allowing scope changes in the language would conflict directly with the architectural control. Since architecture is at a higher level of purview in the design chain, and provides direct visualization of independence, scope is best determined by the architecture.

Simplifying Subroutines

A *Process* defines the instructions that act on data. At first glance, a process may look like a *function* or *subroutine* in a typical programming language. However, many of the problems we deal with today are dominated by sets of rules that operate on complex data structures, not just mathematical functions. This implies many large dissimilar transformations that must be tailored to the complex data structures upon which they operate.

The lack of strict one-in one-out control structures (as described by Mills, [7]) in typical programming languages creates problems in the support phase of a software product. Using a C-based language, the resulting code can contain complex conditional statements implemented with many layers of nested brackets. This becomes a problem when new features are added to a function and the number of conditions must grow to support these features. Unless one takes the time to break up the C function, or resorts to the use of GOTOs, nesting increases.

Since breaking up a complex nested conditional statement is difficult, and creating a new function means deciding what data must be available to the new function, the nesting typically grows. Using C, the local data becomes global. If C⁺⁺ is used, it can be shared among “friends.” These decisions are all practical deterrents to breaking up the nests. And as they grow, they become more difficult to maintain.

DECLARATION OF DATA

The Problem of Global Types

If one tries to use some of the more complex C-code functions, e.g., those for graphics or Inter-Process Communications (IPC), one typically runs into the problem of variable data type definitions, or *defined data types*. These types are defined in one C routine, and used in another. In C⁺⁺, the type can be in a “class” that is shared.

The problem encountered with user defined data types occurs when trying to locate the data type definition in the routine of interest, and determining that it is defined elsewhere. One must then locate the source code for the defining routine to determine the data type. Having found the source code for the defining routine, and then the data type definition, one may discover that this definition depends upon yet another defined data type in yet another routine.

When one finally gets to the end of such a chain, one should not be surprised to discover that the data type is simply an integer. Such mystery chains are not uncommon in large systems when user defined data type definitions are permitted, and people elect to redefine the dictionary.

When data declarations can be imbedded in code with instructions, one is inclined to minimize the size of data declarations, or just minimize the keystrokes. This is most evident when reading C-based code, where programmers use terse data definitions that allow them to *type* the data as they go, within arithmetic or conditional constructs. Such practices minimize understandability along with keystrokes.

Clarity is enhanced with a single dictionary for all data types. Enforcement of this principle becomes simple if data typing is not mixed with the executable statements, but placed in a separate entity. Since resources are separate entities that are defined and shared by processes that want access to them, then by virtue of connect lines in the architecture, there are no data declarations in a process. Therefore, the re-typing problem does not exist.

The presence of global data types generally creates a dependency with any subprogram that shares them. It creates severe bottlenecks for parallel processing. With the CAD approach described above, the architect determines explicitly - by design - what processes have access to what resources. The connectivity is clear, visually, from the drawing. There are *no global types*.

Machine Independent Standards (What You See Is What You Get!)

Older languages, e.g., C, C⁺⁺ and their derivatives, perform *word boundary alignment* of arithmetic type data. (Data has not been organized as words since the birth of the byte in the IBM 360.) Word alignment causes data structures to get *padded* with “slack” bytes, implying that group level structures are not stored as defined by the programmer. Moving group level structures will generally cause data to be inserted incorrectly into another structure, unless great care is taken to define the structures precisely the same. This is a very undesirable restriction when working with databases or performing message or symbol string processing. Even when compilers offer an option to ignore the alignment, programmers ignore the option for fear of incompatibilities across a team.

Using hierarchies also implies a language in which such data structures are easily created, referenced, and understood. An example is shown in Figure 2. More importantly, the structures are organized to match the application - not by type.

Resource structures are byte oriented and fully compatible with current and planned machines. Since data on today's machines is byte addressable, a designer can create complex hierarchical structures that are most convenient for meeting requirements. What You See (in your attribute structure) Is What You Get (in memory), independent of the machine you are using. The language translators are more complex since the burden is shifted from the user to the computer - at translation time. However, the real savings occur at run-time.

