ON THE CONCEPT OF MODULES IN PROGRAMMING SYSTEMS

D.D. Cowan
University of Waterloo

C.J. Lucena
A. von Staa
Pontifical Catholic University of Rio de Janeiro

CS-76-05

January, 1976

ON THE CONCEPT OF MODULES IN PROGRAMMING SYSTEMS

D.D. Cowan
University of Waterloo

C.J. Lucena
A. von Staa
Pontifical Catholic University of Rio de Janeiro

CS-76-05

January, 1976

ON THE CONCEPT OF MODULES IN PROGRAMMING SYSTEMS

by

D.D. Cowan
Computer Science Department
University of Waterloo
Waterloo, Ontario
Canada

C.J. Lucena
A. von Staa
Informatics Department
Pontifical Catholic University
of Rio de Janeiro
Brasil

Key phrases: Modularity, Programming languages, Programming systems.

CR Categories: 4.2

This research was supported in part by grants and contracts from:
The National Research Council of Canada,
The Canadian International Development Agency,
IBM (Canada).

Abstract

The effective and widespread use of methods for the design and construction of programs will require support by a collection of automatic or semi-automatic procedures often called a system's environment for program development. The design of such systems requires a clear understanding of the semantics of programs and the rules for their synthesis, particularly the concept of modularity.

This paper is intended to contribute further to the understanding of the concept of modularity since program units called modules will allow the synthesis of complex programming systems from off-the-shelf program components. Such a method of system construction should significantly reduce the cost of software production.

The concept of a module is presented in several stages in this paper. First, it is characterized by showing that modularity is a property of certain programming units at a given linguistic level. Once this axiom of modularity is established, then modules are further distinguished from other types of program units by five properties. A program unit which satisfies these five properties is said to have strong modularity. It is then determined that three linguistic levels appear adequate to characterize most programming situations and there is a general discussion of the type of language required at each of these linguistic levels. At this point, contemporary language features both implemented and suggested, such as procedures, clusters and module interconnection languages are surveyed to determine how well they satisfy the modularity properties at the appropriate linguistic levels. Finally, the concept of sharing is examined and it is shown that a module which is shared is a module in a weak sense.

1. Introduction

The effective and widespread use of methods for the design and construction of programs will require support by automatic or semi-automatic procedures; such a set of procedures has often been called a system's environment for program development. When designing such systems a clear understanding of the semantics of programs and of the synthesis rules for creating such programs is required.

Recent research in software engineering has led to some new concepts in the area of programming language design, a better understanding of program structure and a better characterization of some of the fundamental properties of software systems. The concept of modularity, for instance, has received a very thorough treatment in the works of Parnas [2,3], Dennis [1], Liskov and Zilles [4,5] and others.

This paper is motivated by a desire to contribute further to the understanding of the concept of modularity, since the authors felt the need for a very systematic characterization of the concept of modularity in the process of designing a programming system for software development. Program units called modules will allow the synthesis of complex programming systems from "off-the-shelf" program components and hence programmers can reduce the cost of producing such systems. Furthermore, modules are powerful mechanisms to model the abstractions which occur at the various levels of development of a programming system.

When trying to characterize the concept of a module a number of issues come into play. Some of the important ones are: system and program structure, programming language design, specification languages and techniques, and management of programming. Previous research in our opinion has not adequately related these issues to the definition and use of the concept of modularity. Related issues such as common base languages also help to complicate the modularity problem.

In this paper we start with a reference to Boebert used by Dennis [1] (on the role of linguistic levels in modularity) and the concept of software module specification proposed by Parnas [2,3]. We give some properties as a base for our definition of modularity and then identify some types of modules that can be used in connection with a programming system. After defining modularity we present a survey of some programming language features and determine if they satisfy our definition. The paper is completed by discussing some pragmatics related to the implementation of modularity.

Although the subject of modularity lends itself to a formal mathematical treatment, we opted, at this stage of our research, for a systematic, but informal presentation of our ideas.

2. Basic Concepts

Discussion of the modularity problem is based on an informal statement by Boebert [1], that we will state here as our basic axiom of modularity.

Axiom. Modularity is a property possessed by certain programming units defined at a given linguistic level of programming.

A linguistic level of programming, in the present context, is a particular notation for the expression of computational algorithms that has a syntax and semantics which distinguishes it from other notations. Different linguistic levels are used to express different phases of the development of a programming system. For example, a set of precompiled programs can be joined together for execution by a linking language, while these various precompiled programs will have been written at some time in high-level languages such as ALGOL 60, PL/1 and COBOL, or perhaps in assembler language. The precompiled programs also may have been described at an early stage of development through a non-executable specification language such as PSL [9] or the more informal language HIPO [17]. Each of the these notations, the linking language, the normal programming language, and the program specification language all represent different linguistic programming levels. In the paper we use the concept of linguistic level to help define modularity but our choice of levels is somewhat unconventional. By associating modularity with a given linguistic level it will allow characterization of a module independent of the linguistic level at which it occurs.

DeRemer and Kron [7], in a recent paper, make implicit use of the concept of linguistic levels; in fact, the difference they describe between LPSs (Languages for Programming in the Small) and MILs (Module Interconnection Languages or Languages for Programming in the Large) supports the axiom of modularity, that modularity is closely related to linguistic level.

One's experience and the literature on software lead to a list of properties which when applied to a program unit at a given linguistic level, defines it as a module. The properties are stated next in a concise manner with a short discussion accompanying each one.

Property 1 - Composition by nesting is the property which allows program units to be invoked and/or declared inside other program units to an arbitrary depth.

The linguistic level defined by FORTRAN for instance, makes no provision for combining separately written FORTRAN programs; a complete FORTRAN program consisting of main program and subprograms cannot serve as a program module because it cannot be further combined with other units to form larger modules.

