PROGRAMMING-IN-THE LARGE VERSUS PROGRAMMING-IN-THE-SMALL

Frank DeRemer
Hans Kron

University of California, Santa Cruz

Key words and phrases
Module interconnection language, visibility, accessibility, scope of definition, external name, linking, system hierarchy, protection, information hiding, virtual machine, project management tool.

Abstract
We distinguish the activity of writing large programs from that of writing small ones. By large programs we mean systems consisting of many small programs (modules), possibly written by different people.

We need languages for programming-in-the-small, i.e. languages not unlike the common programming languages of today, for writing modules. We also need a "module interconnection language" for knitting those modules together into an integrated whole and for providing an overview that formally records the intent of the programmer(s) and that can be checked for consistency by a compiler.

We explore the software reliability aspects of such an interconnection language. Emphasis is placed on facilities for information hiding and for defining layers of virtual machines.

1. Introduction

Programming a large system in any typical programming language available today is an exercise in obscuration. We work hard at discovering the inherent structure in a problem and then structuring our solution in a compatible way. Research into "structured programming" (Dijkstra 1972) tells us that this approach will lead to readable, understandable, provable, and modifiable solutions. However, current languages discourage the accurate recording of the overall solution structure; they force us to write programs in which we are so preoccupied with the trees that we lose sight of the forest, as do the readers of our programs!

Let us refer to typical languages as "languages for programming-in-the-small" (LPSs). Let us use the term "module" to refer to a segment of LPS code defining one or more named resources. Each "resource" is a variable, constant, procedure, data structure, mode, or whatever is definable in the LPS. Preferably a module is one to a few pages long and is easily comprehensible by a single person who understands the intended environment and function of the module.

Work reported herein was supported in part by the National Science Foundation via grant number GJ 36339.

We argue that structuring a large collection of modules to form a "system" is an essentially distinct and different intellectual activity from that of constructing the individual modules. That is, we distinguish programming-in-the-large from programming-in-the-small. Correspondingly, we believe that essentially distinct and different languages should be used for the two activities. We refer to a language for describing system structure as a "module interconnection language" (MIL); it is one necessity for supporting programming-in-the-large.

An MIL should provide a means for the programmer(s) of a large system to express their intent regarding the overall program structure in a concise, precise, and checkable form. Where an MIL is not available, module interconnectivity information is usually buried partly in the modules, partly in an often amorphous collection of linkage-editor instructions, and partly in the informal documentation of the project. Aside from the issue that each of these three areas is ill-suited to express interconnectivity, the smearing of the relevant information over disjoint media is highly unreliable. Even more unsatisfactory are the facilities for specifying and enforcing module disconnectivity via informating hiding, limiting access to resources, establishing protection layers, closing off subsystems, etc. The lack of such facilities invites undisciplined or even unsocial programming, as shown by one of Weinberg's case studies (Weinberg 1971, pp. 71-75), since there is no automated means of enforcing the surface consensus of the programming team.

That current languages fail to support the global task of composing large systems was well argued by Wulf and Shaw in their paper entitled "Global variables considered harmful" (Wulf 1973). A responding paper (George 1973) proposed a scheme of declarations to augment block structure as a solution to the problems associated with global variables. The scheme provided mechanisms to protect variables from violations of various sorts by contained blocks, and to allow limited access to certain variables by selected internal blocks. Similar approaches have been suggested by others (Clark 1971, White 1972, and Ichbiah 1974). We believe that some of the mechanisms proposed were appropriate, but that they were inappropriately placed in the LPS.

That is, we distinguish between block structure and module interconnectivity. Block structure works well on a small scale, but humans simply cannot keep track of nesting levels after a few pages. Furthermore, and perhaps most important, module interconnectivity must in many cases take the shape of a graph or partial ordering. The more limited tree structure of nested blocks forces us to place some low-level modules at high-level places, extending their scope of definition to inappropriate places. It follows, then, that we need a separate language, or at least separate language constructs, for describing module interconnectivity, rather than complicating existing constructs that are well suited for programming-in-the-small.

Similarly, we reject Liskov's strong association of module interconnectivity with data abstraction (Liskov 1974). We believe that both are necessary, but that they should not be tied together inextricably. The programmer is the best judge of when they ought to be used in concert; he should be able to state his intents and to rely on a compiling system to see that they are carried out correctly. We regard data abstraction as a technique that should be supported by an LPS, rather than by a global mechanism such as an MIL.