RESOURCE NAME: MESSAGE_FORMATS			
MESSAGE			
1	SYNC_CODE	CHARACTER 6	
	ALIAS VALID	VALUE '101010',	'010101'
1	TYPE	STATUS FORMAT_A	FORMAT_B
1	CONTENT	CHARACTER 46	
FORMAT_A REDEFINES MESSAGE			
1	PAD	CHARACTER 14	
1	HEADER		
	2 PRIORITY	STATUS FLASH	IMMEDIATE
			ROUTINE
	2 ORIGIN	INDEX	
	2 DESTINATION	INDEX	
	ALIAS BROADCAST	VALUE 0	
1	BODY		
	2 LENGTH	INTEGER	
1	TRAILER		
	2 MESSAGE_NUMBER	INTEGER	
	2 TIME_SENT	REAL	
	2 TIME_RECEIVED	REAL	
	2 ACKNOWLEDGMENT	STATUS RECEIVED	NOT_RECEIVED
	2 LAST_SYMBOL	CHARACTER 2	
	ALIAS TERMINATOR	VALUE '\\', '//', '<<', '>>'	
FORMAT_B REDEFINES MESSAGE			
1	PAD	CHARACTER 14	
1	HEADER		
	2 SOURCE	INDEX	
	2 SINK	INDEX	
1	BODY		
	2 CONTENTS	CHARACTER 42	

7/13/06

Figure 2. Example of hierarchical data structures in a resource.

Hierarchical Group Moves

The ability to move a complete hierarchical structure, or any substructure within a hierarchy, with a simple MOVE statement is important to the implementation of transformations in complex systems. This permits group moves of one complex structure or substructure to another with a single instruction fetch, dramatically improving running times. A good example is moving complete messages, or fields containing subfields, as illustrated in Figure 2. One may want to move subfields into a packet, and packets into a frame. These group moves are executed simply by referring to the attribute name of the highest level group to be moved. One need not worry about its size or structure. The implementer need only ensure that the receiving structure is organized in a way that receives the data being moved. This reduces instruction code as well as running time.

The ability to use hierarchical group moves results from the *machine independence* properties described above. Hierarchies are easy to define and easy to understand using the approach shown in Figure 2. Ease of use is also helped by minimizing the qualification of names in a hierarchy to that sufficient for unique identification.

Reusable Names And Qualifiers

It is common to have many thousands of attributes in a large system, and the selection of names in large modules becomes a critical part of helping others to understand the module. Names can be reused without qualification except in the same process. Even in a single process, one should not have to make up different names to distinguish the same type of object from another when they are obviously used in different contexts.

For example, when reading the specifications for two different equipment locations, one would expect to find the same words reused, possibly with other names as qualifiers, e.g., RADIO LOCATION or ANTENNA LOCATION. One can easily understand the meanings in accordance with the context of each separate specification.

Translators can recognize the context of attribute names and qualifiers. This frees the user from having to create needlessly different names that mean the same thing in different contexts. What's more, this is done without injecting special delimiters, e.g. an underscore or period, to signify qualification. This implies translation based upon context, putting the burden on the translator not the user.

Reuse of names in a process is thus allowed provided the intended use can be uniquely resolved. Consider the following example from a resource description.

```

1  ENTRANCES
2  FRONT
   3  DOOR                STATUS OPEN
                           CLOSED
   3  WINDOW              STATUS OPEN
                           CLOSED
2  BACK
   3  DOOR                STATUS OPEN
                           CLOSED
   3  WINDOW              STATUS OPEN
                           CLOSED

```

A conditional statement using that resource can be written as follows.

```

IF FRONT DOOR IS OPEN OR BACK DOOR IS OPEN
  EXECUTE ENTRY
ELSE IF FRONT WINDOW IS OPEN OR BACK WINDOW IS OPEN
  SET FRONT DOOR TO CLOSED
ELSE ...

```

Although names (e.g., DOOR) are reused in the resource, the intended use is resolved by using the qualifiers FRONT or BACK in the process. In the case of OPEN or CLOSED, reuse of STATUS names is qualified automatically by the particular status attribute FRONT DOOR or BACK DOOR.