Property 2 - Syntactic non-interference is the property which allows program units to be invoked and/or declared in a program text, written in a given programming language, without requiring any syntactic changes in the program text in which it is being placed.

The linguistic level defined by ALGOL 60, for instance, violates this property. A situation may occur in ALGOL 60 where a clash of names occurs when two procedures are placed in the program as declarations within the same enclosing procedure. Thus the use of nonlocal references in an ALGOL 60 program unit (procedure) violates property 2.

Property 3 - Semantic context-independence is the property which allows a program to have an invariant meaning independent of the location in which it is declared and/or invoked within an algorithm expressed at a given linguistic level.

Suppose that a program unit, defined at a given linguistic level, is specified by two first-order predicate-calculus formulae that represent respectively its input and output assertions in the Floyd sense [18]. If the program unit's specification, as given by these two assertions, is invariant within the program text where it is invoked and/or declared, then we say that the program unit satisfies property 3.

Property 4 - Data generality is the property which allows a program unit to communicate with other program units of the same linguistic level, using arbitrary data structures.

Satisfying this property means that the program unit implements Parnas' hiding principle [2]. Through this principle no program unit (programmer) should have any information about the inner workings of the modules with which it communicates. The hiding principle is, of course, a fundamental property required in off-the-shelf programming.

Property 5 - Definitional completeness is the property which forces a program unit defined at a given linguistic level to carry a semantic definition which is comprehensible to a class of users.

A program unit's definition is an inherent part of the concept of a module. In fact, no off-the-shelf programming is viable unless modules carry a complete operational definition. In fact, when a modular program is being synthesized, the specification of the module (its definition) should precede its implementation. For instance, an operational module definition should carry information about the type, form and nature (whether input or output) of parameters, a clear statement about the precision, cost and restrictions (input assertions) and a description of the actions to be taken when something unexpected occurs (sometimes called the module's robustness). When engineering a modular system it is important to express the definition in such a way that it can also be handled automatically.

A program unit must also be correct in order to be called a module. Properties 3 and 5 imply correctness and so this is not explicitly stated as another property.

3. The Concept of Modularity

The concept of modularity is stated here as a notion which is independent of particular language features and programming methods. The reasons for this method of presentation appear in the discussion following the definition. Modularity is defined as follows:

Definition A program unit defined at a certain linguistic level is a module, or a linguistic level allows a program unit to exhibit strong modularity (or simply, modularity) if and only if the program unit satisfies properties 1 through 5.

By defining modules in this abstract manner, a principle is provided for the software engineer in designing and constructing programming tools. When designing languages for constructing programming systems the software engineer can choose the linguistic levels and methods of constructing and manipulating modules using the properties stated in section 2. In general, each linguistic level will contain features for encapsulating program units into modules, a set of control structures and data types.

Several approaches to program development can be discussed abstractly in terms of our definition of modularity with specific reference to properties 1 through 5. For example, top-down programming, bottom-up programming, incremental program verification, and interchangeability and efficiency can be characterized.

Our definition of linguistic level is somewhat different than the ones presented in the literature [14] on top-down structured programming. Top-down structured programming in its "classical" form is usually practised at a single linguistic level that is considered to support modularity, because of property 1. The use of decreasing linguistic levels allows an interesting extension of the concept of top-down programming, since we are able to support this type of programming both within and between levels. A hint of this extension of top-down programming was contained in [15].

Bottom-up programming is characterized by properties 2 to 5 since we require modules to be independent and well-described entities, which allow for the composition of programs from ready-made (off-the-shelf) program units.

Properties 3 and 5 allow for incremental program verification since each module may be proved correct independent of its environment. In fact, if we examine the program as a whole, expressed at a given linguistic level, the properties of individual modules can be proved correct independently and then the properties of the modules can be abstracted when the whole program is proved correct. Of course, the same remarks apply to the testing of modules.

Properties 2 to 5 allow a module to be replaced by an equivalent one. Furthermore, property 4 (data generality) allows not only the replacement of a module by one that implements a different algorithm on the same data structures but replacement by one that uses the same or different algorithm on a different data structure. Optimization, that is, the selection of the most efficient implementation for a given specification becomes a widely applicable technique.

4. Levels of Language for Modular Programming

In previous sections there was no attempt to pre-define the number and form of the linguistic levels required for the complete expression of a programming system. Although it is now generally accepted that programming should take place through a spectrum of inter-communicating linguistic levels, which within levels proceed from various stages of specification to various stages of implementation, it is difficult to establish criteria to specify the number of linguistic levels and their respective characteristics. In previous sections only the conditions for a given linguistic level to support modularity have been stated; in this section a specific model is proposed that fixes the number of linguistic levels and tries to characterize modularity in each of these levels. Of course, this is only one possible model which might be proposed but it seems to work well in explaining the role of various programming constructs in modularity.

There are a number of reasons that support our proposed partitioning into three linguistic levels, which is also the partitioning we propose for our programming system for program development [19]. They are:

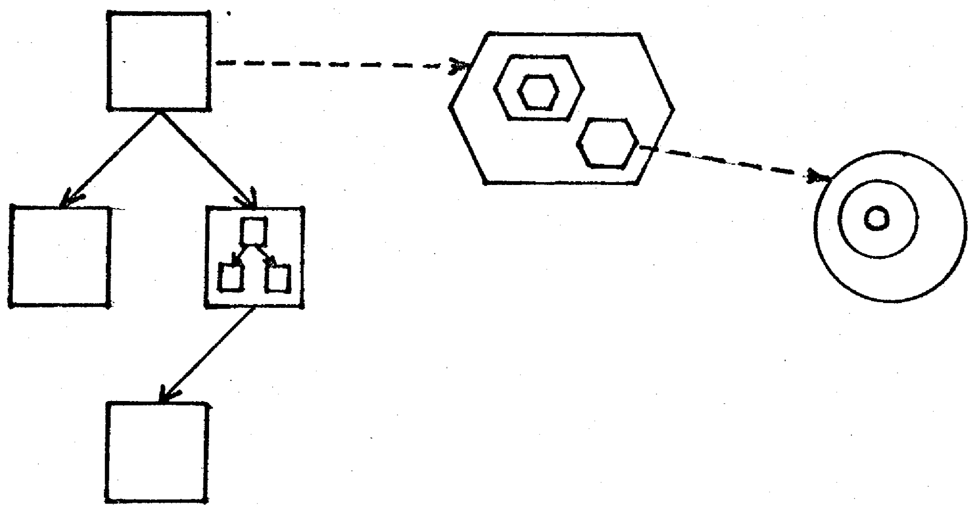
a) Some reasonable theoretical arguments in favor of similar three level models [16].

b) Most of the current language features and programming techniques in the literature can be comfortably classified into the three levels.

Modularity is examined at three linguistic levels that are called respectively systems, program and data-abstraction levels. As mentioned before, the reader will note some similarities between these three levels and the three levels called relational, access path and machine levels, that Earley [16] defines for data structures.

In a working modular system the three linguistic levels (the system-level, the program-level and the data-abstraction level) can be visualized in terms of contemporary program structures. At the system-level, modules (which we will call system-modules) are usually pre-compiled programs which are connected together by a module-interconnection language (using the terminology in [7]). The program-level, supports program-modules, which generally speaking stand for procedures in the ALGOL sense. The data-abstraction level, supports data-abstraction modules which stand basically for implementations of abstract data types. Figure 1 sketches the inter-relationships between our three selected kinds of modules.

The description in the previous paragraph is an analytical presentation of the levels of modularity. From a synthesis point of view, using a top-down approach, the three levels relate to each other in the following way: system-modules are specified together with their inter-connections; program-modules are defined so as to implement system-modules (data abstractions are left unspecified); data abstractions are later implemented and they define the programming system's machine level. It is important to emphasize at this point the differences between the control structures used at the various levels. The module interconnection language used at the system-level will probably look like a graph language,



□ System linguistic level (nesting of system interconnections)

⬡ Program linguistic level (nesting of procedures)

○ Data abstraction linguistic level (nesting of data abstractions)

$- - - \rightarrow$ Inter-level communications links

Figure 1 Module interconnection at the system, program and data abstraction linguistic levels.

describing the connection between the various nodes (system-modules) and the content of each node. At the program-level the control structures will be the standard control structures found in an ALGOL-like language (IFTHENELSE, DOWHILE, CASE, etc.). The data-abstraction level, illustrated in the next section, will call for a module interconnection language similar to the type used at the system-level.

5. Language Features for Modular Programming

We feel that most of the language features that exist or have been proposed to support the concept of modularity can be satisfactorily examined through the use of our three-level model (not that the model is completely general but probably because the problem of modularity has not been sufficiently explored). Furthermore, we think that our general concept of modularity and the use of the three-level model of linguistic levels, helps classify and clarify specific language features for modular programming.

Although we are considering the three-level model as the basis for our programming system for program development, no engineering decisions about specific system features will be discussed in this paper. By specific system features we mean special notation for module specification and programming mechanisms for module encoding. It can be observed that the higher the linguistic level, the closer the specification and the programming mechanisms will be.

5.1 The System-Level

The system-level is the highest level in our model. System-modules are specified (when they need to be built) or identified (when they are in a library waiting for off-the-shelf utilization) through a notation that can be well characterized by the expression: module interconnection language (MIL) [7]. Ready-made modules at this level are often termed load modules and object modules. When the newly specified modules are finally built (at lower linguistic levels), the MIL program is "compiled" into a linking language which is supported by some operating system.

It is possible to identify in the literature some notational systems which are candidates to support some form of modularity at this linguistic level. We will examine two of these notational systems in the context of our definition of modularity.

The MIL proposed in [7] was designed to define "system-modules". Top-down and bottom-up programming methodologies as well as some form of incremental proof and interchangeability can be achieved in that system. For each "module" m at a node n of the MIL system tree, two statements are required in the MIL program:

a) the "statement of origin", listing the resources defined in m;

b) the "statement of usage", listing the resources that are used, but not defined in m.

A complete MIL program consists of a sequence of "system-descriptions", each assumed to be compilable alone, or in conjunction with others. Figure 2 shows a system module specification example taken from [7].

The PSL (Problem Statement Language) is also a system-level design language developed by Teichroew and others at the ISDOS Project [9]. PSL uses the following concepts to describe a system-level design: _interface_ objects produce _inputs_ for the system and receive _outputs_ from the system; _sets_ (files) consist of _entities_ (records) which in turn consist of _groups_ and/or _elements_ (fields) of data; _processes_ produce and receive data subject to _events_, _conditions_ and _time intervals_ which describe dynamic behavior; _system parameters_ provide size information. The example in Figure 3 taken from [9] illustrates a system-level design of a very simple payroll programming system.

```
system Input
author 'Sharon Sickel'
date 'July 1974'
provides Input-parser
consists of
  root module
     originates Input-parser
     uses derived Parser, Post-processor
     uses nonderived Language-extensions
  subsystem Scan
     must provide Scanner
  subsystem Parse
     must provide Parser
     has access to Scan
  subsystem Post
     must provide Post-processor
```

Figure 2 DeRemer and Kron's system description

A description of a system-level design in PSL is composed of sections of various types. Sections and statements within sections can occur in any order.