2. Objectives of any module interconnection language (MIL)

In general, an MIL should serve as (1) a project management tool, (2) a means of communication between members of a programming team, (3) a design tool for, and actual means of establishing overall program structure, and (4) a means of documenting that structure in a clear, concise, formal and checkable way.

Top-down. As a communications device it should be especially effective from the top down. That is, it should facilitate the structuring of a system by a project manager (Weinberg 1971) or chief programmer (Mills 1970) or "modularizer" (Maynard 1972), and it should facilitate the communication of that structure to the relevant programmers.

One level at a time. Furthermore, it must encourage the structuring on one level at a time, since we humans do not usually deal effectively with several levels simultaneously. Given a job a programmer should be able to switch to the role of chief programmer for his subtask, assigning subproblems to his assistant programmers via an MIL program, just as he was given his job assignment by his superior. Such separate assignment of task and subtasks, via separate MIL programs, is perhaps most important when the chief programmer and his assistants are, in fact, one and the same person. The most likely person to attempt to violate system structure, whether intentionally or accidentally, is the one who is familiar with more than one level.

Bottom-up. Of course, an MIL should also support bottom-up programming. After all, we frequently gain understanding of a problem by first working on some of its details. Also, we often can make use of subsystems that have already been written to solve parts of other problems. In such cases we must be able to compose systems from existing subsystems.

Horizontal. Similarly, programmers at a single level may need to communicate. Each may need a resource supplied by the other. Or they may be creating co-subsystems, in the sense of coroutines. Or they may be writing mutually recursive subsystems. Thus, module linkage in the horizontal, as well as the vertical direction is needed.

Composition. Finally, when all the partial MIL programs for a system are put together, they should constitute a complete definition of the overall system structure. The description should be readable by humans to aid in understanding the system. Moreover, the compiling system should print graphical representations of the system structure, preferably in several layers of detail, for ease of human consumption.

Linkage. One may regard an MIL as being a higher-level language for specifying how a "linker" is to prepare for "loading" a program comprised of separately compiled segments (Presser 1972). Roughly, the linker must resolve static references to "external" names; i.e. names defined externally to each separately compiled module. A distinction, however, is that we do not expect the linkage to happen after compilation but rather as part of it.

Proving-in-the-small. Thus, an MIL might help to alleviate the disadvantages of independent compilation of modules, as addressed by Hoare (Hoare 1973). Modules can be small without sacrificing a proper description of the problem structure, since the latter is adequately and explicitly expressed in the MIL program. Working with small modules, we may find it less prohibitive to prove their correctness.

Proving-in-the-large. After establishing the correctness of the modules, we may be able to prove separately, on the MIL level, that correct modules work together correctly. Due to formalizing the description of system structure via a language, we may build a compiler that can check that a system of modules does indeed conform to the intended structure. That structure might be designed, for example: (1) to hide certain information (Parnas 1971) or (2) to build layers of virtual machines (Dijkstra 1972). Any MIL should inherently support these two concepts.

Independent of formal and mechanized proofs of correctness, the protection and documentation provided by the MIL program enhance the likelihood of correct construction in the first place and correct modifications later on.

Trade-off. One cost of this approach will be a more complex compiling system than has been necessary heretofore. However, the compiler can be more helpful to us by providing more feedback at early stages of system development. The cost should be more than offset by the increased speed and accuracy with which we will be able to construct and maintain large systems.

In summary, then, an MIL should:
(1) encourage the structuring of a system before starting to program the details;
(2) encourage the programming of modules assuming a correct environment, but without knowledge of the irrelevant details of that environment;
(3) encourage system hierarchy, while allowing flexible, if disciplined connections between modules;
(4) encourage information hiding and the construction of virtual machines, i.e. subsystems whose

internal structure is hidden, but which provide desired resources; and
(5) encourage descriptions of module interconnectivity that are separate from the descriptions of the modules themselves.

3. An example

To demonstrate the use of an MIL we graphically display a sample program structure in Figure 1. The particular system illustrated is a theorem proving program written in Algol/W (Wirth 1966) by Professor Sharon Sickel at the University of California, Santa Cruz. The program was written without the aid of an MIL; we tediously reviewed it after it was complete and factored out the overall structure. The exercise took several hours because the structure was not completely clear in its author's mind, nor was it apparent from the program listing, and because the exercise suggested some ways to improve the structure; in fact, what is presented here is the improved version.
The missing part of Figure 1 is developed in Section 5, culminating in Figure 5. In that section, we also define the concepts that are only sketched next.