Reuse of names in a process requires qualification only to the extent sufficient to insure uniqueness at the lowest (innermost) level in the resource hierarchy. For example:

```

HIGH_POWER
  1  TRANSCEIVER
    2  HEIGHT                REAL
    2  LOCATION
      3  X_LOCATION          REAL
      3  Y_LOCATION          REAL
LOW_POWER
  1  TRANSCEIVER
    2  HEIGHT                REAL
    2  LOCATION
      3  X_LOCATION          REAL
      3  Y_LOCATION          REAL

```

One can then write the following statements.

```

MOVE HIGH_POWER HEIGHT TO LOW_POWER HEIGHT
MOVE HIGH_POWER X_LOCATION TO LOW_POWER X_LOCATION

```

This is also true for ALIAS names as defined below. They are automatically qualified by the attribute names that use them.

Defining Properties That Can Be Tested Easily

When large quantities of conditional statements are nested to specify what action must be taken, clarity is lost. One must be able to represent compound conditions easily. One also wants to read and understand the conditions quickly and reliably. This requires potential states of attributes to be represented by names of groups or sets of states.

Consider,

```

CALENDAR_INFORMATION
  1  MONTH                    CHARACTER 9
    ALIAS THIRTY_DAY_MONTH    VALUE 'APRIL  ',
                                'JUNE   ',
                                'SEPTEMBER',
                                'NOVEMBER'
    ALIAS THIRTY_ONE_DAY_MONTH VALUE 'JANUARY ',
                                'MARCH  ',
                                'MAY   ',
                                'JULY  ',
                                'AUGUST ',
                                'OCTOBER',
                                'DECEMBER'
  1  YEAR                     STATUS NORMAL,
                                LEAP_YEAR

```

The attribute structure above supports the following conditional statement:

```

IF MONTH IS A THIRTY_DAY_MONTH
  EXECUTE THIRTY_DAY_RULE
ELSE IF MONTH IS A THIRTY_ONE_DAY_MONTH
  EXECUTE THIRTY_ONE_DAY_RULE
ELSE IF YEAR IS A LEAP_YEAR
  EXECUTE TWENTY_NINE_DAY_RULE
ELSE EXECUTE TWENTY_EIGHT_DAY_RULE.

```

Here, the name of the month is used as the value of the attribute, and can be tested or printed directly. In addition, the ALIAS qualifier defines the sets of discrete values the character string can take on, and the STATUS attribute type provides discrete character states that can be SET as well as tested directly. The reader of this conditional statement does not have to refer to any other documentation to understand what the conditions are for setting and testing these attributes.

Sharing Data – Elimination of Argument Lists

Most older programming languages use argument lists when calling subroutines or functions. A simple example is SIN(THETA). This approach is handy for built-in scientific and Boolean functions. However, when passing multiple arguments, things can easily get confused. Unless otherwise specified, the data values in the argument list are copied to new memory locations before being operated upon by the function. Upon return from the function call, they may or may not be copied back into memory locations associated with the calling routine.

When nonnumeric data structures are used, passing data becomes cumbersome and error-prone. For medium size data structures, processing time can become troublesome. In these cases, many languages provide for passing pointers to the data, instead of passing the data. However, it is up to the programmer to manage and pass the pointers. Using the CAD approach, resources are shared directly by those processes to which they are connected in the architecture. Access is always by pointers that are managed automatically behind the scenes.

If the designer wants to make a copy of an attribute structure, he simply moves that structure to a corresponding structure, typically in a different resource. The original attribute structure remains in tact. Often one may want to look at the data using a different structure in the receiving resource. This can only be accomplished if the attribute structures preserve their size and structure when moved. Because of the well specified nature of hierarchical data structures in a resource (What You See Is What You Get), and the corresponding group move property, this is simple to do. To gain speed, one may simply REDEFINE the structure as shown in Figure 2, eliminating the MOVE.