Some comments are in order about the two system-level specification languages. The goal of PSL appears to be more general than DeRemer and Kron's MIL. Since both notations are machine-readable, a system for

PAYROLL-EXAMPLE

FORMATTED PROBLEM STATEMENT

```
PARAMETERS FOR   FPS

  FILE NOINDEX PRINT NOPUNCH SMARG=5 NMARG=20 AMARG=10 BMARG=25 RNMARG=70 CMARG=1 HMARG=40 DESG
  ONE-PER-LINE DEFINE COMMENT NONEW-PAGE NONEW-LINE

  1 INTERFACE                               departments-and-employees;
  2     GENERATES     employee-information;
  3     RECEIVES      paysystem-outputs;
  4
  5 INPUT                                   employee-information;
  6     GENERATED BY  departments-and-employees;
  7     RECEIVED BY   payroll-processing;
  8
  9 SET                                     payroll-master-information;
 10     UPDATED BY    payroll-processing;
 11
 12 PROCESS                                 payroll-processing;
 13     RECEIVES      employee-information;
 14     GENERATES     paysystem-outputs;
 15     UPDATES       payroll-master-information;
 16
 17 OUTPUT                                  paysystem-outputs;
 18     GENERATED BY  payroll-processing;
 19     RECEIVED BY   departments-and-employees;
 20
 21 EOF EOF EOF EOF EOF
```

Figure 3

specification analysis (called PSA in the case of the ISDOS project) can be built and consistency checks can be applied to the specification. Both specification languages are open-ended, in the sense that they may be further described (implemented) in terms of lower-level linguistic levels.

Examining both systems using our definition of modularity, the following conclusions can be drawn: the program units called section (both processes and data) are modules at the linguistic level defined by PSL. MIL's modules are not strong modules in the sense of our definition. To justify this statement is necessary to examine each of the properties that are required by our definition.

Property 1 (Composition by nesting) - Both sections and MIL's modules satisfy it. In fact, data definition and process sections in PSL may refer to other sections of the same type to an arbitrary degree of nesting. In MIL, modules can be composed to form systems of modules.

Property 2 (Syntactic non-interference) - Since no two PSL's sections or MIL's modules can bear the same name, syntactic non-interference is satisfied by the definition of the respective notational systems.

Property 3 (Semantic context-independence) - The linguistic level defined by PSL satisfies this property. Let us take a process section first. Its general form is the following:

```
Process Section
PROCESS name
RECEIVES inputs
GENERATES outputs
USES data TO DERIVE/UPDATE data
MAINTAINS relation and/or subsetting criteria
```

A process in PSL can both USE data generated by any other process and DERIVE (UPDATE) data to be used by any other process. The semantics of the data definitions are fixed (data is always defined in terms of ENTITIES, SETS, GROUPS and ELEMENTS). The semantics of a process section are therefore self-contained and property 3 is satisfied for these constructs. By the same token, data-definition sections, e.g. Set Section, are program units that satisfy property 3.

An MIL program defines the scope of definition of resource names across their module and subsystem boundaries. MIL allows for a controlled-access mechanism through which the power and the responsibility to establish channels for transmitting names of resources between siblings in its system tree rests solely with their parent. Since, in this case, the semantics of one module may have to be defined in terms of the resources defined in another module, we say that MIL modules do not satisfy property 3 (we will review this issue in section 6 of this paper).

Property 4 (data generality) - Both PSL and MIL program units satisfy this property. The interpretation of this statement needs special attention. PSL data and process sections do satisfy the property because they communicate via data structures whose meanings are universal to the PSL system. The same happens in MIL because there we have a trivial form of communication: resources are typeless. Data generality only holds in the latter case if we consider this linguistic level as a terminal one, that is, providing it does not communicate automatically with lower linguistic levels.

Property 5 (definitional completeness) - The linguistic level defined by PSL and MIL is suppported by language features that we can basically call definitional. The program text is identical to the program's documentation. In this sense we can say that both PSL and MIL satisfy property 5.

It should not be too difficult to "compile" languages at the system level such as MIL and PSL into linking languages which can be "interpreted" by an operating system, since the difference between non-executable specification languages and standard linking and job control languages appears to be minimal.

5.2 The Program-Level

At the program-level, a lower linguistic level, there are at present two types of interrelated languages. These are the specification languages such as plain assertion languages and Parnas' specification language [3] which describe how a module should behave and the implementation or so-called programming languages which describe how a module achieves its specification. Ideally one would like to compile a specification directly but efficiency considerations based on current computer architectures require that specifications be rewritten in procedural forms before code can be generated.

In order to bridge the present gap between specification (intent) and programming (implementation) at the linguistic level our system will use a very high-level programming language which will support abstract data structures [4]. This means that this language will have control structures to manipulate data structures and their operations but the definition of the data structures and operations will occur at another linguistic level (the data-abstraction level discussed in the next section). The gap between intent and implementation is bridged in the sense that the statement of the implementation and its specification can easily be proven equivalent by some known techniques (e.g.[21]). This is so because operational specification notations do not usually carry the description of implementation data structures. A standard high-level language which does describe these details is therefore level-incompatible with such specifications (one being far more abstract than the other) and therefore equivalence proofs are hard to obtain (even informal ones).

The next few paragraphs examine a number of different notations for specifying and implementing modules in the light of our definition of modularity. Specifically plain assertion languages, Parnas' specification language [3], the classical ALGOL procedures [20], forms [6] and clusters [4] are considered.