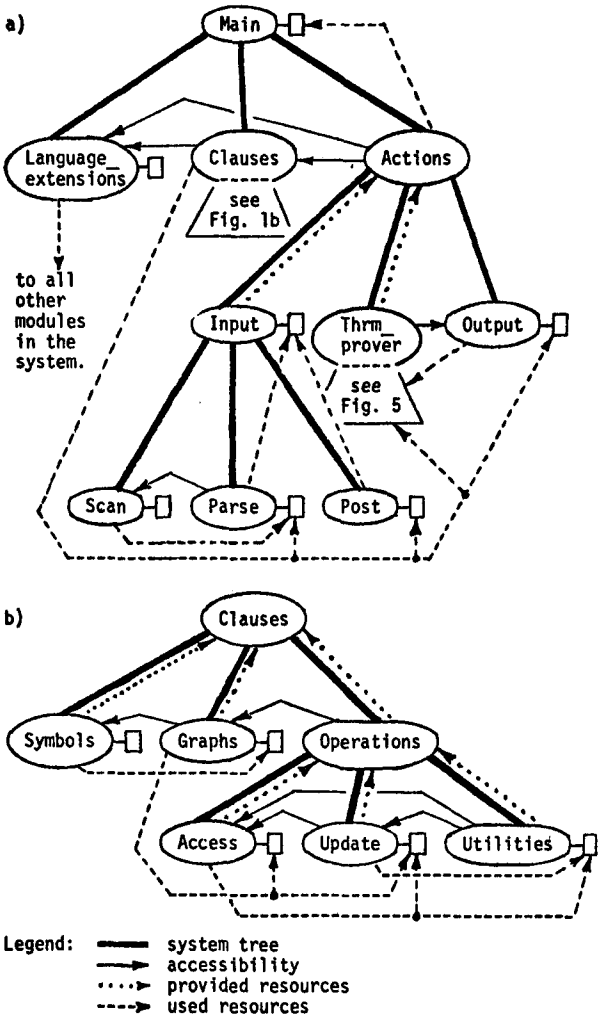

Figure 1. The system structure for a theorem proving program.

Graphical representation. Any system structure is represented as follows:
(1) Nodes and bold edges constitute the "system tree"; each edge connects a "parent" (system) to one of its "children" (subsystems).
(2) Dotted lines going upward along the tree edges from a child to its parent indicate that resources "provided" by the child are also "provided" by the parent. In other words, some of the resources provided by the child are passed via the parent to siblings and/or ancestors of the parent.
(3) Solid arrows, always between siblings, denote "sibling accessibility links". These are established by the parent, viz. chief programmer, so that the siblings may use each other's resources as necessary.
(4) Rectangular boxes attached to the nodes indicate LPS modules. Every leaf has a module. In our sample system, each module attached to a non-leaf serves the purpose of being the "driver" of its (sub-) system.
(5) Dashed arrows indicate that some resources provided by the node at the tail of the arrow are used by the module at the head of the arrow. The global aspects of the flow of resource names are addressed in detail in Section 5.

Refinements. Figure 1 represents the program at a high level of abstraction. As the diagram is further developed, specific resource names would start to appear and refinements would be made to accessibility. For examples, a parent might give a child accessibility to only a subset ("group") of the resources provided by siblings of the child, or the parent might hide some or all of its environment from the child.

Programming versus structuring. It is to be emphasized that "top-down programming" of a system would not necessarily proceed top-down relative to such a system diagram. Indeed, we are likely to start by programming the essential algorithms first, e.g. those inside Thrm_prover here, and deducing from their descriptions what would be the best form for the data to take and what innate operations should be supplied with the data, e.g. in the Clauses subsystem here. Adding the names of such data and operations to our diagram and appropriately refining related links would be working in a distinctly bottom-up fashion relative to the diagram. Nonetheless, we regard it as obvious that this overall approach is a top-down one relative to the problem being solved.

Misuse of structure. We now regard the modified Algol/W program as being reasonably well structured and the diagram of Figure 1 as a good display of its overall structure. One exception is that we believe the Clauses subsystem to be over-structured, in the sense that the Algol/W program structure is being used to try to achieve data abstraction. Unfortunately, this technique is both incomplete and uncheckable.
It is incomplete in that all of the irrelevant details of implementation of "clauses" cannot be hidden in Algol/W; one must rely upon programmer self-discipline as regards the correct use of data manipulation procedures provided with the virtual data type "clause". The technique is uncheckable by the compiler because it does not know that this

part of the program structure is being used for data abstraction, which in turn is due to Algol/W having no data abstraction facility to hide the details of implementation of new data types.

An LPS with general data abstraction facilities would obviate the need for such misuse of program structure. Likewise, an MIL would eliminate the necessity of using block structure to describe overall system structure.