AN ELEGANT SOLUTION TO FLOW OF CONTROL

Nesting of control structures is a feature of virtually all conventional programming languages. For example, it is not uncommon to see

```
An if-statement
    containing an if-statement
        which contains a while-loop
```

Such an example by itself is not especially problematic, but does suggest the mental complexity of keeping track of code with nested control. Moreover, the mental complexity increases as the length of the code and the length of nested sequences grows. It is not uncommon for single blocks of code to extend over more than one page

Things can get complex even without a nested loop. When there is nesting and the statements contained in the IF are of some length, getting a clear picture of the entire structure is difficult. When nested IF's cover many lines, the logic becomes hard to understand.

We propose a fresh look at something so commonplace that we take it for granted, that is, flow of control. To control the complexity of highly conditional transformations, we must be able to deal with multiple layers of hierarchy. One approach is using nested IFs. Our approach is to group statements into RULEs within processes. This solves multiple problems. Each process is decomposed into rules that share the same resources as the process. An example is shown in Figure 3. The hierarchical rule facility is provided through a simple *one-in, one-out* control structure embodied in the EXECUTE statement, which takes on various forms. This statement allows the designer to deal with rules that are at an "equal level" in the hierarchy of logical operations, without resorting to the dangers of GOTO statements. The control structures are strictly *one-in, one-out*. We note that "one-in one-out" implies control is transferred to a rule, then transferred back to the statement following the one that invoked the rule.

```

PLACE_CALL                                     level 1
  IF CLOCK_TIME IS GREATER THAN ONE_HOUR
    STOP.
  IF ACTIVITY(SOURCE) IS WAITING_TO_CALL
    EXECUTE ATTEMPT_CALL
  ELSE EXECUTE RETRY_LATER.

-----
ATTEMPT_CALL                                   level 2
  INCREMENT CALLS_ATTEMPTED
  IF LINES_IN_USE(OFFICE(SOURCE)) ARE LESS THAN
    LINES_IN_OFFICE(OFFICE(SOURCE)) THEN
    EXECUTE MAKE_CALL
  ELSE EXECUTE BLOCK_CALL.

RETRY_LATER
  SET ACTIVITY(SOURCE) TO RETRY_LATER
  CALL TERMINATE_CALL

-----
MAKE_CALL                                     level 3
  INCREMENT LINES_IN_USE(OFFICE(SOURCE))
  IF CALLERS_PLAN(SOURCE) IS PLACE_NEW_CALL
    SET PHONE_NUMBER TO UNKNOWN
    EXECUTE LOOK_UP_NUMBER UNTIL PHONE_NUMBER IS FOUND.
  OFFICE_NUMBER = OFFICE(DESTINATION)
  CALL CONNECT_CALL

BLOCK_CALL
  INCREMENT CALLS_BLOCKED
  SET SIGNAL_TO_SUBSCRIBER TO BUSY
  MOVE 'BLOCKED AT SOURCE' TO CALL_STATE

-----
LOOK_UP_NUMBER                               level 4
  DESTINATION = (TOTAL_SUBSCRIBERS * RANDOM) + 1
  IF DESTINATION IS NOT EQUAL TO SOURCE
    SET PHONE_NUMBER TO FOUND.

```

Figure 3. Example of the hierarchical rule structure of a process.

Controlling Complexity With Rule Hierarchies

A collection of statements that is given a name is called a "rule". Figure 3 shows an example of a process that has six rules. Generally a process will consist of a number of rules. Rules that are specific to an algorithm's data structure are all usually contained in the same spot, are easier to build, and much easier to understand during the support years

Each process has a top-level rule, e.g., PLACE_CALL in Figure 3. When the statements in the first rule have been executed, control returns to the calling process. Any rule may contain EXECUTE statements that invoke other rules within the process.

This dual structure has several advantages:

- a. At the end of a rule, control returns to the statement following the EXECUTE statement that invoked that rule. This guarantees the 1-in, 1-out property.
- b. Flow of control is linear within a rule.
- c. A process may contain one or more rules, each identified by a name. This provides flexibility in the number of conditional statements that a process can support.
- d. One may EXIT a rule at any time within an IF statement, e.g.,

```
IF THIS_RULE IS COMPLETE
    EXIT THIS RULE
```
- e. There is no nesting of IF statements.
- f. Except for the first rule, rules may be placed in any order.
- g. There is no recursion of rules.