We have indicated that the very high-level programming language for our system operates on abstract data structures. Hence, there are two concepts of abstraction present at this level (which must not be confused): the module as an abstraction mechanism and data abstractions. It is not the intention at this linguistic level to consider the implementation of data structures but rather to use data in its abstract form. That rules out the consideration of clusters at this linguistic level since they are mechanisms for the implementation of abstract data types. The example presented in Figure 4 illustrates a program at the program-level using abstract data types. The PL/I extension used in this example was proposed in [22].

```
EX:  PROC OPTIONS (MAIN);
      DCL stack(type ABSTRACT_TYPE);
      DCL p stack(BIN FIXED);
        :
        :
      IF input='(' THEN stack@push(p,k)
                 ELSE IF input=')' THEN DO;
                                        PUT SKIP LIST(k,stack@top(p));
                                        stack@pop(p);
                                        END
```

Figure 4 Programming with abstract data types in PL/1

Forms and ALGOL-type procedures (functions) and the several variations of this concept are considered as the natural modeling facility for the concept of modularity at the program level. Although forms can also be seen as a modeling capability for data abstractions (acting as clusters) this particular capability does not belong at the linguistic level we are considering in this section. Forms were designed as a hybrid between a macro and a class (in SIMULA [23]). It subsumes the notions of macro, procedure, generator, and/or coercion (and as we mentioned before the notion of type as well).

At this point we need to find out if standard procedures and forms define modules, in our sense, at this linguistic level. It is straightforward to conclude that they do not. Both procedures and forms (through some control mechanisms) allow the possibility of sharing and hence violate properties 2 and 3. Suppose that forms are not considered further for this reason (this concept of sharing will be reviewed later), and that a version of procedure is adopted in which only data communication through parameters is allowed. Procedures would have then satisfied properties 1 to 3, but would still fail to satisfy the property of data generality. In fact, current programming languages do not support features which allow for the free interchange of data structures between procedures with no knowledge of the communicating procedures' internal representations. This question will be addressed separately in section 5.4.

Even though procedures are not able to satisfy rigorously our concept of modularity at this linguistic level, some notational features which can easily be incorporated into procedures allow fulfillment of property 5 (definitional completeness) and hence achieve a reasonable approximation to modularity. In fact, it has been suggested that the assertions that specify programs expressed at this level should be made an integral part of the program text. Although this may be a considerable improvement over the conventional program documentation techniques, this approach is still simplistic because it does not allow expression of several fundamental properties of programs (such as fault tolerance) since they are not normally expressible through assertion languages. Parnas [3] has proposed a notation for program specification that when used in assocation with procedures appears to be adequate for the attainment of property 5. The notation requires some improvement since it does not allow specification of precision or cost of the module being defined and is not amenable to mechanization in its present form. Figure 5 shows a self-explanatory example of Parnas' notation taken from [3].

```
Function PUSH(a)

possible values: none
integer: a
effect: call ERR1 if a > p2 v a < 0 v 'DEPTH' = p1
        else [VAL = a; DEPTH = 'DEPTH'+1;]

Function POP

possible values: none
parameters: none
effect: call ERR2 if 'DEPTH' = 0
        the sequence "PUSH(a); POP" has no net effect if no error
        calls occur.

Function VAL

possible values: integer initial; value undefined
parameters: none
effect: error call if 'DEPTH' = 0

Function DEPTH

possible values: integer; initial value 0
parameters: none
effect: none
p1 and p2 are parameters.  p1 is intended to represent the maximum
depth of the stack and p2 the maximum width or maximum size for
each item.
```

Figure 5 Parnas' module specification notation

5.3 The Data-Abstraction Level

In the previous section consideration was given to language features to be used at the program-level. At this point it was decided to treat data structures as an abstract entity and delay consideration of their implementation to another linguistic level. As a consequence discussion of the cluster mechanism was delayed until this section.

At the data-abstraction level we are interested in specification notations and programming mechanisms to describe and implement data abstraction modules, which sometimes are called types. A cluster [4,5] is one such mechanism and is composed of an operation list (names of all operations applicable to objects of the type), an object description (representation that implements the type) and a sequence of operation definitions. Most important of all, a user of a data abstraction modeled through a cluster can manipulate objects only through the operations names and not directly via their representation. This establishes a satisfactory link with program modules (program modules do not have side-effects from this source) and allow data abstraction modules modeled through clusters to satisfy properties 2 and 3. Since clusters can be used inside clusters to an arbitrary degree of nesting, property 1 is also satisfied.

As in the previous section, we cannot conclude that the proposed mechanism satisfies the definition of modularity, because clusters do not allow data generality. Data generality can be characterized for the present case, in the following way. If two variables a and b are declared to be of the same abstract data type T, it is conceivable that T would be implemented by two different clusters using different data representations. If we now define a binary operation $a \oplus b$ we would expect the operation to take place in a manner transparent to the user. In other words, data generality would guarantee that variables of the same type with different underlying representations could communicate without any knowledge about their respective implementations. Clusters do not admit this possibility.

The concept of clusters can be conveniently defined via a number of specification techniques for data abstractions. Liskov and Zilles survey some of these specification techniques in [5]; Figure 6 illustrates two of them. It is reasonable to conclude that clusters satisfy property 5 since they can be described by either of these methods of specification.

The reader may have concluded that we are being quite permissive about the requirements to satisfy property 5. In fact, clusters as they are currently specified, do not have all the important properties required to satisfy the definitional completeness criteria. Two examples are presented to illustrate this point. First, cluster specifications certainly do not indicate error conditions and exceptions both of which could be encountered in data abstraction modules. Second, it can be observed that the only free variables in a cluster are other cluster names, which must eventually be bound to object modules of clusters. Here we have a specification problem which is close to the specification problem at the system-level. One can conceive of a cluster interconnection language which specifies who knows whom within a collection of data-abstraction modules modeled by clusters. Certainly current cluster specifications do not completely define their interconnections. Thomas [8] has actually called MIL a programming system that helps the expression of this type of specification.

One should also note that a set of clusters used to program another cluster characterizesa virtual machine. Such a characterization is a useful abstraction when one considers a common base language (section 5.4).