4. The universe of discourse of MIL 75

We now present a particular language, MIL 75, for describing interconnections among modules. Being quite new, it has recieved little use thus far, so we expect it to evolve further toward an optimally useful language for describing system structure. Of course, the language is intended to satisfy the objectives stated in Section 2 above. It would, however, be a poor tool for enhancing reliability if it innately included "harmful" or unreliable features. Avoiding the latter must, of course, be a design objective of any language.

Names. The universe of discourse of MIL 75 consists of names: the names of resources originating in the separate modules, the names of the modules themselves, and the names of systems of modules.

External scope. An MIL 75 program addresses the question of who knows whom within a collection of modules. It defines the scopes of definitions of names across module and subsystem boundaries. It has nothing to say about the scopes of definitions within modules, these being defined by block structure and/or other constructs in the LPS.

Static, not dynamic. We emphasize that the interconnections addressed in MIL 75 are static ones, just as the names used to establish the connections are statically known, rather than being computed at run-time. In effect, the MIL program structures a global region through which the modules communicate. Thus, for example, MIL 75 could be used as a high-level language for programming the "global" declarations that are included in the compilation of each BCPL module (Richards 1969). However, MIL 75 admits of rather less restricted implementations, as may be deduced from Section 5.

No loading. Furthermore, it is to be stressed that MIL 75 does not address the problem of loading. That is, it has nothing to say about when modules or subsystems are to be loaded, nor does it say what, if any, overlay scheme is to be used. Perhaps we need, in addition to an MIL, a "subsystem loading language" to address exactly those issues. Presumably, the MIL program provides many of the right points of reference for describing loading and overlay strategies.

No functional specification. An MIL 75 program does not specify the nature of resources; it only specifies what those resources are to be named. Of course, the functional specification of modules and subsystems is important, but that is a separate issue not dealt with in this paper. A good solution may be to coalesce an MIL and a "function specification language". Perhaps the latter would be a language of axioms (Parnas 1972, Hoare 1972, Guttag 1974).

No types. Similarly, MIL 75 does not provide any ways of specifying the type of an object or defining language extensions. Rather it is used to specify paths for transmitting relevant information from one module to another. Such paths may be defined for any named entity that has its defined and applied occurences distributed over different LPS modules.

It is assumed, however, that the total LPS + MIL compiling system will do as much bookkeeping as necessary to do all static checking as soon as the necessary information is available. Clearly, this will require a non-trivial file system so that the compiler may keep summaries of each module and its external connections for the compiler's own use in subsequent compilations and recompilations. A modification of Liskov's "description units" (Liskov 1974) seem to be appropriate for such bookkeeping.

Accessibility. Finally, an MIL 75 program gives a particular module either unrestricted access to an object or none at all. We assume that any restrictions such as "read only", "write only", "read before write", "execute only", and other more general monitorings, are appropriate to the domain of, and programmable in, the LPS.

5. The semantics of MIL 75

The module interconnection language MIL 75 can be defined via attribute grammars (Knuth 1968); essentially, an MIL 75 program specifies a tree whose nodes are augmented by attributes. The latter are sets of resource names. In this section, however, we define the language by starting with a simple algebraic structure (a tree) and refining it stepwise. Simultaneously, the language concepts are presented and motivated by the stepwise development of the subsystem, Thrm_prover, which is to become part of the system in Figure 1.

5.1 System hierarchy

We concentrate on the overall system structure first.Since MIL 75 is intended to encourage structured programming, it imposes a tree structure on the system under construction. This "system tree" expresses nothing but the hierarchical relation between systems and subsystems. For now, we do not contemplate modules or resources at all. This will happen later, possibly forcing us to refine the system tree during the development of an actual system. Figure 2 shows the tree of the system Thrm_prover.

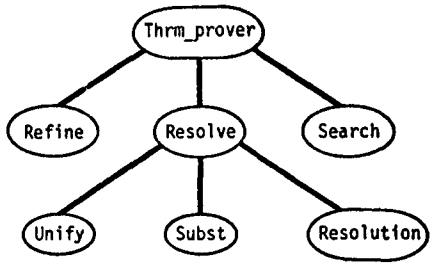

Figure 2. A sample system tree.

Our guideline for the rough decomposition of the project into a tree is that each node should finally encompass an intellectually manageable part of the whole problem, assuming that adequate support is provided by other nodes. Conceptually, there is

for each system tree node a designer who is responsible for the programming associated with "his" node and who supervises the designers of its children.