The example of Figure 3 involves four top levels of control, yet is simple to understand. It also shows the ability to "push down" the complexity of rule sets into a hierarchy of logical levels. As a result, a process is typically somewhat larger than a "well written" C++ or Java function that are more the size of a rule. But it should be much more understandable, and will require many fewer comments, more likely none. Furthermore, a process with 20 rules may take a number of C++ or Java functions to implement.

Understandability of Complex Conditional Situations

One of the most important benefits of this approach is the simplicity of complex conditional situations. Consider the following example:

```
IF SYMBOL IS AN UNDERSCORE
OR SYMBOL IS A PERIOD
    EXECUTE CHECK_WORD_BLOCK
ELSE
    EXECUTE SCAN_FOR_SPECIAL_CASES.

IF STATEMENT IS A SPECIAL_CASE
    EXIT THIS RULE
ELSE ...
```

Here we see the equivalent of the case statement. However, the statements that are normally contained within a case statement may be placed in a separate rule, in this case CHECK_WORD_BLOCK and SCAN_FOR_SPECIAL_CASES. This adds great clarity, as we can read and understand the top level of control without being distracted by nested details that themselves may be quite complex.

Figure 3 provides an example of a process structure that follows the rule for grouping hierarchical logical levels. The built-in rules eliminate strings of call statements that cause unstable designs because of the complex dependencies created. The rules also run faster. They also support ease of use of parallel processors, since a large set of rules shares only the data available to the process. Since the *logical levels are totally independent of position*, the process may be organized in any manner the designer deems most understandable. Except for the first rule appearing first, the rest of the rules may be shuffled like a deck of cards.

The largest benefits of the hierarchical rule structure of processes are the understandability of complex conditional statements, and the ease with which one can add new conditions as the software is enhanced. Additional features of the process language, particularly the EXIT THIS RULE statement, serve to flatten conditions within a rule, making the logic apparent. This eliminates nested IF statements as well as call strings to small fragments of code that are used as the alternative.

Understandability of Loop Structures

A related property of the approach to control structures is the isolation of the body of a loop (the statements to be repeated) in a separate named rule. Only the rule name is used within the control structure itself.

Thus we must write something like

```
EXECUTE LOOK_UP_NUMBER
UNTIL PHONE_NUMBER IS FOUND
```

and place the body of the loop elsewhere

```
LOOK_UP_NUMBER
  DESTINATION = (TOTAL_SUBSCRIBERS * RANDOM) + 1
  IF DESTINATION IS NOT EQUAL TO SOURCE
    SET PHONE_NUMBER TO FOUND.
  . . .
```

After the LOOK_UP_NUMBER rule is executed, control automatically returns to the EXECUTE statement. Here again, rather than a sequence of nested structures, we can easily read the control at a single level.

Rule Pointers

A step towards speed and understandability of processes is the ability to assign the name of a rule to a “rule pointer” .

A RULE clause is used to define the allowed rule names that a rule pointer can assume during execution. Consider:

```
NEXT_ACTION      RULE  INITIALIZE_NETWORK,
                   START_TRANSMISSION,
                   START_RECEPTION,
                   DISCONNECT_CALL
```

Here, NEXT_ACTION defines a set of allowed rules.

The rule pointer NEXT_ACTION will likely be set in a conditional statement prior to a point where the rule is to be executed, such as:

```
IF TRANSCEIVER(TRANSMITTER) IS TRANSMITTING
    SET NEXT_ACTION TO START_TRANSMISSION
ELSE IF TRANSCEIVER(RECEIVER) IS RECEIVING
    SET NEXT_ACTION TO START_RECEPTION
```

Later one can then simply have.

```
EXECUTE NEXT_ACTION
```

This mechanism provides a direct, and therefore fast, transfer of control similar to the computed GOTO in FORTRAN. The improvement of this approach is twofold. First is the direct return of control to the next statement (one-in one-out versus GOTO). Second, the name of the rule, e.g., START_RECEPTION is used in the set statement, instead of a number - making it clear what the pointer is being used for. As above, meaningful names can be used and the choice of action can be set when an appropriate condition is met.