1 $\mathrm{CREATE}\ (\mathrm{STACK})$

2 $\mathrm{STACK}(S)\ \&\ \mathrm{INTEGER}(I) \supset \mathrm{STACK}(\mathrm{PUSH}(S,I))\ \&$

$\qquad [\mathrm{POP}(S) \neq \mathrm{STACKERROR} \supset \mathrm{STACK}(\mathrm{POP}(S))]\ \&$

$\qquad [\mathrm{TOP}(S) \neq \mathrm{INTEGERERROR} \supset \mathrm{INTEGER}(\mathrm{TOP}(S))]$

3 $(\forall A)[A(\mathrm{CREATE})\ \&$

$\qquad (\forall S)(\forall I)[\mathrm{STACK}(S)\ \&\ \mathrm{INTEGER}(I)\ \&\ A(S)$

$\qquad\qquad \supset A(\mathrm{PUSH}(S,I))\ \&\ [S \neq \mathrm{CREATE} \quad A(\mathrm{POP}(S))]]$

$\qquad \supset (\forall S)[\mathrm{STACK}(S) \supset A(S)]]$

4 $\mathrm{STACK}(S)\ \&\ \mathrm{INTEGER}(I) \supset \mathrm{PUSH}(S,I) \neq \mathrm{CREATE}$

5 $\mathrm{STACK}(S)\ \&\ \mathrm{STACK}(S')\ \&\ \mathrm{INTEGER}(I)$

$\qquad \supset [\mathrm{PUSH}(S,I) = \mathrm{PUSH}(S',I) \supset S=S']$

6 $\mathrm{STACK}(S)\ \&\ \mathrm{INTEGER}(I) \supset \mathrm{TOP}(\mathrm{PUSH}(S,I)) = I$

7 $\mathrm{TOP}(\mathrm{CREATE}) = \mathrm{INTEGERERROR}$

8 $\mathrm{STACK}(S)\ \&\ \mathrm{INTEGER}(I) \supset \mathrm{POP}(\mathrm{PUSH}(S,I)) = S$

9 $\mathrm{POP}(\mathrm{CREATE}) = \mathrm{STACKERROR}$

a. Axiomatic specification of the stack abstraction.

Functionality:

CREATE	:	$\rightarrow \mathrm{STACK}$
PUSH	:	$\mathrm{STACK} \times \mathrm{INTEGER} \rightarrow \mathrm{STACK}$
TOP	:	$\mathrm{STACK} \rightarrow \mathrm{INTEGER} \cup \mathrm{INTEGERERROR}$
POP	:	$\mathrm{STACK} \rightarrow \mathrm{STACK} \cup \mathrm{STACKERROR}$

Axioms:

1' $\mathrm{TOP}(\mathrm{PUSH}(S,I)) = I$
2' $\mathrm{TOP}(\mathrm{CREATE}) = \mathrm{INTEGERERROR}$
3' $\mathrm{POP}(\mathrm{PUSH}(S,I)) = S$
4' $\mathrm{POP}(\mathrm{CREATE}) = \mathrm{STACKERROR}$

b. Algebraic specification of the stack abstraction.

Figure 6 Specification techniques for data abstractions

5.4 Data Generality and Common Base Languages

It has been recognized that the most critical property of modularity is data generality. That is, the definition and use of module interfaces to allow the passing of data between modules independent of data representation. Looking at our three-level model, it can be seen that problems occur with the available program features at all the levels.

At the system-level, for instance, data generality is only achieved when a universally known data representation (normally supported by a physical medium) is provided. Figure 7 illustrates this fact.

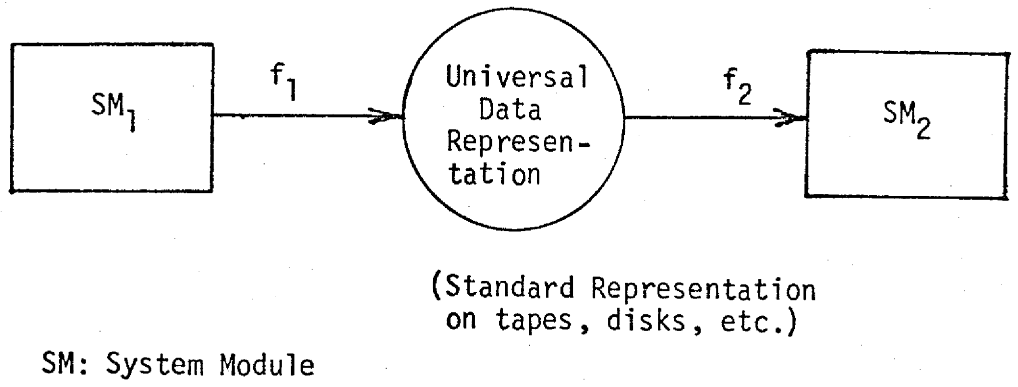

SM1
f1
Universal Data Represen-tation
f2
SM2
(Standard Representation on tapes, disks, etc.)
SM: System Module

Figure 7 Data Generality

The functions f1 and f2 map the data structures in SM1 to a standard representation and from a standard representation to SM2 respectively. The idea of a universally known data representation allows the achievement of data generality at all linguistic levels. The previous situation can be simulated, in general, even in the case in which the same physical medium such as main memory is being used for programs and data by choosing a universal data representation and adopting a specific programming discipline. The programming discipline would require the programmer of a module to write the functions f1 and f2 and incorporate them into the module. The communication, with data generality, between two modules would take place as illustrated in Figure 8.