Definition: A "system tree" is a tuple $T = (N, S, Pa, Sn, t)$ where
(1) N is a finite set of "nodes";
(2) t is a distinguished member of N called the "root";
(3) Pa: $(N-\{t\}) \rightarrow N$ is a total function, called the "parent function", such that for any node n_1 in N there is a sequence of nodes $n_1, n_2, \ldots, n_k$ $(k \geq 1)$ with $n_k = t$ and $Pa(n_i) = n_{i+1}$ $(1 \leq i < k)$.
The terms "child", "sibling", etc. are defined in the obvious manner.
Finally,
(4) S is a finite set of "system names"; and
(5) Sn: $N \rightarrow S$ is a bijection; i.e. each node in N is associated with a unique system name.

When pre-existing subsystems are used in a new system, or when a very large system is constructed, the uniqueness of system names may prove difficult to achieve. Therefore, the syntax of MIL 75 allows "qualified names" (e.g. Resolve.Unify) for unambiguity and "aliases" for renaming.

5.2 Provided and derived resources

The next decisions to be made during the top-down development presumably concern the function of each subsystem. As the function of a subsystem can be completely described in terms of the resources it uses and provides, we now consider the association of resources with system tree nodes.

Ultimately, resources will originate in the LPS modules. Pursuing a top-down approach, however, the designer of any system tree node p states the set of resources provided by p. Then the question is where these resources come from. Some might originate in a module later to be attached to the node p, and thus are the direct responsibility of the designer of p. All other resources must come from any children of p. Therefore, the designer of p states the set of resources each child q must provide. This statement specifies the desired function of q, provided that all resources are adequately specified.

As seen from the node p, the resources it demands from its children are called "derived resources". The node p may derive resources from a child q and provide them to its own parent, in turn. In diagrams, such a case is indicated by a dotted arrow from q to p (cf. Figure 3 below).

Definition: A "resource-augmented system tree" is a tuple $T_R = (T, R, Pr, Mp)$ where
(1) $T = (N, S, Pa, Sn, t)$ is a system tree;
(2) R is a finite set of resources;
(3) Pr: $N \rightarrow 2^R$ is a total function (2^R denotes the powerset of R);
we say that "n provides r" iff $r \in Pr(n)$;
(4) Mp: $N \rightarrow 2^R$ is a total function;
we say that "n must provide r" iff $r \in Mp(n)$;
(5) $Mp(n) \subseteq Pr(n)$ for all n in N; and
(6) $Mp(p) \cap Mp(n) = \emptyset$ for all pairs of siblings p, n.

Naturally, it is a task of an MIL compiler to check that condition (5) is satisfied, i.e. that the bottom-up flow of derived resources is consistent. We allow set inclusion in (5) for facilitating a bottom-up approach, where pre-existing subsystems are used in a new parent system.

5.3 Accessibility

The next refinement is concerned with the interaction between siblings. The power and the responsibility to establish channels for transmitting names of resources between siblings rests solely with their parent. Here we follow Parnas' policy of a "designer controlled information distribution" (Parnas 1971). Consider Figure 3, where the "sibling accessibility links" are drawn as solid arrows between siblings.

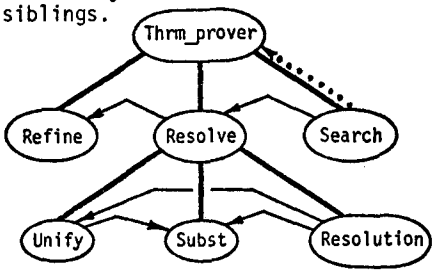

Figure 3. Sibling accessibility links.

These links do not represent individual connections between modules and resources. Rather, they allow the sibling at the tail of the arrow to access any resource provided by the sibling at the arrow head. In Figure 3, for example, Search has access to (any resource provided by) Resolve. For reliability reasons, access rights are non-transitive; for instance, Search has no access to Refine. Also, the children of Resolve are invisible to Search. Thus, Search can access a resource provided by Unify if and only if this resource is also provided by Resolve. In short, the substructure of one sibling is not apparent to another.

The accessibility links between a set of siblings may form any directed graph. Thus, the parent may allow mutual recursion between resources (e.g. procedures, coroutines, or data structures) of its children.

Inherited access. Typically, the access rights granted to a node are also useful for most of its children. In MIL 75, a child inherits by default all access rights that have been granted to its parent. In Figure 3, Resolution inherits access to its "uncle" Refine; in Figure 1a, Parse inherits access to its "granduncle" Clauses.