Process Pointers

The PROCESS pointer clause is similar to the RULE pointer clause. It is used to define each of the allowed process names that a PROCESS pointer can assume during execution. It is used to support the PROCESS pointer version of the CALL statement for executing processes.

The mechanism is almost identical to rule pointers. For example, the process pointer is defined in a resource:

```
NEXT_PROCESS      PROCESS COMPUTE_TIMERS,
                   DRAW_TERRAIN,
                   COMPUTE_MEASURES
```

It can then be used in a process as shown below.

```
IF INPUT_OPTION IS INITIATE
    SET NEXT_PROCESS TO COMPUTE_TIMERS
ELSE IF INPUT_OPTION IS CALCULATE
    SET NEXT_PROCESS TO DRAW_TERRAIN
...
CALL NEXT_PROCESS
```

A “CONTROL SPECIFICATION” LANGUAGE

In contemporary software environments, a project will require invoking different kinds of facilities, e.g., libraries, file systems, directories, and so forth. These facilities are generally invoked by an OS specific language generally known as a control language or script.

This requirement is handled using the third high level language known as the “Control Specification” language. It contains a sequence of labeled sections. Each section allows the developer to specify properties of the external environment, including file and directory specifications, and graphics. The notation used for the syntax is based on the same simplified English-like format as the process language. This language allows the developer to invoke system facilities in a manner that is independent of both the machine and operating system.

SUMMARY

Current programming languages encourage minimum keystrokes and terse (economy of) expression. From the above discussion, there is no correlation between terseness of language and run-time speed. Furthermore, one may have to read a line of code multiple times to understand terse logic. This is the opposite of reading prose, where good authors take pain to ensure the reader can quickly comprehend the details. If one cannot read fast, one may lose the “train of thought.” When developing and enhancing software, speed and reliability of comprehension directly affect productivity

Understandability was considered important to programming back in the 1960s; today it represents an opportunity for vast change, one that can open the use of software to a much wider range of professionals. Emphasis on understandability can help us deal with significant increases in software complexity, and provide great benefits for the computer field in general.

We do not expect subject area experts to learn language syntax sufficiently to write a program. However, it is our experience that a programming background is unnecessary for understanding complex algorithms. Given a language with the properties described here, code can be written so that experts in their field can understand the algorithms, and validate the implementation of a complex design.

Invariably, many of the issues addressed here provide food for vigorous debate. We encourage that such debate be focused is on improving speed and productivity.

REFERENCES

1. Anselmo, Donald and Henry Ledgard, Measuring Productivity in The Software Industry, Communications of the ACM, Vol. 46. No.11, Nov 2003.
2. Cave, W.C., *Software Survivors*, Software Developer & Publisher, W. Cave, July/Aug1996.
3. Cave, W.C., et al, *Engineering Software*, Visual Software International Technical Report S-3207, Mar, 2007. http://www.VisiSoft.US/PDF_Files/EngineeringSoftware.pdf
4. Cave, W.C., *Increasing Software Speed and Productivity*, Visual Software International Technical Report S-7107, July, 2007. http://www.VisiSoft.US/PDF_Files/IncreasingSoftwareSpeedAndProductivity.pdf
5. Ledgard, H., et al, "The Natural Language of Interactive Systems," CACM No. 10, October 1980.
6. Ledgard, Henry F., *The Emperor with No Clothes*, Communications of the ACM, Vol. 44, No. 10, Oct 2001.
7. Mills, H.D., Mathematical Foundations of Structured Programming, Technical Report FSC 72-6012, IBM Federal Systems Division, 1972.
8. Shannon, C.E., "A mathematical theory of communication," Bell System Tech. Journal, vol 27, July and October 1948.
9. Stroustrup, Bjarne, "More Trouble with Programming," Technology Review, MIT, MA, December 7, 2006.