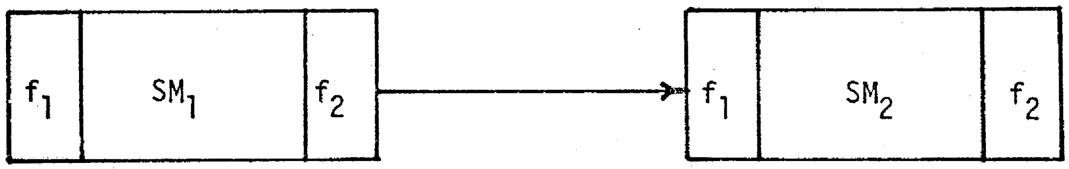


Figure 8 A form of communication with data generality

The same solution can be transported from the system-level to the lower linguistic levels. For instance, if we always incorporate the two functions f1 and f2 into clusters, the binary operation $\oplus$ between the variables a and b and the binary operation $\leftarrow$, that assigns a value to c in the statement

$$c \leftarrow a \oplus b ,$$

could be executed automatically even if a, b and c, of the same type T, used different implementations for T. This is the application of the same data generality mechanism to the data-abstraction level.

Less expensive solutions can also be proposed to allow the language features discussed before to satisfy our definition of modularity. One efficient solution is based on a generalized use of _type descriptors_ [10,11] of the kind defined by clusters. It consists in the _transmission_ of type descriptors between modules. Through this approach values of type _type_ (data abstractions) can be transmitted, i.e., there are variables, parameters and functions in the programming language of type _type_. The transmission of type descriptors allows the postponement of strong type checking until the moment when modules are assembled together. Since type descriptors typically do not change during execution, it is possible to perform static type checking. Conditions for static type checking and parameter association can be found in [11]. Von Staa's approach allows the communication with data generality at any level of our model, providing only that the different modules follow the specification (definition) of the operations in the data abstraction.

In Figure 9 we show how a data type descriptor (cluster-like) implementing strings can be defined. To reduce the size of the example, several simplifying assumptions were made. It is assumed that a (1:0) vector is the null vector as in APL. It is also assumed that vector assignments are possible as in ALGOL 68 or PL/I. Also, the assumption is made that some form of dynamic memory management exists such as in LISP or SNOBOL 4.

```
Type string of (type user_type which defines (vector, copy) is
   begin string;
       user_type vector value(1:flex size:=0);
       outside_scope functions;
           null is value:=vector.null;
           function copy(string from) is value:=from.value;
           function convert(user_type vector from) is value:=from
           function concat(string head, tail) is value:=convert
                         (vector.concat(head,tail));
       end functions;
   end string;
```

Figure 9 Data descriptor "string" receives a type descriptor as parameter

To properly implement the above programming mechanism a method was devised through which it is possible to combine modules which receive parameters of the same abstract type that have different implementations. The method requires the existence of a table of conversion functions and an algorithm that establishes how to get from one representation to the other (based on existing representations).

After the characterization of programming features for data generality in the context of our definition of modularity and of our three-level model for program development, it is important to clarify the requirements for a common base language.

In both methods for achieving data generality discussed above, the problem of conversion between representations is present. Both the encoding of the program segments that implement the functions f_1 and f_2 in the first method and the encoding of the programs that constitute the conversion functions in the conversion function table used in connection with the second method can be, in general, very complex tasks.

If programmers were able to encode the implementation of clusters in a set of instructions in a common base language perhaps the solution to our problem would become more tractable. Specifically, the common base language would probably contain certain basic data types such as integers, reals, lists, etc. and operations on them such as add, subtract, insert and delete. There could be several implementations of each basic data type and its corresponding operations. All clusters will be encoded in the basic data types either directly or through nesting. Different clusters may use different representations of the same basic data type and so there still is the problem of data generality to be conquered. This problem may now be manageable since only conversion routines have to be provided between the basic data types and not between all data types used.

Dennis [1] proposes to borrow these operations from VDL. Standish's operations on basic data structures, which resemble VDL also serve as a basis for the language. We opted for the practical approach of using Tompa's [24] and Low's [25] type of operation primitives to be able to compile clusters into any representation from a predefined library of representations. In summary, the advantages of writing the low-level details of clusters in terms of a common base language are many fold:

a) It supports portability.

b) It allows for the automatic selection of the best underlying representation [25].

c) It facilitates the encoding of conversion functions.

Conversion functions can naturally be encoded through the use of the following set of primitives (common base language).

ADD:	adds an element to the structure
SUB:	subtracts an element from the structure
SELECT:	selects an element from the structure
INSERT:	inserts a new element into the structure
REPLACE:	replaces an old element in the structure
LINK:	links two sub-structures
DETACH:	detaches two sub-structures
COPY:	generates a copy of the structure
SUCC:	finds the successor of a given element in the structure
PRED:	finds the predecessor of a given element in the structure

The operations can be easily axiomatized as in [12]. A cluster that refers to a base representation level has the following form (in extended PL/I [22]):

```
name: CLUSTER ON REP1 (parameter list) IS op1,...,opn;
      [declaration of global (to the cluster) variables]
      CREATE
         DCL r REP;
         [create body]
      ENDCREATE
      op1: PROC (parameter list) RETURNS (type);
         [declaration of local variables]
         [op1 body]
      END op1;
      :
      opn: PROC...
      :
END name;
```

In op_1 to op_n REP1 is handled in terms of the primitive operations. In the PL/I extension version, the programmer has access at this level to all PL/I data types with the exception of pointer and base variables. This is meant to delay the use of implementation details to the base representation level (handled by the standard operations in the common base language). A base representation-level cluster is then written in the following manner, where the symbol template stands for the PL/I data types used to implement the concrete representation.

```
repr: REP (parameter list) USES <template>;
          [declaration of global (to the cluster) variable]
        CREATE
          [create body]
        ENDCREATE
        ADD: PROC (parameter list);
          [declaration of local variables]
          [body of standard operation ADD]
        ENDADD;
        :
        SUB: PROC...
        :
        SELECT: PROC...
        :
    END repr;
```

By the presentation of the above language features it was our intent to explain our views on how the concept of common base languages relates to the concept of modularity by helping its effective implementation.