Alternatively, any parent may "will" a child nothing or an explicitly specified subset of its own access rights, thus making the child less "privileged" and formally asserting that the child and all its descendants cannot exploit or disturb certain resources. If a child is to be partially disinherited, the parent must list all access rights left to the child. Thus, if the parent later obtains additional access rights to vulnerable resources, they do not inadvertently shine through to the less privileged child.

Derived access. Naturally, a parent has access to the resources that it demands from any of its children. However, all descendants of its children are invisible to the parent. Thus, we can build layers of virtual machines as in Figure 4, where

the most privileged nodes are at the bottom.

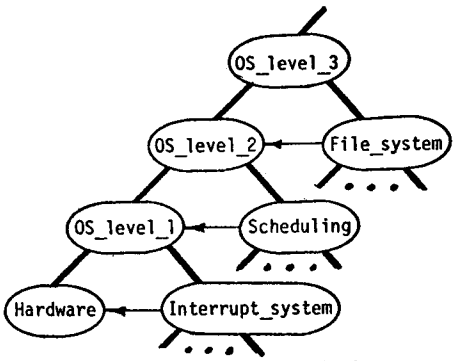

Figure 4. Privileged nodes at the bottom.

Definition: An "access-augmented system tree" is a tuple T_A = (T, Sac, Iac) where

(1) T = (N, S, Pa, Sn, t) is a system tree;
(2) Sac and Iac are relations on N-{t};
(3) p Sac n (pronounced "p has sibling-access to n") implies that p and n are siblings;
(4) p Iac n (pronounced "p inherits access to n") implies that either Pa(p) Sac n or Pa(p) Iac n.

Definition A node p "has access to" a node n iff either p Sac n, p Iac n, or p = Pa(n).

5.4 Module placement

We proceed to place modules into the system tree. With each node, we may associate at most one LPS module, as indicated by the following.

(1) With each leaf n, we must associate a module, the "leaf module" at n. A leaf without module is not allowed, since it cannot provide any resources.
(2) A module associated with a non-leaf n may act as a "driver" or "monitor" of the system named Sn(n). Such a "root module" at n must define all resources in Pr(n) that are not derived from the children of n.
(3) A non-leaf without root module serves as a structural entity only, making an integrated whole out of its subsystems and establishing a single interface to the outside.

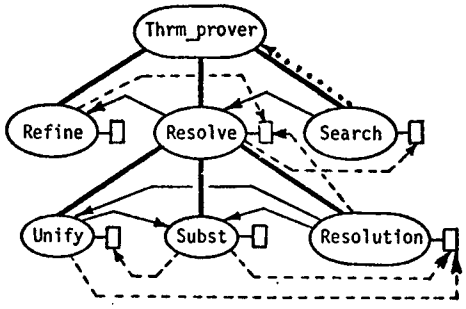

Figure 5. Module placement and usage links.

Different modules might be programmed in different LPSs. The site of the modules is primarily determined by their intellectual manageability, vis-a-vis programming-in-the-small, and secondarily by the quality of the MIL program that binds them together. For if the modules are too small, the MIL program is inconcise and introduces overhead; if the modules are too large, the MIL program does not give enough information about the system structure.

Origin and usage of resources. For each module m at a node n, there must be two statements in the MIL 75 program: (1) the "statement of origin", listing the resources defined in m, and (2) the "statement of usage", listing the resources that are used, but not defined, in m. For clarity, the latter statement is divided into a list of the "derived resources" provided by children of n, and a list of all others, i.e. those obtained through sibling or inherited access.

The compiling system must check that (1) the actual usage of resources by module m conforms to the access rights granted to node n, and that (2) any resource provided by n either comes from a child or originates in module m. No node may provide a resource that is obtained through sibling access or inherited access; such a flow of resources would probably have deleterious effects on reliability.

Usage links. The compiling system can now derive and graphically display the "usage links", drawn as dashed arrows in Figures 1 and 5. If a node n has access to a node p, and the module m at n uses a resource provided by p, then a usage link points from the node p to the module m. Figure 6 shows the three possible cases. Recall that the resource(s) provided by p might not originate in the module at p, if any. However, this is irrelevant to, and hidden from, the module m.

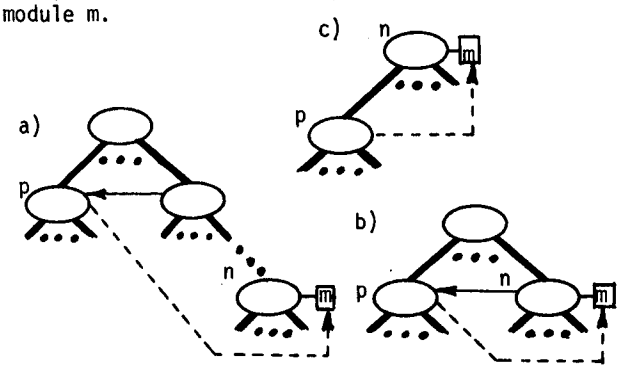

Figure 6. Usage links.

Figure 6 also suggests a graphical check on consistency: we can always form a cycle by traversing the usage link, zero or more tree edges upward, and finally one sibling accessibility link (parts a and b), or by traversing the usage link and one tree edge downward (part c).

Definition: A "module interconnection structure" is a tuple
T_M = (T, T_R, T_A, M, Mod, Or, Ud, Und) where

(1) T = (N, S, Pa, Sn, t) is a system tree, T_R = (T, R, Pr, Mp) is a resource-augmented system tree, and T_A = (T, Sac, Iac) is an acess-augmented system tree;
(2) M is a finite set of modules;
(3) Mod: N → M is a partial, injective function, such that Mod(q) is defined for every leaf q; we say that Mod(n) is the "module at n";
(4) Or, Ud, and Und are total functions N → 2^R, such that Or(n) = Ud(n) = Und(n) = $\emptyset$ if Mod(n) is undefined; we say that r ∈ Or(n) is a

"resource originating in Mod(n)", $r \in Ud(n)$ is a "derived resource used in Mod(n)", and $r \in Und(n)$ is a "non-derived resource used in Mod(n)";
(5) for every p,n in N: $Or(p) \cap Or(n) = \emptyset$;
(6) we define for all $p \in N$ the set of "derived resources"
$D(p) = \{r \in R \mid \exists q \in N: Pa(q) = p \text{ and } r \in Mp(q)\}$;
then, for all $p \in N$,
(7) $Pr(p) \subseteq Or(p) \cup D(p)$;
(8) $Ud(p) \subseteq D(p)$; and
(9) $r \in Und(p)$ implies $(\exists n)[(p \text{ Sac } n \text{ or } p \text{ Iac } n) \text{ and } r \in Mp(n)]$.

5.5 Programming in MIL 75

A complete MIL 75 program consists of a sequence of one-level "system descriptions". Each is assumed to be (re-) compilable alone, or with any others. Put together, they can be translated into a module interconnection structure T_M.

A system description for a node p consists of statements specifying
(1) Sn(p), the designer's name, and a relevant date;
(2) Pr(p);
(3) Mod(p), Or(p), Ud(p), and Und(p);
and for each child q of p:
(4) Sn(q), Mp(q), and $\{n \in N \mid q \text{ Sac } n\}$; and
(5) the set $W = \{n \in N \mid q \text{ Iac } n\}$ either by enumeration (such that $W \subseteq I$ is satisfied) or by default (then W = I is assumed), where
$I = \{n \in N \mid p \text{ Iac } n \text{ or } p \text{ Sac } n\}$.

Phrases specifying empty sets in (3) and (4) above may be omitted. Also, the name of Mod(p) may be left out if it is identical with the system name Sn(p). There follows a sample system description.

```
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
```

Groups. To increase the compactness of MIL 75 programs and to encourage an even more refined granting of access rights, MIL 75 also contains the concept of grouping (cf. George 1973). The designer of a parent can define named subsets or "groups" of the set of derived resources provided by its children. Then, the parent can grant its children access rights to these groups. A group acts as a virtual child of the parent, but has neither module, children, nor access rights. The compiler must check that no child q has access to a group that provides a resource also provided by q, lest the descendants of q inherit access to that resource.

6. Conclusions: The gain in reliability

A module interconnection language can be a significant asset. It embodies a design methodology for reliable software. It provides much needed forms of abstraction, expression, and verification, vis-a-vis programming-in-the-large. It also enhances the effects on reliability that are gained by other methods: management styles; top-down, modular, and structured programming; data abstraction; and information hiding. Reliability is enhanced during system design, the actual programming, system testing, and maintenance and modification.

System Design. Modularization, as a forethought rather than an afterthought, helps in finding reliable solutions to complex problems by applying the traditional method of "divide and conquer". The advantages of modularization, however, are often apparently offset by added complexity in the connections among modules (Liskov 1972). This is not surprising; our problems are not solved by design methodologies that concentrate only on issues of programming-in-the-small. We will obtain more reliable software only if programming-in-the-large is recognized as a separate activity that relies, as heavily as does programming-in-the-small, on a language providing abstraction, structure, and style.