6. On the Problem of Sharing

So far, we have been characterizing language features' modeling capabilities according to our very strict definition of modularity. We have seen that it appears possible to design language features to achieve full modularity. Such a form of modularity will certainly contribute to reduced costs and enhance reliability in a number of programming areas. We believe that the so-called applications programming area will be the greatest beneficiary of this approach. Nevertheless, we think that the so-called systems' programming area could be hampered by the amount of discipline which is required. Efficiency problems can appear in rather unique ways in systems programming. This is a good reason for weakening the requirements on modularity, providing it is clear that this weakening is taking place.

The concept that systems' programming requires is the concept of sharing (particularly data sharing, since program sharing can be obtained with full modularity). We believe that a programming system which supports modularity, should also support a version of modularity that could be called modularity with sharing. Of course, the sharing should be controlled through access rights mechanism in order that a high degree of modularity be maintained. In this version of modularity DeRemer and Kron's MIL and the form concept are welcome. In particular, they provide for access rights control mechanisms.

Since sharing implies the loss of semantic context-independence, a number of new specification features are required. We envisage a notation system in which the properties that were subtracted from the independent modules (the ones that cannot be proven independently because of the interrelationship with other modules) are factored out and verified together. This is similar to the proof of certain properties (deadlocks, etc.) of systems modeled through their graph models of computations (which display only control relationships). Once the system's data interconnections are proven correct we can then move to the verification of the individual modules. The extra burden imposed on the system design and implementation process is, in this case, probably not as serious as in the applications area, since it can be claimed that systems programmers are usually far more specialized in programming techniques than applications programmers.

7. Conclusions

We have proposed a definition of modularity based on five properties that try to capture the complete semantics of the concept. We also studied several implications of the definition with respect to the design of language features to support modularity. Several of the current proposed language features do satisfy our very strict definition of modularity providing that a few changes are introduced. The changes have mostly to do with supporting the property of data generality. We acknowledge the need for sharing in the design of programming systems and claim that a weak concept of modularity should also be accepted providing that it is reinforced by some additional specification features. We believe that our approach to the modularity issue contributes to the better understanding of the concept and that in particular it helps designing systems to support modular programming.

References

[1] Dennis, J., "Modularity" in Advanced Course on Software Engineering, Springer Verlag, 1973.

[2] Parnas,D.L., "On the Criteria to be used in Decomposing Systems into Modules",CACM, vol.15, No.12, 1972.

[3] Parnas,D.L., "A Technique for the Specification of Software Modules with Examples", CACM, vol.15, No.5, 1972.

[4] Liskov, B.H. and Zilles, S.N., "Programming with Abstract Data Types", Proceedings of ACM SIGPLAN Symposium on Very High Level Languages, SIGPLAN Notices, vol.9, No.4, 1974.

[5] Liskov, B.H. and Zilles, S.N., "Specification Techniques for Data Abstractions: IEEE Transactions on Software Engineering, vol.1, No.1, 1975.

[6] Wulf, W.A., "ALPHARD: Toward a Language to Support Structured Programs", Technical Report, Carnegie-Mellon University, 1974.

[7] DeRemer, F., Kron, H., "Programming-in-the-Large Versus Programming-in-the-Small", Proceedings of the International Conference on Reliable Software, Los Angeles, 1975.

[8] Thomas, J.W., "The Basis for a Module Interconnection Language for CLU", Report CS-9, Computer Science Group, Brown University, 1975.

[9] Teichroew, D., Bastarache, M.J., "PSL User's Manual", ISDOS Working Paper No.98, Department of Industrial Engineering, The University of Michigan, 1975.

[10] Staa, A.v., Lucena, C.J., "On the Implementation of Data Generality", Research Report IS-1-75, Departamento de Informatica, Pontificia University Catolica do Rio de Janeiro, 1975.

[11] Staa, A.v., "Data Transmission and Modularity Aspects of Programming Languages", Research Report CS-74-17, Department of Computer Science, University of Waterloo, 1974.

[12] Lucena, C.J., "On the Synthesis of Reliable Programs", Technical Report UCLA-ENG-7505, Computer Science Department, University of California at Los Angeles, 1975.

[13] Guttag, J.V., "The Specification and Application to Programming of Abstract Data Types", Technical Report CSRG-59, 1975.

[14] McGowan, C.L., Kelly, J.R., "Top-Down Structured Programming Techniques", Petrocelli/Charter, New York, 1975.

[15] Mills, H.D., "OS 360 Job Control Language Programming", Classroom notes.

[16] Earley, J., "Relational Level Data Structures for Programming Languages", Acta Informatica 2, 1973.

[17] IBM Report GC 20-1851-1, "HIPO - A Design Aid and Documentation Technique", 1975.

[18] Floyd, R., "Assigning Meaning to Programs", American Mathematical Society, vol.19, 1967.

[19] Lucena, C., Cowan, D.D., "Toward a Systems Environment for Computer Assisted Programming", to appear.

[20] Naur, P. et al., "Revised Report on the Algorithmic Language ALGOL 60", Comm. ACM 6, 1, 1963.

[21] Hoare, C.A.R., "Proofs of Correctness of Data Representations", Acta Informatica, vol.1, No.1, 1972.

[22] Schwabe, D., Lucena, C., "Specification and Uniform Reference to Data Structures in PL/I", to appear.

[23] Dahl, O.J. et al., "The Simula 67 Common Base Language", Norwegian Computing Centre, Oslo, 1968.

[24] Tompa, F.W., "Choosing a Data Storage Schema", Ph.D. Thesis, University of Toronto, 1974.

[25] Low, J.R., "Automatic Coding: Choice of Data Structures", Stanford University, Computer Science Department, STAN-CS-74-452, 1974.