Programming. The results of the system design phase must be communicated to and among the members of the programming team or project. Such communication concerns, among other things, the modules that constitute the system, the position of each module in the hierarchy, the resources each module must provide, and the access rights with which each module is endowed. It is unreasonable to expect that this information can be reliably transmitted via anything but a formal language, or enforced by anything less than a rigorous compiling system.

Testing. A substantial amount of testing can be done by independently exercising each module. However, there is a wide gap between testing individual modules and testing the system as a whole. Performing only these two kinds of tests involves too big a jump in levels of abstraction and results in a gap in confidence. The hierarchical subsystem concept of an MIL suggests a more gradual bottom-up development of the testing phase. Also, the MIL program states clearly, for each subsystem, the connections to other subsystems and who is responsible for testing. In short, the MIL supports "structured testing".

Maintenance and modification. Each connection and dependency between modules is durably documented in the MIL program. No link between modules can be left out of the documentation and be forgotten--the compiling system will complain. Thus, system modifications are more likely to be successful, since their implications are more likely to be foreseen when using an MIL. Furthermore, the hierarchical structure imposed by the MIL makes it easy to replace modules and/or subsystems; and the compiler can support system modification by providing cross-references and graphs of system structure, accessibility links, and usage links. Finally, the modified system structure is automatically checked for consistency.

Outlook. It is obvious that we need languages for programming-in-the-large. An MIL is but a first approximation to such a language, since it does not include facilities for the specification of the function of modules. An MIL may even be regarded as only a "language feature" in the sense of Hoare (Hoare 1973). It seems, however, powerful enough to increase software reliability, even in the

absence of other needed extensions to current languages.

Acknowledgments

We are grateful to Frank Frazier, Nico Habermann, Jim Horning, Jean Ichbiah, Bernard Lorho, Bill McKeeman, Doug Michels, Dan Ross, Sharon Sickel, and Bill Wulf for many helpful comments and stimulating discussions.

References

Clark, B.L., and Horning, J.J. "The system language for project SUE." SIGPLAN Notices 6, 9 (October 1971).

Dijkstra, E.W. "Notes on structured programming." In: Dahl, O.J., Dijkstra, E.W., and Hoare, C.A.W. "Structured Programming." Academic Press, London, New York, 1972.

George, J.E., and Sager, G.R. "Variables--Bindings and protection." SIGPLAN Notices 8, 12 (December 1973).

Guttag, J. "The use of type for the definition of abstract data objects." Dept. of Computer Science, Univ. of Toronto, Ontario, Canada (March 1974).

Hoare, C.A.R. "Proof of correctness of data representations." Acta Informatica 1, 271-281 (1972).

Hoare, C.A.R. "Hints on programming language design." Memo AIM-224, Computer Science Dept., Stanford University (December 1973). Also in: Proc. Symposium on Principles of Programming Languages, Boston (October 1973).

Ichbiah, J.D. "Visibility and separate compilations." Proc. of IFIP WG 2.4, La Grande Motte, France (May 1974).

Knuth, D.E. "Semantics of context-free languages." Mathematical Systems Theory 2, 2 (1968).

Liskov, B.H. "A design methodology for reliable software systems." Proc. Fall Joint Computer Conference, 191-199 (1972).

Liskov, B.H., and Zilles, S. "Programming with abstract data types." Proc. Symposium on Very High Level Languages, SIGPLAN Notices 9, 4 (April 1974).

Maynard, J."Modular Programming". Petrocelli Books, New York, 1972.

Mills, H.D. "Chief programmer teams: Techniques and procedures." IBM Internal Report (January 1970).

Parnas, D.L. "Information distribution aspects of design methodology." Technical Report, Dept. Computer Science, Carnegie-Mellon Univ. (February 1971).

Parnas, D.L. "A technique for software module specifications with examples." CACM 15, 5 (May 1972).

Presser, L., and White, J.R. "Linkers and loaders." ACM Computing Surveys 4, 3 (September 1972).

Richards, M. "The BCPL reference manual." Memo 69/1, The University Mathematical Laboratory, Cambridge, England (January 1969).

Weinberg, G.M. "The psychology of computer programming." Van Nostrand Reinhold Co., New York, 1971.

White, J.R., and Presser, L. "A tool for enforcing system structure." Report CS-11, Dept. of E.E., Univ. of California, Santa Barbara (April 1972).

Wirth, N., and Hoare, C.A.R. "A contribution to the development of ALGOL." CACM 9, 6 (June 1966).

Wulf, W., and Shaw, M. "Global variable considered harmful." SIGPLAN Notices 8, 2 (February 1973).